Stanford Artificial Intelligence Laboratory
Memo AIM-262

July 1975

Computer Science Department
Report No. STAN-CS-75-502

Synchronization of ConCurrent Processes

by

Odd Pettersen

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University

SYNCHRONIZATION OF CONCURRENT PROCESSES

by

O. Pettersen

Stanford University
Artificial Intelligence Laboratory *

ABSTRACT:

The paper gives an overview of commonly used synchronization primitives and literature, and presents a new form of primitive expressing conditional critical regions.

A new solution is presented to the problem of "readers and writers", utilizing the proposed synchronization primitive. The solution is simpler and shorter than other known algorithms. The first sections of the paper give a tutorial introduction into established methods, in order to provide a suitable background for the remaining parts.

Key Words and Phrases:

Scheduling, process scheduling, synchronization, mutual exclusion, semaphores, critical regions, parallel programming, multiprogramming, concurrent processes, process communication, shared variables.

CR Categories: 3.80, 3.82, 4.30, 4.32

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract DAHC 15-73-C-0435.

The view and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Agency or the US Government.

* Present address:
Stanford Artificial Intelligence Laboratory
Computer Science Dept.
Stanford University
Stanford, CA. 94305

Address after August 1975:
The Technical University of Norway
Div. of Engineering Cybernetics,
7034 Trondheim - NTH
Norway

1. INTRODUCTION.

It has been shown by several authors, for example [1] to [7], that the internal synchronization between concurrently executing processes in a multiprogramming and/or multiprocessor system can be performed by the use of semaphores exchanged between the processes. Such synchronization becomes necessary when processes interact intentionally through operation on the same set(s) of data. Commonly, "critical regions" are being used to provide indivisibility, necessary for proper functioning of operations on semaphores and other concurrent operations on shared data.

Hoare [3] has proposed a variant of critical regions, by the introduction of conditions for the entry into a critical region. This combines the effect of semaphores and critical regions, and this principle can be used as an effective alternative to semaphores and simple critical regions, since it is more elegant and powerful when the synchronization requirements are more complex. The difference between the two concepts is explained and discussed by Brinch-Hansen in [4].

This paper will give a short review of the principles mentioned, and show how a proposed solution in [4] to a commonly encountered example, the "readers and writers" problem, has certain undesirable effects, as partly also shown in [5]. A new form of conditional critical regions is proposed, and examplified by a new solution to the "readers and writers" problem. A proof of the solution is also included. The new solution is hardly more complicated than the one proposed in [4].

A recent paper by Hoare [11] considers the synchronization from the operating system's viewpoint and presents a semaphore-based solution to a new version of the "readers and writers" problem. I will comment briefly on that paper and show that also the new version of the problem is easier and simpler solved by the method proposed in the present paper.

Program constructs will be presented in the high-level language PASCAL ([8]), which was also used in [4] and [11].

2. CRITICAL REGIONS.

Writing a set of data into a section of memory, or reading it out, generally takes some time and is performed through execution of a series of primitive operations.

More than one computational process may have a legitime need to operate on the same set of data, and these executions may overlap in time. If at least one such concurrent process modifies the shared data, the results will be wrong, because reading processes have no way of knowing whether read data are "old" or "new". The solution to this problem is to provide facilities to prevent such harmful simultaneous operations on shared data. One widely accepted method to prevent simultaneous operations is to let such critical operations be performed associated with a "critical region".

A critical region of some designation v is an abstract concept which can be assigned to some different parts of different programs, but *only to one program at a time*. Different critical regions, however, with different designations, are completely decoupled and have no mutual relationships.

In PASCAL, a critical region can be associated with a shared variable v, declared as follows:

var v : *shared* T

A critical region is defined, and entered by the notation

region v *do* S (1)

where S is a statement executed during the critical region. S can consist of several statements by enclosing them between *begin* and *end*.

