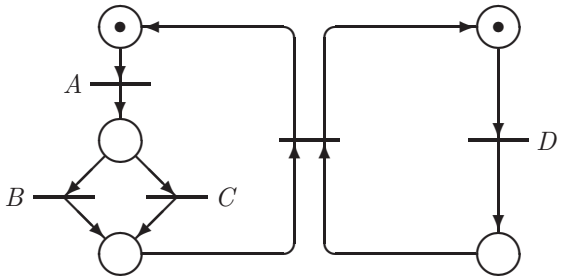
Solutions for Exercise Class 1

1. Solution for Petri.2

(a) The simplest Petri Net becomes:



It is seen that it is necessary to introduce an anonymous transition that ensures that the two sequential processes are synchronized in each round. (This synchronization may also be expressed in other, less obvious, ways.)

(b) From the above net it is seen that $(A, D), (B, D), (C, D)$ can be executed concurrently. (Since there exists behaviours of the net in which the corresponding transitions may fire simultanuously.)

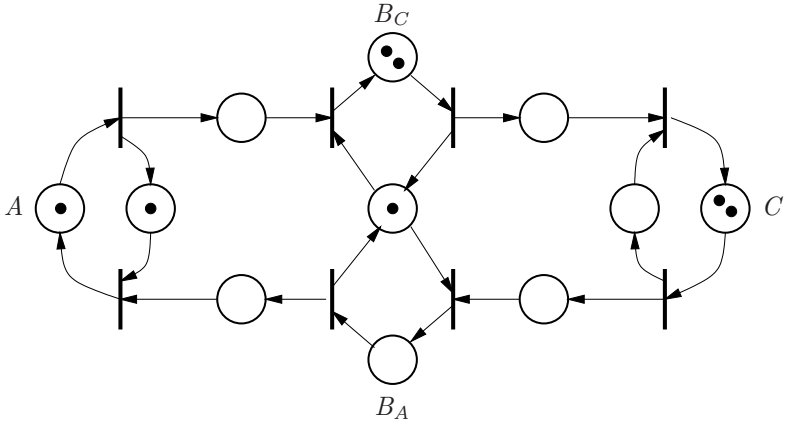
(c) For the first round, we get the following six possible interleavings:

A, B, D	A, C, D
A, D, B	A, D, C
D, A, B	D, A, C

2. A place with a single token is added, and a loop from/to this place is added to each of the transitions A, B, C, and D.

3. Solution for Petri.1

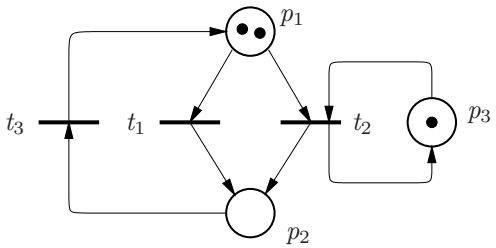
The jetties A and C are represented by places marked with the current number of boats. The jetty B is represented by two places, since boats are to continue towards A or C. All jetties have a place modelling the capacity of the jetty. Inbetween the jetties, there are places representing boats on their way on the sea. Here, the five boats have been placed arbitrarily at the jetties:



4. Advantages of formal models are clearly that they are precise and unambigious. Furthermore, graphical models like Petri Nets may even be very intuitive. A disadvantage is that formal models requires the reader to know the modelling language well. Futhermore, the relationship with the real-world phenomenon being modelled may be less obvious.

5. A fork over a place-node reflects a *choice*. A fork over a transition-node reflects *process creation*.

6.



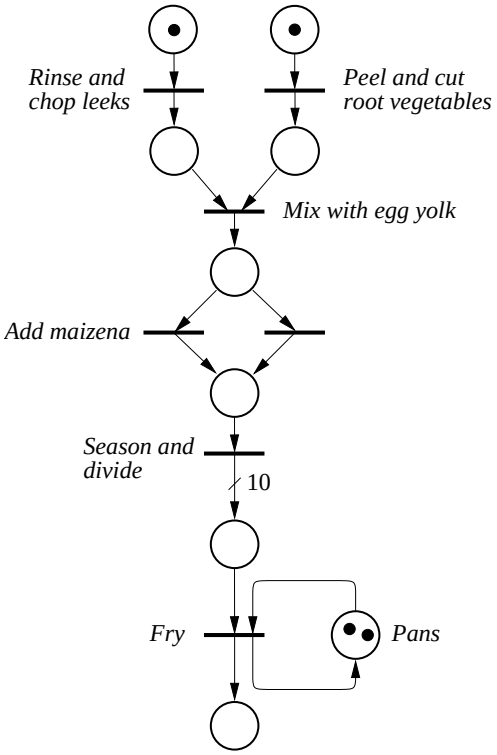
Possible firings: $M_0 \xrightarrow{\{t_1\}} (1,1,1)$, $M_0 \xrightarrow{\{t_2\}} (1,1,1)$, $M_0 \xrightarrow{\{t_1,t_2\}} (0,2,1)$, $M_0 \xrightarrow{\{t_1,t_1\}} (0,2,1)$

Note that t_1 may fire together with itself, but t_2 may not!

Solutions for Exercises, Week 1

1. Solution for Petri.3

The recipe can be modelled to various detail. Here is a proposal which reflects most of the desription:



Remarks: Whether to add maizena or not is represented by a choice with a void branch. The the crossed arc with a 10 represents 10 arrows corresponding to the division into portions. According to the way we define simultaneous firings of a transition with itself, the last part allows for frying two pancakes at a time.

2. Solution for Petri.5

Figure 1.1 in [Basic] is represented by Petri Net $N = (P, T, F)$ where

$$\begin{aligned} P &= \{p_1, p_2, p_3, p_4, p_5\} \\ T &= \{t_1, t_2, t_3, t_4, t_5\} \\ F &= \{(p_1, t_1), (p_1, t_3), (p_2, t_3), (p_3, t_2), (p_4, t_5), (p_5, t_4) \\ &\quad (t_1, p_3), (t_2, p_1), (t_3, p_4), (t_4, p_2), (t_5, p_3), (t_5, p_5)\} \end{aligned}$$

[In order to represent multiple arrows between a place/transition pair, the relation F is understood to be a *multiset* (ie. a set in which each element may occur more than once).]

Solutions for Exercises, Week 2

1. Solution for Trans.1

(1) $a_1\ a_2\ a_3\ b_1\ b_2$
(2) $a_1\ a_2\ b_1\ a_3\ b_2$
(3) $a_1\ a_2\ b_1\ b_2\ a_3$
(4) $a_1\ b_1\ a_2\ a_3\ b_2$
(5) $a_1\ b_1\ a_2\ b_2\ a_3$
(6) $a_1\ b_1\ b_2\ a_2\ a_3$
(7) $b_1\ a_1\ a_2\ a_3\ b_2$
(8) $b_1\ a_1\ a_2\ b_2\ a_3$
(9) $b_1\ a_1\ b_2\ a_2\ a_3$
(10) $b_1\ b_2\ a_1\ a_2\ a_3$

2. Solution for Trans.2

The number of interleavings is given by:

$$\left(\begin{array}{c} n_1+n_2 \\ n_1 \end{array}\right) = \left(\begin{array}{c} n_1+n_2 \\ n_2 \end{array}\right) = \frac{(n_1+n_2)!}{n_1! * n_2!}$$

An argument:

> Each interleaving must contain n_1 actions from P_1 and n_2 actions from P_2, ie. in total $n_1 + n_2$ actions. We may say that each interleaving has $n_1 + n_2$ places and an interleaving is the uniquely given by selecting n_1 of these for the actions of P_1 (the actions are supposed to come in the given order). As known from combinatorics, the number of ways n_1 elements can be selected out of $n_1 + n_2$ elements is given by the above expression.

An other argument:

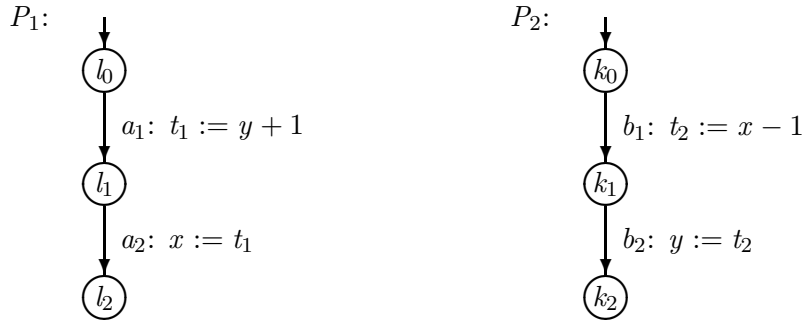
> For any interleaving, the actions from P_1 can be permuted in $n_1!$ ways and the actions from P_2 in $n_2!$ ways. From any interleaving, we may thus generate $n_1! * n_2!$ permuations of the total of $(n_1 + n_2)!$ permutations of all actions of P_1 and P_2. From this, the expression follows.

