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

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

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. (A) Yes (B) No

```
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 }
```

When one thread is in the critical section, the other thread will keep looping checking if `free` becomes true without making any progress.

(1p) 1.6. (A) Yes (B) No

```
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 }
```

Here each thread will always make progress in every iteration, as the critical section will be executed.

(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.

There are six important differences:

- A Java 5 lock can have many associated condition variables, unlike an intrinsic lock that has one condition variable
- Java 5 locks can provide fairness
- Locking an unlocking using the `synchronized` keyword is tied to methods or blocks of code
- Using `synchronized` blocks ensures that the lock is unlocked whenever an exception is thrown and control leaves that block of code
- Java 5 locks provide the `tryLock()` method that can be used to acquire the lock without blocking
- Java 5 locks provide the `lockInterruptibly()` method that can be interrupted by thread cancellation

Mentioning any three of these points gives a full mark.

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

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. (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.

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.

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

We will again use the abbreviated version of the program where critical sections are part of the statements following them (p5 and q5).

We make the assumption that the critical sections theselves cannot deadlock. Deadlock occurs if at least one of p or q finishes the non-critical section, but none of them is able to reach the critical section. That is, if the program's state is of the form `(p2, ?, ?)`, `(p3, ?, ?)`, `(?, q2, ?)` or `(?, q3, ?)` and no state of the form `(p5, ?, ?)` or `(?, q5, ?)` is reachable from that state. Freedom from deadlock is the opposite property, that is when the program is able to reach one of the latter states from the former ones.

Technically, we should also show that leaving the critical section does not deadlock. Concretely, this means that a state of the form `(p1, ?, ?)` is reachable from any reachable state of the form `(p5, ?, ?)` and a state of the form `(?, q1, ?)` is reachable from any reachable state of the form `(?, q5, ?)` in the unabbrieviated program. When translating the condition to one for the abbreviated program we must take into account that the non-critical sections may not terminate.

The other condition that we need to show (no deadlock on leaving the critical section) for the abbreviated program is that when the program's state is of the form `(p5, ?, ?)`, it will eventually reach a a state of the form `(p2, ?, ?)` unless p's non-critical section does not terminate; and similarly that when the program's state is of the form `(?, q5, ?)`, it will eventually reach a a state of the form `(?, q2, ?)` unless q's non-critical section does not terminate. Showing this is straightforward, as statements in p5 and q5 will never block.

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

First, looking at the table we show that starting from any reachable state of the form `(p2, ?, ?)` the program can progress to a state of the form `(p3, ?, ?)`,

and similarly from any state of the form `(?, q2, ?)` the program can progress to a state of the form `(?, q3, ?)`. Secondly, from the table we can see that for any reachable state of the form `(p3, ?, ?)` the program can progress to a state of the form `(p5, ?, ?)` or `(?, q5, ?)`. Similarly, starting with `(?, q3, ?)` the program will also reach a state of the form `(p5, ?, ?)` or `(?, q5, ?)`.

Showing no deadlock in leaving the critical section is straightforward, as statements in p5 and q5 will never block.

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

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

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

Part 3: Concurrent Java 1 (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.

The code can deadlock. For example, consider that there are two accounts in the table: account 0 and account 1. If two threads try to transfer money between these accounts, a deadlock may occur.

Here is a concrete situation. Both accounts have 10 dollars on them. Thread 1 wants to transfer 10 dollars from account 0 to account 1, while thread 2 wants

to transfer 10 dollars the other way round. Thread 1 locks the lock of account 0, followed by thread 2 locking the lock of account 1. Then both threads will deadlock waiting for the locks of the accounts that were locked by the other thread.

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

We fix the problem by introducing a global locking order on all locks in the accounts table. We will ensure that if a thread locks two locks at the same time, the lock of the account with a lower ID will be locked first. This way we will avoid the circular locking situation.

