UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	IN5170 — Models of Concurrency
Day of examination:	9. December 2019
Examination hours:	14:30 – 18:30 (4 hours)
This problem set consists of 11 pages.	
Appendices:	None
Permitted aids:	No written or printed material

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

Some general advice and remarks:

- Before you start to solve the problems, take a look at the whole problem set to schedule your time.
- The number of points stated on each part indicates the weight of that part.
- You can make your own clarifications if needed. Please write down any such clarifications.
- Make brief and clear explanations!
- Make your program implementations as simple as possible!
- Use underlined keywords **if**, **then**, **else**, and **fi** for if-statements, so that it is easy to see if there is an else-branch and where an if-statement ends.

Good luck!

(Continued on page 2.)

Problem 1 Semaphores: Philosophers (weight 20)

Consider the problem of the dining philosophers, but where the chopsticks are shared: There are five chopsticks placed in the middle of the table, to be shared by the five philosophers, so that *any philosopher can take any remaining chopstick*. For instance, two philosophers sitting next to each other should be able to eat at the same time if no one else has a chopstick.

1a Program the Dining Philosophers (weight 10)

You are given the following outline:

```
process Phil[i=0 to 4]{
  while (true) {
   think;
   acquire one chopstick;
   acquire another chopstick;
   eat;
   release chopsticks;
}}
```

1. Program the Phil process definition as explained and outlined. Show how you choose to model the chopsticks and declare shared variables.

   Solution:

   ```
   sem sticks :=5; // initially 5 chopsticks

   process Phil[i=0 to 4]{
     while (true) {
      think;
      P(sticks);
      P(sticks);
      eat ;
      V(sticks);
      V(sticks);
   }}
   ```

2. Is deadlock possible. Explain briefly!

   Solution:
   Yes. For instance, all Phil.s may take one stick each, and then there is a deadlock.

1b A Fair and Deadlock-Free Solution (weight 10)

Make a new solution to the problem above (in question 1a) that is deadlock-free and fair to all philosophers. You may add semaphore variables.

(Continued on page 3.)

Explain briefly why the program is fair and deadlock-free.

Solution:

```
sem sticks :=5; // initially 5 chopsticks
sem napkins := 2; // at most two can eat at the same time
// could also be 3 since 2 will succeed anyway

process Phil[i=0 to 4]{
  while (true) {
   think;
   P(napkins);
   P(sticks);
   P(sticks);
   eat ;
   V(sticks);
   V(sticks);
   V(napkins);
}}
```

You may also use an extra sem variable (mutex) instead of napkins, and do a P(mutex) instead of P(napkins); and then do a V(mutex)-operation on this semaphore after the two P(sticks)-operations.
This will give somewhat more rigid synchronization since a Phil can only do P(sticks) inside a critical region. The solution above is without (exclusive) critical regions.
Another solution, which is more elegant, is to use two semaphores, say stick1 and stick2, which are initialized to 3 and 2.

```
sem stick1 := 3; // used for first stick
sem stick2 := 2; // used for second stick

process Phil[i=0 to 4]{
  while (true) {
   think;
   P(stick1);
   P(stick2);
   eat ;
   V(stick2);
   V(stick1);
   V(napkins);
}}
```

This will not give deadlock since only 3 can get the first stick. Then two can still eat at the same time.

(Continued on page 4.)

Problem 2 Shared Variable Concurrency (weight 50)

Consider a simple card game with two teams, each consisting of two persons. The game starts with an arbitrary card on the table. One of the two players on the starting team tries to beat it by putting a card on the table (for instance a higher card of the same suit). If he beats it, his team earns a point. Then the other player on the same team puts a card on the table, and one of the players on the other team tries to beat that card, and then the other player on that team puts a new card on the table, and so on. Note that the two players in one team may play in any order (without looking at each others cards). We will not care about how the cards are played and which team wins, but we will care about the synchronization:

> Both players on one team have played before the players on the other team play.

We consider the setting of shared variables with critical regions and define the four players by the four processes

```
process P[i=0 to 3]{ while (more cards) {...} }
```

You do not need to implement the *more cards* test. One team is given by P[0] and P[2] and the other team by P[1] and P[3]. The two teams are therefore called *the odd team* and *the even team*. We use the following shared variables to keep track of the game:

```
bool played[i=0 to 3] // to record who has played in this round
bool evennext // to control which team is next
```

The even team may only start when *evennext* is true, and the odd team when it is false. We assume that boolean variables are initially set to false. Note that the partner of player i is given by $(i + 2)$ **mod** 4 where **mod** is the modulo operation.

