UNIVERSITY OF OSLO

Faculty of mathematics and natural sciences

Examination in	INF3140/4140 — Models of Concurrency
Day of examination:	17. December 2007
Examination hours:	14.30 – 17.30

This problem set consists of 14 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 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 Cocktail Bar (weight 50)

Read the following description of a synchronisation problem:

> In a cocktail bar we find two bartenders. Both randomly dispense *vodka* and *orange juice*. Three customers with different tastes want to be served: the first one prefers pure vodka, the second one likes *screwdrivers*, a mixture of vodka and orange juice, and the last one only drinks orange juice.
>
> Drinking and serving occur in turns. A drink comprises two units. For example, an orange juice consists of two units of orange juice. Each bartender dispenses one unit at a time. Depending on what they serve, one of the customers will drink. Once the drink has been consumed, the bartenders serve the next drink, and so on. Only drinks consisting of two units are drank, and a glass will only be refilled once it is completely empty.

Model this problem with five processes. Each customer and each bartender shall be modelled by its own process. The drink shall be modelled by two *shared variables* d_1 and d_2, which each represents one unit of the drink. The values of these variables range over *empty*, *vodka*, and *orange*. Initially, the glass is empty, corresponding to the valuation $d_1 = \mathit{empty}$ and $d_2 = \mathit{empty}$. The state $d_1 = \mathit{vodka}$ and $d_2 = \mathit{orange}$ corresponds the state after each bartender has served one unit. In this case they have produced a screwdriver.

All processes may read any shared variable freely, and customers are also allowed to write to each of the variable d_1 and d_2. The first bartender, `Bartender1`, is not allowed to write to d_2 and the second bartender, `Bartender2`, is not allowed to write to d_1.

Drinking a glass can be modelled by setting both d_1 and d_2 to empty. The screwdriver drinker may be modelled as follows:

```
process ScrewdriverDrinker() {
  while (true) {
    ... // coordinate with other processes
    d1 = empty;
    d2 = empty;
    ... // coordinate with other processes
  }
}
```

The bartenders may use the non-deterministic choice operator `[]` to select which ingredient to pour into the glass.

(Continued on page 3.)

1a Implementation using semaphores (weight 15)

Implement the customer and bartender processes. Use *semaphores* for coordinating access to the shared variables. Make sure, that the implementation is correct and that it does not contain deadlocks.

Solution: A possible solution uses the design pattern of "passing the baton" of Andrews.[1] It uses one semaphore for each process, called `c1` for the semaphore of the vodka drinker, `c2` for the semaphore of the orange juice drinker, `c3` for the semaphore of the screwdriver drinker, `b1` for the semaphore of the bartender, and `b2` for the semaphore of the second bartender.

Initially, all semaphores are closed, i.e., 0, except for the semaphore of the first bartender. This means that all processes will block at their `P(...)` operations, immediately before they will drink, with the exception of the first bartender.

The first bartender will pass the baton to the second bartender after her has filled is part of the glass. The second bartender will fill its half and look at the content of the glass. Depending on that content, the baton is passed to one of the drinkers. After the drinker with the baton has emptied his glass, he passes the baton back to the first bartender.

```
sem b1 = 1; // semaphore for the first bartender
sem b2 = 0; // semaphore for the second bartender
sem c1 = 0; // Have a binary semaphore for each customer
sem c2 = 0; // initialised to 0, i.e., they will block on that
sem c3 = 0; // semaphore.
enum { empty, vodka, orange } d1 = empty, d2 = empty // the glass.

process Bartender1() {
  while (true) {
    P(b1);
    d1 = vodka [] d1 = orange;
    V(b2);
  }
}

process Bartender2() {
  while (true) {
    P(b2);
    d2 = vodka [] d2 = orange;
    if (d1 == vodka && d2 == vodka) V(c1);
    else if (d1 == orange && d2 == orange) V(c2);
    else V(c3);
```

[1] See Sect. 4.4.3, p. 171 of Gregory R. Andrews, *Foundations of Mutlithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

(Continued on page 4.)

```
  }
}

process VodkaDrinker() {
  while (true) {
    P(c1);
    d1 = empty; d2 = empty;
    V(b1);
  }
}

process OrangeJuiceDrinker() {
  while (true) {
    P(c2);
    d1 = empty; d2 = empty;
    V(b1);
  }
}

process ScrewdriverDrinker() {
  while (true) {
    P(c3);
    d1 = empty; d2 = empty;
    V(b1);
  }
}
```

1b Correctness (weight 5)

Explain why your implementation is correct, i.e., why a glass is only refilled after it has been completely empty, and why each customer will only drink what he likes to drink.

