UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Exam in:	INF4140 — Models of Concurrency
Day of examination:	17. December 2013
Examination hours:	14.30 – 18.30
This problem set consists of 8 pages.	
Appendices:	Inference rules for program verification
Permitted aids:	None

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

Some general advice and remarks:

- This problem set consists of 4 independent problems and one optional extra problem which gives *extra* points.
- The points from problems 1 – 4 sum up to a total of 100 points. The number of points stated on each part indicates the (estimated) difficulty resp. time for solving that part.
- Use your time wisely and take into consideration the weight of each question.
- You should read the *whole* problem set before you start solving the problems.
- 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 and Merry Christmas!

(Continued on page 2.)

Problem 1 Miscellaneous (weight 10)

1a Synchronization primitives (weight 4)

Compare the P-operation on a counting semaphore with the `wait`-operation on a condition variable for monitors. Shortly characterize relevant similarities and differences. Don't make your list of items longer than 3.

1b Concurrent execution and termination (weight 6)

As known from the lecture: a program terminates properly means, all processes of a program reach their respective "end" at which point the execution stops. Now consider the following 2 programs and discuss the given questions for each of them. Start by stating your answer clearly ("yes"/"no") and add a *short* justification, why your answer is the case.

1. Is it possible that the program terminates?
2. Is it guaranteed that it terminates?
3. Does your answer to those 2 points depend on the assumption of weak/strong fairness?

Listing 1: Program 1

```
1  x = 0; y = 0; b = true;
2  co
3      while (b) x = x + 1;
4    ||
5      while (b) y = x + x ;
6    ||
7      await (y==5) < b = false>;
8  oc
```

Listing 2: Program 2

```
1  x = 1; y = 1; b = true;
2  co
3    while (b) <h = x; x =y ; y = h>;
4   ||
5    while (b) x = x + y + 1;
6   ||
7    await (y > x)  < b = false>;
8  oc
```

(Continued on page 3.)

Problem 2 Sushi bar (weight 30)

Assume a small sushi bar with 5 seats. Customers can `enter` and `leave` the bar. The general behavior of one customer process looks as follows:

```
1  process customer
2  begin
3      while true do
4        enter();
5        # eat sushi
6        leave();
7      do
8  end
```

As social persons, the sushi customers behave as follows:

1. in principle, when a seat is free a customer can enter.
2. However, if the sushi-bar is full, the (then 5) customers form a group who want to finish eating together without disturbance by newcomers. So, no new customers are allowed to enter until the sushi-bar is empty again.

2a Semaphore solution (weight 25)

Solve this synchronization problem by

- filling out the bodies of `enter` and `leave` and
- using *semaphores* for your solutions.

Don't forget to introduce and initialize appropriately all variables you need for our solution, including the semaphore(s).

Hint: you can use a semaphore V-operation with extended functionality of the form $V(sem, n)$: It's an abbreviation for doing the V-operation n-times on semaphore *sem*, where n is a number $n \geq 0$.

2b Invariant (weight 5)

State a reasonable invariant for the sushi-bar program, capturing *both* conditions for sushi-bar clients stated above.

(Continued on page 4.)

Problem 3 Crossing baboons (weight 25)

In South Africa's Kruger National Park, there's a canyon spanned by a single rope. The local baboons (= kind of monkeys) can cross the canyon by swinging hand-over-hand on the rope. However, if two baboons go in opposite direction and meet in the middle, they get into a fight and fall to death, so that needs to be avoided. Furthermore, the rope is not too strong and can carry *at most 5* baboons at the same time. Assuming that we can train the baboons to use monitors, you are required to program a synchronization scheme such that

- never more than 5 baboons are on the rope.
- once a baboon has started to cross, he can reach the other side without encountering on the rope another baboon going the opposite direction

```
1  monitor Rope{
2    ...
3    procedure go-north() {...}
4
5    procedure end-north() {...}
6
7    ...
8  }
9
10 process baboon-tonorth[i=1 to N]{ # South going is anologous
11   while (true) {
12     ...
13     # cross the canyon
14     ...
15   }
16 }
```

3a Basic monitor solution (weight 15)

Solve the described problem using a *monitor* (assuming standard signaling & continue scheduling).

3b Invariant (weight 5)

State a reasonable invariant for the program.

3c No-starvation (weight 5)

The problem formulation in the first sub-task does not require to solve the problem that a constant stream of baboons in one direction may prevent baboons in the opposite direction to cross the rope forever. Extend your solution to repair that weakness.

(Continued on page 5.)

