Improvement of parallelism in a finite buffer sharing policy

R. Devillers and G. Louchard

Laboratoire d'Informatique Théorique, Faculté des Sciences, Université Libre de Bruxelles, Avenue F.-D. Roosevelt, 50, 1050 Bruxelles, Belgium.

When parallel processes are linked in producer-consumer pairs and share a finite buffer where every portion is accessible to each process, it appears that a slow consumer may considerably delay the entire system.

Using conditional critical sections, Dijkstra has proposed to reserve for each producer-consumer pair the adequate number of portions for a normal working and to dedicate the rest of the buffer to absorb the production peaks of the various pairs. L. W. Cooprider *et al.* then developed solutions which avoid systematic inspection and only use the now classical synchronisation primitives *P* and *V*.

The present paper is devoted to the elaboration of solutions which improve parallelism and, when useful, discharge processes of administrative tasks.

We have used and compared four synchronising methods to this aim: conditional critical sections, semaphores, path expressions and monitors.

(Received June 1974)

1. Introduction

When parallel processes are linked in producer-consumer pairs and share a finite buffer where every portion is accessible to each process, it appears that a slow consumer may considerably delay the entire system; in extreme cases, when a consumer is blocked, the messages generated by the associate producer will invade the whole buffer and will block the system. To avoid such messy behaviour, Dijkstra proposed (1972) to reserve for each producer-consumer pair the adequate number of portions for a normal working and to dedicate the rest of the buffer to absorb the production peaks of the various pairs. He also proposed a simple algorithm which realises this policy with the aid of the conditional critical section technique, as developed by Hoare (1971) and Hansen (1972a; 1972b; 1973). But, besides being difficult to implement, the use of conditional critical sections implies a systematic examination of every blocked process at each termination of a critical section while

(*a*) when a producer has generated a message, only its associate consumer has to be activated,

(*b*) when a consumer has used a message, at most one process (which will often be the associate producer) has to be activated.

Courtois and Heymans (1973) and Cooprider *et al.* (1973) then developed solutions which avoid this systematic inspection and only use the now classical synchronisation primitives *P* and *V*.

With the use of a single semaphore for the mutual exclusions, this solution still limits the parallelism of processes which belong to different pairs*.

The present paper is devoted to the elaboration of solutions which improve parallelism and, when useful, discharge processes of administrative work.

We have used and compared four synchronising methods to this aim: conditional critical sections, semaphores, path expressions and monitors. Starting from a simple solution, written with critical sections, we have introduced several refinements with the help of these three other methods.

2. Hypotheses

n groups of parallel processes are considered; each group is composed of a family of processes producing messages and a family of processes receiving messages; the consumers of a group may only treat the messages created by the producers of the same group.

For their communications, these processes dispose of a buffer containing N frames; each frame may receive a message from the producers.

It will be supposed that $N > n$

and that a number $N_i\,(\geqslant 1)$ of frames in the buffer are reserved for the group i's messages ($i = 1, 2, \ldots, n$); the identity of these frames is not defined yet: it will be able to vary with time.

These quantities N_i are chosen so that

1. They grant good communications in each group in periods of normal traffic
2. $C = N - \sum_i N_i > 0$ [if $N = \sum_i N_i$, the groups are totally disjointed and independent: one can subdivide the buffer in n fixed sub-buffers and use, for each, Habermann's algorithm (1972)]; the C remaining frames will be used to receive the supplementary messages of the various groups (when there is a production peak, immobilisation of consumers, . . .).

3. Solution with conditional critical sections

First, we have tried to develop a solution with adequate characteristics by using Hoare's structured primitives of synchronisation.

With slight generalisations of the original primitives, we can write the following acceptable solution.

3.1. *Solution*

Using a mixture of Hoare (1971) and ALGOL notation, we get

```
Declarations:
  integer array N[1:n] (such that N[i] = N_i);
  integer f (initially C);
  integer array m[1:n] (initially 0);
  {resource array m[1:n]; producers||consumers};
  {resource f; producers||consumers};

Consumer of group i:
  while true do
  begin
  with m[i] when m[i] > 0 do
    begin pick up a full frame of type i;
      collect the message from the frame; release the frame;
```

*This drawback may in fact be made rather slight, from a pure practical point of view, if the processes are such that they stay just a very little moment in the various critical sections.

```
        $m[i] := m[i] - 1$;
        **if** $m[i] \geqslant N[i]$ **then with** $f$ **do** $f := f + 1$
      **end**;
      *receipt of the message*
    **end**;
```

Producer of group i:

```
    while true do
    begin
    production of a message;
    with $m[i]$ and $f$ when $m[i] < N[i]$ or $f > 0$ do
      begin if $m[i] \geqslant N[i]$ then $f := f - 1$;
        $m[i] := m[i] + 1$;
        pick up a free frame;
        place the message in the frame; attach the frame
      end
    end;
```

Comments

(*a*) $m[i]$ gives the total number of existing group i's messages, f is the total number of free frames among the C common frames;