3. SEMAPHORES.

A semaphore is a shared single integer variable, declared as follows in PASCAL:

var s : semaphore

A semaphore is initialized to a value, Cs, determined by the intended type of synchronization.

Two primitive (indivisable) operations for the manipulation of a semaphore are wait(s) and signal(s). Their operation can be described very simply by:

```
wait(s):
        s:=s-1;
        while s < 0 do SUSPEND;                                    (2)

signal(s):
        s:=s+1;                                                    (3)
```

The use of this can be demonstrated by the following example, borrowed from [2].

A communication buffer is organized as a circular linked list of frames, at least 2 frames long. Two pointers indicate:

F The first empty frame to insert a message into
R The frame before next frame from which a message is to be withdrawn.

A function, succ(x), supplies the link of the next element.

The synchronization must guarantee that the buffer neither overflows nor underflows. The latter involves that a message can not be withdrawn before it is deposited. Since we are concerned with two constraints, two semaphores must be used: Deposit is preceded by wait(frame), and followed by signal(ready). Accept is preceded by wait(ready) and followed by signal(frame). The initial constants and conditions are: frame = Cframe = buffersize, and ready = Cready = 0, and F = succ(R).

The two programs could be:

```
deposit:    wait(frame);
            buffer[F] := message;
            F := succ(F);
            signal(ready);

accept:     wait(ready);
            R := succ(R);
            received := buffer[R];
            signal(frame);
```

The operations on semaphores may be visualized as follows:

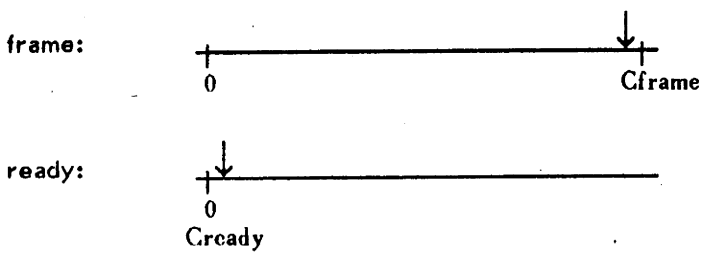

Fig. 1.

The two pointers shown indicate the positions after one deposit more than the number of accepts. Initially, the pointers are located at Cframe and Cready. It is easy to see that deposit can be traversed several times (i.e. Cframe times) before congestion occurs, and accept must be activated. Until then, the wait(frame) operation will not activate SUSPEND. Similarly, as long as accept lags behind deposit, signal(ready) will have been traversed more than wait(ready), consequently ready ≥ 0. However, as soon as one more accept is attempted, ready will become -1, and the further processing will be deferred.

As already mentioned, the semaphore operations wait(s) and signal(s), or at least parts of them, must be indivisable. Otherwise, if for example two different processes simultaneously were performing the operation s:=s+1 on the same variable s, the result could be s(k+2) = s(k)+1 or s(k+2) = s(k)+2, depending on the arbitrary interleave of the basic primitives constituting the operation s := s+1. The correct result of n operations, obviously, should be s(k+n) = s(k)+n, but, if these n operations are arbitrary interleaved in time, the result may be anything between s(k) and the correct one. The problem is resolved by ensuring that the wait(s) and signal(s) operations are indivisable. This preserves the integrity.

If the operations wait(s) and signal(s) themselves are not indivisable, then the consistence is preserved by performing the operations on semaphores within critical regions, which, by definition, are indivisable.

The two programs of the example should then be modified to:

```
deposit:    region v do wait(frame);
            buffer[F] := message;
            F := succ(F);
            region v do signal(ready);

accept:     region v do wait(ready);
            R := succ(R);
            received := buffer[R];
            region v do signal(frame);
```

An alternative way of expressing essentially the same would be to require the wait and signal subroutines to be handled by a scheduler (monitor), for example like [11]. Also then, however, some mechanism must be provided to ensure the integrity, for example by granting monitor access to only one process at a time.

