I.2**.** Final value: $2 \le x \le 200$
The upperbound is obtained by letting the programs run sequentially. For the lowerbound (and everything in between) consider the following sequence of actions.

r := x		{ *r = 0* }
s := x; s := s+1; x := s; j := j+1;	(99 times)	{ $j = 99 \wedge r = 0$}
r := r+1; x := r; i := i+1;		{ $x = 1 \wedge j = 99$ }
s := x		{ $s = 1 \wedge j = 99$ }
r := x; r := r+1; x := r; i := i+1;	(99 times)	{ $s = 1 \wedge j = 99$ }
s := s+1; x := s; j := j+1		{ *x = 2* }

A.1a. The specification reduces to *N* synchronization requirements.
#X(i+1) $\le$ *#X(i)*, for $0 \le i < N-1$
$\#X(0) \le \#X(N-1)+1$
We associate *N* semaphores with these requirements, *s[0..N-1].* Initial values are all 0 except semaphore *N-1* which is initially 1. Action *X(i)* occurs always in two equations, once on the left and once on the right of the inequality.
for $0 < i < N$: $X(i) \rightarrow P(s[i-1]); X(i); V(s[i])$
for *i=0:* $X(0) \rightarrow P(s[N-1]); X(0); V(s[0])$
The synchronization requirement already guarantees exclusion.

A deadlock would be possible only if all processes are blocked. This would mean that all semaphores are 0 (because of *S3*). Topology gives that the total number of *P*-operations equals the total number of *V*-operations in such a state. Using the semaphore properties:
0
= { deadlocked state: all processes trying a *P* }
$(\Sigma i: 0 \le i < N: \underline{c}P(s[i]) - \underline{c}V(s[i]))$
= { semaphore property: *S1* }
$(\Sigma i: 0 \le i < N: s_0([i]-s[i]))$
= { *S3*: blocked implies *s[i]=0*; given: sum of initial values is 1 }
1
In other words, assuming a deadlock leads to a contradiction.

Fairness is no issue. This completes the solution.

A.1b. In this case the equation
$\#X(0) \le \#X(N-1)+1$
and corresponding semaphore drops out. The requirements no longer guarantees exclusion, hence an extra semaphore m, initially 1, is introduced for that purpose.
for $0 < i < N-1$: $X(i) \rightarrow P(s[i-1]); P(m); X(i); V(m); V(s[i])$
for *i=0:* $X(0) \rightarrow P(m); X(0); V(m); V(s[0])$
for *i=N-1:* $X(N-1) \rightarrow P(s[N-2]); P(m); X(N-1); V(m)$
Notice that *P(m)* and *V(m)* are placed around *X(i).* Absence of deadlock with respect to *m* now follows from the fact that the critical sections (*X(i)*) terminate. Swapping the order of the *P* operations (*P* on *s* with *P* on *m*) leads to the danger of deadlock. As is often the case, such a deadlock does not always happen; it is just possible.

Deadlock analysis and fairness are similar as in part **a**.

A.2. The problem statement implies two synchronization conditions:

$\#A1 \leq 2\#A0 \leq \#A1+2$

We introduce two semaphores, *s* initially 0 and *t* initially 2 for the respective inequalities. Action synchronization gives us:

A0 $\rightarrow$ *P(t); P(t); A0; V(s); V(s)*
A1 $\rightarrow$ *P(s); A1; V(t)*

(i.e., the left-hand-sides should be replaced systematically by the right-hand-sides).

Exclusion of *A0* is guaranteed but exclusion of *A1* is not: two processes may call *A1* and pass semaphores *s*. In order to obtain exclusion, we need another semaphore, *m* say, initally 1.

A1 $\rightarrow$ *P(s); P(m); A1; V(m); V(t)*

We place the *P(m)-V(m)* as close to *A1* as possible in order to obtain a terminating critical section (viz., *A1*). Then we know that the exclusion does not *introduce* deadlock.

Two different processes may call *A0*. Although we know that exclusion is guaranteed, a deadlock is still possible: both processes may execute a single *P(t)* and then become blocked. From this we see that actually the two *P(t)* operations should be indivisible. Therefore, we introduce one more semaphore n, initially 1 to achieve this.

A0 $\rightarrow$ *P(n); P(t); P(t); V(n); A0; V(s); V(s)*

Finally, the solution is as fair as the semaphores are.