(*b*) we have no clear separation between critical sections in the producer, which has to block f even when it is raised by the associated consumer (our producer is in fact identical to the send procedure used by Hansen (1973)); the producer can then uselessly delay other producers;

(*c*) it is not possible to separate collection (or extraction) of the message from other operations;

(*d*) we have used a 'parallel with' in the producer's code: **with** $m[i]$ **and** f **when** ...; this case was in fact not included in Hoare's proposition; it seems to us that this extension would not be much more difficult to implement than the original primitives;

(*e*) the implementation of these primitives may however strongly decrease the apparent parallelism of the solution; note also that care must be taken to avoid deadlocks in the implementation of the 'parallel with';

(*f*) Hansen (1973) solves the same problem with a single critical region. We have tried to keep as much separation as possible between groups. See however (*c*);

(*g*) picking, attaching and releasing are not precisely defined; appropriate lists should be constructed for practical and efficient use.

3.2. *Adequacy of the solutions*

We shall not develop detailed proofs to show that the proposed solution 'fits the considered problem'; even in the simplest cases, complete demonstrations of the well-formed synchronisation of parallel processes are extremely tedious and intricate (see for example Habermann, 1972a; 1972b). That is why we shall only extract some invariant properties and general considerations, showing that the algorithms work correctly (note however point (*e*) of our comments).

1. If the various producers-consumers are blocked outside their critical sections, it appears that:

$$f + \sum_i (m[i] - N[i])_+ = C^*$$

indeed:

(*a*) it is true initially

(*b*) each time a producer passes through its critical section

(i) either $m[i] < N[i]$ and then

$m[i] \leftarrow m[i] + 1 \leqslant N[i]$

f is left unchanged

(ii) or $m[i] \geqslant N[i]$ and $f > 0$ and then

$m[i] \leftarrow m[i] + 1 > N[i]$

$f \leftarrow f - 1$

(*c*) each time a consumer passes through its critical section

(i) either $m[i] > N[i]$ and then

$m[i] \leftarrow m[i] - 1 \geqslant N[i]$

$f \leftarrow f + 1$

(ii) or $m[i] \leqslant N[i]$ and then

$m[i] \leftarrow m[i] - 1 < N[i]$

f is left unchanged.

2. Under the same conditions, if λ is the number of free frames in the buffer

$$\lambda = f + \sum_i (N[i] - m[i])_+$$

the proof is similar to the preceding one.

3. f never becomes negative.
Indeed f is only decreased (by 1) when $m[i] \geqslant N[i]$ in the producer critical section and, in this case, the critical section can only be entered when $f > 0$. Consequently,

(*a*) relation 2 implies that $\lambda \geqslant 0$, and thus that it will never be granted more frames than are existing

(*b*) relation 1 implies that $\sum_i (m[i] - N[i])_+ \leqslant C$, i.e. that the overflows will never invade the proper reserve of the various groups.

4. $m[i]$ never becomes negative.
Indeed, the only operation decreasing $m[i]$ (by 1) appears in the consumers of group i and their critical section is only entered when $m[i] > 0$.
It follows also that consumers of group i will not all wait forever if a producer of group i has created a message.

5. Producers of group i will not all wait forever if $m[i] < N[i]$: one of them will eventually enter the critical section (in fact consumers of group i can get control of the critical section a finite number of times only and can only decrease $m[i]$).
Producers will not all wait forever if $f > 0$; one of them will enter the critical section (meanwhile, the consumers can only increase f). If a consumer of group i frees a frame, two cases may occur

(*a*) if $m[i] < N[i]$ (after decreasing $m[i]$), then, if producers of group i are waiting, one of them will eventually enter its critical section

(*b*) if $m[i] \geqslant N[i]$, then f becomes positive and, if producers are waiting, one of them will eventually enter its critical section.

6. Continuous exchange between producers and consumers of group i in their reserve N_i will never eternally delay exchanges in other groups: one of them will eventually get control of the f-critical section.

4. Solution with semaphores

To improve our first conditional critical section solution (see our comments in Section 3), we have tried to introduce several refinements. Our first tool to develop these refinements will be the now classical semaphore. We shall proceed in a structured way, starting from the solution in Section 3.

1. To improve parallelism and to separate free frames administration, we have first introduced a 'distributor' process that has to assign the C common frames to the producers of the groups having emptied their own reserve.
This distributor is activated both by producers waiting for a free frame and by consumers freeing common frames; it

*We shall use the notation x_+ for $\max\{x, 0\}$.

moreover allows the consumers to discharge tasks that can be run separately and parallel, without suspending the work of these processes.

Distributor activation is done by producers with the help of the semaphore *distr*. Indication of common frames being released is done with the semaphore *f*, which still represents the total number of free frames among the *C* common frames.

2. Administration of free frames among the N_i frames reserved for group *i* is done immediately by consumers of group *i* without involving other groups or common free frames. Transfer of critical section control to a producer of group *i* is done at the same time.
3. The following semaphores will be used.

mutex[*i*]:	to protect group *i*'s critical sections
cons[*i*]:	to inform a consumer that a type *i* message has been created
prod[*j*]:	to inform producer *j* that a free frame has been reserved for it.