2. Solution for Trans.5

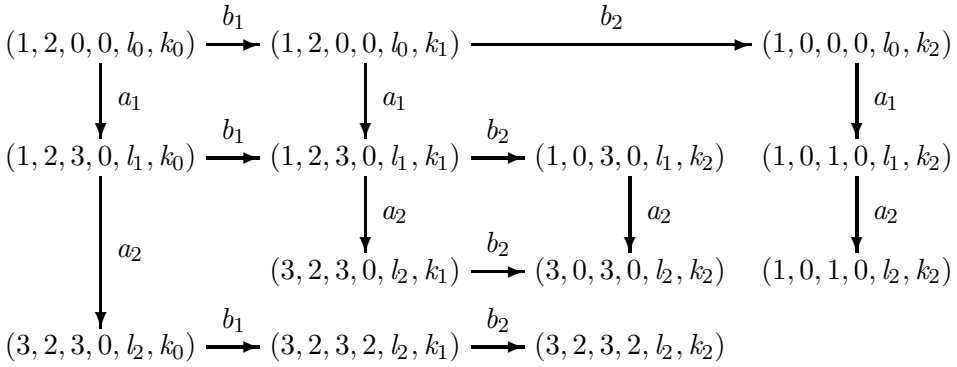
Rewriting to atomic assignment statements:

$x := 1;\ y := 2;$
co
$\langle\, t_1 := y + 1 \,\rangle;\ \langle\, x := t_1 \,\rangle \parallel \langle\, t_2 := x - 1 \,\rangle;\ \langle\, y := t_2 \,\rangle$
oc

Or equivalently as transition diagrams:



Let the global state be given by a vector $(x, y, t_1, t_2, \pi_1, \pi_2)$ where π_i is the control pointer for process P_i. Assuming arbitrarily t_1 and t_2 to be initially 0, the transition system for the concurrent systems is given by the transition graph:

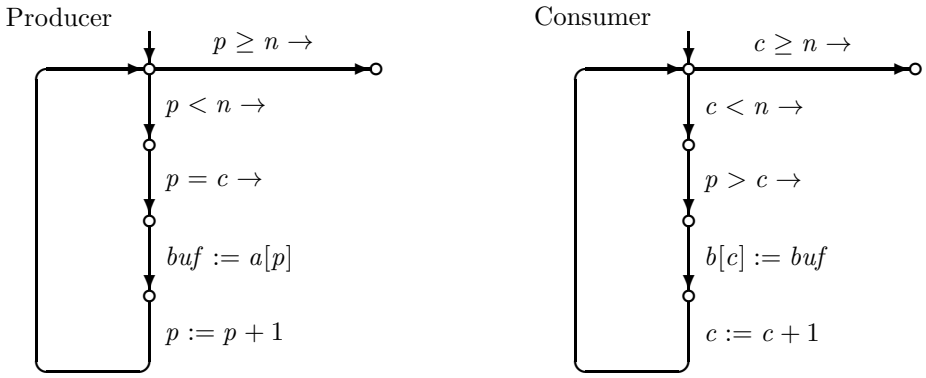


By inspection of the final states ($\pi_1 = l_2 \wedge \pi_2 = k_2$) we find the possibilities for (x, y):

$$\{(1,0),(3,0),(3,2)\}$$

3. Solution for Trans.6

Transition diagrams of the processes in Andrews Fig. 2.2:



Solutions for Exercises, Week 3

1. Solution for Trans.3

(1)	$a_1\ b_1\ b_2\ c_1$	(7)	$b_1\ c_1\ b_2\ a_1$
(2)	$a_1\ b_1\ c_1\ b_2$	(8)	$b_1\ b_2\ a_1\ c_1$
(3)	$a_1\ c_1\ b_1\ b_2$	(9)	$b_1\ b_2\ c_1\ a_1$
(4)	$b_1\ a_1\ b_2\ c_1$	(10)	$c_1\ a_1\ b_1\ b_2$
(5)	$b_1\ a_1\ c_1\ b_2$	(11)	$c_1\ b_1\ a_1\ b_2$
(6)	$b_1\ c_1\ a_1\ b_2$	(12)	$c_1\ b_1\ b_2\ a_1$

2. Solution for Trans.4

Number of interleavings:

$$\frac{(n_1 + n_2 + ... + n_k)!}{n_1! * n_2! * ... * n_k!}$$

This forumula can be obtained by first selecting places for P_1, then for P_2 from the remaining slots etc. and then multiply these possibilities together. The formula then follows by reduction.

Another, more direct, way to get the formula is to generalize the second argument from the solution for Trans.2.

3. Solution for Andrews Ex. 2.11

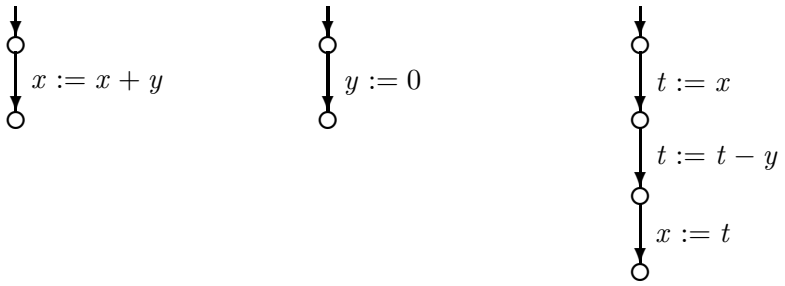
(a) The expression evaluation must be divided into three atomic reading steps. Now, each variable may or may not have been changed when read. Since all changes incidentally increment each variable by 3, zero to three increments may be seen. Thus, the possible final values are $\{3, 6, 9, 12\}$.

(b) Since the variables are updated by independent processes, each variable still may or may not have been changed when read irrespective of the ordering of the readings. Thus, the possible results are again: $\{3, 6, 9, 12\}$

4. Solution for Andrews Ex. 2.14

(a) No, the statement of the last process, $x := x - y$, has three critical references, since both x and y are read and written by other processes. The statement $\langle\, x := x + y \,\rangle$ is by definition atomic and thus not subject to the rule.

(b) Rewriting the last process to atomic statements $\langle\, t := x \,\rangle$; $\langle\, t := t - y \,\rangle$; $\langle\, x := t \,\rangle$, the three processes are represented by the transition diagrams:



These give rise to 20 interleavings. Of course, all of these lead to $y = 0$. Analyzing the interleavings, we find that x may get the values $\{0, 1, 2\}$.

5. Solution for Share.1

The problem is to get two processes to make a true synchronization, i.e. the processes should "meet" inbetween they perform their operation.

Proposal 1

An obvious way to ensure the invariant is to use directly the variables that appear in the invariant, i.e. the number of times OP_A respectively OP_B have been executed. This results in the following program:

var a, b : *integer*;
$a := 0;\ b := 0;$

process A =	**process** B =
repeat	**repeat**
while $a > b$ **do** ;	**while** $b > a$ **do** ;
...	...
OP_A;	OP_B;
$a := a + 1$;	$b := b + 1$;
forever	**forever**

Notice that both tests and assignments are atomic according the the rule of critical references. Note also that the test cannot be changed to $a \neq b$ since deadlock may then occur (how?). Furthermore, it is important that both a and b are incremented *after* the execution of the respective operation.

This solution, however, has the drawback that it increments a and b indefinitely. An attempt to count **modulo** somthing is very difficult to do correctly. Another problem is that the synchronization code is spread over the process. This is readily solved by moving the increments up before the while-loops.

Proposal 2

An attempt to synchronize using flags could be to use a flag for each process according to the following strategy: When a process arrives at the "meeting place", it raises its flag and

waits for the flag of the other process to come up. When this is the case, it lowers its own flag and continues:

var $flag_A, flag_B$: *boolean*;
Initialization: $flag_A := false$; $flag_B := false$;

$SYNC_A$:	$SYNC_B$:
$flag_A := true$;	$flag_B := true$;
while $\neg flag_B$ **do** ;	**while** $\neg flag_A$ **do** ;
$flag_A := false$;	$flag_B := false$;

This solution is not correct! If a process is suspended after having raised its flag, the other process may run without blocking.

The solution cannot be patched by having the processes wait for the other process to take its flag down, before lowering its own, since this solution is subject to deadlock.

A correct solutionis obtained by having the processes take down the flag of the *other* one and then wait for their own flag to be lowered. It is, however, sufficient with only "half" of this solution. See proposal 4, below.

Proposal 3

One may try to use a variable to alternate the execution of the two processes:

var *turn* : (A, B);
Initialization: $turn := A$; – Arbitrary

$SYNC_A$:	$SYNC_B$:
$turn := B$;	$turn := A$;
while $turn = B$ **do** ;	**while** $turn = A$ **do** ;

This solution satisfies the invariant as no operation will get ahead of the other, but it will not utilize the concurrrency of the problem. A concurrent solution is presented below:

Proposal 4

Here we make an assymmetric solution with only one flag. The flag is used as follows: When a certain of the processes (say P_A) arrives at the meeting place, it raises the flag and waits for it to be lowered. The other process will start by waiting for the flag to be raised and will then lower it. In this way it is ascertained that both process will wait for the other one to arrive at the meeting place.

var *flag* : *boolean*;
Initialization: $flag := false$;

$SYNC_A$:	$SYNC_B$:
$flag := true$;	**while** $\neg flag$ **do** ;
while $flag$ **do** ;	$flag := false$;

In this solution there is no risk of overflow and the synchronization code has been gathered in the beginning of the processes.

Solutions for Exercise Clas 2

1. The final value of x may range from 2 (!) to 10.

Assuming the two processes to be called P_1 and P_2, this is how it can get as low as 2:

	x
Initially:	0
P_1 reads 0 from x.	0
P_2 increments x four times.	4
P_1 writes 1.	1
P_2 reads 1 from x.	1
P_1 increments x four times.	5
P_2 writes 2.	2

It can be shown (using invariants — not imagination!) that this is the smallest result.

2. First question: NO. $x := x + 2$ and $x := x + 1$ can be executed in any sequential order, but are not atomic.

Second question: NO. $x := 1$ and $x := 2$ are each atomic, but the order is important.

3. The the number of critical references within statements a to f are 2, 1, 1, 2, 1, and 2 respectively. Hence only statements b, c, and e can be considered atomic.