2a Solution with Ignorant Players (weight 10)

Program the player process $P[i]$ without critical regions, or with as small critical regions as possible. The player need not be aware of whether he plays before his partner or not. Use the command *play* to indicate that a card is being played.

Explain why any critical regions are needed, and why they cannot be made smaller.

Solution:

```
process P[i=0 to 3]{
while  (more cards) {
```

(Continued on page 5.)

```
if ((evennext=even(i)) and played[i]=false) {
  play; // play a card
  < if (played[(i+2) mod 4]) { // my partner has played
       // reset booleans and flip evennext
       evennext:=not evennext; // now the other team will be next
       played[(i+2) mod 4]:=false; }
    else {played[i]:=true; }
  >
}}}
```

Both players may reach the first then-branch. But as long as at least one of the teammates will do that, the then-branch is fine. Not both should reach the second if-statement at the same time, then they both could do the else-part and there could be be a deadlock. In this solution at most one of the two players can reach the second then-branch. Thus the CR is needed in this solution and cannot be made smaller.
In a solution with $played[i] := true$ before the second if-statement, both players could reach the second then-branch. Then when one of these then-branches is executed, the other team may play, and the second player on the team could do his then-branch later (if not properly protected in a CR), which could mess up a later round. Then a CR would be even larger than above.

2b Solution with Aware Players (weight 10)

Program the player process $P[i]$, such that when the player is playing a card, he is aware of whether he is the first person on the team to play in this round or not. You do not need to program which card to play nor define any strategy. Use the command *playfirst* to indicate that the first card of the team is being played in this round, and the command *playsecond* to indicate that the second card of the team is being played in this round. [1]

Make the critical regions as small as possible. Explain why any critical regions are needed, and why they cannot be made smaller.

Aware Solution:

```
process P[i=0 to 3]{
while  (more cards) {
  if ((evennext=even(i)) and played[i]=false) {
   <if (played[(i+2) mod 4]) { // the teammate has played
      playsecond;
      evennext:=not evennext;
      played[(i+2) mod 4]:=false;}
    else {playfirst; played[i]:=true; } > // play a winning card if possible
   }}}
```

[1] The strategy for *playfirst* could be to *play a winning card if possible*, and for *playsecond* to *play a good starting card* for the next round. (Do not go into further detail about this.)

(Continued on page 6.)

The critical section cannot be made smaller since then the awareness could be lost, and also we must be sure than both players play before the other team takes over, as above.

2c Aware Player Solution: Correctness (weight 10)

Consider the second version of the program (as described in question 2b). It is required that when a new round starts (for either team) the boolean *played* variables should all be false ($played[i] = false$ for $i = 0$ to 3).

1. Show how to add pre- and postconditions, and/or invariants, in the program to ensure this requirement. Explain which Hoare triples that need to be verified.
2. In what way could we be able to prove this: by sequential reasoning, by shared variable reasoning with interference, by a global invariant, or by a combination of local and/or global invariants. Justify your answer briefly. (You do not need to do a proof.)
3. Explain briefly how a proof of the Hoare triples could be carried out and why it would work in this case.

Solution:
To prove that all played values are false when a new round starts, we prove that all played values are false when *evennext* is negated, i.e., as a postcondition Q of the then-branch of the critical region:

$$evennext \neq even(i) \wedge \forall i : [0..3] \; . \; played[i] = false$$

The precondition P of the critical region is that of the first if-test and that of the second if-test since this is the condition that makes the execution go through the branch that flips *evennext*:

$$evennext = even(i) \wedge played[i] = false \wedge (played[(i + 2) \; mod \; 4])$$

Thus the postcondition refers to variables changed by other processes. This requires verification taking interference into account. We must show that the local proof of any statement s' in another process does not violates Q. The interference proof looks at a triple $\{P'\}s'\{Q'\}$ part of some other process and we must prove

$$\{P \wedge P'\}s'\{Q'\}$$

There are four processes in this program, each with one critical region. Thus there are 3 proofs. Each of this will have a precondition stating that the (outer) if-test is true. i.e., $evennext = even(i') \wedge played[i'] = false$ where i' is the process. For either of the two process i' of the other team we have that $evennext \neq even(i')$ where $even(i') \neq even(i)$. Thus the precondition

(Continued on page 7.)

