Chalmers | GÖTEBORGS UNIVERSITET
Karol Ostrovský, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Friday, October 29, 2007, 14.00-18.00

(including example solutions to programming problems)

Daniel Hedin, tel. 772 5422

- 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.

Question 1. (a) Give an example (in words) of a safety property that you would expect to hold for a correct solution to the train controller assignment (Lab 1: Trainspotting). *(2p)*

(b) Explain why the `wait` operation in Java is always found inside a while-loop. Are there situations where it is not necessary to have a `wait` inside a while-loop? Explain your answer. *(3p)*

Solution 1. First is the issue of signal-and-continue semantics, which can lead to invalid wait conditions by the time the waiter is allowed to execute. Using the "passing the condition" could avoid the while loops but not in Java. Java allows spurious wake-up! In Java `wait` can wake-up on its own.

Question 2. *Shared variable update* is one of the key issues in concurrent programming.

(a) Give an example program in a programming language of your choice (Java/JR/MPD/Erlang) that illustrates a problem that can be caused by concurrent updates of a shared variable. *(2p)*

(b) Give or describe (if it is infinite) a sequence of execution steps (a trace) of your example program that leads the problem. *(2p)*

(c) Modify your example program to eliminate the occurrence of the problem. *(1p)*

Question 3. (a) A resource allocator makes a process wait only when there are insufficient resources to satisfy its allocation request. Explain what is wrong with the following implementation of a resource allocator: *(3p)*

```
public class ResourceAllocator {

   public op void allocate(int n);
   public op void replace(int n);

   private sem available = XXX;

   public void allocate(int n) {
      for(int i=0; i<n; i++)
         P(available);
   }

   public void replace(int n) {
      for(int i=0; i<n; i++)
         V(available);
   }
}
```

(b) Suppose that we modify the body of the above resource allocator by adding an additional private semaphore `sem mutex=1` and by modifying the body of `allocate` to:

```
P(mutex);  for(int i=0; i<n; i++) P(available);  V(mutex);
```

Is this version correct? Explain your answer. *(3p)*

Solution 3. As explained in the lectures: assume we have 2 resources, that is `available = 2`, and two processes P1 and P2 that both make a call to allocate 2 resources concurrently, that is `allocate(2)`. Both processes P1 and P2 start running the operation `allocate` and each manages to acquire the semaphore `available` once, that is they each do `P(available)`. Then, both processes will try to acquire the semaphore again but become dead-locked.

The fix in (b) is not much of a fix as the following situation can happen: assume that the resource allocator currently holds one resource, that is `available = 1`. Process P2 wants to acquire 2 resources: calls `allocate(2)`, and blocks because there is currently only one resource available. Now, another process P2 wants to acquire one resource: calls `allocate(1)`, and blocks because there is another process inside the critical section protected by `mutex`. But this call was supposed to succeed as the resource allocator was one unit available.

Question 4. Write code which implements a *bounded buffer* for at most *M* integers. Your interface to the buffer should be provided by two methods/functions:

- `void put(int i)`
- `int get()`

You should provide an implementation of these methods/functions (and anything else you might need). Your buffer should allow arbitrary number of producers and consumers to use these operations. You do not have to write code representing the producer or consumer processes.

You should make use of JR asynchronous message passing (i.e. the non-blocking `send`, together with `receive` or input statements) as the only form of process synchronisation. Alternatively, you can provide a solution in Erlang which only supports asynchronous message passing. *(10p)*

Solution 4. A question from exam 2003-10-24.

Question 5. Using monitor synchronisation, implement the body of your own class `CyclicBarrier` which provides similar features as the Java class of the same name. The skeleton code looks as follows:

```
public class CyclicBarrier {

   public CyclicBarrier(int N);

   public void await();
}
```

The cyclic barrier allows *N* processes to synchronise in the following way:

- The cyclic barrier has one method `await()`;
- when a process calls `await()` it blocks;
- then, when all *N* processes entered the barrier they are unblocked and the next barrier cycle can begin.

(4p)

Solution 5. Exercise 2.

Question 6. You need to implement a simulation of access to a Japanese onsen (a Japanese version of spa where you soak in a natural hot-spring). The onsen features two separate areas (let us call them area 1 and area 2) of capacity 30 and 40, respectively. Women and men are admitted to the onsen. This is a modern-style onsen where women and men are not allowed to mix in a single area. When the onsen opens, area 1 is used by women, and area 2 is used by men.

The main interface for the onsen simulation consists of a sex enumeration and the main access operation:

```
public enum Sex { Male, Female };
public op void doBathing(Sex s, cap void(int) notify);
```

Using message-passing (and no other synchronization primitives), implement a server that grants access to this onsen.

