Chalmers | GÖTEBORGS UNIVERSITET
Karol Ostrovský, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Wednesday, October 21, 2009, 14.00 – 18.00, V.

(including example solutions to programming problems)

Karol Ostrovský, tel. 772 1065
Nicholas Smallbone, tel. 772 1079

- Grading scale (Betygsgränser):

  Chalmers: 3 = 20–29 points, 4 = 30–39 points , 5 = 40–50 points
  Chalmers ETCS: E = 20–23, D = 24–29, C = 30–37, B = 38–43, A = 44–50
  GU: Godkänd 20–39 points, Väl godkänd 40–50 points

  Total points on the exam: 50

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):**
  - Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answer in either Swedish or English. The exact syntax of program code is not important as long as it is clear to us that you understand what you are doing.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of each programming language you are going to use is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.

Question 1. Implement a simulation of exam grading for concurrent programming. There are two kinds of processes: the examiner (one process) and assistants (*N* processes). The assistants grade the exam sitting at the grading desk. There is only one desk for grading (university budget cuts), which needs to be shared by the assistants and the examiner. The assistants grade one exam at a time taking turns at the desk and having coffee in the mean time. The examiner does not actually grade the exam, but he needs exclusive access to the desk when he brings new exam papers. Furthermore, the examiner is a gentle person, he may only bring `20` exam papers to the grading desk at a time. His job is to bring a new batch of exams only when the assistants have finished grading the previous one.

(a) Implement an infinite simulation (meaning that there are an infinite number of exams) of the above pattern using semaphores. No monitors or message passing may be used. You do not need to think about any fairness issues. *(8p)*

(b) Write an LTL property which holds in the simulation described above. When writing properties which refer to the state of the simulation you can use just ordinary text. For example, you can write

$$\Box(\text{The examiner holds 30 exams})$$

(Although note that this particular property is false.)
To get full points the LTL property should be a liveness property. *(2p).*

Solution 1.

```
public class Grading {
  int exams = 0
  sem desk = 0
  sem more = 1

  process examiner {
    while (true) {
      P(more);
      //get another batch of exam papers
      exams = 20;
      V(desk);
    }
  }

  process assistant((int i = 0; i<N; i++)) {
    while (true) {
      P(desk);
      exams--;
      //grade
      if (exams==0) V(more);
      else          V(desk);
      JR.nap(1000); //coffee time
    }
  }
}
```

Question 2. During the course we have seen several solutions to the cyclic barrier problem. You should be very familiar with the problem. In this question you are asked to implement a small variation of this problem using rendezvous communication (hint: input statement in JR). Here is the skeleton of the class you should implement:

```
public class CyclicBarrier {

  public op int await();

  private int N;

  public CyclicBarrier(int anN) {
    N = anN;
  }

  private process server {
    //
    // This needs some work...
    //
  }
}
```

The CyclicBarrier should work in the standard way blocking each caller of `await` until all the `N` processes arrive. In this particular implementation you also have to return the arrival index of the process with `0` returned to the first process and `N-1` returned to the last process arriving at the barrier. For example the following test program should print "`1, 0, 2.`"

```
  static CyclicBarrier cb = new CyclicBarrier(3);
  process P1 {
    JR.nap(2000);
    int index = cb.await();
    JR.nap(1000);
    System.out.print(index+", ");
  }
  process P2 {
    JR.nap(1000);
    int index = cb.await();
    JR.nap(2000);
    System.out.print(index+", ");
  }
  process P3 {
    JR.nap(3000);
    int index = cb.await();
    JR.nap(3000);
    System.out.print(index+".");
  }
```

(6p)

Solution 2.

```
private process server {
  while (true)
    inni int await() st await.length() >= N {
        for(int x=1; x<N; x++)
          inni int await() {
            return x;
          }
        return 0;
    }
}
```

Question 3. The objective of this question is to design and implement an Erlang module which provides a Linda-style tuple space and some of the standard tuple-space primitives. The interface to the module is given be the following collection of functions:

- `new()` – returns the `Pid` of a new (empty) tuplespace.
- `in(TS, Pattern)` – returns a tuple matching `Pattern` from tuplespace `TS`. The operation blocks until a tuple in the tuple space matches the `Pattern`. Then the matching tuple is atomically removed from the tuple space `TS` and returned.
- `out(TS, Tuple)` – atomically puts `Tuple` into the tuplespace `TS`.