4. A waiting queue for producers must be introduced, the detailed treatment of which will be given later.

A first sketch of the solution is now developed, starting from the first solution in Section 3.

```
Declarations:
  integer array $m[1:n]$ (initially 0),
    $N[1:n]$ (such that $N[i] = N_i$);
  semaphore $f$(initially $C$), distr(initially 0);
  semaphore array mutex$[1:n]$ (initially 1),
                   cons$[1:n]$, prod$[1:np]$ (initially 0; np is the
                                       total number of
                                       producers);
```

```
Consumer of group i:
  while true do
  begin
  P(cons[i]);
  P(mutex[i]);
  pick up a full frame of type i;
  collect the message from the frame; release the frame;
  $m[i] := m[i] - 1$;
  if $m[i] \geqslant N[i]$ then begin V(f); V(mutex[i]) end
              else if $m[i] = N[i] - 1 \wedge$ producer j of
                    group i waiting
                   then begin remove j from the waiting
                              queue;
                              V(prod[j])
                        end
                   else  V(mutex[i]);
  consumption of the message
  end;
```

```
Producer j, of group i:
  while true do
  begin
  production of a message;
  P(mutex[i]);
    if $m[i] \geqslant N[i]$ then begin insert j in the waiting queue;
                                V(mutex[i]; V(distr);
                                P(prod[j])
                          end;
  $m[i] := m[i] + 1$;
  pick up a free frame;
  place the message in the frame; attach the frame;
  V(mutex[i]); V(cons[i])
  end;
```

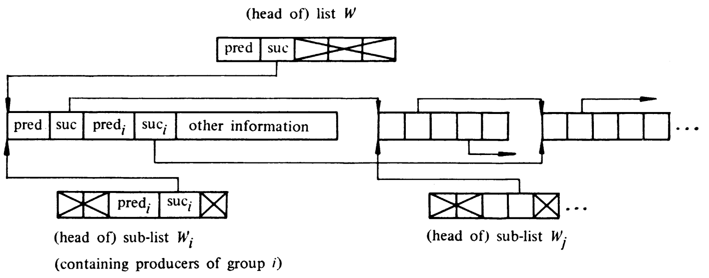

Fig. 1

```
Distributor:
  integer i, j;
    while true do
    begin
    P(distr); P(f);
    j := first producer of the waiting queue;
    if $j \neq 0$ then begin i := group of producer j;
          P(mutex[i]);
          if j still is in the waiting queue*
            then begin remove j; V(prod[j]) end
            else begin V(mutex[i]); V(f) end
        end
      else V(f)
    end;
```

Picking, attaching and releasing may now be precisely and efficiently defined. We have associated to each group *i* a list *P*[*i*] of the frames used by this group (i.e. produced but not yet received messages); the free frames are gathered in another list: *L* (protected by semaphore *mutexL*). Producers waiting for a free frame will enter a list *W* (protected by semaphore *mutexW*).

Consumers waiting for a message will automatically be placed in queues controlled by private semaphores.

We don't define how the lists are handled (picking out may be FIFO, LIFO, random, . . .); we suppose only that the following procedures are present:

insert(*j*):	that inserts producer *j* in the *W* queue (at the end if a FIFO policy is wanted)
remove(*j*):	that removes producer *j* out of *W*
first:	function that identifies the first (following the adopted policy) producer in *W* its value will be 0 if *W* is void
firstg(*i*):	function that identifies the first producer of group *i* in *W* it will be 0 if *W* does not contain producers of this group

and that we can 'pick up' (or 'attach') a frame from (or to) the *L* and *P*[*i*] queues.

To make the protection of *W* easier and to speed up its consultation, we will moreover suppose that procedures *first* and *firstg* may work by straight examination, without having to follow the whole queue; such a purpose may be obtained for instance with the use of a nested list structure as shown in **Fig. 1.**

So, examination of sub-list W_i can be done without bothering other *W* sub-lists. The next step in solution then appears as:

```
Declarations:
  integer array $m[1:n]$ (initially 0),
    $N[1:n]$ (such that $N[i] = N_i$);
  semaphore mutexL, mutexW(initially 1), $f$(initially $C$),
    distr(initially 0);
```

*This test is necessary because a consumer may have freed producer *j* between *j* computation and *mutex*[*i*] blocking.

```
semaphore array mutex[1:n] (initially 1),
                cons[1:n], prod[1:np] (initially 0);
queue L(initially: the whole buffer), W(initially void);
queue array P[1:n] (initially void);
```

Consumer of group i:

```
while true do
begin
P(cons[i]);
P(mutex[i]);
pick up a full frame from P[i];
collect the message from the frame;
P(mutexL); attach the frame to L; V(mutexL);
m[i] := m[i] - 1;                                        (1)
if m[i] ≥ N[i] then
      begin V(f); V(mutex[i]) end
   else if m[i] = N|i| - 1 ∧ firstg(i) ≠ 0
      then begin integer j; j := firstg(i);
            P(mutexW); remove(j); V(mutexW);
            V(prod[j])
            end
      else V(mutex[i]);
consumption of the message
end;
```