4. If a variable spans more than one memory word, it has to be accessed using several bus cycles. If these words are accessed by other processors or devices, intermediate memory states may be seen. Even if used only by a single processor, the access to a larger memory area (e.g. a record/structure) is likely to be divided into interruptable steps.

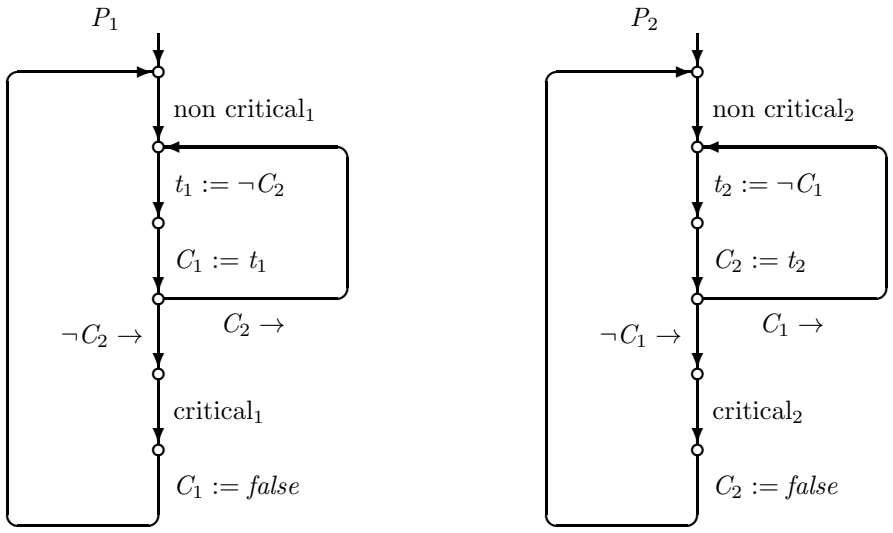
5. Usually the least addressable unit of memory is a byte. Thus to change a boolean variable represented as a bit of a byte, it is necessary to read the whole byte into a register, change the bit by masking and finally store the byte againg. This will not be atomic.

6. Solution for Share.2

(a) First we note that the statement $C_1 := \neg C_2$ cannot be considered atomic since C_1 is a shared variable and C_2 is a variable read by the other process. Rewriting to atomic actions we the following entry protocol for P_1:

```
repeat
  < t1 := ¬C2 >;
  < C1 := t1 >;
until < ¬C2 >;
```

Correspondingly for process P_2. Transition diagrams:



(b) The algorithm *does not* ensure mutual exclusion. We now see that with the initializations given, an execution in which the atomic actions of the two processes alternate will first set both C-s to true and in the next repetition, both variables false after which both processes will enter their critical section!

(c) Since the idea of the algorithm is to set ones flag to the opposite of the flag of the other process, it is tempting to believe that the algoritm will work, if the statements $C_1 := \neg C_2$ and $C_2 := \neg C_1$ are executed atomically. But even assuming these to be atomic, the following execution is possible:

	C_1	C_2
Initially:	*false*	*false*
P_2 executes nc_2, its entry-protocol and enters cs_2.	*false*	*true*
P_1 executes nc_1 and (atomically) sets $C_1 := \neg C_2$.	*false*	*true*
P_2 leaves cs_2 and executes $C_2 := false$.	*false*	*false*
P_1 tests C_2 and enters cs_1.	*false*	*false*
P_2 executes nc_2, enters its entry-protocol, sets C_2 to true, finds C_1 to be false and enters cs_2.	*false*	*true*

Both processes are now in their critical sections!

The trouble is that the value of C_2 that is tested is not the same as the one that C_1 is set relative to (and vice versa).

[If the **until**-test in P_1 is changed to C_1 and correspondingly in P_2 to C_2, the algorithm ensures mutual exclusion given atomic assignments.

To actually prove this we need the following auxiliary invariants:

$$\begin{array}{lcll} G_i & \triangleq & in\ cs_i \Rightarrow C_i & i = 1, 2 \\ I & \triangleq & \neg(C_1 \wedge C_2) & \end{array}$$

Now assume that both processes are in their critical sections

$$in\ cs_1 \wedge in\ cs_2$$

According to G_i this would mean that both C variables were true. This, however, would be in contradiction with I. Thus, if G_i and I are invariants, mutual exclusion is ensured.

We are now going to show the auxliary invariants. G_1 and G_2 are seen to be local invariants.

I is shown by an inductive argument:

- Initially I holds since both C_1 and C_2 are false.
- Since e_1 obviously preserves I, the only potentially dangerous action in P_2 is a_1:

  a_1**:** This actions will preserve I, as C_1 is set to the negation of C_2 and hence one of them will be false after the action.
- By symmetry, all actions in P_2 will also preserve I.

Thus I is an invariant of the program.]

7. In this course, we define a *critical region* to comprise a set of *critical sections* which are pieces of code among which there must be mutual exclusion. In the literature, this distinction is not always made.

8. Yes. The only constraint is that there cannot be two processes active within the same region at the same time.

9. Yes. For instance there may be a region protecting the use of a printer and a region protecting some shared variables. Critical regions may even overlap.

10. (a) $\Box\neg Snows(Bermuda)$

(b) $\Box(Snows(Helsinki) \Rightarrow Snows(Finland))$

(c) $\Box(Snows(Norway) \Rightarrow \Diamond Snows(Sweden))$

(d) $\Box\Diamond Snows(DK) \wedge \Box\Diamond Snows(NZ) \wedge \Box\neg(Snows(DK) \wedge Snows(NZ))$

(e) $\Box(Snows(Sahara) \Rightarrow \Box Snows(Sahara))$

(f) $\Box\exists x : Snows(x)$

Solutions for CP Exercises, Week 4

1. Solution for Andrews Ex. 3.3

(a)

```
var l : integer := 1;

process P[i : 1..n] =
  var r : integer := 0;
  repeat
    nc1:  non-critical section_i;
          repeat
              Swap(r, l);
          until r = 1;
    cs1:  critical section_i;
          Swap(r, l);
  forever;
```

We are now going to prove that the above solution does ensure mutual exclusion.

First, we assume that the local variables r are renamed to global variables r_i $(i = 1..n)$ that are all initialized to 0;

Next, we prove some auxiliary invariants:

$$\begin{array}{lcll} F_i & \triangleq & in\ cs_i \Rightarrow r_i = 1 & i = 1..n \\ G & \triangleq & l \in 0, 1 & \\ H_i & \triangleq & r_i \in 0, 1 & i = 1..n \end{array}$$

Since r_i is changed only in P_i, F_i is a local invariant By induction, H_i and G are easily seen to be invariants since 0 and 1 are the only values being swapped around.

Now we define

$$I \triangleq r_1 + r_2 + \ldots + r_n + l = 1$$

This holds intially and since any of the variables are changed only by atomic swapping of two of them, their sum will remain constant. Therefore, I is an invariant of the program.

Now, if two of the processes P_i and P_j $(j \neq i)$ should be in their critical sections at the same time, F_i and together with G and H_i would give us

$$r_1 + r_2 + \ldots + r_n + l \geq 2$$

contradicting the invariant I. Thus, we conclude that this cannot be the case, ie. the algorithm ensures mutual exclusion.

If two or more processes execute $Swap(r_i, l)$ at the same time, one of the will get the "token" first and thereby obtain access to the region. Which of them it is not determined. Thus, the algorithm cannot deadlock nor livlock, but it is *not fair*. Starvation can occur if other process manages to enter the region inbetween a given process attempts to execute $Swap(r_i, l)$.

(b) To avoid memory contention by writing to l, its value may be checked before an attempt is made to change it with *Swap*:

```
repeat
    while l = 0 do skip;
    Swap(r, l);
until r = 1;
```

This will not effect the proof in (a).

(c) *Not included*

2. Solution for Andrews Ex. 3.2

```
var l : integer := 1;

process P[i : 1..n] =
  var s : integer;
  repeat
    non-critical section_i;
    DEC(l, s);
    while s > 0 do {
      INC(l, s);
      delay;
      DEC(l, s);
    }
    critical section_i;
    INC(l, s);
  forever;
```

Here, the lock l is used as in the test-and-set solution. However, if the lock is already "set" ($l < 1$), the effect of *DEC* must be undone by *INC*, before trying again. The correctness argument (or proof) follows the same line as for the test-and-set solution.

Solutions for Exercises, Week 5

1. Solution for Theory.1

Question 1.1

(a) I holds initially since $x = 0 \wedge y = 0$.

All three a-actions are potenially dangerous for I:

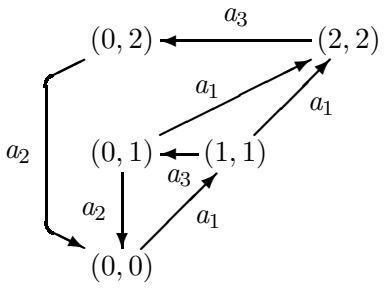
a_1: Assuming I to hold before the actions, the precondition ensures that $0 < y \leq 2$ after the actions. Further $x = y$ and therefore I holds after execution of a_1.

a_2: The precondition $x = 0$ and the effect ensures $x = 0 \wedge y = 0$ after the actions. Thus, I holds.

a_3: If I holds before the actions, we must have $0 = x \leq y \leq 2$ after the actions, thus I still holds.

Since I holds initially and is preserved by all atomic actions, I is an invariant of the program.

(b) Transition graph:



Further there are a_3 self-loops on states $(0,0)$, $(0,1)$, and $(0,2)$ plus an a_2 self-loop on state $(0,0)$. Since these are relevant only for liveness properties and only under less than weak fairness assumptions, they are not shown.

