UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Examination in:	INF4140 — Models of Concurrency
Day of examination:	17. December 2013
Examination hours:	14.30 – 18.30

This problem set consists of 19 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
```

Solution 1 *In all cases and questions: they key to termination is the third branch, so: is it possible that the third branch is executed, is it guaranteed, and what about fairness.*

(Continued on page 3.)

1. *possible termination: it's possible for both. For program 1: the trick is to realize that* `x+x` *is* not atomic. *If it where, y might be even "always", and under this min-conception, the answer "program cannot terminate" seems plausible. That the second one may terminate is obvious.*

2. *Termination guarantee: it's clear that the answer is* no *for both.*

3. influence of fairness: *The key to fairness is to think about programs running "forever" (it's a condition for infinite execution): For program 1: fairness (whichever) obviously has no influence, if the* $y = 5$ *is missed, it's not executed and won't be ever enabled again. For the second program: That's more tricky.*

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).

(Continued on page 4.)

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.

Remark 1 (Sushi bar) *The problem is decribed in [**?**, 7.1.2]. The book provides first a "wrong" solution. We decided not to show the wrong solution and to have it analyzed, because it's not clear whether such a task is confusing (and then puts the student on a wrong track etc).*

There is a certain similarity with the Roller-coaster problem. In the lecture the RC problem was given as part of oblig-3, which they should therefore be familiar with. A difference is that the RC was intended to be solved by monitors, *here we ask first for a semaphore solution. The roller-coaster is also (as semaphores) described in [**?**, Section 5.7]. The "interface" (i.e., the way the clients are supposed to interact) of the monitor solution and the one for the semaphores are not the same. One difference (at least in [**?**]) between the RC and the sushi is that the RC care is "*active*" in the sense that it triggers "unloading" of the passengers. In the sushi-bar the customers are allowed to leave themselves.*

Let's identify the synchronization needs (first the logical ones, and then afterwards, when introducing some shared variables, some mutexes for the specific synchronization to protect them). The logical synchronizations are described in the task. The way the skeleton is given indicates: the sushi-bar is not really an "entity" of itself (for instance a monitor-module). All the functionality is in the procedures (and appropriate shared vars plus semaphores). From the task description, it should be obvious that the implementation needs to keep track (probably explicitly, i.e., via dedicated shared variables) of two entities

- *customers eating, and*
- *customers waiting.*

The obvious invariant is that

$$eating \leq 5$$

It also should be a goal (liveness) that if $eating = 0$, then the number of waiting clients is reduced. For leaving: no client must be refused from

(Continued on page 5.)

leaving, i.e., the exit-procedure must be non-blocking.[1] *The specific additional complication/synchronization condition is: it's not that a client can enter the bar as soon as there's a free chair: once the sushi-bar is full, it needs to go empty again until the first new one can enter (if there are people waiting, preferably from the waiting queue, if fairness is an issue.) That means, we* cannot *use the number of eating person in the corresponding synchronization condition. Basically, we need at least one more bit to indicate how to interpret if the bar is not empty nor full, i.e., if* $0 < eating < 5$. *The border-cases are clear. And it's at the border cases, where that condition toggles. In the code below, the boolean* `must_wait` *is plays that role: it's initially false, it's set to true when (if ever) hitting 5, and reset to false when (if ever) the number of eaters hit 0 egain.*

Those considerations about the logical synch. conditions (and adding some mutex protection when doing some manipulations) may lead to the code of Listing 3.

Listing 3: Erroneous solution

```
1   enter() {
2     P(mutex);
3     if   must_wait
4     then waiting = waiting + 1;
5          V(mutex);
6          P(block);
7          P(mutex);      # !!
8          waiting -= 1
9     else skip;
10    eating += 1
11    must_wait = (eating == 5)
12    V(mutex);
13  }
14
15
16  leave() {
17    P(mutex);
18    eating -= 1
19    if   eating == 0:
20    then n = min(5, waiting)
21         V^n(block);
22         must_wait = false;
23    else skip;
24    V(mutex);
25  }
```

Unfortunately, it's wrong. The error illustrates one "challenge" of the task. The general core of the problem should be known from various places from the lecture (where? also with the sem's?): if an enters the "critical part" of such problems (here the sushi bar), that involves 2 things:

[1] Probably there will be some mutex-handing when exiting, so technically a blocking operation is being done. Nonetheless, assuming that no one stays in its mutex-section forever and that the mutex-protections are used appropriately, it's not a problem, the delay is negligible.

