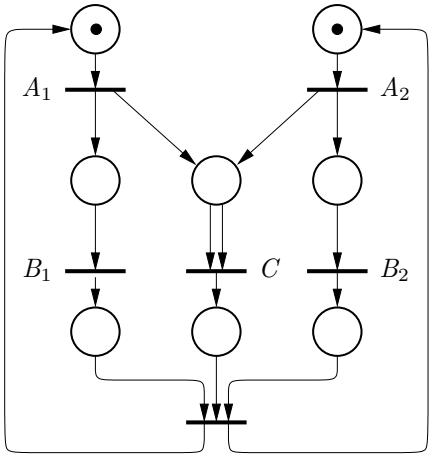
Solutions for Exercises, Week 12

1. Solution for Concurrent Systems Exam December 2006, Problem 2

Question 2.1



Question 2.2

Corresponding to the Petri-net, we introduce a semaphore *DoneA* that counts the number of A-operations executed. C may then be executed after n P-operations on *DoneA*. Q controls the final synchronization by awaiting a signal from each finished B-operation on a semaphore *DoneB* and then signalling each process P_i on a private semaphore $GoA[i]$:

```
var DoneA : semaphore;          // Counts no. of A's done
    DoneB : semaphore;          // Counts no. of B's done
    GoA[1..n] : semaphore;      // OK to start A_i again.
```

All semaphores are initialized to 0

```
process P[i : 1..n];              process Q;
  repeat                            repeat
    A_i ;                             for j in 1..n do P(DoneA);
    V(DoneA);                         C;
    B_i ;                             for j in 1..n do P(DoneB);
    V(DoneB);                         for j in 1..n do V(GoA[j])
    P(GoA[i])                       forever
  forever
```

[It is **not** possible to replace $GoA[1..n]$ with a common semaphore since a P process may wait again immediately after a wait and thereby could consume a token destined for another process.]

Question 2.3

```
monitor Sync

  var adone : integer := 0;          // No. of A's done
      done : integer := 0;           // No. of B's and C done
      OkC : condition;               // Wait for all A's done
      Alldone : condition;           // Wait for all B's and C done

  procedure EndA()
    adone := adone + 1;
    if adone = n then signal(OkC)

  procedure StartC()
    while adone < n do wait(OkC);
    adone := 0

  procedure Done()
    done := done + 1;
    if done < n + 1 then wait(Alldone)
                    else done := 0;
                         signal_all(Alldone)

end
```

```
process P[i : 1..n];
  repeat
    A_i ;
    Sync.EndA();
    B_i ;
    Sync.Done()
  forever
```

```
process Q;
  repeat
    Sync.StartC();
    C ;
    Sync.Done()
  forever
```

[Solution assumes no spurious wake-ups.]

2. Solution for Concurrent Systems Exam December 2008, Problem 3

Question 3.1

The operation must be declared as:

op *get_users*() **returns** *integer*;

and be accepted unconditionally by adding the following branch to both the inner and outer **in** statements:

[] *get_users*() **returns** *integer* → **return** *users*

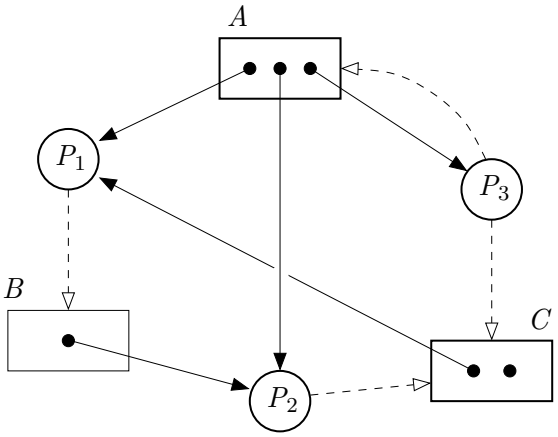
Question 3.2

The module *VarReg* must be initialized with $N = m$.

Writer:	*set*(0);	Readers:	*acquire*();
	writing		reading
	set(*m*);		*release*();

Question 3.3

(a)



(b) If the free C instance is granted to P_2, it may finish. Then P_1 and P_2 can finish in arbitrary order. Since all processes can finish, the situation would normally be called *safe*.

(c) If P_3 calls $Reg_C.acquire()$ (before P_2 does) and henceforth P_1 calls $Reg_B.acquire()$ and P_2 calls $Reg_C.acquire()$, all processes will have standing requests which cannot be fulfilled, since all instances are acquired. Hence the system has deadlocked.

(d) An attempt is made to reserve the resources according to a strict ordering:

In P_1, $Reg_C.acquire()$ and $Reg_A.acquire()$ are exchanged.
In P_2, $Reg_B.acquire()$ and $Reg_C.acquire()$ are exchanged.
In P_3, $Reg_C.acquire()$ and $Reg_A.acquire()$ are exchanged.

Hereby, the resources are reserved in the order: A, C and B.

There is a problem though, since P_3 reserves its two A instances in two rounds and hence the principle of deadlock prevention by strict ordering does not apply directly. However, since P_1 and P_2 only need one A instance each, there is always one instance "reserved" for

P_3. We may think of this as being taken by the first call of $Reg_A.acquire()$ in P_3 and may henceforth be ignored. Deadlock freedom then follows from the ordering principle applied to the remaining resources.

Question 3.4

The operations are assumed to act upon the following shared variables:

var *users* : *integer* := 0;
max : *natural* := N;
setting : *boolean* := *false*;

Now, the operations may be specified by:

acquire() :	$\langle\, users < max \rightarrow users := users + 1 \,\rangle$
release() :	$\langle\, users := users - 1 \,\rangle$
set(k : *natural*) :	$\langle\, \neg setting \rightarrow max := k;\quad setting := true \,\rangle$; $\langle\, users \leq max \rightarrow setting := false \,\rangle$

Question 3.5

(a)

```
monitor VarReg

  var users : integer := 0;
      max : natural := N;
      setting : boolean := false;
      Room, SizeOk, Done : condition;

  procedure acquire() {
    while users ≥ max do wait(Room);
    users := users + 1;
    if users < max then signal(Room)          — Cascade wakeup
  }

  procedure release() {
    users := users − 1;
    if users = max then signal(SizeOk);
    if users < max then signal(Room);
  }

  procedure set(k : natural) {
    while setting do wait(Done);
    max := k;
    setting := true;
    while users > max do wait(SizeOk);
    setting := false;
    signal(Done);
    if users < max then signal(Room);
  }

end
```

(b) Calls of *acquire*() should wait only if there is no room in the region:

$$I \triangleq waiting(Room) > 0 \Rightarrow users \geq max$$

However, with the chosen cascade wakeup, this has to be relaxed in order to take leaving calls into account:

$$I' \triangleq waiting(Room) > 0 \wedge woken(Room) = 0 \Rightarrow users \geq max$$