Producer j, of group i:

```
while true do
begin
production of a message;
P(mutex[i]);
   if m[i] ≥ N[i] then
      begin P(mutexW); insert(j);
      V(mutexW);
         V(mutex[i]; V(distr);
         P(prod[j])
      end;
m[i] := m[i] + 1;                                        (2)
P(mutexL); pick up a free frame from L; V(mutexL);
place the message in the frame;
attach the frame to P[i];
V(mutex[i]); V(cons[i])
end;
```

Distributor:

```
integer i, j;
   while true do
   begin
   P(distr); P(f);
   P(mutexW); j := first; V(mutexW);
   if j ≠ 0 then
      begin i := group of producer j;
         P(mutex[i]);
         if j = firstg(i)
            then begin P(mutexW); remove(j); V(mutexW);
                       V(prod[j])
                 end
            else begin V(mutex[i]); V(f) end
      end
      else V(f)
   end;
```

If picking up or attaching a frame, placing or collecting the message are slow operations, we can improve speed and parallelism with the aid of another semaphore array, *mutexP*[1:*n*], by extracting some manipulations out of the *mutex*[*i*]-critical sections.

Our final solution is deduced from the preceding one as follows.

4.1. *Solution*

Make the following alterations in the preceding solution:

In the declarations, add:

```
semaphore array mutexP[1:n] (initially 1);
```

In the consumer, before line 1, read:

```
P(cons[i]);
P(mutexP[i]); pick up a full frame from P[i]; V(mutexP[i]);
collect the message from the frame;
P(mutexL); attach the frame to L; V(mutexL);
P(mutex[i]);
```

In the producer, after line 2, read:

```
V(mutex[i]);
P(mutexL); pick up a free frame from L; V(mutexL);
place the message in the frame;
P(mutexP[i]); attach the frame to P[i]; V(mutexP[i]);
V(cons[i])
end;
```

The instructions modifying *m*[*i*] have been inserted at the earliest possible place for the producers and at the latest possible place for the consumers to insure correct communications within each group.

4.2. *Adequacy of the solution*

As in 3.2, to show the adequacy of that solution, we shall extract only some invariant properties and general considerations of the solution. They are mostly the same as in 3.2 but the demonstrations have to be adapted, due to the *P*, *V*-expression of the synchronisation and to the distributor introduction.

1. If the various producers-consumers are blocked outside their *mutex*[*i*]-section (going from the *P*(*mutex*[*i*]) to the last *V*(*mutex*[*i*]) and the distributor is blocked outside its *f*-section (going from the *P*(*f*) to the *V*(*f*)), it appears that:

$$f + \sum_i (m[i] - N[i])_+ = C$$

indeed

(*a*) it is true initially

(*b*) each time a producer passes through its *mutex*[*i*]-section

(i) either $m[i] < N[i]$ and then

$m[i] \leftarrow m[i] + 1 \leqslant N[i]$

f is left unchanged

(ii) or it is freed by an associate consumer, going through its *mutex*[*i*]-section and observing that $m[i] = N[i]$ and that there is a waiting producer of the same group; in this case, $m[i]$ and f are left unchanged

(iii) or it is freed by the distributor, going through its f-section, and then $f \leftarrow f - 1$

$$m[i] \leftarrow m[i] + 1 > N[i]$$

because there are no producers waiting if $m[i] < N[i]$

(*c*) each time a consumer passes through its *mutex*[*i*]-section

(i) either $m[i] > N[i]$ and then: $m[i] \leftarrow m[i] - 1$ and $f \leftarrow f + 1$

(ii) or $m[i] = N[i]$ and there is an associate producer waiting: see (*b*)(ii)

(iii) or $m[i] \leqslant N[i]$ and there are no associate producer waiting, then $m[i] \leftarrow m[i] - 1 < N[i]$ and f is left unchanged

(*d*) each time the distributor passes through its f-section

(i) either it wakes up a waiting producer: see (*b*)(iii)

(ii) or it does nothing: f and $m[i]$ are left unchanged.

2. Under the same conditions, if λ is the number of free frames in queue L:

$$\lambda = f + \sum_i (N[i] - m[i])_+$$

the proof is similar to the preceding one.

3. As f, being a semaphore, never becomes negative, conclusions from 3.2.3 are still valid.

4. $m[i] \geqslant 0$

   Indeed, the only operation decreasing $m[i]$ (by 1) appears in consumers of group i, which can be raised *only* by $cons[i]$. $cons[i]$ is itself increased in producers of group i *only* after the operation $m[i] := m[i] + 1$. Conclusions of 3.2.4 are still valid.

