UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in: INF4140 — Models of Concurrency

Day of examination: 14. December 2009

Examination hours: 14.30 – 17.30

This problem set consists of 5 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 Chatting (weight 50)

We here consider the following description of a synchronization problem:

> A number of clients are interested in communicating with other clients by finding a chat-partner. It does not matter which other client a client communicates with, but if client A has client B as a partner then client B should have client A as a partner. Identities must be exchanged in order to communicate.
>
> Note that clients know their own id and the task is to find the partner's id.

1a Semaphore Solution (weight 20)

Provide an implementation of the synchronization part of the clients by filling in code for the dots in the sketch below. Use semaphores for synchronization. Remember to declare and initialize the semaphores.

```
...                                    # global variables
process Chat_Client[i = 1 to M]{
  int partner_id;
  ...
  [chatting]
}
```

Solution:

```
int exchange_id;
sem enter = 1;
sem continue = 0;

process Chat_Client[i = 1 to M]{
  int partner_id;
  While (true){
    P(enter);
      if (exchange_id == 0){
        exchange_id = i;
        V(enter)
        p(continue)
        partner_id = exchange_id;
        exchange_id = 0;
        V(enter);
      }
      partner_id = exchange_id
```

(Continued on page 3.)

```
      exchange_id = i;
      V(continue)
  }
}
```

1b Monitor Solution (weight 20)

The synchronization will now be moved from the clients to a chat server that handles requests from the clients. When a client wants to establish a communication link, it invokes a `findPartner` call on a chat server. Whenever two clients have invoked `findPartner`, the chat server provides the clients with their partner's id.

Provide a monitor implementation of the chat server by filling in code for the dots in the sketch below. Specify which signalling discipline you use.

```
monitor Chat_Server{
   ...
   procedure findPartner(int id, int &partner_id){
     ...
   }
}
```

Solution: Signal and Continue:

```
monitor Chat_Server {
  int partner;
  bool busy = false;
  cond not_alone, not_busy;

  procedure findPartner(int id, int &partner_id) {
    while (busy) wait(not_busy);

    if (empty(not_alone)) {
      partner = id;
      wait(not_alone);
      partner_id = partner;
      busy = false;
      signal_all(not_busy);
    }
    else {
      busy = true;
      partner_id = partner;
      partner = id;
      signal(not_alone);
```

(Continued on page 4.)

```
    }
  }
}
```

Signal and Wait:

```
monitor Chat_Server {
  bool waiting = false;
  int wait_id;
  cond not_alone;
  procedure findPartner(int id, int &partner_id) {
    if (!waiting) {
      wait_id = id;
      waiting = true;
      wait (not_alone);
      partner_id = wait_id;
      waiting = false;
    } else {
      partner_id = wait_id;
      wait_id = id;
      signal(not_alone);
    }
  }
}
```

1c Fairness (weight 5)

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

> A call to `findParter` will always terminate.

Solution: This is true if we have an even and finite number of calls. Even because everybody needs a partner. If we have an infinite number of calls, a call may stay in the while loop forever.

1d Symmetry (weight 5)

Does your solution to Problem 1b fulfill the following property?

> If client A has client B as a partner then client B has client A as a partner.

(Continued on page 5.)

Explain briefly!

Solution: Yes, because when the second call has passed the while loop, busy is set to true. Then no other call may interfere. The two calls that are allowed to pass the loop will exchange id.

Problem 2 Asynchronous Communication: Chat Server (weight 50)

Reconsider the chat problem:

> A number of clients are interested in communicating with other clients by finding chat partners. When a client wants to establish a communication link, it sends a *request* message to a chat server. When the chat server has received two request messages, it sends *open* messages to the two clients providing their partner's identity as a parameter.

Below you are asked to make a solution to certain parts of the chatting problem, based on asynchronous message passing using the language with **send** and **await** statements.[1]

2a Implementing the Chat Server (weight 15)

Program the chat server as explained above so that any number of clients can ask for partners. A client wishing to get a partner could be programmed according to the following sketch:

```
send S:request; await S:open(X); ...
```

where "..." indicates communication (i.e., chatting) with X. The *open* message acts as an answer to the *request* message, giving the client a partner (i.e., X).