(Continued on page 6.)

1. *"testing": checking that the condition to wait for is satisfied*
2. *"setting": once the thread wishing to enter has been granted access, the condition changes again, for instance by setting appropriate variables governing the condition.*

The general challenge is to get the testing and setting either atomic, resp. make sure that if it's not atomic, it works nontheless.

That's the problem also here: ignoring the boolean variable discussed above, the two shared vars needed for getting the logic to work is to keep track of the ones eating *and the ones* waiting.

In is instructive to compare it with the readers-writers problem (especially when done with semaphores), which will highlight the particular challenge here. At the surface, the problem seems comparable: in both cases, one needs to keept track of a pair of variables, both counting numbers of processes: here eating and waiting and in the readers/writers case: reading and writing processes. The more problematic variable is the readers *in the latter case and the ones* waiting *the case here.*[2] *The* ***crucial*** *difference between the two problems is: the* readers *refer to processes* inside *the critical section, the* waiting *refers to processes* not inside *the critical section. As a consequence: for the R/W problem*

> *the reader trying to access the critical section could adapt the number of readers with* `nr++` before *getting potentially suspended at the guarding semaphore.*

Here, the same is done for the `waiting` *counter. However, and that's the point: once the suspended process is woken up and allowed to proceed to the critical section,* this variable conceptually is to be decreased. *In the readers-writers problem, the reader*[3] *can proceed to the critical section* without touching this (or any other) shared var again! *Here, in entry protocol of the customer increases* `waiting` *before getting suspended (which is unproblematic), but one may be tempted to* also *decreases it afterwards. That will lead to the erroneous code shown, violating the basic safety invariant.*

*The correct solution resembles (for instance) the way from the lecture/book how to implement P and V operations for FIFO semaphores using S&C monitors (*passing the condition*, see [?, Figure 5.3]), except that here we have no monitors.* □

Solution 2 (of Sushi) *A solution that avoids the mentioned problem is given in Listing 4. [?] also presents a second possible solution.*

[2] why?

[3] The writer is less problematic.

(Continued on page 7.)

Listing 4: Sushi

```
1    enter() {
2      P(mutex);
3      if   must_wait
4      then waiting = waiting + 1;
5           V(mutex);
6           P(block);
7      else eating = eating +1;
8           must_wait = (eating == 5)
9           V(mutex);
10   }
11
12
13 leave() {
14   P(mutex);
15   eating -= 1
16   if   eating == 0:
17   then n = min(5, waiting)
18        waiting = waiting - n;  // !
19        eating  = eating  + n;  // !
20        must_wait = (eating == 5); // !
21        V^n(block);
22   else skip;
23   V(mutex);
24 }
```

(Continued on page 8.)

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

(Continued on page 9.)

baboons in the opposite direction to cross the rope forever. Extend your solution to repair that weakness.