5. Producers of group i will not all wait forever if $m[i] < N[i]$: one of them will eventually enter the critical section $mutex[i]$ (in fact, consumers of group i can only use a finite number of these critical sections) and will jump over the test (consumers which would have worked meanwhile can only reduce $m[i]$); producers will not all wait forever if $f > 0$: the distributor will raise the first of the queue (and meanwhile, the consumers can only increase f). If a consumer of group i frees a frame, then three cases may occur

   (*a*) if $m[i] < N[i] - 1$, no producer of group i is waiting and the frame can be returned to the proper pool

   (*b*) if $m[i] = N[i] - 1$ and producers of group i are waiting, the first of the queue W_i will be raised directly (if not, the frame is returned to the proper pool)

   (*c*) if $m[i] \geqslant N[i]$ and producers are waiting, the first of the queue will be raised by the distributor, which is always alerted (with *distr*) by any waiting producer (if not, the frame is returned by the distributor to the common pool).

6. Continuous exchange between producers and consumers of group i in their reserve N_i will never delay exchanges in other groups (this type of exchange in group i is controlled through $mutex[i]$).

7. It may finally be pointed out that deadlocks are avoided by the hierarchy

$$\begin{tikzcd} mutex[i] \arrow[d] \\ mutex W \end{tikzcd}$$

8. It is possible to avoid the introduction of an additional process (the distributor) by inserting its procedures in the critical section of each of the producers-consumers.

   We can also increase the degree of parallelism for the processes in sub-critical situations (when there are few accesses to the common part of the buffer) by the introduction, for each group i, of a queue listing the free frames reserved for this group (which allows a reduction of the accesses to *mutexL*-critical sections).

   Unfortunately, the resulting solutions are rather intricate; so we shall not write them out here. Moreover, it may be pointed out that it should not be necessary, from a practical point of view, to search ultimate parallel solutions.

9. When f, the total number of common free frames, is nearly always positive, some time is lost in the producer when entering W uselessly. One can refine the solution in the following way:

   (*a*) introduce f as an integer, protected by semaphore *mutexf*

   (*b*) when $m[i] \geqslant N[i]$, the producer first checks if $f > 0$; if it is, the producer can proceed immediately, if not, the producer enters W and waits

   (*c*) the distributor is only alerted by a consumer (when $m[i] \geqslant N[i]$).

   We do not give all details here, adequacy can be shown as in the previous solution.

5. Solution with synchronisation paths

Our second tool to introduce refinements in the solution in Section 3 will be the synchronisation paths of Campbell and Habermann (1974):

1. The protection of critical sections is now done in various paths, one protecting the queue L (1)*, another protecting $m[i]$ (2) and a third one protecting $P[i]$ (3) (We have used the same lists and notations as in Section 4). Two other paths are necessary to introduce the adequate synchronisation: producer-consumer (4) (after a message production) and consumer-producer (5) (after liberation of a frame belonging either to the N_i frames of group i or to the common pool of C frames).

2. Procedures must be introduced

   (*a*) in the *i*-consumer for full frame pick-up (*cpick-i*), for frame releasing (*return*) and $m[i]$-decreasing (*sub*1-*i*)

   (*b*) in the *i*-producer for free frame pick-up (*detach*), frame attaching (*cattach-i*) and $m[i]$-increasing (*cadd*1-*i*)

3. To introduce separation of the two types of free frames, two null procedures (*signal* and *signal-i*) are used for synchronisation purposes.

Starting from our first solution in Section 3, we can now get a first approach of the solution:

```
Declaration:
  type frame; . . .
  endtype;
  type sharedpool;
    queue L (initially, the whole buffer);
    queue array P[1:n] (initially void);
    integer array m, N[1:n]
      (initially N[i] = N_i and m[i] = 0);
    path return, detach end;                    (1)
    path sub1-i, add1-i end;                    (2) ⎫
    path attach-i, pick-i end;                  (3) ⎪
    path {cattach-i; cpick-i} end;              (4) ⎬ i = 1, 2, . . ., n
    path {(signal, signal-i); caddl-i}              ⎪
      end;                                      (5) ⎭
    operations
    procedure cpick-i (fullframe); pick-i (fullframe);
    procedure pick-i (fullframe); fullframe := full frame of
              P[i];
    procedure cattach-i (fullframe); attach-i (fullframe);
    procedure attach-i (fullframe);
              P[i] := P[i] ∨ fullframe;
    procedure return (freeframe); L := L ∨ freeframe;
    procedure detach (freeframe); freeframe := free frame of L;
    procedure caddl-i; add1-i;
    procedure add1-i; m[i] := m[i] + 1;
    procedure sub1-i; begin m[i] := m[i]-1;
      if m[i] ⩾ N[i] then signal else signal-i end;
    procedure signal; null;
    procedure signal-i; null
  endtype;

Initialisation:
sharedpool B; frame fram; integer j, k;
for j := 1 step 1 until n do for k := 1 step 1 until N[j] do
   B. signal-j;
for k := 1 step 1 until C do B.signal;

Consumer of group i:
  while true do
  begin
  B.cpick-i(fram);
```

*The numbers refer to path numerotation in the solution.

```
    collect the message from fram;
    B.return(fram);
    B.sub1-i;
    consumption of the message
    end;
```

Producer of groupi:

```
    while true do
    begin
    production of a message;
    B.cadd1-i;
    B.detach(fram);
    place the message in fram;
    B.cattach − i(fram)
    end;
```

Comments:

1. In our solution, we have used restricted path expressions (with the exception of the last family, which is considered below) as suggested by Campbell and Habermann (1974) (a procedure name occurs only once in any path expression). Without this restriction, we could write a shorter solution, with less procedures. We can for instance use a single procedure for each of the pairs *cpick-i, pick-i; cattach-i, attach-i*; *cadd*1-*i*, *add*1-*i*.
2. The proof of validity of this solution is now shifted: correctness of consumer and producer is easy to verify but correctness of path expressions is more complicated, we don't write it here.
3. Implementation of *i*-indices in procedures is left undefined: it depends on the way the path expressions are implemented.
4. In the path family, *signal* appears *n* times: this family must be treated separately. Moreover, if an efficient use of the pool is wanted, it is strongly advised that *cadd*1-*i* 'absorbs' *signal-i*, instead of *signal*, if both are present, so as to leave as many free frames as possible in the common pool. We must in fact express the statement:
   if $m[i] \geqslant N[i]$ **then with** f **do** $f := f + 1$ of our critical section solution. This special interpretation of path expression can be translated in 'normal' paths, at the price of some complication; we could for instance proceed as follows (in practical life, an ad hoc implementation should be used):

5.1. *Solution*

The following refinements are introduced into the preceding solution:

(*a*) introduce two arrays: $w[i]$ giving the total number of waiting producers of group i and $fg[i]$ giving the total number of free frames among the N_i frames of group i

(*b*) reintroduce f giving the total number of common free frames

(*c*) in the producer code, insert a call to *B. test-i* before *B. cadd*1-*i*,

(*d*) replace, in the declaration of the sharedpool type, the last family of paths by the following procedures, paths and variables:

```
  integer array w[1:n] (initially 0), fg[1:n]
     (initially fg[i] = N_i);
  integer f(initially C);
  path add1f, distr end;                          (6)
  path{(eadd1f, add1w); cdistr} end;
  path{signal; cadd1f} end;
  path test-i, distr-i, test2-i end;   (7)    ⎫
  path{signal-i; cdistr-i} end;               ⎬ i = 1, . . ., n
  path{(sub1fg-i, sub1w-i); cadd1-i} end;     ⎭
  procedure cadd1f; add1f;
  procedure add1f; begin f := f + 1; eadd1f end;
  procedure eadd1f; null;
  procedure cdistr; distr;
  procedure distr; if f > 0 then begin j := 0;
                                    for j := j + 1 while
                                    f > 0 do test2-j
                                 end;
  procedure test2-i; while w[i] > 0 and f > 0 do
                        begin f := f − 1;
                            sub1w-i
                        end;
  procedure cdistr-i; distr-i;
  procedure distr-i; if w[i] = 0 then fg[i] := fg[i] + 1
     else sub1w-i;
  procedure sub1w-i; w[i] := w[i] − 1;
  procedure add1w(i); w[i] := w[i] + 1;
  procedure test-i; if fg[i] > 0 then sub1fg-i else add1w(i);
  procedure sub1fg-i; fg[i] := fg[i] − 1;
```

(*e*) and add the three processes:

```
  while true do cadd1f
  while true do cdistr-i
  while true do cdistr
```

Comments:

1. Path (6) protects f, path (7) protects $fg[i]$ and $w[i]$. Other paths introduce synchronisation.
2. We have used a procedure *distr* which simulates (with less efficiency) the distributor of the semaphore-solution; it introduces however an a priori priority among the various groups, which may be unwanted. (It can be released by some randomisation).
3. Deadlocks are avoided with the hierarchy

$$\begin{tikzcd} f \arrow[d] \\ fg[i],\ w[i] \end{tikzcd}$$

4. Again with unrestricted path expressions, less procedures are needed to write the solution.
5. Transfer of critical sections is not easy.
6. Proof of our addition is not easy; we do not give it here.

6. Solution with monitors

Our third tool to introduce refinements in the solution in Section 3 will be the monitors, as described by Hoare (1974). Notations are similar to Sections 4 and 5.

New variables will be:

1. $res[i]$ giving, when positive, the number of frames still reserved for group i, and, when negative, − (number of common frames used for group i)
2. Full and free$[i]$ are 'conditions' (see Hoare, 1974): the first one says to a consumer that a full frame is available, the other says to a producer that a free frame is available.

A solution can now be written as follows:

```
monitor private monitor;
begin integer m (initially 0); queue P (initially void);
  condition full;
  procedure pick (fullframe);
    begin if m = 0 then full.wait;
      m := m − 1;
      fullframe := full frame of P
    end;
  procedure attach (fullframe);
    begin P := P ∨ fullframe;
      m := m + 1;
      full.signal
    end
end;


monitor common monitor;
```

```
begin queue L (initially the whole buffer), W (initially void);
  integer f (initially C);
  integer array res[1:n] (initially, res[i] = $N_i$);
  condition array free[1:n];
  procedure return (freeframe, i);
    begin L := L ∨ freeframe;
      res[i] := res[i] + 1;
    if res[i] ⩽ 0 then
      begin integer j;
        j := first of W;
        if j ≠ 0 then free [groug of j].signal
                 else f := f + 1
      end
      else if res[i] = 1 then free[i].signal
    end;
  procedure detach (freeframe, i, currentproducer);
    begin if res[i] ⩽ 0 then
      begin if f > 0 then f := f − 1
        else begin insert in W (currentproducer);
                   free[i].wait;
                   remove from W (currentproducer)
             end
      end;
      res[i] := res[i] − 1;
      freeframe := free frame of L
    end
end;

private monitor array own[1:n];
common monitor B; frame fram;
Consumer of group i:
while true do
  begin own[i].pick(fram);
    collect the message from fram;
    B.return (fram, i);
    consumption of the message
  end;

Producer j of group i:
while true do
  begin production of a message;
    B.detach (fram, i, j);
    place the message in fram;
    own[i].attach (fram)
  end;
```