Solution: The drinker will just wait for his baton. The second bartender will decide which drinker will receive the baton. Since the reasoning is symmetric, we will only consider the vodka drinker. He is waiting at his semaphore `P(c1)`. This semaphore is signalled by the second bartender in the code fragment

```
if (d1 == vodka && d2 == vodka) V(c1);
else if (d1 == orange && d2 == orange) V(c2);
else V(c3);
```

The `V(c1)` operation is only executed, if `d1 == vodka` and `d2 == vodka`, i.e., if the drink is pure vodka. Since there are no other situations in which that semaphore is signalled, we conclude, that `c1 == 1` implies `d1 == vodka`

(Continued on page 5.)

`&& d2 == vodka`, `d1 == empty && d2 == vodka`, or `d1 == empty && d2 == empty`, where the letter to conditions hold, while the vodka drinker is actually drinking. After passing the baton on, the vodka drinker is blocked again at `P(c1)`. Also observe, that while executing `V(b1)` in any customer, the glass is empty, i.e., `d1 == empty && d2 == empty` and therefore, the glass is only refilled *after* it has been emptied.

For a more formal proof, the condition for signalling the `b2` semaphore is `d1 != empty && d2 == empty`, which never holds for any of the customers.

1c Safety (weight 5)

Argue that your implementation is *safe*, i.e., it is never the case that two processes write to the same variable at the same time.

Solution: Following from the *passing the baton* pattern, each process, e.g., `Bartender1`, have to wait until it can pass its semaphore. This happens as soon as the corresponding `V(...)` operation is executed. Therefore, at most one process is executing its critical section.

1d Deadlock freedom (weight 5)

Explain, why your implementation is free of deadlocks.

Solution: Deadlock is avoided, since at any given moment, one statement is enabled for execution. Assuming progress (if a statement is enabled, it will eventually be taken), the system will never stop executing.

Observe, that the system is also free of starvation. Whenever a glass has been filled, it will be emptied.

For each process, however, we cannot guarantee, that it will drink infinitely often. This only holds if pouring is *fair*, which we did not assume.

Some solution had another source of starvation when competing for access to the glass. The code of such a solution is similar to:

```
sem b1 = 1; // semaphore for the first bartender
sem b2 = 0; // semaphore for the second bartender
sem g = 0;  // Have a binary semaphore for the glass.
enum { empty, vodka, orange } d1 = empty, d2 = empty // the glass.

process Bartender1() {
  while (true) {
    P(b1);
    d1 = vodka [] d1 = orange;
    V(b2);
```

(Continued on page 6.)

```
  }
}

process Bartender2() {
  while (true) {
    P(b2);
    d2 = vodka [] d2 = orange;
    V(g);
  }
}

process VodkaDrinker() {
  while (true) {
    while (true) {
      P(g);
      if (d1 != vodka && d2 != vodka)
        break;
      else
        V(g)
    }
    d1 = empty; d2 = empty;
    V(b1);
  }
}

process OrangeJuiceDrinker() {
  while (true) {
    while (true) {
      P(g);
      if (d1 != orange && d2 != orange)
        break;
      else
        V(g)
    }
    d1 = empty; d2 = empty;
    V(b1);
  }
}

process ScrewdriverDrinker() {
  while (true) {
    while (true) {
      P(g);
      if ((d1 != orange && d2 != vodka) || (d1 != vodka && d2 != orange))
        break;
      else
```

(Continued on page 7.)

```
        V(g)
    }
    d1 = empty; d2 = empty;
    V(b1);
  }
}
```

This solution is correct, safe, free of deadlocks, but allows processes to starve: If the second bartender signals the `g` semaphore, one process, which is not allowed to drink, may pass the "baton" to itself infinitely often. Such a solution should be accepted, too.

1e Implementation using monitors (weight 15)

Implement the customer and bartender processes, but you must use *monitors* for synchronising access to the shared variables. Make sure, that the implementation is correct and that it does not contain deadlocks.

Solution: In this solution we assume the *signal and continue* signalling discipline, i.e., executing *signal* will not release the processor.[2] An explicit signal is necessary.

```
monitor Glass {
  cond fill;  // Condition to indicate need for a refill.
  cond drink; // Condition for the second bartender
  enum { empty, vodka, orange } d1 = empty, d2 = empty // the glass.

  procedure fill1 {
    while (d1 != empty) wait(fill);
    d1 = orange [] d1 = vodka;
    if (d2 != empty)                // Now the glass is full, signal
      signal(drink);                // customers.
    else                            // Glass is not full, signal the
      signal(fill);                 // other bartender.
  }

  procedure fill2 {
    while (d2 != empty) wait(fill);
    d2 = orange [] d2 = vodka;
    if (d1 != empty)
      signal(drink);
    else
      signal(fill);
  }
```

[2] Id. at Sect. 5.1.3, pp. 208–211.

