UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF4140 — Models of Concurrency
Day of examination:	15. December 2011
Examination hours:	14:30 – 18:30

This problem set consists of 9 pages.

Appendices:	None
Permitted aids:	All written and printed

Please make sure that your copy of the problem set is complete before you attempt to answer anything.

Some general advises and remarks:

- This problem set consists of three independent parts. It is wise to make good use of your time.
- You can score a total of 100 points on this exam. The number of points stated on each part indicates the weight of that part.
- You can make your own clarifications if you find the examination text ambiguous or imprecise. Such clarifications must be written clearly in the delivered answer.
- Make short and clear explanations!

Good luck!

(Continued on page 2.)

Problem 1 The Ice Cream Shop (weight 55)

We here consider the following description of a synchronisation problem:

> An ice cream shop serves ice cream to customers. One ice cream cone consists of two scoops. First one chocolate flavored scoop is made, and then one with vanilla flavor. The ice cream shop has two employees, one for each flavor. The two employees should be able to work in parallel, but not serving the same customer.

1a Semaphore Solution (weight 20)

Provide an implementation of the synchronization part of the problem by filling in code for the dots in the sketch below (where needed). Use semaphores for synchronization. Remember to declare and initialize the semaphores and other variables.

Remark that making ice cream for a particular customer is initiated by that customer. Thus, a customer should not be given an ice cream unless he or she has asked for it. Make sure that the ChocolateMaker and the VanillaMaker knows which customer it should give the scoops to. Make sure that your solution is free of unnecessary delay. For instance, the chockolate maker should not delay if there are newly arrived customers.

Note: The ice cream cones are represented by an integer array. The values of this array in position i is the number of scoops currently in the cone of customer i.

code/icecreamshop-skeleton

```
...                                   # global variables
int scoops[N] = ([N] 0);

process Customer[i = 0 to N - 1]{
  while(true){
    ...
    scoops[i] = 0;                   #eat
    ...
  }
}

process ChocolateMaker{
  while(true){
    ...
    scoops[...] = scoops[...] + 1  #make scoop
    ...
  }
}
```

(Continued on page 3.)

```
process VanillaMaker{
  while(true){
    ...
    scoops[...] = scoops[...] + 1  #make scoop
    ...
  }
}
```

Solution:

```
sem chocAvail = 1;
sem vanillaAvail = 1;
sem makeVanilla = 0;
sem makeChoc = 0;
sem gotChoc = 0;
sem gotVanilla = 0;

int chockCustomer, vanillaCustomer;
int scoops[N] = ([N] 0);

process Customer[N]{
  while(true){
    P(chocAvail);
    chocCustomer = i;
    V(makeChock);
    P(gotChoc);
    V(chockAvail);
    P(vanillaAvail);
    vanillaCustomer = i;
    V(makeVanilla);
    P(gotVanilla);
    V(vanillaAvail);

    ///eat
    scoops[i] = 0;
  }
}

process VanillaMaker{
  while(true){
    P(makeVanilla);
    scoops[vanillaCustomer] = scoops[vanillaCustomer] + 1
    V(gotVanilla);
  }
}

process ChocolateMaker{
```

(Continued on page 4.)

```
  //as above
}
```

1b Deadlock (weight 5)

Explain briefly, why your solution to Problem 1a is free of deadlocks.

Solution:

```
After taking one semaphore we always release another
semaphore which someone is waiting for. For instance
when we take chockAvail, we
release makeChock which the ChocolateMaker waits for.
```

1c RPC/Rendezvous Solution (weight 25)

Write a module IceCreamShop that exports a getIceCream-operation. Calls to getIceCream() should return when the ice cream is ready. The module should be able to make one chocolate scoop and one vanilla scoop in parallel, but not for the same customer.

You may use remote procedure calls (RPC) or rendezvous or a combination, but not semaphores, locks etc.

As we have no shared memory we can not use an array to count the number of cones anymore. Use pseudo code as:

```
#make chocolate scoop
```

to show where the scoops are made.

Solution:

```
module IceCreamBar
  op getIceCream()


body IcecreamBar

  proc getIceCream()
    call startChock();
    #make the scoop
    send doneChock();
    call startVanilla();
    #make the scoop
    send doneVanilla();


  process chocController
    bool availChock = true;
```

(Continued on page 5.)

```
    in startChoc() and availChock -> availChock = false ;
    [] doneChock -> availChock = true;
    ni


  process vanillaController
    bool availVanilla = true;
    in startVanilla() and availVanilla -> availVanilla = false;
    [] doneVanilla -> availVanilla = true;
    ni
```

1d Fairness (weight 5)

Under what conditions is the following statement true for your solution to Problem 1c?

A call to *getIceCream* will always terminate.

Solution:

```
If there is a finite number of calls to getIceCream or
if the calls are handled in a FIFO manner, they
will always terminate.
```

(Continued on page 6.)

Problem 2 Program Analysis (weight 30)

Given two integer variables x and y, we will in this problem consider the example code S, defined by the following sequence of statement:

```
S: x = 0; y = 10;
   while (x < y) {
     x = x+1; y = y−1;
   }
```

2a Interpretation (weight 5)

Give a short explanation of the (partial correctness) *interpretation* of the following triple:

$$\{\mathsf{true}\}\ \mathsf{S}\ \{\mathsf{x{=}{=}y}\}$$

Solution:
The triple is true if for any state (since the precondition is true) that execution of S starts in, then x==y holds in the resulting state assuming that S terminates.