To implement this, we will only need to modify the `transfer()` method. We add this code between lines 7 and 8 of the `Accounts` class.

```
int lower = Math.min(source, target);
int upper = Math.max(source, target);
```

We also modify lines 8, 10, 17 and 18 of the `Accounts` class (the line numbers refer to the original ones before our additions).

```
 8      store[lower].l.lock()
10        store[upper].l.lock()
17        } finally {store[upper].l.unlock() }
18      } finally {store[lower].l.unlock() }
```

Instead of `Math.min` and `Math.max()`, `if`-statemets may be used. Here are some other ways in which this problem can be solved.

- By using `tryLock()` with a timeout on to lock the second lock, and when that fails unlocking the first lock and retrying. This solution would not get the full points, because it might end up performing polling.
- By using the non-blocking `tryLock()` and waiting some (random) amount of time before retrying. This solution might also end up polling, and would not get the full points.

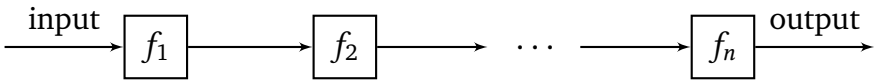
Here is the full code of the solution for reference.

```
 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      int lower = Math.min(source, target);
 9      int upper = Math.max(source, target);
10      store[lower].l.lock()
11      try {
12        store[upper].l.lock()
```

```
13        try {
14          int tmp = store[source].getBalance();
15          if (tmp < amount) return false; // Transfer failed
16          store[source].updateBalance(tmp - amount);
17          store[target].updateBalance(store[target].getBalance()
18                                      + amount);
19        } finally {store[upper].l.unlock() }
20      } finally {store[lower].l.unlock() }
21      return true;
22    }
23  }
```

Part 4: Concurrent Java II (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.

```
1  class Pipeline {
2    SThread[] threads;
3
4    // A thread that runs a single stage of the pipeline
5    // The object contains a one-slot buffer
6    static class SThread extends Thread {
7      Stage s;
8      SThread next;
9      // Input slot for the stage
10     boolean free = true;
11     T slot;
12
13     SThread(Stage ss, SThread n) {
14       s = ss;
15       next = n;
16     }
17     synchronized void put(T t) throws InterruptedException {
18       while (!free) wait();
19       slot = t;
20       free = false;
21       notifyAll();
22     }
```

```
23      synchronized T get() throws InterruptedException {
24        while (free) wait();
25        T tmp = slot;
26        free = true;
27        slot = null;
28        notifyAll();
29        return tmp;
30      }
31      public void run() {
32        try {
33          while(true) {
34            T cur = get();
35            T res = s.compute(cur);
36            if (next != null) next.put(res);
37          }
38        } catch (InterruptedException e) {
39          System.err.println("Thread interrupted");
40        }
41      }
42    }
43    // Initialize the pipeline
44    public Pipeline(Stage[] stages) {
45      int n = stages.length;
46      threads = new SThread[n];
47      SThread s = null;
48      // Create threads starting from the last one
49      // Each thread gets a reference to the next one
50      for (int i = n-1; i >= 0; --i) {
51        SThread tmp = new SThread(stages[i], s);
52        threads[i] = tmp;
53        s = tmp;
54      }
55    }
56    // Start the threads
57    public void run () throws InterruptedException {
58      for(int i = 0; i < threads.length; ++i) threads[i].start();
59    }
60    // Feed one more input to the pipeline
61    // Don't allow any more data after join() has been callled.
62    public void feed(T input) throws InterruptedException {
63      threads[0].put(input);
64    }
65  }
```

The above code handles the `InterruptedException`, but you can ignore it without

losing points.

(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.