(a) Since the water in this onsen is rather hot visitors are not allowed to stay inside for more than 20 minutes. But each visitor really wants to thoroughly enjoy the experience so everyone stays for the maximum allowed time. Your initial implementation should service the operation `doBathing(Sex s, cap void(int) notify)` which is serviced only when there is a space in the appropriate area. When the visitor is allowed to enter a message with the area number is send on the channel `notify`, and an internal timer for 20 minutes is started. The operation `doBathing` still keeps the caller blocking. It only returns after the visitor spent the 20 minutes in the area (simulating a 20 minute stay in the appropriate area of this onsen). (Hint: JR has a very useful forward statement.) *(5p)*

(b) When the onsen opens, area 1 is used by women, and area 2 is used by men. However, area 2 has a stunning view over a winding river. In order to be fair to both women and men, the onsen master swaps areas by calling `swap()` every now and then. As an effect of this call, the server should swap the areas between women and men. Of course, an area swap should be enforced correctly: no women are allowed to enter until all men have left, and vice versa. (Hint: JR has capabilities, which are references to operations) *(5p)*

As a further hint we provide an example code that represents a possible visitors behaviour:

```
private process bathMale((int i=0;i<10; i++)) {
   op void notify(int a);
   send printer(Sex.Male, i, notify);
   doBathing(Sex.Male, notify);
   //Done bathing.
   send notify(0);
}

private process bathFemale((int i=0;i<10; i++)) {
   op void notify(int a);
   send printer(Sex.Female, i, notify);
   doBathing(Sex.Female, notify);
   //Done bathing.
   send notify(0);
```

```
    }

    private op void printer(Sex s, int i, cap void(int) notify) {
       int area;
       receive notify(area);
       System.out.println(i+s.toString()+" bathing in area "+area);
       receive notify(area);
       System.out.println(i+s.toString()+" done");
    }
```

Solution 6.
```
public class Spa {
    public enum Sex { Male, Female };
    public op void doBathing(Sex s, cap void(int));
    public op void swap();

    private op void timeout(int index);
    private op void a1(int index);
    private op void a2(int index);
    private cap void(int) area1, area2;
    private int size2, size1, index = 0;
    private boolean swapping = false;

    public Spa() {
      area1 = a1;
      area2 = a2;
      size1 = 3;
      size2 = 4;
    }

    private op void timer(int ms, int index) {
      edu.ucdavis.jr.JR.nap(ms);
      send timeout(index);
    }

    private boolean isFree(Sex s) {
      boolean ret = false;
      switch (s) {
      case Male:
        ret = area2.length() < size2;
        break;
      case Female:
        ret = area1.length() < size1;
        break;
      }
      return ret;
    }
```

```
private process server {
  while (true) {
    inni void doBathing(Sex s, cap void(int) notify) st isFree(s) && !swapping {
        send timer(2000, index);
        switch (s) {
        case Male:
          send notify(area2==a2 ? 2 : 1);
          forward area2(index);
          break;
        case Female:
          send notify(area1==a1 ? 1 : 2);
          forward area1(index);
          break;
        }
        index++;
      }
    [] void timeout(int index) {
      inni void area1(int ix) st index==ix {
        }
      [] void area2(int ix) st index==ix {
        }
      if (area1.length()==0 && area2.length()==0 && swapping) {
        cap void(int) tmp = area2;
        area2 = area1;
        area1 = tmp;
        int tmpSize = size2;
        size2 = size1;
        size1 = tmpSize;
        swapping = false;
        System.out.println("Swapped");
      }
    }
    [] void swap() {
      swapping = true;
      System.out.println("Swapping...");
    }
  }
}

private process bathM((int i=0;i<10; i++)) {
  op void notify(int a);
  send printer(Sex.Male, i, notify);
  doBathing(Sex.Male, notify);
  //Done bathing.
  send notify(0);
}
```

```
  private process bathF((int i=0;i<10; i++)) {
    op void notify(int a);
    send printer(Sex.Female, i, notify);
    doBathing(Sex.Female, notify);
    //Done bathing.
    send notify(0);
  }

  private op void printer(Sex s, int i, cap void(int) notify) {
    int area;
    receive notify(area);
    System.out.println(i+s.toString()+" bathing in area "+area);
    receive notify(area);
    System.out.println(i+s.toString()+" done");
  }


  public static void main(String[] args) {
    Spa s = new Spa();
    edu.ucdavis.jr.JR.nap(1000);
    send s.swap();
  }
}
```

Question 7. In this question you should implement a *merge sort algorithm* using Linda tuple-space. For the purposes of this question, assume that there is a tuple-space and the standard tuple-space operations described below.