4. CONDITIONAL CRITICAL REGIONS

Conditional critical regions represent a method to synchronize interacting processes, more advanced than those methods explained in the previous paragraphs.

As suggested by Brinch-Hansen [4], regions could be made conditional by changing the form (1) to

region v *when* B *do* S (4)

with the symmetrical complement:

region v *do* S *await* B (5)

The first form allows the program to enter its critical region v. If condition B does not hold, the critical region will be exited immediately. The article calls it "busy waiting", indicating that the program will loop, testing for the condition B to occur. This "busy waiting" is obviously a great disadvantage. Fortunately, it can very easily be avoided, as will be explained later in this paragraph.

The complementing construct (5) causes statement S to be executed, and then further execution of the process to be delayed, until condition B becomes true.

Apparently, conditional critical regions are quite different from semaphores and unconditional critical regions. It is then appropriate to ask: what are their relative advantages, and when is the one method better suited than the other? As Brinch-Hansen has discussed in [4], semaphores are well suited for simple cases, and conditional critical regions superior when the synchronization structure is more complex.

To demonstrate the difference, paper [4] gives two solutions to the so-called "readers and writers" problem, one with semaphores and unconditional critical regions, and one with conditional critical regions.

4.1. The "readers and writers" problem.

The "readers and writers" problem tends to become a classical example, and has appeared in several papers, as for example [4], [5], [6], [9], [10], [11]. It was apparently mentioned first by Courtois et al. in [6]. It is stated as follows:

> Several writers are depositing messages into a buffer, from which several readers will read. Any number of readers may access the buffer simultaneously, but a writer shall have exclusive access. Further, writers have priority over readers.

Several possible solutions exist. One of the simplest encountered is Brinch-Hansen's solution with conditional critical regions in [4]. Although his solutions represent a somewhat simplified example, this fact does not affect the ability to compare the two synchronizing concepts. It is shown in [4] that conditional critical regions give a far simpler solution than the use of only semaphores. The solution presented in [4] is:

declaration:
var v : *shared record* rr, aw : integer *end*

reader:
region v *when* aw = 0 *do* rr := rr + 1;
read;
region v *do* rr := rr - 1;

writer:
region v *do* aw := aw + 1 *await* rr=0;
write;
region v *do* aw := aw - 1;

where the identifiers are:

v	is the critical region
rr	denotes number of "running readers"
aw	indicates the number of "active writers", i.e. writers that have been granted access or are actually writing.

In his later work [10], Brinch-Hansen uses a somewhat different form, apparently as an effort to eliminate certain undesired effects. This will be discussed later in this paragraph. For our purpose here, to explain the operation of conditional critical regions, the earlier form is chosen, since this is more similar to the form I will propose in the following.

Unfortunately, as pointed out by the authors of [6], Courtois, Heymans, and Parnas, in a comment [5] to Brinch-Hansens article [4], Brinch-Hansens simple solution is incorrect, or has at least certain undesireable effects:

As far as the algorithm is concerned, the order of admitting waiting readers and writers into the critical region is quite unpredictable. Thus, it is possible that a writer may wait indefinitely during a stream of incoming readers. This conflicts with the requirement of priority for writers. Paper [5] points out the error in [4] but gives no solution, other than referring again to the solution in [6], with semaphores. Another consequence, but not mentioned i [5], is that *region*-calls from outgoing readers (second *region*-call) may well be blocked from the region by a burst of incoming readers, thus preventing the number of "running readers" to be counted down.

4.2. Discussion of undesirable effects.

It is not mentioned in [4], but it seems necessary to require the dispatcher to release region v at the entrance of the await function. Otherwise, a deadlock will occur: The controlled variable of B is a shared variable, changed by some other computational process. This operation will usually be placed within region v. If region v were not released, no other process could enter it, and condition B would remain false for ever.

