

UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Exam in: INF4140 — Models of Concurrency

Day of examination: 15. December 2011

Examination hours: 14:30 – 18:30

This problem set consists of 7 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 .

```

...                               # 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
    ...
  }
}

process VanillaMaker{
  while(true){
    ...

```

(Continued on page 3.)

```
        scoops [...] = scoops [...] + 1 #make scoop
        ...
    }
}

...
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 4.)

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

1b Deadlock (weight 5)

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

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.

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.

(Continued on page 5.)

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:

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

2b Verification (weight 20)

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

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

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

$$I : x \leq y \wedge \text{even}(y - x)$$

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

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:

$$\{\text{true}\} S \{x==y \wedge x==5\}$$

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

(Continued on page 6.)

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**?

(Continued on page 7.)

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.