Remark 2 *The problem can be found in [?]. In the book [?, page 256] and in exercise 4, we had a pretty related problem ("one lane bridge"). The difference is that the monkey problem here is "bounded" whereas the one-lane-bridge was not.* □

Solution 3 (Baboons & monitors)

Code and invariant: *Here's a possible solution. It's a direct adaptation of the* bridge *problem discussed in the exercises (see [?, Exercise 5.7]). The sync-needs are pretty clear. In terms of monitor invariants, the one from the bridge in the book was formulated not only on the shared variables but also on the* condition variables *using the* `empty` *condition-variable inspection function. We cannot directly do that here, since we have not officially introduced that there is a function that can find out the number of queued processes on a condition variable.*[4] *The invariant for the bridge is*

$$(n_s = 0 \vee n_n = 0) \wedge (n_s > 0 \rightarrow \mathit{empty}(\mathit{gosouth})) \wedge \\ (n_n > 0 \rightarrow \mathit{empty}(\mathit{gonorth})) \qquad (1)$$

One obvious generalization of the invariant is the following: That does not work like that any more for two reasons. One obvously is that there cannot be more than 5 baboons, so the first one needs to be refined and also the other to factors containing the condition on the condition variables cannot be like that. However, the straightforward generalization

$$(5 \geq n_s = 0 \vee 5 \geq n_n = 0) \wedge \\ (5 > n_s > 0 \rightarrow \mathit{empty}(\mathit{gosouth})) \wedge (5 > n_n > 0 \rightarrow \mathit{empty}(\mathit{gonorth}))$$

However, we have to be slightly careful (and be aware what a monitor invariant actually is). At first sight, equation (1) from the old problem holds because the implementation in the solution to the exercise used `signal_all` *for signalling:* `signal_all` *unblocks all waiting cars* atomically *and that seems (at first sight) be captured in the implications.*

In the monkey-problem we cannot just free 5 threads in the same way because we do not have a operation that can free a specified number of processes atomically.[5] *One has to remember however what a* monitor

[4] Of course here we are talking about formulating an invariant (which is logic) in principle we can talk about everything we want, but perhaps it's not obvious for the students.

[5] Also in the previous task for the semaphores, the more expressive V-operation there was not claimed to be atomic, but intended as abbreviation for a sequence of V's.

(Continued on page 10.)

invariant *is. It must hold at all times* if not process is actually inside the monitor. *Therefore, whether the signalling is atomic or not is irrelevant.*

The general pattern *for the monitor should be familiar from the lecture (if not even from the bridge-exercise in particular): The monitor has as big advantage built-in mutex protection. On the down-side, the familiar "complication" for the monitors (e.g. compared to await-statements) is that one has to be* inside *the monitor to check whether or not the conditions (= monitor invariant) hold to allow to proceed (and of course one has to be aware of the signalling discipline). Compared to the previous task (with semaphores): the monitor variables have a queue, and somehow work like "semaphores" in that one can wait on it, but the condition variables are of course not "counting", in particular the waiting-operation waits* unconditionally.

Listing 5: Baboons (too simplistic)

```
1  monitor Rope {
2    int ns = 0;          # number of apes going south
3    int nn = 0;          # number of apes going north
4    cond gosouth gonorth;
5    ## Invariant:
6    ## (ns == 0 or nn == 0) and
7    ## (ns <= 5) and (nn < 5) and
8    ## (ns > 0 => empty(gosouth)) and
9    ## (nn > 0 => empty(gonorth))
10
11   # called by apes wanting to go north
12   procedure go-north() {
13     while (ns > 0 ∧ nn ≥ 5 ) wait(gonorth)
14     nn = nn + 1
15   }
16   # called by apes finished going north
17   procedure end-north() {
18     nn = nn - 1;
19     signal(go-north)         ## added
20     if (nn == 0) signal_all(gosouth);
21   }
22
23   # called by apes wanting to go south
24   procedure go-south() {
25     while (nn > 0) wait(gosouth)
26     ns = ns + 1
27   }
28   # called by apes finished going south
29   procedure end-south() {
30     ns = ns - 1;
31     signal(go-south);       ## added
32     if (ns == 0) signal_all(gonorth);
33   }
34 }
```

No-starvation: *The code show has indeed the problem as stated in the task: a constant stream of north-going monkeys precludes any south-going*

(Continued on page 11.)