The code below is a modification of the answer for 4.1.

```
1  class Pipeline {
2    SThread[] threads;
3    boolean finishing = false;
4
5    // A thread that runs a single stage of the pipeline
6    // The object contains a one-slot buffer
7    static class SThread extends Thread {
8      Stage s;
9      SThread next;
10     // Input slot for the stage
11     boolean free = true;
12     int waitingPuts = 0;
13     T slot;
14     boolean terminating = false;
15
16     SThread(Stage ss, SThread n) {
17       s = ss;
18       next = n;
19     }
20
21     synchronized void put(T t) throws InterruptedException {
22       ++waitingPuts;
23       while (!free) wait();
24       slot = t;
25       free = false;
26       --waitingPuts;
27       notifyAll();
28     }
29
30     // If get() gets woken up by terminate() and there
31     // is not element, then it will return null
32     synchronized T get() throws InterruptedException {
33       while (!(!free || passedLastOne())) wait();
34       T tmp = slot;
35       free = true;
36       slot = null;
37       notifyAll();
38       return tmp;
```

```
39      }
40
41      // Set the terminating flag to true and
42      // wake up a potentially waiting get()
43      synchronized void terminate() {
44        terminating = true;
45        notifyAll();
46      }
47
48      // This condition is true if there will be no
49      // more elements
50      synchronized boolean passedLastOne() {
51        return waitingPuts == 0 && free && terminating;
52      }
53
54      public void run() {
55        try {
56          while(!passedLastOne()) {
57            T cur = get();
58            if (cur == null) break;
59            T res = s.compute(cur);
60            if (next != null) next.put(res);
61          }
62          if(next != null) next.terminate();
63        } catch (InterruptedException e) {
64          System.err.println("Thread interrupted");
65        }
66      }
67    }
68
69    // Initialize the pipeline
70    public Pipeline(Stage[] stages) {
71      int n = stages.length;
72      threads = new SThread[n];
73      SThread s = null;
74      // Create threads starting from the last one
75      // Each thread gets a reference to the next one
76      for (int i = n-1; i >= 0; --i) {
77        SThread tmp = new SThread(stages[i], s);
78        threads[i] = tmp;
79        s = tmp;
80      }
81    }
82    // Start the threads
83    public void run () throws InterruptedException {
```

```
84      for(int i = 0; i < threads.length; ++i) threads[i].start();
85    }
86    // Feed one more input to the pipeline
87    public void feed(T input) throws InterruptedException {
88      synchronized (this) {
89        if (finishing) throw new RuntimeException();
90      }
91        threads[0].put(input);
92    }
93
94    public void join() throws InterruptedException {
95      synchronized (this) {
96        finishing = true;
97      }
98      // Send the terminate signal to the first stage
99      threads[0].terminate();
100     // Wait for the last stage to finish
101     threads[threads.length-1].join();
102   }
103 }
```

As previously, any handling of `InterruptedException` may be omitted. The code above requires the locks to be fair, which is not possible with intrinsic Java locks (`synchronized`), but instead of rewriting it using Java 5 locks we will assume that intrinsic locks do provide fairness. It is fine to do it on the exam as long as you explicitly say that you assume fairness.

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.

The problem occurs when two processes run `start()` concurrently, and both of them execute `whereis(myserver)` one after another. Then both of them receive `undefined` from `whereis(myserver)`, and choose the first clause of the case-statement. Then both spawn a new process and try to register it, which ends with one succeeding and one failing to do it. The failing process will crash with a `badarg` exception, while the succeeding one will finish executing the function properly. Furthermore, there will be an additional server process running, which will not be registered.

Note that the failing process will throw an exception instead of returning `{error, already_started}` as expected.