Comments:

1. Each group's full frames handling is protected by their own monitor. Free frames handling is done under common monitor *B* supervision; it seems difficult to improve parallelism for free frames without risking deadlocks.
2. Adequacy of the solution is rather easy to show but care must be taken with *wait* and *signal* operations peculiar behaviour; note also that consumer and producer procedures are nearly the same as in path expression solutions.
3. Producer continuation by immediate checking is of course in the same spirit as indicated in our remark in Section 4.2.9.
4. In *return* procedure, instruction: *free* [*group of j*].*signal* is sure to alert producer *j* and no other waiting on free-producer. Indeed 'if more than one program is waiting on a condition, we postulate that the *signal* operation will reactivate the longest waiting program' (note 3 on p. 550 of (Hoare, 1974)). Similarly, we do not need *firstg*(*i*) anymore, *free*[*i*] condition assumes the same role.
5. We must still use an explicit queue *W* for the waiting producers because a simple condition (*freeW*, for instance) could not insure proper alerting in group *i* when *res*[*i*] = 1.

7. Conclusions

If we try to extract the advantages and disadvantages of each type of synchronisation primitives, we may write the following remarks.

1. *Conditional critical regions*:
(*a*) these primitives lead to a rather nice solution, which is easy to verify,
(*b*) but, in order to obtain it, we needed some extensions of the original proposal and the solution is in fact less efficient than was hoped; moreover, we may mention the old problems of the implementation and of the (controlled) busy form of waiting which are attached to these primitives.

2. *P and V*:
(*a*) these primitives seem the most versatile and the most efficient ones,
(*b*) but they lead to rather complex solutions, which are not easy to verify.

3. *Synchronisation paths*:
(*a*) when all the specifications and initialisations are made, the resulting solution is usually the most easy to write and to verify,
(*b*) clear separation of synchronising specifications and problem operations is a good concept which will lead to better structured programs,
(*c*) but these specifications appear themselve rather complicated and delicate and a need for more general paths arises in the elaboration of the solution; some of the specifications seem sometimes rather artificial.

4. *Monitors*:
(*a*) the synchronising specifications are easy to write and check,
(*b*) the resulting solution is very simple; it is nearly the same as for synchronisation paths,
(*c*) again, separation of synchronising specifications and problem operations is a good concept; *wait* and *signal* operations are useful,
(*d*) processes parallelism is clearly restricted by the fact that no parallelism is possible inside each monitor. See also remarks on p. 557 of Hoare (1974).

Starting from the critical section structured primitives and introducing several step by step refinements with other primitives permits us to get more efficient and better structured solutions.

However such introduction is not direct and proofs must be adapted.

To be conclusive about the primitives comparison, more examples will have to be considered. We intend to follow this direction in another paper.

Acknowledgements

We are indebted to P. J. Courtois for helpful comments and discussions and for his communication before publication (Cooprider, Courtois, Heymans and Parnas, 1973).

We are also grateful to the referee for valuable suggestions concerning the structure of this paper.

References

DIJKSTRA, E. W. (1972). Information Streams sharing a finite Buffer, *Information Processing Letters*, Vol. 1, pp 179-180.
HOARE, C. A. R. (1971). Towards a Theory of parallel Programming, in International Seminar on Operating System Techniques, Belfast,

Northern Ireland, 1971; and *Operating Systems Techniques*, pp. 61-71, edited by C. A. R. Hoare and R. H. Perrot, Academic Press, New York, 1973.

HANSEN, P. B. (1972a). A comparison of two synchronizing Concepts, *Acta Informatica*, Vol. 1, pp. 190-199.

HANSEN, P. B. (1972b). Structured Multiprogramming, *CACM*, July 1972, Vol. 15, No. 7, pp. 574-578.

HANSEN, P. B. (1973). Concurrent Programming Concepts, *Computing Surveys*, December 1973, Vol. 5, No. 4, pp. 223-245.

COURTOIS, P. J., and HEYMANS, F. (1973). *Information Streams sharing a finite Buffer*; *another Implementation*, MBLE, Technical Note N85, January 1973.

COOPRIDER, L. W., COURTOIS, P. J., HEYMANS, F., and PARNAS, D. L. (1973). Information Streams sharing a finite Buffer: other Solutions, *Information Processing Letters*, Vol. 3, No. 1, July 1973.