monkey ever to cross. In exercise 4*, we dealt with the problem in the context of the one-lane-bridge. The core of the solution there was to count also the* delayed *cars/monkeys.*

The "non-fair" solution for the one-lane bridge in exercise 4(a) used the standard while-wait pattern. Actually that could have been simplified in that the while-loop is replaced by a simple conditional. That was possible, because the invariant was simpler (the condition that the number in the requested direction is ≤ 5 *was missing). Now, the rest of the body of the enter-procedure (e.g.,* `go-noth`*)* may invalidate the invariant again. *Therefore, we must have the while-loop here. Further in that one-lane-bridge exercise, the solution for the* `FairBridge` *argued that one should avoid the while-loop since while loops "don't prevent sneaking". In that simpler setting, the control could be directly transferred from the "last north car" to potentially "south-waiting" cars, and one was* guaranteed *to enter. Therefore, the conditional-code without while loop was able to prevent that form of sneaking. As mentioned, we cannot use this simple approach, we need a while-loop to recheck the condition.*

However, the task is formulated to avoid the weakness that a constant stream of baboons may prevent *the opposite baboons from crossing. Depending on how one interprets that (and depending on fairness) also a while-solution (with sneaking) is acceptable.*

In that exercise we also had an alternative *solution without signal-all, but a while-loop over* `signal`*. We cannot directly use that, but we could do a solution that wakes up all the waiting but not more than 5. In order to port the idea of the solution, we need the following key insight: To avoid re-checking the invariant at the entry procedure makes it necessary that the signaler makes sure that the entry condition is satisfied upon signalling. Therefore, the signaller cannot use* `signal_all`*. Since now, in contrast to the bridge, the invariant requires that there are no more than 5 baboons on one direction, the signalling must* cap *the number of "wake-up" by 5.*

(Continued on page 12.)

Listing 6: Fair-baboons: *no*-while-loop & individual `signal`s

```
1  monitor Rope {
2    int ns = 0;          # number of apes going south
3    int nn = 0;          # number of apes going north
4    cond gosouth gonorth;
5    ## Invariant:
6    ## (ns == 0 or nn == 0) and
7    ## (ns <= 5) and (nn < 5) and
8    ## (ns > 0 => empty(gosouth)) and
9    ## (nn > 0 => empty(gonorth))
10
11   # called by apes wanting to go north
12   procedure go-north() {
13     if ((ns > 0 ∧ nn ≥ 5) ∨ ¬empty(gosouth))
14     then  wait(gonorth)
15     else  nn = nn + 1
16   }
17   # called by apes finished going north
18   procedure end-north() {
19     nn = nn - 1;
20     if (nn == 0)
21     then { m  = max(5,size(gosouth));
22            ns  = ns + m       # adapt number already here
23            signal^m(gosouth);
24          }
25   }
26
27   ...
28 }
```

□

(Continued on page 13.)

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 14.)

Solution 4 *Local event of* `B`*:*
$B \uparrow C : request(tonorth); C \downarrow B : cross(permission)$ *and* $B \uparrow C : arrive$

$\{h$ **is** $Cycle_B\}$
while `true` **do** $\{h$ **is** $Cycle_B\}$
  $\{\forall$ *permission.* **if** `(permission)` **then** $h; B \uparrow C : request(\texttt{tonorth});$
    $C \downarrow B : cross(permission); B \uparrow C : arrive$ **is** $Cycle_B\}$
    else $h; B \uparrow C : request(\texttt{tonorth}); \; C \downarrow B : cross(permission); \varepsilon$ **is** $Cycle_B\}$
  send `C:request(tonorth);`
  $\{\forall$ *permission.* **if** `(permission)`
    then $h; C \downarrow B : cross(permission); B \uparrow C : arrive$ **is** $Cycle_B$
    else $h; C \downarrow B : cross(permission); \varepsilon$ **is** $Cycle_B\}$
  await `C:cross(permission);`
  $\{$**if** `(permission)` **then** $h; B \uparrow C : arrive$ **is** $Cycle_B$ **else** $h; \varepsilon$ **is** $Cycle_B\}$
  if `(permission)` **then**
    `// ''cross the canyon''`
    $\{h; B \uparrow C : arrive$ **is** $Cycle_B\}$
    send `C:arrive;`
  fi
  $\{h$ **is** $Cycle_B\}$
od

Entry: *Initially,* h *is emtpy, it is trivial that* ε **is** $Cycle_B \Rightarrow h$ **is** $Cycle_B$

Loop:

h **is** $Cycle_B \Rightarrow \forall$ *permission.* **if** *(permission)*
  then $h; B \uparrow C$ *: request(tonorth);* $C \downarrow B$ *: cross(permission);* $B \uparrow C$ *: arrive* **is** $Cycle_B$
  else $h; B \uparrow C$ *: request(tonorth);* $C \downarrow B$ *: cross(permission);* ε **is** $Cycle_B$

It holds because h **is** $Cycle_B$ *and we know that*

$$B \uparrow C : request(tonorth), C \downarrow B : cross(permission), B \uparrow C : arrive$$

is $Cycle_B$*. Therefore,*

$$h; B \uparrow C : request(tonorth), C \downarrow B : cross(permission), B \uparrow C : arrive$$

is $Cycle_B$*.*

Analogous for the **else***-branch.*

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.)