The Linda Tuple-space Primitives Assume you have tuple-space primitives `in` and `out`. These work in the usual way. The specific syntax that you should assume in this question is as given in the following examples:

- If variable `X` has value `42`, then `out(27,"Hi",X)` will place the tuple `{27,"Hi",42}` into the tuple-space.
- The `in` operation blocks until a suitable tuple is found. For example, `in(?Y,99)` will block until a matching tuple is found. Assuming that `Y` is of string type, an example of a tuple which matches this in operation is `{"hello",99}`. If `{"hello",99}` were present in the tuple-space, then the above in operation could remove it, after which the variable `Y` would be bound to the value `"hello"`.

Merge Sort Algorithm This is a standard sorting algorithm, which is highly parallelisable. Its description is given here to avoid any misunderstandings.

The input of the merge sort algorithm is a list of numbers. Conceptually, merge sort works as follows (a top-down description):

- divide the unsorted list into two sublists of about half the size;
- sort each of the two sublists;
- merge the two sorted sublists back into one sorted list.

An alternative bottom-up description of merge sort is as follows:

- split the list into singleton lists containing exactly one number;
- repeatedly merge pairs of lists until you have just one final sorted list.

The Question Merge sort in Linda should work in such a way that there will be many worker processes running in the background waiting for merging tasks in the tuple space. When such a task appears in the tuple space a worker process will take it, merge the two lists in the task, and return the result to the tuple space for further processing. The worker process then waits for the next task. The main sort function should submit the tasks to the tuple space and collect the results, possibly coordinating the different tasks during the run.

This means that you need to implement the *worker process* and the *main sort function*.

To obtain maximum points your solution should provide the maximum level of concurrency while maintaining merge sort complexity $O(n \log n)$. *(10p)*

Solution 7.
```
-module(merge_sort).
-import(ts,[in/2,out/2]).
-export([sort/2,worker/1]).

sort(TS, List) ->
    lists:map(
      fun(X) ->
      Ref = make_ref(),
      out(TS, {list, 0, Ref}),
      out(TS, {Ref, X, tail})
      end,
      List),
    tasks(TS, 0, length(List)).

tasks(_, _, 0) ->
    [];
tasks(TS, Level, 1) ->
    {list, Level, Ref} = in(TS, {list, Level, any}),
    retrieve(TS, Ref, []);
tasks(TS, Level, N) ->
    Next = N div 2,
    Mod = N - Next*2,
    if (Mod>0) ->
    out(TS, {list, Level, head}),
    out(TS, {head, infinity, tail});
       true-> ok
    end,
    insert(TS, Level, Next+Mod),
    collect(TS, Next+Mod),
    tasks(TS, Level+1, Next+Mod).

insert(_, _, 0) ->
    ok;
insert(TS, Level, N) ->
    out(TS, {work, Level}),
    insert(TS, Level, N-1).

collect(_, 0) ->
    ok;
collect(TS, N) ->
    in(TS, {taken, any}),
    collect(TS, N-1).

retrieve(TS, Ref, Acc) ->
    case in(TS, {Ref, any, any}) of
{Ref, N, tail} ->
    lists:reverse([N|Acc]);
{Ref, N, Next} ->
    retrieve(TS, Next, [N|Acc])
    end.

worker(TS) ->
    {work, Level} = in(TS, {work, any}),
    out(TS,{taken, Level}),
    {list, Level, RefN} = in(TS, {list, Level, any}),
    {list, Level, RefM} = in(TS, {list, Level, any}),
    Ref = make_ref(),
    out(TS, {list, Level+1, Ref}),
    order(TS,
  Ref,
  in(TS, {RefN, any, any}),
  in(TS, {RefM, any, any})).

order(TS, Ref, {_, infinity,_}, {_, M, RefM}) ->
    out(TS, {Ref, M, RefM}),
    worker(TS);
order(TS, Ref, {_, N, RefN}, {_, infinity, _}) ->
    out(TS, {Ref, N, RefN}),
    worker(TS);
order(TS, Ref, {_, N, tail}, Tuple = {RefM, M, _}) when N<M ->
    out(TS, {Ref, N, RefM}),
    out(TS, Tuple),
    worker(TS);
order(TS, Ref, {_, N, RefN}, Tuple = {_, M, _}) when N<M ->
```

```
    RefNext = make_ref(),
    out(TS, {Ref, N, RefNext}),
    order(TS, RefNext, in(TS, {RefN, any, any}), Tuple);
order(TS, Ref, Tuple = {RefN, _, _}, {_, M, tail}) ->
    out(TS, {Ref, M, RefN}),
    out(TS, Tuple),
    worker(TS);
order(TS, Ref, Tuple, {_, M, RefM}) ->
    RefNext = make_ref(),
    out(TS, {Ref, M, RefNext}),
    order(TS, RefNext, Tuple, in(TS, {RefM, any, any})).
```