A.3. From the program text we obtain two properties.

I2: $x = 2 \cdot \#A - \underline{c}C$
I3: $y = 2 \cdot \#D - \underline{c}B$

We rephrase *I0* and *I1* as follows.

I0: $\#B \leq 2 \cdot \#D$
I1: $2 \cdot \#A \leq \#C+10$

Introduce semaphore *s*, initially 0, for *I0* and *t*, initially 10 for *I1*. Do the following replacements according to the principle of action synchronization.

$A \rightarrow P(t)^2; A$ $\quad$ $B \rightarrow P(s); B$
$C \rightarrow C; V(t)$ $\quad$ $D \rightarrow D; V(s)^2$

while *true* **do** *P(t); P(t);* ***A****: x := x+2* **od**

while *true* **do** *P(s);* ***B****: y := y-1* **od**

while *true* **do** ***C****: x := x-1; V(t);* ***D****: y := y+2; V(s); V(s)* **od**

There can be no deadlock since the third program produces *V*-operations on both semaphores, without limit. Fairness is no issue.

It is not easy to say which restrictions have danger of deadlock. Certainly restrictions that would cause blocking of the third component are dangerous. This means limiting *y* from above and *x* from below. Putting a large factor in front of it, like $0 \leq 10 \cdot x - 5 \cdot y$, ensures blocking in a few steps.

B.1. We assume the implementation with the circular array. We can imagine this bounded buffer as having two “sides” (called *ports*). In case of sharing a “side” of the buffer between several

processes, this “side” must be used under exclusion. In case of two producers, for example, this can be implemented by

P(n); Send (b, ...); V(n);

where *n* is a fresh semaphore, initially 1, that is associated with the *Send* operation on this particular buffer. If we also have several consumers we need a similar adaptation for the Receive operation *but with a different semaphore* (otherwise, a deadlock could result, check this).

Alternatively, we can put the semaphores within the implementation of the buffer as before. Notice that by having separate semaphores for consumers and producers, there is no mutual influence of consumers and producers with respect to accessing the queue.

If we use an implementation with a general queue in it for which exclusion is provided we already have a multi-producer - multi-consumer solution.

B.2. Assuming that we have safe solution for multiple producers and consumers (see B.1) we can implement this as follows.

1st consumer: *Receive(b, ...); Receive(b, ...); Receive(b,...)*
2nd consumer: *Receive(b, ...); Receive(b, ...); Receive(b,...); Receive(b,...)*

This, however, has the disadvantage that when there are four items available, both consumers may take two and then block. Therefore, the receives should be critical sections which can be implemented using a fresh semaphore *m*. Then, there is still the problem that the 2nd consumer passes *P(m)* while there are only three items available. A case in which this might lead to a deadlock is when there is some sort of a cycle in the dependencies of the processes. The simplest case is when a producer is also a consumer. This last problem cannot be solved with the techniques we discussed until now. The solution is along the lines of *condition synchronization*.

In general we would like to have a more general version of *Receive* (and *Send*) that supports receiving *n* items by integrating the repetition and the exclusion within the implementation of *Receive* like the one to the right.

```
proc Receive (var b: buffer; var y: array of elem; n: int) =

|[  var i: int;
    with b,q do
        P(m); i := 0;
        while i ≠ n do
          P(t); y[i] := b[r]; r := (r+1) mod N; i := i+1; V(s)
        od;
        V(m)
    od
]|
```

The sketched approach works for any implementation, also for a circular buffer of size 2. Other solutions might not have this property. For example, if we take the *P(t)* outside the repetition by performing *n P(t)* operations beforehand: the buffer might not be able to store *n* items!

For the case of a size 1 buffer we have a single variable *b*. The repetition remains but the need for semaphore *m* disappears as the synchronizing semaphores already establish exclusive acces.

B.3. The *N* producers share a buffer which can be of any positive size. The *Send* operation must provide exclusion among the producers. Let $Send_i$ denote the Send action of producer *i* and $Receive_i$ a *Receive* action in which the consumer consumed an item from the buffer that

was written by producer i. After a $Send_i$, producer *i* must excecute some synchronization action $Sync_i$ such that $\#Sync_i \leq \#Receive_i$
This gives us an array *x* of *N* semaphores, one per producer *i*, all initially 0. Action $Sync_i$ (which can itself be "*skip*") is preceded by *P(x[i]);* action $Receive_i$ is followed by *V(x[i]).* In order for the consumer to be able to retrieve the identity of the producer, this identity should be stored together with the item. We obtain the following.