Your task is to program the chat server S. The program should have the form of a loop (since it should be able to handle any number of requests) with a loop invariant expressing that all requests seen so far have been answered. You will later be asked to write the loop invariant.

Solution:

```
var X,Y: Agent;
loop
 await X? request;
 await Y? request;
 send X.open(Y);
```

[1] The statement **send** $X : m(par)$ sends the message m with parameters par to agent X, and the statement **await** $X : m(y)$ waits (actively) for a message m from agent X and receives the parameters in variable y, and the statement **await** $X?m(y)$ waits (actively) for a message m from any agent X and receives the parameters in variable y and stores the identity of the sending agent in variable X.

(Continued on page 7.)

```
 send Y.open(X)
endloop
```

(Continued on page 8.)

2b Events (weight 3)

What are the events of the chat server S (i.e. α_S)?

Solution:

$A \downarrow S : request$
$S \uparrow A : open(B)$

2c Functions over the history (weight 7)

Define a function *waiting* over the local history of S to calculate the set of clients waiting to get a partner.

Solution:

$$waiting(\varepsilon) = \emptyset$$
$$waiting(h; A \downarrow S : request) = waiting(h) \cup \{A\}$$
$$waiting(h; S \uparrow A : open(B)) = waiting(h) - \{A\}$$

2d Loop Invariant (weight 15)

Formulate a loop invariant of S over its local history h. The invariant should express the informal property explained above (i.e., "all requests seen so far have been answered"). **Hint:** You may use the *waiting* function from the previous question (even if you have not defined it).

Verify that this loop invariant is satisfied by your implementation, by means of Hoare Logic (with the extension to **send** and **await** statements).

Solution: Loop Invariant: $\{waiting(h) = \emptyset.\}$

Verification:

```
var X,Y: Agent;
loop  {waiting(h) = ∅} =>
{∀ X,Y. (waiting(h) ∪ {X} ∪ {Y}) − {X} − {Y} = ∅} =>
{∀ X,Y. (waiting(h; X ↓ S : request) ∪ {Y}) − {X} − {Y} = ∅}
 await X? request; {∀ Y. (waiting(h) ∪ {Y}) − {X} − {Y} = ∅} =>
{∀ Y. waiting(h; Y ↓ S : request) − {X} − {Y} = ∅}
 await Y? request; {waiting(h) − {X} − {Y} = ∅} =>
{waiting(h; S ↑ X : open(Y)) − {Y} = ∅}
 send X.open(Y); {waiting(h) − {Y} = ∅} =>
{waiting(h; S ↑ Y : open(X)) = ∅}
 send Y.open(X) {waiting(h) = ∅}
endloop
```

(Continued on page 9.)

where implication is used to weaken or simplify a state condition.

It must then be proved that the invariant $\{waiting(h) = \emptyset\}$ implies $\{(waiting(h) \cup \{X\} \cup \{Y\}) - \{X\} - \{Y\} = \emptyset\}$, which is trivial.

And the loop invariant holds initially (since then h is empty).

2e History Invariant (weight 5)

Formulate an invariant of S over its local history h, which always holds, and show that this invariant holds after each interaction point (informal argumentation suffices here).

Solution:

`History invariant:` $\{size(waiting(h)) \leq 2\}$

By reconsidering the proof of the loop invariant (and the simplified state conditions), it follows that after each send or await statement the set of waiting clients can at most be 2.

2f Comparison with chatting in Creol (weight 5)

Assume now that the clients may send *open* messages to each other: For instance if a client receives two *open* messages from S, say with A and B as parameters, it may notify A of B (by executing **send** A : $open(B)$) so that also A and B may chat.

A client may therefore want to accept such *open* calls at any time, and at the same time chat with its partners, and possibly ask S for more partners.

Discuss briefly if such a client is easy to program in the **send** /**await** language with [] for non-deterministic choice, and discuss if it is easier to program such a client in the Creol language.

Solution: Creol would be more easy to program due to its notion of release points. In the **send** /**await** language one may need to encode the same non-deterministic choice at several points in the code to get a similar flexibility.

(Continued on page 10.)