But, one could ask: Could not the operation on the controlled variable of B be performed outside region v? No, that would only exceptionally be possible. Generally, that would contradict the purpose of applying critical regions: To prevent spurious errors, due to uncontrolled interleave of operations on shared variables. Since the controlled variable of B is a shared variable, it would impede the correct synchronization, if it was changed outside the critical region.

It should be concluded, then, that for statement (5), the critical region should be released upon entrance of the waiting state.

Then, no reason remains for keeping the *await* function linked to the *region* call of form (5). They should be separated, making them two individual statements:

region v *do* S (6)

and

await B (7)

Statement (6) is identical to the original unconditional *region* call, of form (1). This splitting would provide some increased flexibility, since it would permit the use of the *await* function more freely. One could argue, that the linking to the *region* call has the advantage that the *await* function can more easily be recognized as a terminal part of the *region* call, so that the dispatcher will not re-enter the process again, until condition B has become true. With two separate statements, an extra and thus unnecessary operating systems call is a natural consequence. However, it should be a trivial task for a moderately intelligent compiler to recognize consequtive *region* and *await* statements, thus eliminating the superfluous user-program entry followed by a new operating system call.

Regarding "busy waiting", it should be noted that a critical region is a resource, competed for by several computational processes. This conflict can only be resolved by some dispatching program, usually a part of the operating system. This means that the region statement (4) is handled as a call to the dispatcher, which generally enters the call into a queue. It should, then, be very easy to implement B as a condition for leaving the queue. Thus, the calling program is completely inactive, until the operating system activates it again, by assigning region v to it, after the condition B has become true.

In a more recent work, [10], Brinch-Hansen has apparently eliminated the "busy waiting", by invoking the dispatcher. He also seems to have tried to avoid the inherent deadlock of his *await*. His solution in [10] has certain other drawbacks, however. He has rearranged the form of the conditional critical region call, to:

region v *do begin await* B; S1 *end* (8)

cooperating with an unconditional region call like (1):

region v *do* S2 (9)

where S2 is supposed to alter condition B. This looks like a deadlock again, referring to the definition of critical regions. The author circumvents this by defining a special *temporary* release of region v while the first process is awaiting for condition B, thus allowing the second process to alter B. This temporary release is, however, inside *begin* and *end* of region v, and this seems rather unlogical. It seems unlikely to guess this kind of operation from merely reading the program text, and it gives an unclean internal operation of the dispatcher, manipulating the calling program from the main queue and over to another, temporary queue.

A remark in [11] points out, that transferring the responsibility for testing of condition B over to the monitor or dispatcher may impede the efficiency, because expression B must be re-evaluated after every exit from a procedure of the monitor. There might even be several similar expressions throughout the program that required similar re-evaluation. Fortunately, this can be improved considerably. Firstly, efficiency can be improved by the user himself, by applying only simple conditions as B, like X = specified integer, boo = true, etc. The second approach to improvement requires some explanation: The inefficiency is hardly linked to *where* a testing is effected, whether this is in the application program, or within the operating system. In any case, this is basically "busy waiting". An alternative to testing of condition B inside the monitor is to enter the function (wait or region) itself, and perform the testing there. This "busy waiting" is definitely no more efficient than doing it inside the monitor. Considerable higher efficiency can be obtained by another and different approach:

> The compiler could generate a list for each procedure, containing controlling variables of wait and conditional critical regions, affected by the particular procedure, together with references to the wait and region functions. At each procedure exit, only the conditions for the wait and region functions referred to in the list should be re-evaluated.

More philosophically, one might perhaps say, that there are totally three different methods to effect an action upon the occurrence of a certain condition or event:

- Interrupt generated by the event.
- "Busy waiting" with repeated testing.
- Prior to run-time, prepare a list showing functions affected by a change of value of a variable within a certain code body, like a procedure. At run time, this list provides the ability to refer actions directly, rather than testing the conditions from the opposite direction.