(Continued on page 8.)

```
  procedure drink_vodka {
      while (d1 != vodka && d2 != vodka) {
        signal(drink);  // Signal to another customer, it may be
        wait(drink);    // something else.
      }
      d1 = empty; d2 = empty;
      signal(fill);
    }
  }

  procedure drink_orange {
      while (d1 != orange && d2 != orange) {
        signal(drink);
        wait(drink);
      }
      d1 = empty; d2 = empty;
      signal(fill);
    }
  }

  procedure drink_screwdriver {
      while ((d1 != orange && d2 != vodka) ||
             (d1 != vodka && d2 != orange)) {
        signal(drink);
        wait(drink);
      }
      d1 = empty; d2 = empty;
      signal(fill);
    }
  }
}

process Bartender1() {
  while (true)
    call Glass.fill1;
}

process Bartender2() {
  while (true)
    call Glass.fill2;
}

process VodkaDrinker() {
  while (true)
    call Glass.drink_vodka;
```

(Continued on page 9.)

```
}

process OrangeJuiceDrinker() {
  while (true)
    call Glass.drink_orange;
}

process ScrewdriverDrinker() {
  while (true)
    call Glass.drink_screwdriver;
}
```

1f Differences between semaphore and monitors (weight 5)

Explain briefly the differences between your solution using semaphores and the solution using monitors. Focus on the different methods for guaranteeing absence of deadlocks.

Solution: The main differences between the semaphore solution and the monitor solution are:

1. The actual behaviour of the processes is implemented in the monitor itself. The processes just call their drinking or pouring procedures.
2. The coordination of processes is accomplished by condition variables. Instead of increasing and decreasing semaphores, we wait and signal condition variables. Two condition variables are sufficient, one to coordinate the customers and one to coordinate the bartenders.
3. Observe that the Bartender has to check whether the `fill` queue is empty and signal it otherwise before it waits on it to avoid deadlock. Similarly, a customer must signal the `drink` condition before he waits on it, because the glass may be full but it may not be his drink and the customer for that drink may be already waiting on that condition.

Nobody thought of using the code from Figure 5.2 (Id. at p. 209) for using monitors to implement semaphores. Such a solution is equally acceptable.

(Continued on page 10.)

Problem 2 Finding partners in an asynchronous network (weight 50)

Consider a network with agents that communicate asynchronously by exchanging messages. Overtaking of messages may happen and there is unbounded buffering of messages. We assume here that it is desirable that (some of the) agents cooperate in pairs: In order to solve a certain task, an agent needs to have exactly one partner. Agents not able to find a partner will not contribute to the task.

The following program sketch, to be run on each agent, is a naive attempt to establish a partner. The main idea here is to invite other agents by `join` messages. An agent may accept such an invitation and respond with a `yes` message, thereby committing himself to partnership with the sender. A `yes` message may be answered by an uncommitted agent with an `ok` message. An agent's attributes include the following variables (not visible to other agents):

> *`agentlist`* `--` a list of agent identities, given initially
> *`partner`* `--` the identity of the partner, initially null
> *`X`* `--` a variable ranging over agent identities, initially null

The program sketch is as follows:

```
<send join messages to all agents in agentlist>;

( (await X?join; send X:yes; await X:ok; partner := X)
[]
  (await X?yes; send X:ok; partner := X) )
```

where the first statement is an abbreviation for a while-loop sending a join-message to each agent in the agentlist (which we may assume has an appropriate initial setting). Assignment is here denoted by :=.

Consider a system of two or more agents. We would like to establish the following properties:

- (**duality:**) *Whenever an agent x is terminating with y as partner, y is terminating with x as partner.*
- (**success:**) *At least one pair will be formed.*
- (**purity:**) *No agent is a partner with itself.*

2a Properties (weight 5)

Consider the three mentioned properties.

(Continued on page 11.)

1. Which ones can be be expressed as local properties to an agent, and which ones as global properties?
2. Which ones are satisfied by the given program sketch?

You need not explain your claims here. Solution:

1. not duality since there is a promise that the other agent will terminate. not success. purity is safety.
2. duality (the two agents may not deadlock), but not success (due to possible deadlock), purity (even if self is member of agentlist)

2b Deadlock (weight 5)

Explain briefly why deadlock is possible!

Solution: Consider agents working at the same speed, and all starting with the first branch. No ok message will be produced and they will all wait forever!

2c Improved Program (weight 15)

Improve the program sketch above so that deadlock is not possible. You may introduce new kinds of messages if needed.