$\{P \wedge P'\}$ is false. For the teammate $(i+2) mod\ 4$ we also have that $\{P \wedge P'\}$ is false since $played[(i+2) mod\ 4]$ is part of P and $played[(i+2) mod\ 4] = false$ is part of P' (since i here is $(i+2) mod\ 4$. [2]

2d Monitor Solution of Aware Players (weight 10)

Program the second version of the program (as described in question 2b) using a monitor "Game" to synchronize the players. The monitor has one procedure *play* that should be called by the players in order to play. A player could then look like

```
process P[i=0 to 3]{ while (<more cards>) { call Game.play(i); }}
```

What is the advantage (if any) of this solution over the one above (in 2b)?

Monitor Solution:

```
monitor Game {

bool evennext // to control which team is next
bool played[i=0 to 3]; // to record who has played in this round
cond turn;


procedure play(Nat i){
  while ((evennext=even(i)) and played[i]=false) {wait(turn);}
  if (played[(i+2) mod 4]=false){
    playfirst;
    played[i]:=true;}
  else {
    playsecond;
    played[i]:=false;
    played[(i+2) mod 4]:=false;
    signal_all(turn) }}
}
```

2e Semaphore Solution of Aware Players (weight 10)

1. Program the second version of the program (as described in question 2b) but using semaphores instead of critical regions. Declare all semaphores and other shared variables needed as well as their initial values.

   Semaphore Solution:

[2] The interference proof looks at the preconditions of two processes and shows that the postcondition of one process is maintained by a statement in another process, when both the preconditions of the statement and that of the process holds.

(Continued on page 8.)

```
sem odd  := 2;
sem even := 0;
sem mutex := 1:
bool played[i=0 to 3];

process P[i=0 to 3]{
while  (<more cards>) {
 if (played[i]=false) {
 if even(i) {P(even);} else {P(odd);}
  P(mutex);
  if (not played[(i+2) mod 4]) {playfirst; played[i]:= true;}
  else {  playsecond;
   if even(i) {V(odd);V(odd);}else{V(even);V(even);}
  \\ played[i]:=false; -- redundant
  played[(i+2) mod 4]:=false;}
  V(mutex); }
}}
```

Note: if (played[i]=false) is needed even with P(even), since otherwise the same player may come in again before his teammate.

2. Compare the three solutions for the aware player problem. Explain briefly the advantages of the different solutions.

   Answer: The first solution uses a busy wait loop. The monitor solution is easier to program/understand/verify than the other solution. In general a monitor solution may have less efficiency due to the built-in synchronization, but here there is not so much to gain in efficiency over the other solutions. The semaphore solution does not need the *evennext* variable, due to the odd/even semaphores. But it uses a mutex to control the synchronization, so there is not so much to gain, and the programming/understanding is more tricky than the monitor solution.

Problem 3 Asynchronous Agents (weight 30)

We consider here the setting of asynchronous agents with **send** and **await** (receive) statements. In particular, we will look at an auction system for managing the sale of some item or property. A new auction may start when any previous auctions are finished. An auction is *open* when it has started and not yet ended. Only the agent that started an auction may close the same auction. The *bidders* are represented by a number of concurrent agents that make bids when an auction is open; and the *winner* of the auction is the bidder with the highest bid at the time when the auction is closed. In addition, bidders may ask the auction agent about the current status of an ongoing auction with *check* messages, and the auction agent should respond with $status(b, n)$ messages, where n is the current highest bid made by bidder

(Continued on page 9.)

b. When an auction is closed the auction agent should send $winner(b, n)$ messages to the owner and the winner b where n is the winning bid value.

3a Programming the Auction Agent (weight 10)

Program an agent called *Auction* as explained above. The auction agent should react to *open*, *close*, *check*, and *newbid(n)* messages where n is the amount of the new bid. The agent should produce *winner(b,n)* and *status(b,n)* messages when appropriate as explained above. The auction agent may have this outline

```
agent Auction {
  nat   highBid; // to hold the current high bid
  agent hBidder; // to hold the current high bidder
  agent owner ; // to hold the auction owner
  ...
  loop ... end } // main program
```

We assume 0 as default value for natural numbers and *null* for agent variables. We also assume that messages arriving at an agent are ignored if there is no matching **await** statement. Thus, they do not appear in the local history.