4.3. Conditional critical regions with priority.

So far, nothing really new has been mentioned about critical regions. I have merely explained certain consequences and restrictions of methods published earlier, although these restrictions do not seem to have been fully recognized in the published articles.

It seems now appropriate to propose a form of conditional critical region calls that has none of the defects mentioned above. The new form is simple to use and to understand, because it is natural and directly attacks the problem, besides it should give a very efficient code.

The new form introduces priority into forms (4) and (6) above and comprises three system calls:

region v:=p *when* B *do* S (10)
region v:=p *do* S (11)
await B (12)

Corresponding to the remarks about flexibility of *await* B, form (7), the await function can arbitrarily be used in connection with the conditional (10) or unconditional (11) region call.

The new element, p, is an integer or integer expression denoting the relative priority for granting the region among competing programs. The assignment v:=p is not effected until the critical region is entered, and the scheduler should arrange the queue of requesting access to the region, according to decreasing values of p, such that that one with the highest value will be picked first. The result is a selection according to relative priority. The value of p must be defined before use, and dynamic priority should be easy to apply.

Naturally, only p-values belonging to calls within the queue are considered, and calls entering the queue after a region is entered will be queued normally and only considered after the region is released, even if the priority of the process currently in the region has lower priority than the approaching process. Although this non-preemptive interpretation of priorities should be quite self-evident, it is mentioned here, to emphasize the fact before starting proving the algorithms in the next section.

With this method, the "readers and writers" problem has a solution that is simpler than other solutions frequently encountered in the literature, besides it has none of the defects cited. In section 4.5, I will present an efficient and simple solution to a new version of the problem, presented in [11].

The solution to the original version of the problem is:

```
declaration:
        var v : shared record rr, aw : integer end
initial values:
        aw:=0;        rr:=0;

reader:
        region v := 1 when aw = 0 do rr := rr + 1;
        read;
        region v := 3 do rr := rr - 1;

writer:
        region v := 2 when aw = 0 do aw := 1;
        await rr=0;
        write;
        aw := 0;
```

Since the forms (10) and (11) are similar to (4) and (5), it should not be surprising to find the new solution quite similar to the one presented in [4]. Some signficant differences exist, however:

* The program for readers deviates only in the inclusion of priorities.
* In the program for writer, a new condition *when* aw=0 is included, making the region call similar to that of the readers.
* The assignment aw:=1 replaces aw:=aw+1, since aw will never need to have values different from 0 or 1. Thus, a simple boolean variable could be adequate, provided the language syntax would permit.
* The assignment aw:=0, terminating the write statement, need not be performed inside the critical region.

4.4. Proof of correctness of new solution for "readers and writers" problem.

Let us use the following definitions of "active" and "running" processes, slightly modified from [4]:

A process is *active* from the moment its request of a resource is acknowledged, until the resource is released.
A process is *running* from the instant it has been given permission to use the resource, until it is released.

The definitions can be visualized:

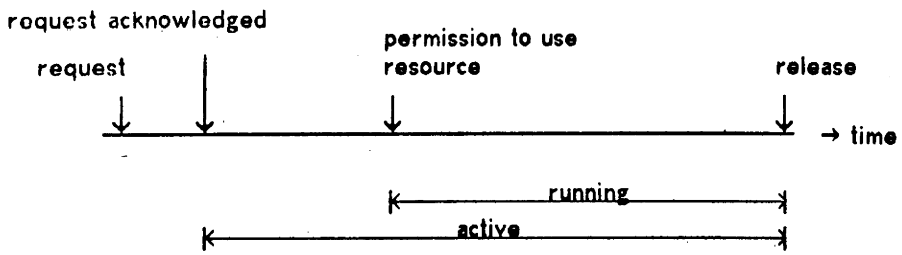

Fig. 2.