(c) From the transition graph, showing the complete reachable state space, we see by inpection that $(x, y) = (1, 2)$ is not reachable and thus $\neg(x = 1 \wedge y = 2)$ is an invariant of the program.

Question 1.2

[Assuming at least weak fairness, the self-loops on the transition graph are irrelevant.]

(a) Given a transition graph, weak fairness ensures that the execution cannot remain forever in a state which can be left by one or more actions. For the transition graph we therefore conclude that any execution must pass through $(x, y) = (1, 1)$ over and over again. Thus, $\Box\Diamond x = 1$ is satisfied.

(b) Consider the infinite execution

$$(0,0) \xrightarrow{a_1} (1,1) \xrightarrow{a_3} (0,1) \xrightarrow{a_2} (0,0) \xrightarrow{a_1} (1,1) \xrightarrow{a_3} \cdots$$

In this execution, all actions are executed infinitely often, thus strong fairness is satisfied. However, no state with $x = 2$ is met.

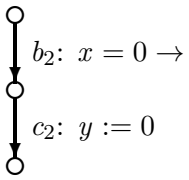
Note: The notion of fairness is related to *actions*. If a particular action is taken *in any state*, fairness is satisfied for that action.

Question 1.3

(a) If a_2 cannot be considered atomic as a whole, by the default assumption of atomic reads and writes, it will correspond to

$$b_2\colon \langle\, \textbf{await}\ x = 0 \,\rangle; \quad c_2\colon \langle\, y := 0 \,\rangle$$

This can be depicted by the transition diagram:



Now, the interleaving

$$(0,0) \xrightarrow{b_2} (0,0) \xrightarrow{a_1} (1,1) \xrightarrow{c_2} (1,0)$$

violates I.

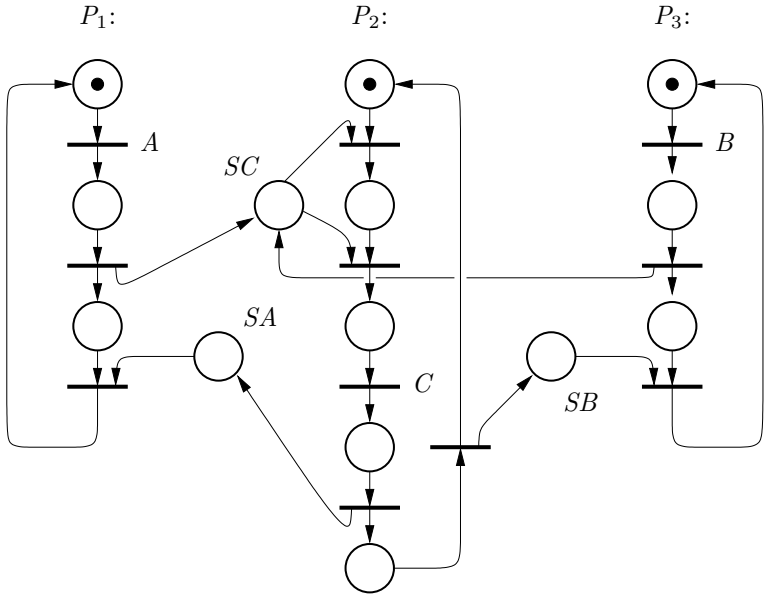
(b) H can be defined as:

$$H \ \triangleq\ I \wedge (at\ c_1 \Rightarrow y < 2) \wedge (at\ d_1 \Rightarrow x \leq t < 2)$$

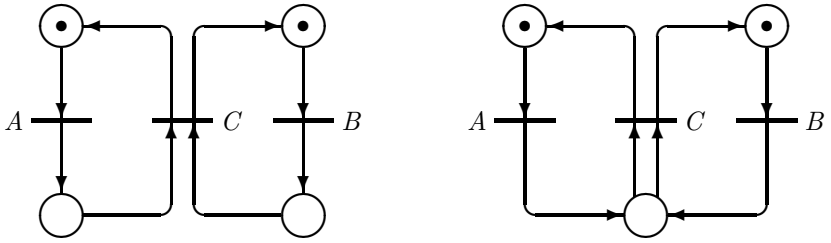
which is readily seen to hold initially and imply I. Further, H is preserved by all atomic actions. Especially, the conjunct $(at\ d_1 \Rightarrow x \leq t < 2)$ is needed to conclude I after d_1.

2. Solution for Sema.1

Direct "translation" of the program to a Petri Net:



Two Petri Nets that both directly expresses how A, B, and C are synchronized:



3. Solution for Sema.2

We here chose P_D to be the "master" that starts up P_A which in turn signals P_B or P_C:

var SA, SBC, SD : *semaphore* := 0;

process P_A;	**process** P_B;	**process** P_C;	**process** P_D;
repeat	**repeat**	**repeat**	**repeat**
P(SA);	P(SBC);	P(SBC);	V(SA);
A;	B;	C;	D;
V(SBC)	V(SD)	V(SD)	P(SD)
forever;	**forever**;	**forever**;	**forever**;

4. Solution for Sema.4

Proposal I

For each pair of processes (P_i, P_j) we introduce a semaphore S_{ij} that is used for signalling from P_i to P_j. Each process signals the two other ones and awaits a signal from each of these:

var $S_{AB}, S_{AC}, S_{BA}, S_{BC}, S_{CA}, S_{CB}$: *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_{AB});	V(S_{BA});	V(S_{CA});
V(S_{AC});	V(S_{BC});	V(S_{CB});
P(S_{BA});	P(S_{AB});	P(S_{AC});
P(S_{CA});	P(S_{CB});	P(S_{BC});

This solution is readily shown to be correct using the semaphore invariant.

Proposal II

Each process has a semaphore to be signalled by the other processes. Each process starts by signalling to each of the two other ones and then waits on its own semaphore for two signals (which are expected to from each of the other processes).

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_B);	V(S_A);	V(S_A);
V(S_C);	V(S_C);	V(S_B);
P(S_A);	P(S_B);	P(S_C);
P(S_A);	P(S_B);	P(S_C);

This solution is correct but works so marginally that it cannot be shown directly by use of the semaphore invariant!

NB: The "dual" solution below, where each process signals its own semaphore twice and then waits on each of the other semaphores *does not work*. The error is that a process may "use" a signal that was supposed for another process.

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_A);	V(S_B);	V(S_C);
V(S_A);	V(S_B);	V(S_C);
P(S_B);	P(S_A);	P(S_A);
P(S_C);	P(S_C);	P(S_B);

The circular solution below *does not work either*:

var S_A, S_B, S_C : *semaphore* := 0;

$SYNC_A$:	$SYNC_B$:	$SYNC_C$:
V(S_B);	V(S_C);	V(S_A);
P(S_A);	P(S_B);	P(S_C);

It has the general fault that a process does not wait for *all* the other processes. If a signalling "the other way round" is added, it will work.

Solutions for Exercises, Week 6

1. Solution for Andrews Ex. 4.24

The idea is that a writer should "take all n coconuts" to make sure that no readers are active. This has to be done through n single P-operations, but this could lead to deadlock if started by more than one writer. Therefore, writers first have to get the right to start this operation by entring a critical region protected by another semaphore $mutex_w$.

```
var rw : semaphore := n;
    mutex_w : semaphore := 1;

process Reader[i : 1..n] =
  ...
  P(rw);
  read the database;
  V(rw);
  ...

process Writer[j : 1..m] =
  ...
  P(mutex_w);
  for k in 1..n do P(rw);
  read the database;
  for k in 1..n do V(rw);
  V(mutex_w);
  ...
```

This solution is fair towards both readers and writers if the semaphores are strongly fair (e.g. FIFO).

2. Solution for Andrews Ex. 4.6

To implement the *sleep/wakeup* mechanism, we need a semaphore for mutual exclusion and one for suspension. A counter keeps track of the number of waiting processes.

```
var S : semaphore := 1;
    Q : semaphore := 0;
    K : integer := 0;

sleep: P(S);
       K := K + 1;
       V(S)
       P(Q);
       K := K - 1;
       if K > 0 then V(Q)
                else V(S)

wakeup: P(S);
        if K > 0 then V(Q)
                 else V(S)
```

Note that a solution in which *sleep* increments K and waits on Q and *wakeup* signals K times does **not** work, since the signals may be taken by new processes arriving later. The above solution using the *baton* technique ensures that new processes are not allowed to interfere during the cascaded wakeup.

3. Solution for Andrews Ex. 4.14

In order to achieve concurrent deposits and concurrent deposits, we may try to do only the slot allocation under mutual exclusion. Then, however, the filling (and emptying) of the slots can occur out of order, rendering the use of the *front* and *rear* pointers unfeasible. We may choose to accept the loose ordering and protect each slot by private full/empty semaphores.

A more conservative option, shown here, is to keep the ordering by passing a signal down the slots indicating that the previous slot has been filled/emptied. This is done through two arrays of semaphores *prev_full* and *prev_empty*:

```
var buf[0..n − 1] : T;
    front, rear : integer := 0;
    full : semaphore := 0;
    empty : semaphore := n;
    prev_full[0..n − 1] : semaphore := 0;
    prev_empty[0..n − 1] : semaphore := 0;
    mutex_P, mutex_C : semaphore := 1;

    V(prev_full[0]); V(pref_empty[0]);          — first slot OK
```

```
process Producer[i : 1..M] =
  var data : T;
      inpos : integer;
  repeat
    data := produce;
    P(empty);
    P(mutex_P);
    inpos := rear;
    rear := (rear + 1) mod n;
    V(mutex_P);
    buf[inpos] := data;
    P(prev_full[inpos]);
    V(full);
    V(prev_full[(inpos + 1) mod n)]);
  forever
```

```
process Consumer[j : 1..N] =
  var result : T;
      outpos : integer;
  repeat
    P(full);
    P(mutex_C);
    outpos := front;
    front := (front + 1) mod n;
    V(mutex_C);
    result := buf[outpos];
    P(prev_empty[outpos]);
    V(empty);
    V(prev_empty[(outpos + 1) mod n)]);
    consume result;
  forever
```

This higher degree of concurrency may be beneficial if the data type T is a large datatype such that buffer insertions/removals take significant time.