To make it simpler than the assignment 3, we provide the matching function and a function `find` that given a pattern and a list of tuples returns either `{value, T, OtherTuples}` when a matching tuple `T` is found and removed from the list (the list `OtherTuples` contains the remaining tuples after `T` was removed from the list `Tuples`) or an atom `nix` when a matching tuple was not found:

```
match(any,_) -> true;
match(P,Q) when tuple(P), tuple(Q) ->
    match(tuple_to_list(P),tuple_to_list(Q));
match([P|PS],[L|LS]) ->
    case match(P,L) of
        true -> match(PS,LS);
        false -> false
    end;
match(P,P) -> true;
match(_,_) -> false.

find(Pattern, Tuples) -> find_a(Pattern, Tuples, []).

find_a(Pattern, [T | Ts], Acc) ->
    case match(Pattern, T) of
        true -> {value, T, Acc++Ts};
        false -> find_a(Pattern, Ts, [T|Acc])
    end;
find_a(_, [], _) ->
    nix.
```

You might find the append function ++ useful. It concatenates two lists, for example: `[1]++[2,3]` gives you the list `[1,2,3]`. No other library functions are necessary to solve this problem. *(8p)*

Solution 3.

```
new() ->
  spawn(fun() -> loop([], []) end).

in(TS, Pattern) ->
  Ref = make_ref(),
  TS!{in, self(), Ref, Pattern},
  receive
    {result, Ref, Tuple} ->
      Tuple
  end.

out(TS, Tuple) ->
  TS!{out, Tuple}.

loop(Ts, Ps) ->
  receive
    {in, From, Ref, Pattern} ->
      case find(Pattern, TS) of
        {value, Tuple, New_Ts} ->
          From!{result, Ref, Tuple},
          loop(New_Ts, Ps);
        nix ->
          loop(Ts, [{From, Ref, Pattern}|Ps])
      end;
    {out, Tuple} ->
      case check_pending(Tuple, Ps, []) of
        {sent, New_Ps} -> loop(Ts, New_Ps);
        nix -> loop([Tuple|Ts], Ps)
      end
  end.

check_pending(Tuple, [P={From, Ref, Pattern} | Ps], Acc) ->
  case match(Pattern, Tuple) of
      true ->
          From!{result, Ref, Tuple},
          {sent, Acc++Ps};
      false ->
          check_pending(Tuple, Ps, [P|Acc])
  end;
check_pending(_, [], _) ->
  nix.
```

Question 4. The following three questions ask you to solve variants of so-called multicast channels. Please, read the introduction carefully as it is relevant in all three questions.

Introduction

A multicast channel provides communication from one sending port to several independent receiving ports. When a sender sends a message to a multicast channel, it is buffered internally, and later received exactly once at each receiving port. A sender atomically appends a message to the end of the channel and continues its execution. Multiple senders are allowed. Each receiver owns its own receiving port to the multicast channel. This way each receiver behaves independently and receives messages at its own pace. A receiver can potentially block if it has read all the messages and it is waiting for a new message to arrive.

The intended interface for multicast channels is as follows:

```
public interface Port<T> {
    public T receive() throws InterruptedException;
}

public interface MChan<T> {
    public void send(T item) throws InterruptedException;

    public Port<T> newPort();
}
```

- `send` – a sender calls this method to atomically append a message to the end of the channel;
- `newPort` – a receiver must first call this method to obtain its own receiving port, which initially points to the current end of the channel;
- `receive` – a receiver calls this method to receive one message from the channel; this method blocks when no message is currently available.

To illustrate the workings of a multicast channel and especially the independent receiving ports consider this simple sequential example, which prints "`P2<-66, P1<-42, P1<-66.`" and then deadlocks.

```
MChan<Integer> mc = new MyMChan<Integer>();
Port<Integer> p1 = mc.newPort();
mc.send(42);
Port<Integer> p2 = mc.newPort();
mc.send(66);
System.out.print("P2<-"+p2.receive()+", ");
System.out.print("P1<-"+p1.receive()+", ");
System.out.print("P1<-"+p1.receive()+".");
System.out.print("P1<-"+p1.receive()+"?");
```

The Problem

In this question 4 you are asked to implement the multicast channel using software transactional memory (STM). The additional requirements are:

- The channel has unbounded size. A sender can always send a new message and is never blocked. However, you need to make sure that you only use memory for messages that have a waiting receiver. (No memory leaks if you want full points.)

- Receivers receiving different messages must be allowed to proceed in parallel. For example if process P1 is trying to receive the first message in the channel while a different process P2 is trying to receive the second message in the channel at the same time, both processes must be allowed to receive their respective messages at the same time. (hint: think about the readers/writers problem with a twist)
- You have a choice of using either Haskell or Java for your solution. The necessary STM functionality for each language is described at the back of the exam.

(7p)

Solution 4. Problem and solution from "Composable Memory Transactions" by T.Harris, S.P.Jones, and M.Herlihy, PPoPP 2005

```
type Chain a = TVar (Item a)
data Item a  = Empty | Full a (Chain a)

type MChan a = TVar (Chain a)
type Port a  = TVar (Chain a)

newMChan   = do c <- newTVar Empty
                newTVar c

newPort mc = do c <- readTVar mc
                newTVar c

readPort p = do c <- readTVar p
                i <- readTVar c
                case i of
                  Empty     -> retry
                  Full v c' -> do writeTVar p c'
                                  return v

writeMChan mc v = do c  <- readTVar mc
                     c' <- newTVar Empty
                     writeTVar c (Full v c')
                     writeTVar mc c'
```

Question 5. Implement the very same multicast channel as in question 4 using Java monitor synchronisation instead of software transactional memory. *(7p)*