Solution:

```
<send join messages to all agents in agentlist>;
al:= agentlist;

while partner=null and al /= emptylist do
  (await X?yes; send X:ok; partner:= X)
  []
  (await X?join; send X:yes;
   waiting:= true; while waiting do
       (await X:ok; partner:= X; waiting:= false)
    [] (await Y?no; al:= al-Y; if Y=X then waiting:= false fi)
    [] (await Y?yes; send Y:no; al:= al-Y)
endwhile) endwhile;

<send no messages to all agents in agentlist>;
```

where the message `no` is used to inform another agent that it currently cannot be accepted as a partner. And `waiting`, `al` and `Y` are additional variables, letting `al` be the remainder of agentlist from which the current agent has not

(Continued on page 12.)

yet received a `yes` or `no`. The `no` is used to control termination of the inner loop, and `al` to terminate the outer loop.

Discuss briefly why your solution is deadlock free and under what conditions.

Solution: Consider a system of two or more agents, running the program above. The program above is deadlock free under the condition of no message loss (but overtaking is OK).

2d Termination and Success (weight 5)

Explain briefly if your solution above is terminating, and under what conditions. If not, indicate a way to make it terminating (but you need not reprogram).

Solution: Consider a system of two or more agents, running the program above. The program above is terminating under the following conditions: no message loss, the agent itself is not in agentlist. (But overtaking is OK). Agentlist is large enough, say all agents in the network.

Consider a system of two or more agents, running your program. Will there be at least one pair (as stated in the success property)?

Solution: Yes, if each agentlist is large enough, say all other agents: Consider the last join message received by the system, say by an agent x from y. x has not terminated, and will send a yes message. y has not terminated, since x has not yet sent a no. y will receive yes and send ok, and both will terminate.

2e Local Postcondition (weight 5)

Consider the given program sketch above for a given agent A. Find a postcondition of the program. The postcondition may refer to A's local attributes and its local history, and should imply that A has been involved in exactly one ok-message with its `partner` as the other party.

Solution: Local postcondition of an agent A:

$$
\begin{aligned}
&X = partner \wedge \\
&(h = A \uparrow agentlist : join, X \downarrow A : join, A \uparrow X : yes, X \downarrow A : ok \\
&\vee\ h = A \uparrow agentlist : join, X \downarrow A : yes, A \uparrow X : ok)
\end{aligned}
$$

where $A \uparrow agentlist : join$ denotes the list of send events to all agents in agentlist.

(Continued on page 13.)

2f Verification (weight 10)

Prove that the postcondition is correct using Hoare Logic.

Solution:

```
<send join messages to all agents in agentlist>;

( ∀ X. P(X, X, h; X ↓ A : join; A ↑ X : yes; X ↓ A : ok)
  (await X?join; send X:yes; await X:ok; partner := X P(X, partner, h))
[]
  (∀ X. P(X, X, h; X ↓ A : yes; A ↑ X : ok)
   await X?yes; send X:ok; partner := X) P(X, partner, h))
```

where $P(X, partner, h,)$ is the postcondition above. The precondition of the [] construct can be simplified to $h = A \uparrow agentlist : join$ which is satisfied after the first statement.

2g Global postcondition (weight 5)

Consider a system of 2 agents, A and B, and assume that the system terminates. Determine a global postcondition by composition of the local postconditions of A and B.

Solution:

$legal(H) \wedge \exists X, partner.post_A(X, partner, H/A) \wedge \exists X, partner.post_B(X, partner, H/B)$

which can be simplified to

$legal(H) \wedge post_A(B, B, H/A) \wedge post_B(A, A, H/B)$

and then to

$legal(H) \wedge (h/A = A \uparrow agentlist : join, B \downarrow A : join, A \uparrow B : yes, B \downarrow A : ok$
$\vee\ h/A = A \uparrow agentlist : join, B \downarrow A : yes, A \uparrow B : ok)$
$\wedge (h/B = B \uparrow agentlist : join, A \downarrow B : join, B \uparrow A : yes, A \downarrow B : ok$
$\vee\ h/B = B \uparrow agentlist : join, A \downarrow B : yes, B \uparrow A : ok)$

and further

$legal(H) \wedge (h = (A \uparrow agentlist : join, B \downarrow A : join, A \uparrow B : yes, B \downarrow A : ok)$
$|||(B \uparrow agentlist : join, A \downarrow B : yes, B \uparrow A : ok)$
$\vee\ h = (A \uparrow agentlist : join, B \downarrow A : yes, A \uparrow B : ok)$
$|||(B \uparrow agentlist : join, A \downarrow B : join, B \uparrow A : yes, A \downarrow B : ok))$

which says that the history must end with the sequence

$B \downarrow A : join, A \uparrow B : yes, A \downarrow B : yes, B \uparrow A : okB \downarrow A : ok$

(Continued on page 14.)

or the same sequence where A and B are swapped. And the two "send join" events must happen before this sequence.