Problem 4 Asynchronous communication: Crossing baboons (weight 35)

We here consider asynchronous message passing using the language with **send** and **await** statements and with the non-deterministic choice statement `S1 [] S2` which chooses either `S1` or `S2` for execution. Consider the following *variation* of the crossing baboons problem. The general problem is the same, but in case a baboon wants to cross when currently impossible, he is not blocked, he should try again. In more detail:

> Assume a traffic controller to decide when a baboon can cross the canyon based on the same criteria than before: no opposite crossing is allowed and no more than *five* baboons are allowed to cross. A baboon who wants to cross sends a message to the controller, and depending on the answer, the baboon crosses, or tries again.

So the cycle of a baboon agent `B` wishing to cross the canyon into the direction of north could be sketched as follows:

```
1 var permission: Boolean;
2 while true do
3   send C:request(tonorth);
4   await C:cross(permission);
5   if (permission) then
6     # cross the canyon
7     send C:arrive;
8   fi
9 od
```

where `C` refers to the traffic controller agent and *tonorth* defines the direction a baboon wants to go (true for going to the north side of the canyon, false for the south).

4a Program Analysis (weight 10)

Let $Cycle_B$ denote the regular expression:

$$\left[\ B \uparrow C : request(tonorth), C \downarrow B : cross(permission), [\ B \uparrow C : arrive \mid \varepsilon\ ]\ ,\ \textbf{some}\ permission\ \right]^*$$

Assume that h is the local history of `B`. Use Hoare Logic to verify that h **is** $Cycle_B$ is a loop invariant for the baboon agent implementation given above. (Remember that h **is** R denote that h matches the structure described by the regular expression R.)

(Continued on page 6.)

4b History invariant (weight 5)

Consider the following local history invariant for a baboon agent `B`:

$$h \leq Cycle_B$$

Show, formally and informally, that this invariant holds after each interaction point in the baboon agent implementation. (Remember that $h \leq R$ denote that h is a prefix of the structure described by the regular expression R.)

4c Implementation (weight 15)

Provide an implememtation of the traffic controller agent `C`. The agent should be always waiting for baboons to ask for permission to cross the canyon.

4d Events (weight 2)

What are the local events of the traffic controller `C`?

4e Properties (weight 3)

Is your solution fair to baboons from both directions? If yes, give your justification; otherwise, indicate briefly how to make it fair.

(Continued on page 7.)

Problem 5 Bonus task: Compare-and-Swap (weight 15)

One low-level synchronization primitive available on some HW architectures is called `compare-and-swap` (one some x86-architectures also known as *compare-and-exchange* `CMPXCHG`). Its meaning can be described as follows:

```
1  compare_and_swap(int* reg, int old, int new) {
2    <                        # atomic begin
3      int old_val = *reg;    # dereference local var reg
4                             # and fetch value into old_val
5      if   old_val == old
6      then *reg   = new      # store at address pointed at by reg
7    >;                       # end of atomic
8    return old_val
9  }
```

In words: the first two arguments contain an *address* in main memory and a value, which are being compared; more precisely, the *content* at the *address* in `reg` is compared to the value `old`. If they are the same, the memory cell at the mentioned address is over-written but the operations 3rd argument. Independent of the comparison: the value as read from the address is returned.

Now: *implement* counting semaphores using *compare-and-swap*. Don't use any other synchronization statements except that one. It's not required that your semaphore offers a FIFO-property and you don't need to take care of any fairness considerations.

(Continued on page 8.)

Appendix: Inference Rules

Assignment

$$\{P_{x \leftarrow e}\}\texttt{x = e}\{P\}$$

Composition

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1\texttt{;}\ \ S_2\{R\}}$$

While

$$\frac{\{I \wedge B\}\texttt{S}\{I\}}{\{I\}\texttt{while (B) S;}\{I \wedge \neg B\}}$$

Consequence

$$\frac{(P' \Rightarrow P) \quad \{P\}\texttt{S}\{Q\} \quad (Q \Rightarrow Q')}{\{P'\}\texttt{S}\{Q'\}}$$

If

$$\frac{\{P_1\}\, S_1\, \{Q\} \qquad \{P_2\}\, S_2\, \{Q\}}{\{\textbf{if}\ \ b\ \ \textbf{then}\ \ P_1\ \ \textbf{else}\ \ P_2\}\, \textbf{if}\ \ b\ \ \textbf{then}\ \ S_1\ \ \textbf{else}\ \ S_2\ \ \textbf{fi}\, \{Q\}}$$

where the precondition **if** b **then** P_1 **else** P_2 is an abbreviation for $(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)$

Send

$$\{Q_{h \leftarrow h; A \uparrow B:m}\}\ \textbf{send}\ \texttt{B} : \texttt{m}\ \{Q\}$$

Receive

$$\{\forall w\ .\ Q_{h \leftarrow h; B \downarrow A:m(w)}\}\ \textbf{await}\ B : m(w)\ \{Q\}$$