UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF4140 — Models of Concurrency
Day of examination:	16. December 2010
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 two independent parts. It is wise to make good use of your time.
- You should read the whole problem set before you start solving the problems.
- 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 Shared variables: House building (weight 50)

We here consider the following description of a synchronisation problem:

> In this exercise we will solve the house building problem. A house consists of three elements: floor, walls and roof. First one floor builder will build the floor. When the floor is finished one wall builder may set up the walls. One roof builder will be able to build the roof when the walls are up. Note that building one house does not involve concurrency, but many houses may be built concurrently.

1a Semaphore Solution (weight 20)

Provide an implementation of the synchronization part of the builder processes by extending the sketch below. Fill in code for the dots where needed. Use semaphores for synchronization. Remember to declare and initialize the semaphores.

```
process FloorBuilder[i=1 to N]{
  while (true){
    ...
    ‘‘build floor’’
    ...
  }
}

process WallBuilder[i=1 to N]{
  while (true){
    ...
    ‘‘build walls’’
    ...
  }
}

process RoofBuilder[i=1 to N]{
  while (true){
    ...
    ‘‘build roof’’
    ...
  }
}
```

Solution:

(Continued on page 3.)

```
sem floor = 0, walls = 0;

process FloorBuilder[i=1 to N]{
    while (true){
    ‘‘build floor’’
    V(floor);
  }
}

process WallBuilder[i=1 to N]{
  while (true){
    P(floor);
    ‘‘build walls’’
    V(walls);
  }
}

process RoofBuilder[i=1 to N]{
  while (true){
    P(walls);
    ‘‘build roof’’
  }
}
```

1b Deadlocks (weight 2)

Does your program (from exercise 1a) contain deadlocks? Explain.

Solution: The solutions should be deadlock free. The floor builder is not depending on any other process an may increase the value of `floor`. The wall builder is only dependent on the `floor` semaphore, which may always be increased.

A similar argument may be made for the roof builder.

1c Monitor Solution (weight 25)

Implement a solution to the house building problem again, but now you should use a monitor to synchronize the builders. Implement both the processes and the monitor to synchronize them. Remember that several houses may be built concurrently.

Solution:

```
process FloorBuilder[i=1 to N]{
  ‘‘build floor’’
```

(Continued on page 4.)

```
  Clerc.doneFloor();
}

process WallBuilder[i=1 to N]{
  Clerc.startWalls();
  ‘‘build walls’’
  Clerc.doneWalls();
}

process RoofBuilder[i=1 to N]{
  Clerc.startRoof();
  ‘‘build roof’’
}

monitor Clerk{ %not fair
  int floor = 0, walls = 0;
  cond buildWalls, buildRoof;

  procedure doneFloor(){
    floor = floor + 1;
    signal(buildWalls);
  }

  procedure doneWalls(){
    walls = walls + 1;
    signal(buildRoof);
  }

  procedure startWalls(){
    while(floor < 1){
      wait(buildWalls);
    }
    floor = floor - 1;
  }

  procedure startRoof(){
    while(walls < 1){
      wait(buildroof);
    }
    walls = walls - 1;
  }
}
```

(Continued on page 5.)

1d Fairness (weight 3)

Does your solution from exercise 1c allow sneaking? Can newly arrived processes get access before processes that are waiting on the queues of the monitor. Explain. If your solution allows sneaking then explain how you can prevent it.

Solution: The code is not needed, but an argument about not increasing the counters unless the queues are empty should be presented. The students should also change from a while loop to an if test when checking if a process should wait or not.

```
monitor Clerk{ %fair
  int floor = 0, walls = 0;
  cond buildWalls, buildRoof;

  procedure doneFloor(){
    if(empty(buildWalls)){
      floor = floor + 1; }
    else  {
       signal(buildWalls);}
  }

  procedure doneWalls(){
    if(empty(buildRoof)){
      walls = walls + 1; }
    else  {
       signal(buildRoof);}
  }

  procedure startWalls(){
    if(floor < 1){
      wait(buildWalls); }
    else  {
       floor = floor - 1; }
  }

  procedure startRoof(){
    if(walls < 1){
      wait(buildroof); }
    else  {
       walls = walls - 1; }
  }
}
```

(Continued on page 6.)

Problem 2 Asynchronous Communication: House building (weight 50)

We here consider asynchronous message passing using the language with **send** and **await** statements, and the following variation of the house building problem:

> Here, a house consists of one floor, four walls, and one roof. A `FloorBuilder` agent build floors. After building a floor, it notifies four different `WallBuilder` agents, each building a wall. After four walls have been build, the `RoofBuilder` may start building a roof.

We have a total of six agents in the system, one `FloorBuilder` agent, four `WallBuilder` agents, and one `RoofBuilder` agent. After building a floor, the `FloorBuilder` agent immediately starts to build the floor of the next house.