2b Verification (weight 20)

Use Programming Logic (*PL*) to verify the triple

$$\{\mathsf{true}\}\ \mathsf{S}\ \{\mathsf{x{=}{=}y}\}$$

Hint. You may use the following invariant I when reasoning about the loop:

$$I: \quad \mathsf{x} \leq \mathsf{y} \land \mathit{even}(\mathsf{y}-\mathsf{x})$$

where $even(n)$, for some number n, is true if and only if n is an even number.

Solution:

Loop entry. Need to verify that the invariant holds on entry to the loop:

$$\begin{aligned} \mathit{true} &\Rightarrow (x \leq y \land \mathit{even}(y-x))_{(x,y \,\leftarrow\, 0,10)} \\ &\Rightarrow 0 \leq 10 \land \mathit{even}(10) \end{aligned}$$

which is true.

Loop Exit. The invariant and the negation of the loop test should imply the postcondition:

$$x \leq y \land \mathit{even}(y-x) \land x \geq y \Rightarrow x == y$$

which is true.

(Continued on page 7.)

Loop iteration. Need to show that I is preseved by the loop, i.e., we have to verify:

$$\{I \land x < y\}\ \mathsf{x{=}x{+}1;y{=}y{-}1}\ \{I\}$$

By the assignment axiom and consequence rule, we arrive at the following implication:

$$\begin{aligned} x < y \land \mathit{even}(y-x) \ &\Rightarrow (x \leq y \land \mathit{even}(y-x))_{(x,y\ \leftarrow\ x+1,y-1)} \\ &\Rightarrow x+1 \leq y-1 \land \mathit{even}(y-1-x-1) \end{aligned}$$

which is true since $x < y \land \mathit{even}(y-x)$ implies $y-x \geq 2$ which is the same as $x+1 \leq y-1$. In addition, since $y-x$ is even, we also know that $y-x-2$ must be even.

2c Strengthening the postcondition (weight 5)

By looking at the program S, the programmer realizes that we can say something more specific about the values of x and y when S terminates. In addition to x==y, we also know that both variables have the value 5. Therefore we consider the following triple:

$$\{\mathsf{true}\}\ \mathsf{S}\ \{\mathsf{x{=}{=}y} \land \mathsf{x{=}{=}5}\}$$

Is it possible to verify this triple given the loop invariant I in Problem 2b? If not, suggest an alternative invariant such that the triple can be verified. It is not necessary to give the verification details for the program with the new invariant

Solution:
No, at loop exit the following implication does not hold:

$$I \land x \geq y \Rightarrow x == 5$$

We can strengthen the invariant with the conjunction $x + y == 10$. This equation holds after initialization and is maintaned by the body of the loop. We can then prove the final implication.

(Continued on page 8.)

Problem 3 The Roller Coaster Problem (weight 15)

In this problem we consider a simple version of the Roller Coaster Problem in the language with **send** and **await** statements. The system consists of one `Car` agent and any number of `Passenger` agents (we assume that there is at least 4 passengers).

For the implementation of `Car`, we assume a `Stack` datatype with the usual operations `push`, `top`, and `pop`, and where `size(s)` returns the number of elements on stack `s`. The agents are implemented as follows:

Implementation of the `Passenger` agents:

```
C : Car; // assumed initialized to the Car agent

while true do
  ...
  send C:embark;
  await C:finished;
od
```

Implementation of the `Car` Agent:

```
pass : Stack[Agent] // assumed initialized to empty stack
P : Agent

while true do
  while (size(pass) < 4) do
    await P?embark;
    pass := push(pass,P)
  od
  // ride!
  while (size(pass) > 0) do
    P := top(pass); pass := pop(pass);
    send P:finished
  od
od
```

Thus operation `push(s,e)` pushes the element `e` on stack `s` and returns the resulting stack. Operation `top(s)` returns the top element without modifying `s`, and `pop(s)` returns `s` after removing the top element.

3a Events (weight 5)

Define the events of the system. You may define the events in terms of `Car` agent `C` and an arbitrary `Passenger` agent `P`. Which of these events are local to the `Car` agent `C`?

Solution:

For any passenger P:

$$P{\uparrow}C:\texttt{embark}, P{\downarrow}C:\texttt{embark}, C{\uparrow}P:\texttt{finished}, C{\downarrow}P:\texttt{finished}$$

(Continued on page 9.)

Local to C:

$$P\downarrow C : \texttt{embark}, C\uparrow P : \texttt{finished}$$

3b Local History of `Car` (weight 10)

Define an extended regular expression which describes the *local* history of the `Car` agent `C` after each iteration of the outermost loop. Thus, define an extended regular expression $Cycle_C$ such that h **is** $Cycle_C$ holds at the end of each iteration of this loop, where h is the local history of `C`. It is not necessary to do any verification.

Solution:

$[P1\downarrow C : \texttt{embark}, P2\downarrow C : \texttt{embark}, P3\downarrow C : \texttt{embark}, P4\downarrow C : \texttt{embark}$
$C\uparrow P4 : \texttt{finished}, C\uparrow P3 : \texttt{finished}, C\uparrow P2 : \texttt{finished}, C\uparrow P1 : \texttt{finished}$
some $P1, P2, P3, P4]^*$