(Continued on page 15.)

Solution 5

- ***After*** **send** *C:request(tonorth)*
  $\forall$ *permission.* **if** *(permission)* **then** *h; C* $\downarrow$ *B : cross(permission); B* $\uparrow$ *C : arrive* **is** $Cycle_B \Rightarrow h \leq Cycle_B$.
  Since $h; a; b$ **is** R *for some regular expression* R*, then* $h \leq R$.

- ***After*** **await** *C:cross(permission)*
  if *(permission)* **then** *h; B* $\uparrow$ *C : arrive* **is** $Cycle_B \Rightarrow h \leq Cycle_B$.
  Similar to the previous case.

- ***After*** **send** *C:arrive*
  h **is** $Cycle_B \Rightarrow h \leq Cycle_B$. *Obvious.*

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.

Solution 6

Listing 7: _

```
1  var direction, tonorth: Boolean;
2  var X: Agent;
3  var n: Int;
4
5  loop
6    await X?request(direction);
7    send X:cross(true);
8    n := n + 1;
9
10   while n > 0 do
11     (await X?request(tonorth);
12       if (tonorth = direction AND n < 5)
13         then
14           send X:cross(true);
15           n := n + 1;
16         else
17           send X:cross(false);)
18      []
19       (await X?arrive;
20       n := n - 1;)
21   od
22 endloop
```

(Continued on page 16.)

4d Events (weight 2)

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

Solution 7
$X \downarrow C : request(direction), C \uparrow X : cross(true)$, $C \uparrow X : cross(false)$ *and* $X \downarrow C : arrive$

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.

Solution 8 *Need to add waiting queues for both south-going and north-going baboons, and a boolean variable to indicate whose turn should be next. As soon as there is a baboon requesting to go to the opposite direction while there are some baboons are crossing the canyon, we can put the request for going to the opposite direction into the queue and indicate with the boolean variable that the opposite direction will be in the next turn. At the same time, any other requests to go to the current direction will be rejected. When the last baboon finishes crossing, we can send permission messages to the first 5 baboons (or all if there are less than 5 are waiting) waiting at the queue for the opposite direction.*

(One can also let at most 5 baboons waiting in the queue, and any further requests will be rejected instead of putting them into the queue.) □

(Continued on page 17.)

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.

Solution 9 *One can probably best do binary semaphores first. Let's do it like that.*

The first observation is: like that operation FA (and TAS) we had in the lecture/exam, the CAS has no "synchronizing power", in the sense of "delaying" a process. Since we are required to use only *CAS and not higher-level stuff, we need to program that "delaying" ourselves on the level of the programming language. The only way to do that is a* loop, *i.e., spinning/busy waiting. Let's assume a cell in main memory pointed at by* `mutex`. *We assume that the only possible (integer) values are 0 and 1. We interpret 1 as "free" and "0" as taken.*

```
1    P(mutex) {
2      while (CAS(mutex,1,0)) {skip}
3    }
4
5    V(mutex) {
6      mutex* = 1
7    }
```

With the standard mutex available, we now need to get the facility of counting.

(Continued on page 18.)

```
1     P_count(sem) {
2         bool safe = false;
3         while (not safe )
4         do P(mutex);
5            if   sem > 0
6            then safe := true
7                 sem  := sem -1
8            V(mutex)
9         od
10        }
```

(Continued on page 19.)

Appendix: Inference Rules

Assignment

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

Composition

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

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

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