With respect to the programs for readers and writers, active and running processes are:

A reader is *active* from the moment it has entered its region v before reading, until it has left region v after reading. A reader is *running* from the moment it leaves region v before reading, until it has left region v after reading.
A writer is *active* from the instant it has entered its region v before writing, until it has executed statement aw:=0 after writing. It is *running* from the instant it has ended the ***await*** function and is to start writing, until it has executed statement aw:=0 after writing.

In addition to identifiers introduced earlier, the following identifier is used in the proof:

rw the number of running writers, according to definition above.

The proof will follow these lines:

1. A set of criteria is established, believed to constitute sufficient conditions for the proof to be complete.
2. A set of lemmas is established.
3. Based on the lemmas, each criterion is shown to be satisfied.

CRITERIA FOR CORRECTNESS OF PROGRAMS:

C1. Mutual exclusion of running processes follows two invariants:
$X1 = (0 \leq rw \leq 1)$ (or: $X1 = ((rw=0) \vee (rw=1))$)
$X2 = \neg(rr>0 \wedge rw>0)$
X1 and X2 are both invariant true.

C2. (overlaps partly what is expressed by C1:)
Several readers can be active simultaneously, but as soon as a writer has applied for access, further access of new incoming readers, as well as other writers, is prevented. Running readers are allowed to conclude. When all running readers have terminated, the pending writer is given exclusive access.

C3. No interference (i.e. unwanted or uncontrolled interaction) exists for use of shared variables.

C4. No deadlock can occur.

C5. Incoming writers have priority over incoming readers. (Also expressed implicitly in C2.)

LEMMAS:

L1. Initially, aw=0 and rr=0. This follows directly from the program text.

L2. Since any change of rr is performed only inside the same critical region, these operations will not interfere.

L3. Because of L2, and since a read function is embraced by the statement rr:=rr+1 ahead of the function and rr:=rr-1 following the function, a completing reader will leave rr undisturbed. Further, rr will always indicate "number of active readers", which is the number of readers having finished their entry critical regions but not yet finished their terminating region. Thus, $rr \geq 0$ is invariant.

L4. What is said about readers in L3, is also true for writers. For a moment, let us assume that the statement aw:=1 were aw:=aw+1. It is performed inside a critical region, where the condition for entering this region is aw=0. If we could ensure that aw would not change, from the moment it was tested aw=0 and until ending the execution of aw:=aw+1, the result would be the same if the last statement were substituted by aw:=1 which is slightly simpler.

L5. After one writer has entered, and made aw←1, no more writers can enter, until the first one has terminated, by executing the statement aw:=0. Thus, there can never be more than 1 writer active, i.e. aw can only attain the values 0 or 1.

L6. When an outgoing writer is to execute the statement aw:=0, other writers can only be inactive or waiting for access, because of L5. Thus, the statement aw :=0 will not interfere with any other operation on aw. Consequently, the statement aw:=0 can be performed outside the critical region, and the condition for the simplification made in L4 is valid, even though aw is changed, by aw:=0, outside the region.

L7. If region v is available, a running reader can always terminate, since the entry to region v is unconditional. It may be delayed if v is unavailable, i.e. v is currently granted to some other process. The duration of such a delay will only be very short, however, since no process executes more than a single assignment statement within v.

L8. Because of the invariant $rr \geq 0$ (see L3), and rr is an integer that only changes its value by unity, the initial value of rr (which is 0), will eventually be attained, if further access for incoming readers (executing rr:=rr+1) were blocked from some point on, and provided that all read operations are executed in a finite time.

L9. A blocking of incoming readers, as mentioned in L8, will be performed by an active writer. Since incoming writers have priority over readers, an incoming writer will be granted access to v before any incoming readers, so that aw:=1 will be performed before further incoming readers will be considered. Then, further access to region v will be denied for all incoming processes, and the blocking mentioned in L8 will be effective. Thus, an incoming writer may be delayed in its await-function, but it will remain here only a finite time, because rr=0 will eventually become true, as stated in L8. It is also important to note that the "blocking" of incoming readers will be effective until the active writer has terminated its writing and executed aw:=0, and this can only happen after it has passed the *await* rr=0, which involves the necessity of rr=0. Thus, the blocking of incoming readers will, once started, remain until rr has been counted down to rr=0, as stated in L8.