4. Solution for Sema.3

(a) The solution to the meeting problem (see [Basic]) does not work for binary semaphores since the following execution is possible:

Process A signals S_B.

Process B signals S_A.

Process A passes $\texttt{P}(S_A)$, executes OP_A and signals S_B.

Now, two signallings on S_B have been performed without an intermediate wait. The precise effect of this depends on the particular kind of binary semaphore and should generally be avoided.

(b) A solution with binary semaphores is obtained by (getting the idea of) interchanging `P` and `V` in one of the processes:

$$\begin{array}{lll} SYNC_A\text{: } \texttt{V}(S_B); & & SYNC_B\text{: } \texttt{P}(S_B); \\ \quad\quad\quad\ \texttt{P}(S_A); & & \quad\quad\quad\ \texttt{V}(S_A); \end{array}$$

That this solution still ensures that the two operations do not deviate from each other is proven by using the semaphore invariants as before. Furthermore, we may show that the values of the semaphores can never exceed 1. From the program, we obtain the following inequalities:

$$\begin{array}{rcccl} \#\texttt{P}(S_A) & \leq & \#\texttt{V}(S_B) & \leq & \#\texttt{P}(S_A)+1 \\ \#\texttt{V}(S_A) & \leq & \#\texttt{P}(S_B) & \leq & \#\texttt{V}(S_A)+1 \end{array}$$

Since S_B is initialized to 0 we have the semaphore invariant $\#\texttt{P}(S_B) \leq \#\texttt{V}(S_B)$. Together with the above we get:

$$\#\texttt{V}(S_A) \leq \#\texttt{P}(S_B) \leq \#\texttt{V}(S_B) \leq \#\texttt{P}(S_A)+1$$

Subtracting $\#\texttt{P}(S_A)$ from both sides we get:

$$\#\texttt{V}(S_A) - \#\texttt{P}(S_A) \leq 1$$

or, as the left hand side is precisely the semaphore value s_a,

$$s_a \leq 1$$

That is, using general semaphores the value of S_A can never exceed 1. Thus, S_A may as well be implemented by a binary semaphore. Correspondingly we can show that $s_b \leq 1$.

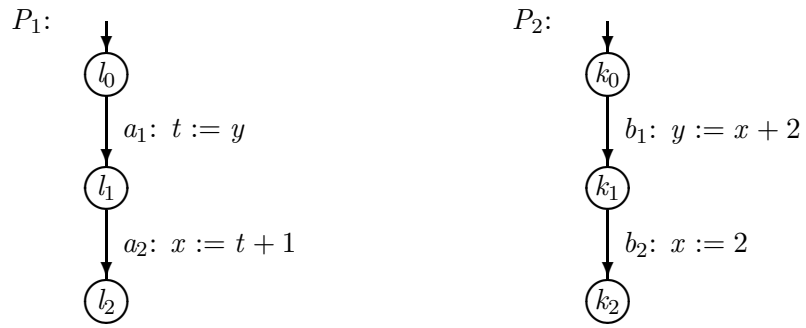
3. Solution for Concurrent Systems Exam December 2003, Problem 2

Question 2.1

The statements b, d, e, and f can be considered to be atomic since they only have only one critical reference each. Both a and c have two critical references.

Question 2.2

(a)



[Location and action labels not required.]

(b) Going through the 6 possible interleavings, the possible results for (x, y) are found to be:

$$(2,3), (2,2), (3,2), (1,2)$$

Question 2.3

P is preserved by a_2 and a_3. [By a_2 since $y < 0$ and P imply $x > 0$ and hence y also becomes positive.]

Q is preserved by all three actions. [Also by a_2 since it cannot be executed when Q holds.]

R is preserved only by a_2.

Question 2.4

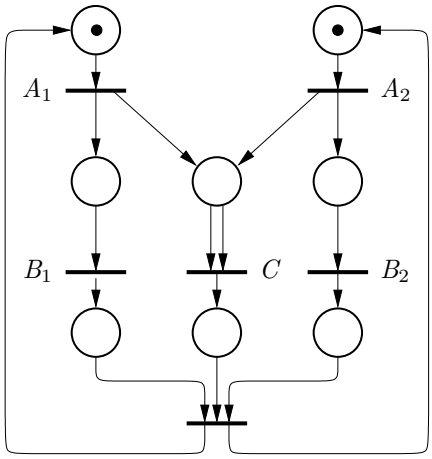
(a) The sequence $(0,1)(1,2)$ repeated forever will satisfy all parts of F.

(b) Assuming F, only the guard of a_3 is constantly true and hence only a_3 is guaranteed to be eventually executed under weak fairness. [If $x \neq 0$ it must be positive due to $\Box x \geq 0$ and hence $y > x > 0$ imply $y > 1$.]

(c) Assuming F, the guards of a_1, a_2, and a_3 will be infinitely often true, and hence they will be eventually executed under strong fairness. [The guard of a_4 is not necessarily true, for instance, it is never true in the state sequence proposed in (a).]

4. 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];
  repeat
    A_i ;
    V(DoneA);
    B_i ;
    V(DoneB);
    P(GoA[i])
  forever
```

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

Solutions for Exercise Class 3

1. Solution for Mon.1

monitor *Meet*

var OK : *boolean* := *false*; — Has the the other arrived?
c : *condition*; — First one waits here

procedure $SYNC_A()$
if OK **then** $\{OK$:= *false*; $signal(c)\}$
else $\{OK$:= *true*; $wait(c)\}$

procedure $SYNC_B()$ — As $SYNC_A$, but coded alternatively
$OK := \neg OK$;
if $\neg OK$ **then** $signal(c)$ **else** $wait(c)$;

end

If we utilize the possibility of asking whether a condition queue is empty or not, we may eliminate the variable OK and both operations become:

procedure $SYNC_{A/B}()$
if $empty(c)$ **then** $wait(c)$ **else** $signal(c)$

There are a number of other (more complex) solution where different queues are used, where each process has a flag etc.

2. See solution to point 6. below.

3. Both are used for letting a process wait until woken up, but they are indeed different:

- Semaphores may be used anywhere. Condition queues are associated with monitors and may only be used within these.
- The *wait* operation on a condition queue *atomically* realeases the monitor while putting the calling process on the queue.
  If a `P`-operation on a semaphore is made within a monitor operation, the process will just wait on the semaphore while the monitor's critical region remains locked.
- If there are no waiting processes, a `V` operation on a semaphore is remembered by incrementing the semaphore value, while signalling an empty condition queue has no effect.

4. Assuming that the variable `b` is protected by the monitor's critical region, this will lead to a deadlock since the critical region is not released during calls of `sleep()`.

5. Based on the second solution idea in point 1. above, adding two monitor variables, *first* and *sum*, the problem may be solved by the following:

monitor *MeetAndSum*

var *first*, *sum* : *integer* := 0;
c : *condition*; — First one waits here

function $SYNC_{A/B}(x : integer)$
if *empty*(*c*) **then** {*first* := *x*; *wait*(*c*)}
else {*sum* := *first* + *x*; *signal*(*c*)};
return *sum*

end

Notice, that since the last process at the meeting may immediately call *SYNC* again before the first of the processes has returned, the sum must be held in a separate variable, *sum*. This variabale will not be overwritten, because the monitor is assumed to be used by two dedicated processes only.

[Since conditions are not rechecked after the wait, this solutions would not work if *spurious wakeups* could occur. But according to the standard definition (in [Andrews]) they do not.]

6. Solution for Mon.4

(a)

monitor *Event*

var *c* : *condition*;

procedure *sleep*()
wait(*c*)

procedure *wakeup*()
signal_all(*c*)

end

[Note that since no conditions are checked after the wait, this solution would **not** work if *spurious wake-ups* could occur.]

(b) To prevent spurious wakeups to let threads pass *sleep()* when no *wakeup()* is called, the current state of the synchronization mechanism must be registered in the monitor. First we try to use a flag *go* in combination with a *count* of the number of waiting threads:

```
monitor Event

  var c : condition;
      count : integer := 0;
      go : boolean := false;

  procedure sleep()
    count := count + 1;
    while ¬go do wait(c);
    count := count − 1;
    if count = 0 then go := false;

  procedure wakeup()
    if count > 0 then
      go := true;
      signal_all(c)

end
```

This solution is robust towards spurious wakeups, but is not quite faithful to the specification, since calls of `sleep()` made before all waiting threads have left the monitor are allowed to pass in the same round. Depending on the actual use, this may be acceptable.

It turns out to be very difficult to control exactly which threads are woken up explicity and which accidentally by spurious wakeups.

However, a simple and precise solution is obtained by using the idea of the *ticket algorithm* where processes keep track of their position themselves.