(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.

```
1  % Try to register the current process and
2  % send the result back to the parent process.
3  % If successful, run init(). If not, quit.
4  init_reg(Parent, Ref) ->
5    case (catch register(myserver, self())) of
6      {'EXIT', _} ->
7        Parent ! {Ref, {error, already_started}};
8      true        ->
9        Parent ! {Ref, {ok, self()}},
10       init()
11   end.
12
13 start() ->
14   case whereis(myserver) of
```

```
15      undefined ->
16        Ref = make_ref(),
17        Parent = self(),
18        spawn(fun () -> init_reg(Parent, Ref) end),
19        receive {Ref, Res} -> Res end;
20      Pid when is_pid(Pid) ->
21        {error, already_started}
22    end.
```

To solve this problem you need to spawn a process, try to register it, and if registering fails, terminate it. The easiest way to do it is to create a wrapper function that will try to perform the registration in the newly-spawned process, and then send the results back to the parent process.

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

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.

```
1  -module(bbuffer).
2
3  -export([start/1, get/1, put/2]).
4
5  start(N) ->
6    spawn(fun () -> loop ([], N) end).
7
8  % L is the queue, N is the number of free slots
9  loop(L, N) ->
10   receive
11     {get, P, Ref} when L /= [] ->
12       [X|Xs] = L,
13       P ! {get_reply, Ref, X},
14       loop(Xs, N + 1);
15     {put, P, Ref, Y} when N > 0 ->
16       P ! {put_reply, Ref},
17       loop(L ++ [Y], N - 1)
18   end.
19
20 get(B) ->
21   Ref = make_ref(),
22   B ! {get, self(), Ref},
23   receive
24     {get_reply, Ref, X} -> X
25   end.
26
27 put(B, X) ->
28   Ref = make_ref(),
29   B ! {put, self(), Ref, X},
30   receive
```

```
31      {put_reply, Ref} -> ok
32    end.
```

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

(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.

```
1  -module(bbuffer).
2
3  -export([start/1, get/1, put/2, unload/1, release/1]).
4
5  start(N) ->
6    spawn(fun () -> loop_prio ([], N) end).
7
8  % L is the queue, N is the number of free slots
9  % First we want to check if there is an outstanding unload message
10 loop_prio(L, N) ->
11   receive
12     {unload, P, Ref} ->
13       loop_unload(L, N, {P, Ref})
14     after 0 -> loop(L, N)
15   end.
16
17 loop(L, N) ->
18   receive
19     {unload, P, Ref} ->
20       loop_unload(L, N, {P, Ref});
21     {get, P, Ref} when L /= [] ->
22       [X|Xs] = L,
23       P ! {get_reply, Ref, X},
24       loop_prio(Xs, N + 1);
25     {put, P, Ref, Y} when N > 0 ->
26       P ! {put_reply, Ref},
27       loop_prio(L ++ [Y], N - 1)
28   end.
29
30 % Get rid of all elements, then reply to the
31 % unload operation, then wait for a release
```

```
32 loop_unload([], N, {P, Ref}) ->
33   P ! {unload_reply, Ref},
34   receive
35     {release, P2, Ref2} ->
36       P2 ! {release_reply, Ref2},
37       loop_prio([], N)
38   end;
39 loop_unload([X|Xs], N, PR) ->
40   receive
41     {get, P, Ref} ->
42       P ! {get_reply, Ref, X},
43       loop_unload(Xs, N + 1, PR)
44   end.
45
46 get(B) ->
47   Ref = make_ref(),
48   B ! {get, self(), Ref},
49   receive
50     {get_reply, Ref, X} -> X
51   end.
52
53 put(B, X) ->
54   Ref = make_ref(),
55   B ! {put, self(), Ref, X},
56   receive
57     {put_reply, Ref} -> ok
58   end.
59
60 unload(B) ->
61   Ref = make_ref(),
62   B ! {unload, self(), Ref},
63   receive
64     {unload_reply, Ref} -> ok
65   end.
66
67 release(B) ->
68   Ref = make_ref(),
69   B ! {release, self(), Ref},
70   receive
71     {release_reply, Ref} -> ok
72   end.
```

We assume the `release()` will only take effect after the buffer is emptied, which is a reasonable assumption.

We use the receive-after construct to receive the `unload` messages out-of-order.

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> 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
```