Producer(i): P(m); Send (b, elem, i); V(m); P(x[i])
Consumer: Receive(b, elem, i); V(x[i])

(Here the *Send/Receive* and the buffer structure must be adapted to store integer *i* as well.) .
Notice that in the current solution, releasing the buffer (by *V(m)*) before waiting on the consumer results in several producers being active.
It is possible to integrate the solution with, for example, a circular buffer by associating a semaphore with each element in the buffer rather than with a process. In this way a dependence on *N* is avoided. Writing this is left to the reader.
Specializing this last idea to the case of a one-place-buffer gives

Producer(i): P(s); b := elem; V(t); P(x)
Consumer: P(t); elem := b; V(s); V(x)

V.2 The implementation is obtained by straightforwardly using the critical section structure as follows.

```
type GenSem = record m: Mutex; c: Condition; val: int end;

proc P (var  s: GenSem) =
|[ with s
   do while val = 0 do Wait (c) od;
        val := val-1;
        V(m)
   od
]|

proc V (var s: GenSem) =
|[ with s do P(m); s := s+1; Signal (c); V(m) od ]|
```

Notice that the data structure must be properly initialized before it is used (also in V.1, of course).

D.3. The effect of buffering at the side of *Split* is that *Split* does not really take decisions based on work that can be performed by one of the two processes. A rather even distribution would be the result. If the input stream to the system stops for a while and if the two processes run at different speeds the system might get into a state in which one of the *f*-processes is idle while the other one has a full buffer. Also, for the pipeline a completely skewed distribution would result.

On the other hand, buffering at the side of *Combine* yields that the *f*-processes do not become blocked on output. This may yield a better utilization of processors.

Adding the request channels double the ports for *Split*: each input is preceded by an output; instead of waiting for an output to become ready, *Split* now waits for a request. Also the implementation of *f* (which we did not write because it's trivial) changes to take this into account.

```
P_f (var c?, rc!, d!: port) =
|[  var x: some type;
    while true do
        send(rc, --);
        { request input }
        receive (c, x);
        send (d, f(x))
    od
]|
```

```
P_Split (var c?, rc!, d!, rd?, e!, re?: port) =
|[  var x: some type;
    while true do
        await (¬(empty(rd) ∧ empty(re)));
        if ¬empty(rd) then
            receive (rd,--); { accept request }
            send (rc,--); receive (c, x);
                                { request and get input }
            send (d, x)       { send reply }
        fi;
        if ¬(empty(re) then
            receive (re,--);
            send (rc,--); receive (c, x);
            send (e, x)
        fi;
    od
]|
```

We use here "—" to denote the communication of an arbitrary value.

Notice that the behavior of this program is that the entire system does not accept an input unless there is a request for it. As a result there is no buffering at all. This might be the other extreme. Usually a small buffering is optimal.

D.4. A passive buffer has an input port and an output port and waits until one of the ports becomes active before it performs a communication. It does not accept an input when it is full and no output when it is empty. The context of such a buffer should naturally be a system that supports just synchronous communication.

If testing on output is not possible a request channel can be used as in the previous exercise.

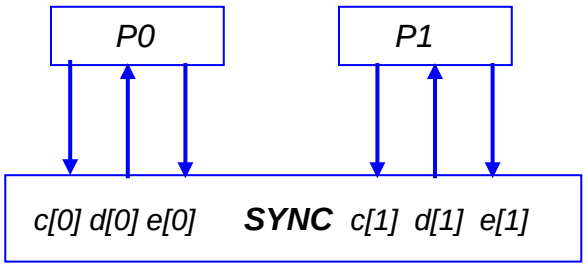
```
P_PassiveBuffer (var c?, d!: port) =
|[  var b: array of some type;
        r,w,c: int;
    r := 0; w := 0; c := 0;
    while true do
        await ((¬empty(c) ∧ c ≠ N) ∨ (¬full(d) ∧ c ≠ 0));
        if ¬empty(c) ∧ c ≠ N then
            receive (c,b[w]); w := (w+1) mod N; c := c+1
        fi;
        if ¬full(d) ∧ c ≠ 0 then
            send (d,b[r]); r := (r+1) mod  N; c := c+1
        fi
    od
]|
```

In order to make an active buffer we must at least have two concurrent processes (this is implied by the activity at both sides). We can do this by having a consumer/procucer implementation and a bounded buffer within this buffer process. Another way is as follows: for *N=1* we have a one-place-buffer (opb) which is a simple copy input to output. For *N=2* we connect two opb's and for *N>2* we connect two opb's to a passive buffer of size *N-2*.