HABERMANN, A. N. (1972a). Synchronization of communicating Processes, *CACM*, March 1972, Vol. 15, No. 3, pp. 171-176.

HABERMANN, A. N. (1972b). *On a Solution and a Generalization of the Cigarette Smokers' problem*, Carnegie-Mellon University Report, August 1972.

CAMPBELL, R. H., and HABERMANN, A. N (1974). *The Specification of Processes Synchronization by Path Expressions*, University of Newcastle Upon Tyne technical report no 55, January 1974.

HOARE, C. A. R. (1974). Monitors: An Operating System Structuring Concept, *CACM*, October 1974, Vol. 17, No. 10, pp. 549-557.

Book reviews

Structured Computer Organisation by A. S. Tanenbaum, 1976; 443 pages. (*Prentice Hall*, $18.50).

This text provides an admirably mature and illuminating study of computer architecture. The subject matter is uniformly well presented, but the feature of most significance is the overall approach to the subject. All too frequently, texts on computer architecture distinguish a level of architecture broadly equated to the demarcation between hardware and software, and concentrate on the implementation of this level, principally by hardware components. This text distinguishes a number of levels, each of which may be supported by, and may itself support, both hardware and software components. The result is a logical exposition of architecture viewed at different levels.

The particular levels on which attention is focussed are divided into two classes, namely, those which are implemented by interpretation and those implemented by translation. In the former category are the microprogramming level, at which hardware interprets sequences of microprogram steps, the conventional machine level, at which the microprogram interprets sequences of basic instructions, and the operating system level, where instructions are interpreted either by microprogram or by the operating system. Levels implemented by translation are typified by the assembly language level; the text does not extend the treatment to a consideration of translated high level source languages, but does discuss architectural features required to implement an interpreted machine level which would support the execution of high level source language statements. A final chapter considers various methods for implementing multi-level machines, including a most illuminating study of 'self virtualising machines', using the IBM VM/370 system as an illustration.

The major examples used throughout the text for in-depth study are the IBM System/360 and System/370, the CDC 6000 and Cyber series, and the DEC PDP-11 series. These provide well-varied illustrations of the subject matter, and are supplemented by additional examples in particular subject areas (e.g. Burroughs B1700 is also used as an example when discussing the microprogramming level). From the viewpoint of a reader in the United Kingdom, it is unfortunate that the illustrations are not extended to include, for example, the ICL 1900 architecture, which provides a further excellent illustration of the various levels. In particular, the composition of the conventional machine level, which can be broadly equated to the ICL 1900 Executive level, provides an excellent contrast with the examples quoted in the text.

It is a tribute to the author's presentation of the material that the reader will regret that the book is not further extended to consider additional levels. The hardware level of interpretation is only touched upon (the reader should beware lest he assume that microprogram interpretation of the conventional machine level is uniformly applied) and operating system structure is not considered in depth; the latter could well be described by means of an interpreted level of machine supporting operating system activities. However, these omissions by no means impact on the stimulating treatment given of structured computer organisation, and the examples and bibliography support the text material in a well chosen manner.

This text is to be recommended to all students of current-day computing system architecture.

D. J. HOWARTH (London)

Mathematical Programming second edition, by C. McMillan, 1975; 650 pages. (*John Wiley & Sons Inc*, £9·75).

When I attended an international mathematical programming symposium in London in 1964 I was surprised at the variety of useful extensions of linear programming methods for nonlinear problems that depend much more on intuition and experience than on mathematical analysis. Since then many good mathematicians have studied nonlinear programming and consequently some techniques have been developed that are often much more powerful than the earlier methods. However, due probably to the fact that the first edition was published in 1970, this book reminds me much more of the London symposium than of recent work. It seems to be written for people who are ignorant of mathematics.

Therefore the few mathematical ideas that occur are introduced in a very detailed manner. For instance, even the definition of a derivative is explained by reference to the speed of a car, and the elementary properties of vectors are stated before they are used. The most advanced method that is given for unconstrained optimisation is the method of steepest ascent. The terms 'positive definite' and 'conjugate' are not mentioned.

Instead the language of the book is that the ideas are developed through examples which are mainly drawn from highly simplified business and ordinary life situations. For instance linear programming is introduced by a scheduling problem where two machines can each manufacture two products. It is fascinating to read how well a subject can be presented through examples, but some people will be irritated by the verbosity and occasional lack of exactitude. The examples account for about half of the length of the book. Most of the remainder is descriptive and there are exercises for the reader at the end of each chapter.

The chapter headings give a fair survey of contents and they are as follows: Fundamental concepts in linear programming, Linear programming extensions, Assignment and distribution methods of linear programming, Classical optimisation, Gradient methods of optimisation, Simplex based methods, Geometric programming, Dynamic programming, The branch and bound algorithm, Integer linear programming, Zero-one programming, Applications of zero-one programming, Network optimisation and Mathematical programming with multiple objectives. I recommend this book as an elementary introduction to all of these subjects because its style makes it easier than usual to understand.

M. J. D. POWELL (Cambridge)