Solution:

```
agent Auction {
  nat   highBid;
  agent hBidder;
  agent owner ;

  bool open; // is the auction open?
  agent X;  // used in receive messages
  nat n;    // used in newbid messages

  loop
     await owner?open(); // owner is assigned here
     open:= true;
     while (open) {
        await X?newbid(n);
        if (n>highBid){highBid:=n; hBidder:=X;} fi;
                     send X!status(hBidder,highBid);
     [] await X?check(); send X!status(hBidder,highBid);
     [] await X?close();
        if (X=owner){send owner!winner(hBidder,highBid);
                     send hBidder!winner(hBidder,highBid);
                     open:=false;
                     owner:= null; highBid:= 0; hBidder := null;} fi
     }
   end }
```

(Continued on page 10.)

One may also use **when** *open* to protect newbid, check and close so that they only are accepted when an auction is open. One could also write await owner?close() in the last selection pattern and skip the if-test. In this case, only relevant close events would appear in the history, something which would simplify 3b and 3c a bit.

3b Events and History Functions (weight 10)

1. What are the events of the auction agent?

   Solution:

   $$(X \downarrow Auction : open()), (X \downarrow Auction : close()),$$

   $$(X \downarrow Auction : newbid(n)), (X \downarrow Auction : check())$$

   $$(Auction \uparrow X : status(b, n))(Auction \uparrow X : winner(b, n))$$

2. Define a function *curowner* such that $curowner(h)$ calculates the owner of the current auction, if there is an ongoing auction, otherwise null. (An auction is ongoing if is opened but not closed.) And define a function *highest* such that $highest(h)$ calculates the highest bid from the history for the current auction. if there is an ongoing one, otherwise 0.

   Solution:

   curowner(**empty**) = **null**
   curowner(h;$(X \downarrow Auction : open())$) = X
   curowner(h;$(X \downarrow Auction : close())$) = **if** X=curowner(h) **then null**
   else curowner(h)
   curowner(h;**others**) = curowner(h)

   highest(**empty**) = 0
   highest(h;$(X \downarrow Auction : newbid(n))$) = **if** n>highest(h) **then** n **else** highest(h)
   highest(h;$(X \downarrow Auction : close())$) = **if** X=curowner(h) **then** 0 **else** highest(h)
   highest(h;**others**) = highest(h)

   We here only include events observed by the auction agent, ignoring messages are not part of the history.

3c Invariant and Verification (weight 10)

1. Suggest a loop invariant for the auction agent. You may use the history functions above (from question 3b) even if you have not defined them.

2. Verify the loop invariant. If your loop invariant is a conjunction of several part involving the history, it suffices to consider one of these parts.

(Continued on page 11.)

Solution:
There are two loops. As long as you look at one of them, that is fine. For both invariants we have

$$highBid = highest(h) \wedge owner = curowner(h)$$

We could also add

$$owner = null \Rightarrow hBidder = null \wedge highBid = 0$$

An invariant for the outer loop could be:

$$highBid = highest(h) = 0 \wedge hBidder = null \wedge owner = curowner(h) = null$$

Or simply $owner = null$ plus (parts of) the two conditions above. An invariant for the inner loop could be:

$$curowner(h) \neq null \wedge highBid = highest(h) \wedge owner = curowner(h)$$

Weaker invariants would also be fine, but at least one clause should involve one of the defined history functions.

Verification is mainly straight forward: As invariant for the inner loop, we here consider

$$I \equiv highBid = highest(h) \wedge owner = curowner(h)$$

There are two []-branches that change $highest(h)$, $curowner(h)$, $owner$, or $highBid$, namely the branches for close and newbid. For the "close"-branch, we get one verification condition for each if-branch. The else-part is trivial since none of $highest(h)$, $curowner(h)$, $owner$, $highBid$ are changed. Consider the verification condition for the then-branch. We need to prove that I implies:

$$\forall X \,.\, (X = owner) \Rightarrow I^{\,owner,highBid,h}_{\,null,0,(h;(X\downarrow Auction{:}close());(..\uparrow..{:}..);(..\uparrow..{:}..))}$$

The quantifier on X is due to the rule for await (combined with a question mark). Consider an arbitrary X and assume $X = owner$. We must prove

$$0 = highest(h;(X \downarrow Auction : close())) \wedge null = curowner(h;(X \downarrow Auction : close()))$$

This is possible using the assumption $owner = curowner(h)$ and I, and the definitions of $curowner$ and $highest$.

The other cases are more or less similar.