Solution 5. There are several possible ways to solve this problem. Here is just one possibility:

```
public class MChan2<T> {

    private class Chain<T> {
        private T item = null;
        private Chain<T> next = null;
        public synchronized Chain<T> write(T anItem)
                throws InterruptedException {
            item = anItem;
            next = new Chain<T>();
            notifyAll();
            return next;
        }
        public synchronized T read()
                throws InterruptedException {
            while (next == null)
              wait();
            return item;
        }
        public synchronized Chain<T> getNext() {
            return next;
        }
    }

    private class PortImpl implements Port<T> {
        private Chain<T> head;
        public PortImpl() {
            head = tail;
        }
        public T read() throws InterruptedException {
            T ret = head.read();
            head = head.getNext();
            return ret;
        }
    }

    private Chain<T> tail;

    public MChan2() {
        tail = new Chain<T>();
    }

    public synchronized void write(T item)
           throws InterruptedException {
        tail = tail.write(item);
    }

    public synchronized Port<T> newPort() {
        return new PortImpl();
    }
}
```

Question 6. As the final twist in the multicast channel saga implement the following variant:

- A channel can store the maximum of `N` messages.
- The key consequence of this new requirement is that the method/function `send` can block if the channel already contains `N` messages. It blocks until all the receivers received enough messages so that less than `N` are still stored in the channel.
- To obtain full points for this question you still need to provide a solution that allows parallel receivers just as in the previous two questions. (No memory leaks either.)

This new variant of multicast channels might seem like a mild variation of the previous two questions. However, depending on your previous solutions, this variant is sufficiently different and more involved. Therefore, in this question you can choose whatever synchronisation mechanism you find best suited for this problem. You should write a motivation for your choice. *(12p)*

Solution 6. Again, there are many ways to implement this. One example is here:

```
import java.util.*;

public class MChanNPlus<T> {

    private T[] buf;
    private int size;
    private volatile int tail;
    private RW sync = new RW();
    private Set<PortImpl> readers = new HashSet<PortImpl>();

    public MChanNPlus(int aSize) {
        size = aSize;
        buf = (T[]) new Object[size];
        tail = 0;
    }

    public synchronized void write(T item)
           throws InterruptedException {
        sync.startWrite();
        buf[tail] = item;
        tail = (tail+1)%size;
        sync.endWrite();
    }

    public synchronized Port<T> newPort() {
        PortImpl ret = new PortImpl();
        readers.add(ret);
        return ret;
    }
    private class PortImpl implements Port<T> {
        private volatile int head;
        public PortImpl() {
            head = tail;
        }
        public T read() throws InterruptedException {
            sync.startRead(head);
            T ret = buf[head];
            head = (head+1)%size;
            sync.endRead();
            return ret;
        }
    }

    private class RW {

        public synchronized void startRead(int head)
               throws InterruptedException {
            while (head==tail)
                wait();
        }

        public synchronized void endRead() {
            notifyAll();
        }

        public synchronized void startWrite()
               throws InterruptedException {
            while (maxxed())
                wait();
        }

        public synchronized void endWrite() {
            notifyAll();
        }

        private boolean maxxed() {
            boolean ret = false;
            for(PortImpl r : readers)
                if (((tail+1)%size) == r.head)
                    ret = true;
             return ret;
        }
    }
}
```

Haskell STM Here are the relevant parts of the standard Haskell library for software transactional memory, which you might need in this exam:

```
module Control.Concurrent.STM
instance Monad STM
data STM a
data TVar a

newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
retry      :: STM a
```

A brief explanation of how the library works: function `newTVar` creates a new shared variable. You can store a value in the shared variable using function `writeTVar` and read the stored value using function `readTVar`. The function `retry` is used for conditional synchronization. When `retry` is called the transaction is aborted and restarted some time in the future.

Since you cannot implement the intended multicast channel Java interface directly in Haskell, your task (if using Haskell) is instead to provide two types `MChan a` and `Port a` that represent a multicast channel and write the following Haskell functions that access the channel

```
newMChan :: STM (MChan a)
send     :: MChan a -> a -> STM ()
newPort  :: MChan a -> STM (Port a)
receive  :: Port a -> STM a
```

Java STM Java does not have any standard STM library. Here is one such possible library that you can use in this exam:

```
class TransactionManager {
  public TransactionManager();
  public void beginTransaction();
  public void endTransaction();
  public TransactionVariable<A> newTransactionVariable();
  public void retry();
}

class TransactionVariable<A> {
  public A read();
  public void write(A value);
}
```

A brief explanation of how the library works: A `TransactionManager` object handles all the low level details of how transactions are handled. Whenever you want to initiate a transaction call the method `beginTransaction` and end the transaction with a call to `endTransaction`. Shared variables, which are only accessed inside a transaction, can be created using method `newTransactionVariable` and are of the type `TransactionVariable<A>` where `A` is the type of the data contained in the variable. The content of the variable is accessed and manipulated by the methods `read` and `write`. The method `retry` is used for conditional synchronization. When `retry` is called the transaction is aborted and restarted some time in the future.