L10. A running writer is also active. I.e. state *running* is a subset of state *active*, implying the invariant true:

$Y = (0 \leq rw \leq aw)$

L11. The Boolean X2 can be changed to:

$X2 = (rr \rangle 0) \vee (rw \rangle 0)$

by De Morgan's theorem. Since rr and rw are non-negative integers, $rr \rangle 0$ implies rr=0 and $rw \rangle 0$ implies rw=0. Thus

$X2 = (rr=0) \vee (rw=0)$

PROOFS:

P1. Because of Lemma L5, stating that aw can only attain the values 0 or 1, invariant Y (see L10) gives immediately

$rw = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}$ as only possible values.

This proves invariant X1 in C1.

P2. Proof of X2=true invariant:
According to L11, the proof is complete if we can show that either rr=0 or rw=0. The program for writer shows, according to the definition for running writers, that when a writer is running, i.e. after the passing of *await* rr=0, then rr=0. This must last at least until the writer is no longer *active*, since aw≠0 blocks incoming readers, according to L9. When a writer is running, then rw≠0, by definition.
This shows that either:

$rr = 0 \quad or \quad rw = 0$

which implies X2 = (rr}0) v (rw}0) is invariant true.
This completes the proof of C1.

P3. The invariants X1 and X2 are proved in P1 and P2. X1 expresses that only one writer can be running at a time. Even stronger, L5 states that only one writer can have access (i.e. be active) at a time. L9 states that further access of incoming readers is blocked when a reader is active. This lemma further states that the active writer becomes running when all running readers have terminated. Thus, C2 is satisfied.

P4. Proof of C3:
Shared variables are: rr, aw, buffer.
L2 and L6 states no interference for rr and aw. The buffer is changed only by a writer, when it is running. We have already proved (P3) that when this occurs, no other process has access to the buffer. This completes the proof of C3.

P5. Deadlock (Criterion C4):
One necessary condition for deadlock is that a program holds resources while waiting for other programs to release resources. If this is proved not to be true, then deadlock will not occur.
Resources common to the program, and of significance for the deadlock problem:

$rr = 0, \quad region\ v, \quad aw = 0$

The only place rr=0 is a condition for proceeding is at the *await* rr=0 in the writer's program. At this point, the writer is neither in the region, nor is aw any condition for the execution which leads to rr=0. Thus, the waiting for rr=0 will not induce deadlock.
region v: Access to region v can be denied, either because the region is granted to some other process, or because aw≠0. If the region is granted to some other process, this will never last long (L7). If aw≠0, a writer must be active, and then aw:=0 remains to be done. Since we have shown that await rr=0 will not involve deadlock, the writer will proceed normally, and finally execute aw:=0. Thus, the awaiting for aw=0 will not cause deadlock. This shows that the cited condition, necessary for deadlock, is not satisfied. Deadlock is prevented, and criterion C4 is satisfied.

P6. Incoming writers have priority over incoming readers:
This follows directly from the definition of conditional critical region with priority, and the fact that p=2 for writers and p=1 for readers. Thus, C5 is satisfied immediately.

This completes the whole proof.

4.5. A solution to a modified version of "readers and writers".

Hoare, in [11], has presented a slight modification of the "readers and writers" problem:

> The writers have priority over readers, as originally. However, readers waiting at the end of a write are given priority over the next writer. The purpose is to avoid the danger of indefinite exclusion of readers, in a burst of successive writers.

A simple solution of this, applying the method of conditional critical regions with priority, is presented without a formal and complete proof:

```
declaration:
    var v : shared record rr, aw : integer end
initial values:
    aw:=0;      rr:=0;

reader:
    region v := 1 do;
    region v := 3 when aw = 0 do rr := rr + 1;
    read;
    region v := 4 do rr := rr - 1;

writer:
    region v := 2 when aw = 0 do aw := 1;
    await rr=0;
    write;
    aw := 0;
```

This solution appears simpler than that in [11] which, moreover, does not deal with the contention problem at all.