Consider the following implementation of the `WallBuilder` agents:

```
F : Agent;  // assumed initialized to the FloorBuilder
R : Agent;  // assumed initialized to the RoofBuilder

while true do
  await F:startWall;
  // build wall
  send R:startRoof
od
```

You may assume that no communication occurs during wall construction.

2a Events (weight 5)

Consider a `WallBuilder` agent `W` implemented as above. What are the local events of `W`?

Solution:

$F \downarrow W : startWall$ and $W \uparrow R : startRoof$

2b Program Analysis (weight 20)

Consider the following loop invariant for the wall builder `W`:

$$\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof}))$$

(Continued on page 7.)

where h is the local history of `W`, the projection $h/(\downarrow\texttt{startWall})$ restricts h to receive `startWall` events, the projection $h/(\uparrow\texttt{startRoof})$ restricts h to send `startRoof` events, and $\#(a)$ returns the length of the sequence a.

Explain why this is a suitable loop invariant, and use Hoare Logic to verify this loop invariant for the `WallBuilder` implementation given above.

Solution:

Taking the invariant as the postcondition for the loop body, a precondition for the loop body may be derived by backward construction:

$$
\begin{array}{l}
\{\#(h/(\downarrow\texttt{startWall})) + 1 = \#(h/(\uparrow\texttt{startRoof})) + 1\} \\
\textbf{await } \texttt{R:startWall} \\
\{\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) + 1\} \\
\texttt{// build wall} \\
\{\#((h; W \uparrow R : \texttt{startRoof})/(\downarrow\texttt{startWall})) \\
\qquad = \#((h; W \uparrow R : \texttt{startRoof})/(\uparrow\texttt{startRoof}))\} \\
\textbf{sent } \texttt{R:startRoof} \\
\{\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof}))\}
\end{array}
$$

We are then left with the trivial verification condition:

$$
\begin{array}{l}
\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) \Rightarrow \\
\qquad \#(h/(\downarrow\texttt{startWall})) + 1 = \#(h/(\uparrow\texttt{startRoof})) + 1
\end{array}
$$

2c History Invariant (weight 5)

Consider the following local history invariant for the `WallBuilder` agent `W`:

$$\#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1$$

Show, formally or informally, that this invariant holds after each interaction point in the `WallBuilder` implementation.

Solution:

For each interaction statement, we can do by proving $Q \Rightarrow I_h$, where I_h is the history invariant and Q is postcondition for the interaction statement in the proof outline above.

For **await** `R:startWall`, we have:

$$
\begin{array}{l}
\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) + 1 \Rightarrow \\
\qquad \#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1
\end{array}
$$

For **sent** `R:startRoof`, we have:

$$
\begin{array}{l}
\#(h/(\downarrow\texttt{startWall})) = \#(h/(\uparrow\texttt{startRoof})) \Rightarrow \\
\qquad \#(h/\uparrow\texttt{startRoof}) \leq \#(h/\downarrow\texttt{startWall}) \leq \#(h/\uparrow\texttt{startRoof}) + 1
\end{array}
$$

(Continued on page 8.)

2d Implementation (weight 15)

Provide an implementation of the `RoofBuilder`. This agent should repeatedly try to build roofs, but it must wait for four walls before building. Write the code under the assumption that the agent does not know the names of the different `WallBuilder` agents, and extend the program sketch below. Fill in code for the dots where needed. You may assume that it takes approximately the same amount of time to build each wall, and that message passing is fast. Thus, the `RoofBuilder` may start building roofs as soon as four walls are completed.

```
while true do
  ...
  // build roof
  ...
od
```

Solution:

```
while true do
  n : Int = 0;
  while n < 4 do
    await X?:startRoof;
    n:= n + 1;
  od
  // build roof
od
```

2e Modified Implementation (weight 5)

Explain, *without* programming, the main changes needed in order to solve the problem in 2d if some `WallBuilder` agents are slower than others.

Hint: The `RoofBuilder` can no longer simply wait for the completion of four walls. For instance, three walls may be completed at the first house and one at the second.

Hint: It may also be necessary to modify the `WallBuilder` agents.

Solution:

Building house 1, three walls may be finished, but the 4th is delayed. Three `WallBuilder` agents may then continue to house 2, and some of these walls may be completed before the last wall of house 1. Thus, `RoofBuilder` gets 4 `startRoof` messages, but from different houses.

One way to deal with this is to introduce house identities. `FloorBuilder` may generate a unique identity for each house. This is passed as an argument

(Continued on page 9.)

of `startWall` to the wall builders, and from them to `RoofBuilder`. Thus, before building roof on house i, `RoofBuilder` must wait for 4 `startRoof`(i) messages. We then need some kind of data structure (e.g., a multiset `ms`) in order to remember the completed walls.

Sketch (not required):

```
while true do
await X? startRoof(x);
ms.insert(x);
if ms.get(x) = 4 then // build roof x fi
od
```