```
monitor Event

  var c : condition;
      round : integer := 0;

  procedure sleep()
    myround : integer := round;
    while myround = round do wait(c);

  procedure wakeup()
    round := round + 1;
    signal_all(c)

end
```

For this we need an indefinite integer to count the current round. In practice though, a 64-bit `long` will suffice.

Also here, we could count the waiting threads and only increment `round` when there are threads actually waiting.

Solutions for Exercises, Week 7

1. Solution for Andrews Ex. 5.4

Given Figure 5.5 in [Andrews] (here in our notation):

```
monitor RW_Controller :

  var nr, nw : integer := 0;
      oktoread : condition;
      oktowrite : condition;

  procedure request_read()
    while nw > 0 do wait(oktoread);
    nr := nr + 1;

  procedure release_read()
    nr := nr − 1;
    if nr = 0 then signal(oktowrite)

  procedure request_write()
    while nr > 0 ∨ nw > 0 do wait(oktowrite);
    nw := nw + 1;

  procedure release_write()
    nw := nw − 1;
    signal(oktowrite)
    signal_all(oktoread)

end
```

(a) The *signal_all* operation can be replaced with repeated signalling:

```
while ¬empty(oktoread) do signal(oktoread);
```

Alternatively, *cascaded wakeup* can be used. Then *signal_all*(*oktoread*) is replaced by a single *signal*(*oktoread*), and *request_read* becomes:

```
procedure request_read()
  while nw > 0 do wait(oktoread);
  nr := nr + 1;
  signal(oktoread);
```

Cascaded wakeup is especially useful in situations where the number processes to be awakened is not known in advance, eg. may depend on parameters of the woken processes.

(b) To give preference to writers, readers should be held back if there are any pending writers in order to prevent starvation of writers. Likewise, writers should be favoured after a writing phase. Thus, *request_read*() and *release_write*() are modified to:

```
procedure request_read()
  while nw > 0 ∨ ¬empty(oktowrite) do wait(oktoread);
  nr := nr + 1;

procedure release_write()
  nw := nw − 1;
  if ¬empty(oktowrite) then signal(oktowrite)
                       else signal_all(oktoread)
```

(c) The solution below attempts to carefully alternate between readers and writers. Thus, the last reader should start a writer and an ending writer should start a group of readers. However, to prevent readers from starving writers, new readers should wait in a *prequeue* if there are writers waiting. Since all waiting readers should start together, the normal condition queue *oktoread* may be used for that purpose as well.

```
procedure request_read()
  if ¬empty(oktowrite) then wait(oktoread);
  while nw > 0 do wait(oktoread);
  nr := nr + 1;

procedure release_write()
  nw := nw − 1;
  if ¬empty(oktoread) then signal_all(oktoread)
                      else signal(oktowrite)
```

This solution, however, allows for new writers to overtake a woken writer and in theory a particular writer may be starved forever. For a truly fair solution, see (d).

It is also possible to use the general fairness technique of alternating a priority between the two groups. The priority is to be used only if both readers and writers are waiting. Here we use a boolean variable *reader_prio* indicating whether readers have priority (if not, writers have).

```
monitor Fairly_Fair_RW_Controller :

  var nr, nw : integer := 0;
      reader_prio : boolean := true;
      oktoread : condition;
      oktowrite : condition;

  procedure request_read()
    while nw > 0 ∨ (¬empty(oktowrite) ∧ ¬reader_prio) do
      wait(oktoread);
    nr := nr + 1;

  procedure release_read()
    reader_prio := false;
    nr := nr − 1;
    if nr = 0 then signal(oktowrite)

  procedure request_write()
    while nr > 0 ∨ nw > 0 ∨ (¬empty(oktoread) ∧ reader_prio) do
      wait(oktowrite);
    nw := nw + 1;

  procedure release_write()
    reader_prio := true;
    nw := nw − 1;
    if ¬empty(oktoread) then signal_all(oktoread)
                        else signal(oktowrite)
end
```

Here the priority is changed when (the first of) a group ends its operation. Again, in theory readers may still be starved, if they do not all get out of *request_read* before the first of the reader group changes the priority. However, in practice this would probably not be an issue if reading is a longer-lasting operation.

(d) [Advanced] In order to get a strict First-Come-First-Served discipline both readers and writers must be processed in some common entrance queue. Further, if the first process of this queue discovers that it cannot start (eg. being a writer when readers are active), it will have to wait being the first to be considered next time. For this to work two condition queues can be used: *pre* where processes queue up in FIFO order and *front* where the (single) front process of the queue waits. To determine which queue to wait at, a count *ne* of the currently entering readers/writers is maintained. Only when being the only one entering, a process it can go directly to the front position. Whenever a process leaves the front position, the next process from the *pre*-queue (if any) is moved to the front.

```
monitor FCFS_RW_Controller :

  var nr, nw, ne : integer := 0;
      pre : condition;
      next : condition;

  procedure request_read()
    ne := ne + 1;
    if ne > 1 then wait(pre);
    if nw > 0 then wait(front);
                   signal(pre);
    nr := nr + 1;
    ne := ne − 1;

  procedure release_read()
    nr := nr − 1;
    if nr = 0 then signal(next);

  procedure request_write()
    ne := ne + 1;
    if ne > 1 then wait(pre);
    if nw > 0 ∨ nr > 0 then wait(next);
                            signal(pre);
    nw := nw + 1;
    ne := ne − 1;

  procedure release_write()
    nw := nw − 1;
    signal(next);

end
```

[Due to the wait conditions not being rechecked, this solutions is not robust towards spurious wakeups.]

2. Solution for Andrews Ex. 5.8

(a) The required invariant must state that the balance never becomes negative:

$$I \triangleq Bal \geq 0$$

The basic problem in this exercise is that the waiting condition for each *withdraw*(*amount*) operation depends on the parameter value *amount*. A general solution to this is to use a *covering condition*, i.e. to wake up all waiting processes, whenever the balance has improved. It is assumed that all amounts belongs to a type of positive integers *PosInt*.

```
monitor SimpleAccount

  var Bal : integer := 0;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    signal_all(positive);

  procedure withdraw(amount : PosInt)
    while Bal < amount do wait(positive);
    Bal := Bal − amount;

end
```

(b) Under the standard assumption the the condition queues are FIFO, the customers may be served FCFS by waking only one at a time, but only as long as the balance is large enough (using the magic *amount* function). Special care must be taken to prevent outside processes from making withdrawals before the woken processes. This could be accomplished by letting the deposit operation do the balance decrementation as in the FIFO Semaphore solution shown in Andrews Figure 5.3. Here, however, we take a more general approach. Whenever a withdrawal process is woken, the monitor is considered *busy*, and new processes will have to wait. Now the processes are started in FIFO order by a cascade wakeup:

```
monitor MagicFSCSAccount

  var Bal : integer := 0;
      Busy : boolean := false;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    if ¬Busy ∧ ¬empty(positive) ∧ amount(positive) ≤ Bal then
      Busy := true;
      signal(positive);

  procedure withdraw(amount : PosInt)
    if Busy ∨ ¬empty(positive) ∨ Bal < amount then
      wait(positive);
      Busy := false;
```

```
    Bal := Bal − amount;                                  — Bal assumed large enough
    if ¬Busy ∧ ¬empty(positive) ∧ amount(positive) ≤ Bal then
      Busy := true;
      signal(positive);
  end
```

Note that deposit processes may increment *Bal*, even when the monitor is busy, but that will not violate the expectations of the woken withdrawal process.

(c) In order to implement the magic function *amount* giving the requested amount of the first withdrawal process, two ideas may be applied:

- A new, separate condition queue is used by the first waiting process and that processes may set a global amount variable. Further processes will have to wait on the old queue. Now great care must be taken to ensure that exactly one and only one of the waiting processes proceed to this queue.
- Within the monitor, a datastructure is maintained giving the amounts of the waiting processes. Thus the datastructure will be parallel to the condition queue. Since the queue is supposed to be FIFO, a list type will be appropiate.

Here we choose the latter approach, extending the above solution with a list type with operations *append*, *head* and *tail*:

```
monitor FSCSAccount

  var Bal : integer := 0;
      Busy : boolean := false;
      amounts : List of integer;
      positive : condition;

  procedure deposit(amount : PosInt)
    Bal := Bal + amount;
    if ¬Busy ∧ ¬empty(positive) ∧ head(amounts) ≤ Bal then
      Busy := true;
      amounts := tail(amounts);
      signal(positive);

  procedure withdraw(amount : PosInt)
    if Busy ∨ ¬empty(positive) ∨ Bal < amount then
      amounts := append(amounts, amount);
      wait(positive);
      Busy := false;
    Bal := Bal − amount;                                  — Bal assumed large enough
    if ¬Busy ∧ ¬empty(positive) ∧ head(amounts) ≤ Bal then
      Busy := true;
      amounts := tail(amounts);
      signal(positive);

end
```

Solutions for Exercises, Week 8

1. Solution for Mon.5

(a) A straightforward solution could be:

```
monitor ChunkSem

  var s : integer := 0;
      Empty : condition;
      NonEmpty : condition;

  procedure P()
    while s = 0 do wait(NonEmpty);
    s := s − 1;
    if s = 0 then signal(Empty)

  procedure V()
    while s ≠ 0 do wait(Empty);
    s := s + M;
    signal_all(NonEmpty)

end
```

(b) For the monitor, the following safety invariant should hold:

$$I_1 \triangleq 0 \leq s \leq M$$

Provided $M \geq 1$, this readily follows from the intialization and the **while** tests.

(c) We now try to express that that no calls of the $P()$-operation are ever "forgotten". This would be the case, if there remained processes but s was still positive. Thus we must require:

$$I_2 \triangleq \mathit{waiting}(\mathit{NonEmpty}) > 0 \Rightarrow s = 0$$

This follows from the initialization, the fact that $s = 0$ when waiting on *NonEmpty*, and the flushing of *NonEmpty*, when s is incremented.

(d) If many processes are waiting on *NonEmpty* and M is small, most of these processes will be unnecessarily woken up. In order to wake up only as many as can carry through the $P()$-operation, either a limited number of signals may be made or *cascaded wakeup* may be applied. Here we show the cascade solution:

```
monitor ChunkSem

  var s : integer := 0;
      Empty : condition;
      NonEmpty : condition;

  procedure P()
    while s = 0 do wait(NonEmpty);
    s := s − 1;
```

if $s > 0$ **then** $signal(NonEmpty)$
else $signal(Empty)$

procedure $V()$
while $s \neq 0$ **do** $wait(Empty)$;
$s := s + M$;
$signal(NonEmpty)$

end

Now, the property I_2 does not necessarily hold at entry to a monitor operation, since there may be processes left on the queue while a woken process is waiting to get back to the monitor. Therefore the invariant will have to be weakened taken the woken processes into account. Due to the cascade, at least one process will be woken as long as $s > 0$. This may be expressed as:

$$I_3 \triangleq waiting(NonEmpty) > 0 \Rightarrow s = 0 \vee woken(NonEmpty) > 0$$

[This can be formulated in a number of equivalent ways.]

For a solution using limited signalling (awakening up to M processes), the invariant should express that enough processes have been woken up:

$$I_4 \triangleq waiting(NonEmpty) > 0 \Rightarrow s \leq woken(NonEmpty)$$

(e) Since both waits recheck their conditions, the solutions shown in (d) is robust towards *spurious wakeups*. Also the invariants have been formulated with inequalities allowing for an spontaneous increase of $woken(NonEmpty)$.

(f) Since we have two waiting conditions the standard solution is to used a mixed condition queue and use a covering condition:

```
class ChunkSem {

  int s = 0;

  public synchronized void P() {
    while (s==0) try {wait();} catch (Exception e) {};
    s--;
    notifyAll();
  }

  public synchronized void V() {
    while (s!=0) try {wait();} catch (Exception e) {};
    s = s + M;
    notifyAll();
  }
}
```

However, since all calls of $P()$ are woken up when s becomes positive, only $V()$ operations can be waiting when $s > 0$. Aside: This may be formally expressed by an invariant:

$$I_5 \triangleq s > 0 \Rightarrow waiting_{P()}() = 0$$

Therefore, the signalling in $P()$ needs only be done when the condition for $V()$ is true and only has to awake a single thread.

```
public synchronized void P() {
  while (s==0) try {wait();} catch (Exception e) {};
  s--;
  if (s==0) notify();
}
```

An attempt to use the cascade solution will render both $P()$ and $V()$ calls waiting in the condition queue and hence will not work.

2. Solution for Mon.6

By introducing a variable, *next*, indicating the smallest waketime of any waiting processes, the number of unnecessary wakeups may be considerably reduced. Using our notation:

monitor *Timer*

var tod : *integer* := 0;
$next$: *integer** := ∞;
$check$: *condition*;

procedure *delay*(*interval* : *integer*)
var *waketime* : *integer*;
waketime := $tod + interval$;
while *waketime* > tod **do**
if *waketime* < *next* **then** *next* := *waketime*;
wait(*check*);

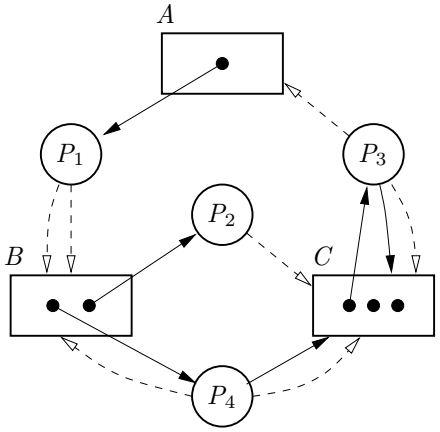
procedure *tick*()
tod := $tod + 1$;
if $tod \geq next$ **then** {*next* := ∞; *signal_all*(*check*)}

end

Here *integer** denotes the set of integers extended with ∞ larger than any integer.

3. Solution for CP Exam December 1998, Problem 4

Question 4.1



Question 4.2

(a) Finishing processes satifying their maximum demands:

Available			*Can be finished*
A	B	C	
0	0	2	P_2
0	1	2	P_4
0	2	2	P_1
1	2	2	P_3
1	2	3	

Since a sequence exists in which all the processes can have their maximal resource demands satisfied, the situation is *safe*.

(b) Even though P_4 is granted a C-instance, the above sequence is still possible and the situation is still safe. Thus, P_4 may be granted a C-instance according the banker's algorithm.

4. Solution for Silberschatz, Galvin & Gagne Exercise 7.11

For a sytem with m inscances of a resource type, a deadlock situation is characterized by a number of processes that are requesting more instances while holding some already, but no more instances are available.

A process P_i can request more instances only if it has not yet reached is maximal claim MAX_i. The maximal number of instances n processes may have reserved without having reached their maximum claim (and thereby be able to finish) is given by:

$$\sum_{i=1}^{n}(MAX_i - 1) = (\sum_{i=1}^{n} MAX_i) - n = MAX - n$$

Thus, no deadlock can occur if this number is less than the number of available instances m:

$$MAX - n < m$$

or

$$MAX < n + m$$

It is assumed that all processes need several instances, ie. $MAX_i > 1$ for all i and, of course, that $MAX_i \leq m$.

Solutions for CP Exercises, Week 9

1. Solution for Andrews Ex. 7.3

(a) Assume that the array to be sorted is put in the variable *input*.

```
var input[1..n] : integer := ...;
    output[1..n] : integer;

chan pass[i : 0..n] : integer;

process Feeder =
  for i in 1..n do
    send pass[0](input[i]);

process HoldMin[i : 1..n] =
  var min, x : integer;
  receive pass[i − 1](min);
  for j in 1..n − 1 do
    { receive pass[i − 1](x);
      if (x >= min) then send pass[i](x)
                    else {send pass[i](min); min := x}
    };
  send pass[i](min);

process Collector =
  for i in 1..n do
    receive pass[n](output[n + 1 − i]);
```

Now, the values sent on *pass*[*n*] will be sorted and can be inserted into the result array *output* by the collector.

Alternatively, the filter processes could directly set "its" element in the result array to the minimum value and just pass on the remaining ones.

2. Solution for Andrews Ex. 7.6

```
type Kind = Read | Write;

chan request : (Kind, integer);
chan release : ();
chan readok[i..n] : ();
chan writeok[i..m] : ();

process Reader[i : 1..n] =
  ...
  send request(Read, i)
  receive readok[i]();
  reading;
  send release();
  ...

process Writer[j : 1..m] =
  ...
  send request(Write, j)
  receive writeok[j]();
  writing;
  send release();
  ...

process RWControl =
  var k : kind;
      id : integer;
      active : integer := 0;
  receive request(k, id);
  repeat
    if k = Read then
      while k = Read do
        { send okread[id]();
          active := active + 1;
          receive request(k, id);
        }
    else
      { send okwrite[id]();
        active := 1;
        receive request(k, id);
      }
    while active > 0 do
      {receive release();  active := active − 1}
  forever
```

Rather than accepting all request at any time and record pending request, it has here been chosen to serve requests in order of arrival as long as possible, ie. either serve a single write request or a consecutive sequence of read requests. Releases need only be considered when a new request is about to be served.

Solutions for Exercise Class 4

1.

```
process Merge;
  var x : integer;
  do A?x → C!x
  [] B?x → C!x
  od
```

2.

```
process Sum;
  var x, y : integer;
  do true →
    if A?x → B?y
    [] B?y → A?x
    fi;
    C!x + y
  od
```

3. In order to meet, all processes must synchronize pairwise by CSP-communications. Care must be taken to avoid deadlock.

```
process P1;          process P2;          process P3;
  repeat               repeat               repeat
    P2!();               P1?();               P2?();
    P3?();               P3!();               P1!();
    ⋮                    ⋮                    ⋮
  forever              forever              forever
```

4. Solution for Andrews Ex. 8.14

```
module Account
  op deposit(amount : posinteger);
  op withdraw(amount : posinteger);
body
  var bal : integer := 0;

  process AccountServer;
    repeat
      in deposit(amount)                          → bal := bal + amount
      [] withdraw(amount) and amount ≤ bal        → bal := bal − amount
      ni
    forever;

end Account;
```

5. Solution for Andrews Ex. 8.15

(a)

```
module ABmeeting
  op MeetA();
  op MeetB();
body

  process MeetingServer;
    repeat
      in MeetA() →
        in MeetB() → skip ni;
        in MeetB() → skip ni;
      ni
    forever;

end ABmeeting;
```

(b)

```
module ABmeeting
  op MeetA();
  op MeetB();
body

  process MeetingServer;
    repeat
      in MeetA() →
        in MeetB() →
          in MeetB() → skip ni
        ni
      ni
    forever;

end ABmeeting;
```

6. Solution for Rendez.1

(a)

```
module Event
  op Pass();
  op Clear(var r : integer);
  op Release(v : posinteger);
body
  var S : integer;
  process EventServer;
    repeat
      in Pass() and S > 0 → skip
      [] Clear(var r)     → r := S; S := 0
      [] Release(v)       → S := S + v;
                            for i in 1..?Pass do
                              in Pass() → skip ni
      ni
    forever;
end Event;
```

[The loop in the *Release* branch ensures that all current calls of *Pass* are processed before a possible call of *Clear* as in the monitor version.]

(b) The semaphore operation P(s) is implemented by:

```
var l : integer;
repeat
  e.Pass;
  e.Clear(l)
until l > 0;
if l > 1 then e.Release(l − 1)
```

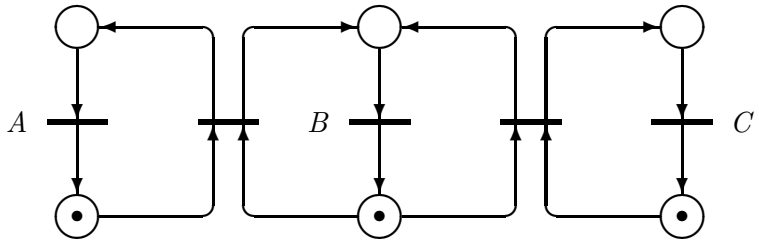
[The call of *Pass* ensures that the semaphore value is not tested (with *Clear*) until it is known to have been positive. Hereby a busy wait is reduced to a semi-busy one being much less resource demanding.]

Solutions for Exercises, Week 10

1. Solution for Exam June 1994, Problem 3

Question 3.1

Before each round, P_2 must synchronize with *either* P_1 or P_3. A Petri-net expressing this is:



Question 3.2

[A synchronization can be implemented by signalling forth and back using two semaphores. A choice between two synchronizations is then implemented by using a common semaphore for the signalling:]

var S_{AC}, S_B : *semaphore* := 0;

process P_A =	**process** P_B =	**process** P_C =
repeat	**repeat**	**repeat**
wait(S_{AC});	*signal*(S_{AC});	*wait*(S_{AC});
signal(S_B);	*wait*(S_B);	*signal*(S_B);
A	B	C
forever;	**forever**;	**forever**;

[Alternatively, *wait* and *signal* may be exchanged in all three processes.]

2. Solution for Andrews Ex. 8.9

(a) A solution giving priority to writers is shown in [Andrews p.388] using the facility to query the number pending calls of an operation *op* to block readers when there are pending writers.

(b) A fair solution can be obtained from the above solution by explicitly processing the readers inbetween the writers. Using our notation, we get:

```
module ReadersWriters
  op read(var T);
  op write(T);
body

  op startread();
  op endread();

  var val : T;

  proc read(var r : T)
    startread();
    r := val;
    endread()

  process Writer =
    var nr : integer := 0;
    repeat
      in startread() and ?write = 0 → nr := nr + 1
      [] endread()                  → nr := nr − 1
      [] write(v : T) and nr = 0    → val := v
                                      while ?startread > 0 do
                                        in startread() → nr := nr + 1 ni
      ni
    forever;

end ReadersWriters;
```

Although extra readers may slip through while the reader group is started in the *write* branch, they cannot recur at the *startread* queue as they cannot pass *endread*. So for a finite number of readers, the *startread* queue will eventually be emptied.

If only the readers waiting when the write has ended should be started, the **while** loop may be replaced by:

```
for i in 1..?startread do
  in startread() → nr := nr + 1 ni
```

3. Solution for Andrews Ex. 8.12

Assuming that a guard using the function that gives the number of pending operation calls is re-eavaluated whenever a call is made, we can do with:

```
module Barrier
  op arrive();
body

  process Control =
    var nr : integer := 0;
    repeat
      in arrive() and ?arrive >= n → for i in 1..n − 1 do
                                       in arrive() → skip ni
      ni
    forever;

end Barrier;
```

If such size-dependent guards were not reevaluated, we could instead nest n accepts within each other by recursion:

```
module Barrier
  op arrive();
body

  procedure meet(k : integer)
    in arrive() → if k > 1 then meet(k − 1) ni

  process Control =
    var nr : integer := 0;
    repeat
      meet(n)
    forever;

end Barrier;
```

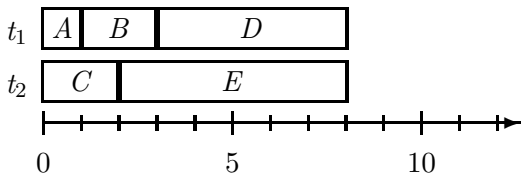
Aside: In Ada neither of these solutions are possible since the *count*-attribute is not reevaluated and **accept**-statements can occur only in the main loop of a task (not within procedures). Instead the *requeue* facility must be used.

[Solutions for Exam Dec. 2018 and Dec. 2020 will appear on the material page.]

Solutions for Exercises, Week 11

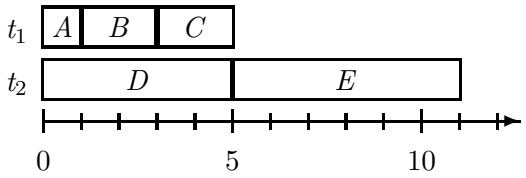
1. Solution for ParComp.1

(a) With the given task execution times it is possible to make a perfect fit:



This gives an execution time of 8 seconds and a speedup of $\frac{16}{8} = 2$.

(b) If task E is postponed, the following scenario is possible:



with an execution time of 11 seconds and a speedup of $\frac{16}{11} \approx 1.45$.

2. Solution for Concurrent Systems Exam December 2002, Problem 3

Question 3.1

(a) $I \triangleq open \Rightarrow waiting(Queue) = 0$

(b) I is a monitor invariant since processes only wait at *Queue* if *open* is false and *Queue* is emptied by *signal_all* whenever *open* becomes true.

(c) If there are currrently less than k processes waiting when $Go(k)$ is called, all of these are woken, but new calls of $Pass()$ still have to wait (if closed). As a special case, if the gate is open, a call of $Go(k)$ has no effect.

Question 3.2

```
module Gate
  op Pass();
  op Set(boolean);
  op Go(integer);
body

  process Keeper;
    var open : boolean := false;
    repeat
      in Pass() and open → skip
      [] Set(b : boolean)   → open := b
      [] Go(k : integer)    → for j in 1..min(k, ?Pass) do
                                in Pass() → skip ni
      ni;
    forever;

end Gate;
```

3. Solution for Concurrent Systems Exam December 2003, Problem 3

Question 3.1

The first three calls of *put*() will enable one of the calls of *unload*() to succeed. The two remaing calls of *put*() will then both succeed leaving the server with $count = 2$. The second call of *unload*() will remain blocked waiting for acceptance by the server.

Question 3.2

```
monitor Batch

  var count : integer := 0;
      NonFull : condition;
      Full : condition;

  procedure unload() {
    while count < N do wait(Full);
    count := 0;
    signal(NonFull);
  }

  procedure put() {
    while count = N do wait(NonFull);
    count := count + 1;
    if count < N then signal(NonFull) else signal(Full);
  }

end
```

In this solution, care has been taken to wake up only the mininum number of waiting *put*-calls using a cascade wakeup on *NonFull*. A solution in which *unload* signals to all on *NonFull* is also acceptable.

4. Solution for Concurrent Systems Exam December 2004, Problem 3

Question 3.1

```
monitor Latch

  var count : integer := 0;
      IsZero : condition;

  procedure set(k : integer) {
    if k ≥ 0 then count := k;
    if count = 0 then signal_all(IsZero)
  }

  procedure down() {
    if count > 0 then count := count − 1;
    if count = 0 then signal_all(IsZero)
  }

  procedure await() {
    while count ≠ 0 do wait(IsZero)
  }

end
```

[The signalling in *set* is necessary since *count* may be set to 0 (!). The **while**-loop in *await* might be replaced by an **if**-statement although in the server-based solution, not all waiting processes are guaranteed to get through before a *set* is called and hence the solution shown here is closer to this semantics.]

Question 3.2

The given SYNCHRONIZE code indicates a solution with two simple (i.e. one-time) barriers in a row. Care must be taken in resetting the simple barriers properly.

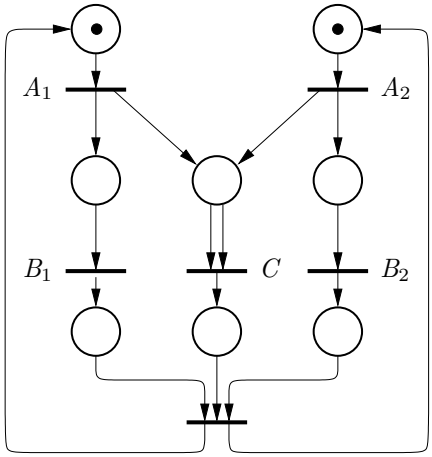
```
process Q;
  repeat
    latch1.await();
    latch3.set(n);  latch4.set(1);          — Prepare for second stop
    latch2.set(0);
    latch3.await();
    latch1.set(n);  latch2.set(1);          — Prepare for next round
    latch4.set(0);
  forever;
```

[$set(0)$ may be replaced by $down()$. Resetting of $latch_1$ and $latch_3$ may be done earlier.]

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*

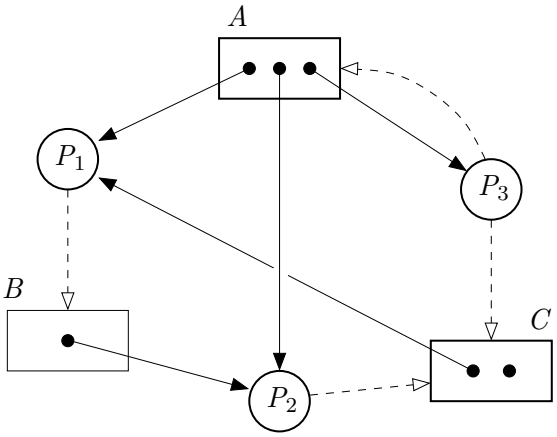
Question 3.2

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*) :	$\langle\, \neg setting \rightarrow max := k;\ \ setting := true \,\rangle$; $\langle\, users \leq max \rightarrow setting := false \,\rangle$

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:

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