When comparing with the solution of the original problem, shown in section 4.3, one note immediately the following details:

* The increase of the highest priority, that one of outgoing readers, from 3 to 4. This should make no difference, since it is the highest priority in each case.
* The "writer"'s program is unchanged.
* "Reader"'s program is extended with a preceding region call, without action statement.
* Following the first region call for incoming readers is the conditional region call, as originally. However, the priority is increased beyond that of the writers.

Region call of priority 1 has the same purpose as that of the original solution: Preventing the continuation in the program if a reader arrives to this point simultaneously with a writer being on the point of entering the region. Then, the writer will prevail. After the writer has left the region, the reader will continue but will be suspended in the next region call, waiting for aw=0, as previously.

If a reader arrives slightly before a writer, the reader will enter its region the first time. After this point, it is guaranteed to continue, also into the next entry of the region, despite the waiting writer, because of the higher priority 3. Thus, the two region-entries will not be separated, and the reader is allowed to continue until termination, together with other active readers, before the write is acknowledged. Readers arriving later must first enter region v with priority 1, however, and this is prevented at this time by the pending writer, which will be granted access first. This separates incoming readers into two groups: Those who have not entered the region the first time: these must wait until the writer has changed aw, upon which they will be trapped at the next entry of the region. The other group consists of those having arrived before the writer; these will continue until termination.

While a writer is active, readers may freely enter region v the first time, since this is unconditional. Then, they will wait for aw=0. Assuming that another writer arrives together with readers during a write, the situation is, at the instant when the active writer terminates: Pending readers applies for region v with priority 3 and thus dominates the waiting writer. Possible new readers, however, having yet not entered the region the first time, will be delayed, because of the low priority =1, until the writer has passed the region. At this time, however, aw=1 and the new readers must wait until completion of the writer.

Consequently, also this time we have effectively separated applying readers into two groups: Those who entered during the previous write, and those arriving after. The effect is as required.

5. REFERENCES

[1] Dijkstra, E.W.: Cooperating Sequential Processes. In *Programming Languages* (F. Genuys, ed.), Academic Press, N.Y. 1968, pp. 43-112.

[2] Habermann, A.N.: Synchronization of Communicating Processes. *Comm. ACM 15*, 3 (March 1972), pp. 171-176.

[3] Hoare, C.A.R.: Towards a theory of parallel programming. *International Seminar on Operating Systems Techniques.* Belfast, Northern Ireland, Aug.-Sep. 1971.

[4] Brinch-Hansen, P.: A Comparison of Two Synchronizing Concepts. *Acta Informatica 1* (1972), pp. 190-199.

[5] Courtois, Heymans and Parnas: Comments on "A Comparison of Two Synchronizing Concepts". *Acta Informatica 1* (1972), pp. 375-376.

[6] Courtois, Heymans and Parnas: Concurrent control with "Readers" and "Writers". *Comm. ACM 14*, 10 (Oct. -71), pp. 667-668.

[7] Coffman, E.G. and Denning, P.J.: OPERATING SYSTEMS THEORY. Prentice-Hall, N.J. 1973. pp. 68-74.

[8] Wirth,N.: The programming language PASCAL. *Acta Informatica 1*, (1971) pp. 35-63.

[9] Griffiths, P.: SYNVER: A System for the Automatic Synthesis and Verification of Synchronous Processes. *Proc. ACM'74* pp. 167-173.

[10] Brinch-Hansen, P.: Operating System Principles. Prentice-Hall 1973.

[11] Hoare, C.A.R.: Monitors, An Operating Systems Structuring Concept. *Comm. ACM 17*, 10 (Oct. -74), pp. 549-557.