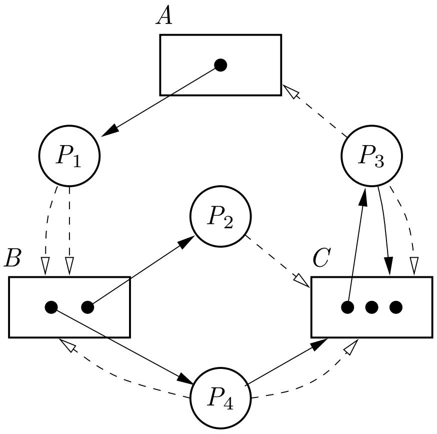
Solutions for CP Exercise Class 6

1. Solution for CP Exam December 1998, Problem 4

Question 4.1



Question 4.2

(a) Finishing processes satifying their maximum demands:

Available A	B	C	*Can be finished*
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 bankman's algorithm.

2.

process *Merge* =
  var x : *integer*;
  do $A\,?\,x \rightarrow C\,!\,x$
  $[\!]$ $B\,?\,x \rightarrow C\,!\,x$
  od

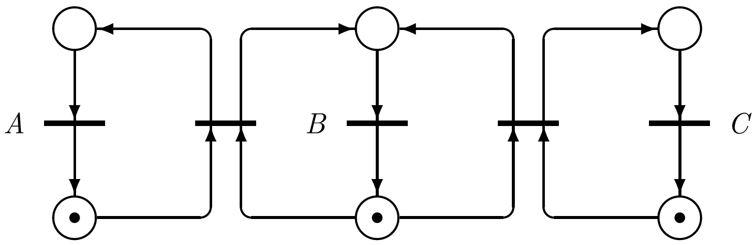
3.

process *Sum* =
  var x, y : *integer*;
  repeat
    if $A\,?\,x \rightarrow B\,?\,y$
    $[\!]$ $B\,?\,y \rightarrow A\,?\,x$
    fi;
    $C\,!\,x + y$
  forever

4. 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**
`P`(S_{AC});	`V`(S_{AC});	`P`(S_{AC});
`V`(S_B);	`P`(S_B);	`V`(S_B);
A	B	C
forever;	**forever**;	**forever**;

[Alternatively, `P` and `V` may be exchanged in all three processes.]

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

process P_1 =	**process** P_2 =	**process** P_3 =
repeat	**repeat**	**repeat**
$P_2\,!\,()$;	$P_1\,?\,()$;	$P_2\,?\,()$;
$P_3\,?\,()$;	$P_3\,!\,()$;	$P_1\,!\,()$;
$\vdots$	$\vdots$	$\vdots$
forever	**forever**	**forever**

6. Solution for Andrews Ex. 7.7

```
type Id = 1..n;
type Time = integer;
type Op = GetClock(Id) | Delay(Id, integer) | Tick;

chan request : Op;
chan reply[Id] : Time;
chan go[Id] : ();

process User[i : Id] =
  ...
  send request(GetClock(i));                          — Get clock
  receive reply[i](clock);
  ...
  send request(Delay(i, period));                     — Delay
  receive go[i]();
  ...


process Timer =
  var op : Op;
      q : PrioQueue<(Time, ID)>;
      time : Time := 0;
  repeat
    receive request(op);
    case op :
      GetClock(id): send reply[id](time);
      Delay(id, p): insert(q, (time + p, id));
      Tick:         time := time + 1;
                    while nonempty(q) ∧ min(q).Time <= time do
                      { send go[min(q).Id](); deletemin(q) }
    end case
  forever
```

Here *PrioQueue*< T > is a priority queue of elements of type T with standard operations *insert*, *min*, and *deletemin*. Tuples are assumed to be ordered lexicographically starting with the first component (here the *Time* component).

7. Solution for Andrews Ex. 7.16

Here, the problem is solved using a single *probe* sent along a ring of processes. In the probe, one of the processes gives a proposal for the least common value. If a process can agree, it passes on the probe, otherwise it makes a new proposal. When a proposal has traversed the ring, a *commitment* message (Done) is sent around.

```
type Id = 1..n;
type Val = integer;
type Op = Probe(Id, Val) | Done

chan in[Id] : Op;

process P[i : Id] =
  var values : set of Val := ...;
      op : Op;
      lcv : Val;                                          — Least Common Value
  lcv := smallest v ∈ values;
  if i = 1 then send in[2](Probe(i, lcv));
  repeat
    receive in[i](op);
    case op :
      Probe(id : Id, val : Val): if id ≠ i then
                                   { lcv := smallest v ∈ values such that v ≥ val;
                                     if lcv > val then { id := i;  val := lcv};
                                     send in[i mod n + 1](Probe(id, val) }
                                 else
                                   { send in[(i mod n + 1](Done);
                                     receive in(op); }

      Done:                      send in[i mod n + 1](Done);
    end case
  until op = Done;
```

When all processes have left the **repeat**-loop, they all have the correct value of *lcv* (assuming it exists) and the channels are empty.

The problem can be solved in many other ways, e.g. by passing several probes simultaneously, or using centralized or symmetric communication schemes.