D.5. There are three major aspects to the design of this system:

- Decide upon the interconnection pattern, the channels between the arbiter and the two processes.
- Design the protocol, the sequence of communication actions and communicated values that make up reserving and releasing the resources.
- Design the arbiter in accordance with these two.

There is no single best solution. We present one and comment on it.



We propose three channels between a process and *SYNC*. Channel *c* is used to request for a certain number of resources (communicated as a value), channel *d* is used by *SYNC* to acknowledge this request and finally, channel *e* is used to release the resources. The protocol in *P0* to use *k* resources is as follows.

send(c[0],k); receive(d[0],-); "use the resources"; *send(e[0],k)*

We use symbol "-" again to denote an arbitrary value. By this choice we decide that *SYNC* does not need to remember the reserved resources but that *P0* will tell this upon releasing them. This admits in principle something like

send(c[0],k0); receive(d[0],-); "use *k0*"; *send(c[0],k1); receive(d[0],-);*
"use *k0+k1*"; *send(e[0],k0+k1)*

in which resources are claimed in two batches.

Process *SYNC* uses variable *free* to record the number of available resources. We must maintain invariant

$0 \leq free \leq K$

Process *SYNC* will maintain the first inequality; the second also depends on correct behavior of *P0* and *P1* (*SYNC* will not be "suspicious"). *SYNC* will be waiting to accept a communication from the *c* and the *e* channels. Its response is as follows:

- a request via *c[i]:* if *free* is large enough, acknowledge it through *d[i]* and adapt *free*; if not, store the request.
- a value via *e[i]:* increase *free*; if there is a request pending, deal with it as above.

This is implemented in the program on the right though in a slightly different order. The *await* statement terminates when there is one communication pending. Then at most one communication is performed with each of the two processes. A request is stored in array *request*. A value of 0 here means: no request pending. Procedure *GiveResources* deals with pending requests.

```
P_SYNC (var c?, d!, e?: array [0..1] of port) =
|[ var free: int;
       request: array [0..1] of int;

   proc Communicate (i: int) =
   |[ var x: int;
      if ¬empty(c[i]) then receive(c[i],request[i])
      else if ¬empty(e[i]) then receive(e[i],x); free := free+x fi
      fi
   ]|;

   proc GiveResources (i: int) =
   |[ if 0 < request[i] ≤ free then
         free := free-request[i]; send(d[i],-);
         request[i] := 0
     fi
    ]|;

   { MAIN }

   free := K; request[0] := 0; request[1] := 0;

   while true do
        await (¬ (empty(c[0]) ∧ empty(c[1]) ∧
                  empty(e[0]) ∧ empty(e[1])));
        Communicate (0); Communicate (1);
        GiveResources(0); GiveResources(1)
   od
]|
```

For the fairness analysis, first assume that communication is synchronous. One might think that the fixed order gives some potential unfairness. However, because in *Communicate* we only deal with one communication at a time it is impossible that a process is given two turns while the partner could have had a turn as well. More precisely, the synchronous communication will result in a communication via *e* in one turn and a subsequent communication via *c* can happen only a next turn. Any pending request of the other partner is handled in between.

It is different when there is buffering. Then it may be the case that a new communication via *c* is received before the previous communication along *e.* Though this is not incorrect it may destroy the fairness. Showing this through an example is left to the reader. If we want to avoid this behavior we may introduce explicit state variables. This is also left as an exercise.

If we would like to add the identity of the resources to the communication, we may use a boolean array for that purpose. *SYNC* records the free resources in such an array and communicates the ones that are assigned to a process through such an array as well.

As a final remark, instead of using three channels we can design a system with only one synchronous channel. The protocol then becomes

send(c,k); send(c,-); “use resources”; *send(c,k)* (or *send(c,-)*)

In case of synchronous communication, the second communication blocks until *SYNC* accepts it. In this way we can use an output to give an acknowledge. The corresponding implementation of *SYNC* is more complicated since more state information needs to be recorded.