to point out that the arguments made earlier in favor of the language construct for simple critical regions can also be made for message buffers.

In Section 3.4.1, we found that a system consisting of processes connected only by buffers can be made functional as a whole. But this is only true if the *send* and *receive* operations are implemented correctly and if they are the only operations on the buffers. A compiler is unable to recognize the data structure B in Algorithm 3.6 as a message buffer and check that it is used correctly. So when message buffers are used frequently, it may well be worth including them as a primitive concept in a programming language.

We now proceed to the next problem, which is due to Courtois, Heymans, and Parnas (1971).

3.4.5. An Example: Readers and Writers

Problem Definition

Two kinds of concurrent processes, called *readers* and *writers*, share a single resource. The readers can use the resource simultaneously, but each writer must have exclusive access to it. When a writer is ready to use the resource, it should be enabled to do so as soon as possible.

The first step is to introduce a terminology which enables us to talk about the problem in a meaningful manner. A process must declare its wish to use the resource, and, since the resource may be occupied at that moment, the process must then be prepared to wait for it. A process must also indicate when it has completed its use of the resource.

So any solution to this kind of resource allocation problem must be of the following nature:

```
request resource;
use resource;
release resource;
```

All processes must go through such a sequence of events, and I would expect the solution to be symmetrical with respect to the readers and writers. To simplify matters, I will start by solving a simpler problem in which I do not bother to ensure that the writers exclude one another, but only that they exclude all readers, and vice versa. They are thus more symmetrical with the readers.

A process is called *active* from the moment it has requested the resource until it has released the resource again. A process is called *running* from the moment it has been granted permission to use the resource until it has released the resource again.

The state of the system can be characterized by four integers, all initialized to zero:

ar	the number of active readers
rr	the number of running readers
aw	the number of active writers
rw	the number of running writers

Correctness Criteria

A solution to the simplified problem is correct if the following criteria are satisfied:

(1) *Scheduling of waiting processes*: Readers can use the resource simultaneously and so can writers, but the number of running processes cannot exceed the number of active processes:

$$0 \leqslant rr \leqslant ar \;\&\; 0 \leqslant rw \leqslant aw$$

This invariant will be called W.

(2) *Mutual exclusion of running processes*: Readers and writers cannot use the resource at the same time:

$$\textbf{not}\ (rr > 0 \;\&\; rw > 0)$$

This invariant will be called X.

(3) *No deadlock of active processes*: When no processes are running, active processes can start using the resource within a finite time:

$$(rr = 0 \;\&\; rw = 0) \;\&\; (ar > 0\ \textbf{or}\ aw > 0)\ \textbf{implies}$$
$$(rr > 0\ \textbf{or}\ rw > 0)\ \textit{within a finite time}$$

(4) *Writers have priority over readers*: The requirement of mutual exclusion means that the resource can only be granted to an active writer when there are no running readers ($rr = 0$). To give priority to writers, we make the slightly stronger condition that the resource can only be granted to an active reader when there are no active writers ($aw = 0$).

Solution With Semaphores

This time we will solve the problem first by means of simple critical regions and semaphores. Two semaphores, called *reading* and *writing*, enable the readers and writers to wait for the resource. They are both initialized to zero. The solution is Algorithm 3.8.

ALGORITHM 3.8 The Readers and Writers Problem Solved With Semaphores

```
type T = record ar, rr, aw, rw: integer end

var v: shared T; reading, writing: semaphore;

"Initially ar = rr = aw = rw = reading = writing = 0"

cobegin
  begin "reader"
    region v do
    begin
      ar:= ar + 1;
      grant reading(v, reading);
    end
    wait(reading);
    read;
    region v do
    begin
      rr:= rr - 1;
      ar:= ar - 1;
      grant writing(v, writing);
    end
    . . .
  end

  begin "writer"
    region v do
    begin
      aw:= aw + 1;
      grant writing(v, writing);
    end
    wait(writing);
    write;
    region v do
    begin
      rw:= rw - 1;
      aw:= aw - 1;
      grant reading(v, reading);
    end
    . . .
  end
  . . .
coend
```

A reader indicates that it is active by increasing *ar* by one. It then calls a procedure, *grant reading*, which examines whether the resource can be granted for reading immediately. Then the reader *waits* until it can use the resource. Finally, it leaves the running and active states by decreasing *rr* and *ar* by one and calls another procedure, *grant writing*, which determines whether the resource should now be granted to the writers. The behavior of a writer is quite symmetrical.

The scheduling procedures, *grant reading* and *grant writing*, are defined by Algorithm 3.9.

ALGORITHM 3.9 The Readers and Writers Problem (cont.)

```
procedure grant reading(var v: T; reading: semaphore);
begin
  with v do
  if aw = 0 then
  while rr < ar do
  begin
    rr:= rr + 1;
    signal(reading);
  end
end

procedure grant writing(var v: T; writing: semaphore);
begin
  with v do
  if rr = 0 then
  while rw < aw do
  begin
    rw:= rw + 1;
    signal(writing);
  end
end
```

The resource can be granted to all active readers ($rr = ar$) provided no writers are active ($aw = 0$). And it can be granted to all active writers ($rw = aw$) provided no readers are running ($rr = 0$).

I will now outline a correctness proof of this solution. The arguments are explained informally to make them easy to understand, but a purist will not find it difficult to restate them formally as assertions directly in the program text.

Let us first verify that the components of variable *v* have the *meaning* intended. Since *ar* and *aw* are increased by one for each request and decreased by one for each release made by readers and writers, respectively, we immediately conclude that they have the following meanings:

ar = number of active readers

aw = number of active writers

It is a little more difficult to see the meanings of the variables *rr* and *rw*. Consider for example *rr*: It is increased by one for each *signal* on the semaphore *reading* and decreased by one for each release made by a reader, so:

rr = number of signals(reading) − number of releases made by readers

From the program structure, it is also clear that the running readers are those which have been enabled to complete a *wait* on the semaphore *reading* minus those which have released the resource again. So

number of running readers =

number of readers which can or has passed wait(reading) −

number of releases made by readers

The semaphore invariant ensures that

number of readers which can or has passed wait(reading) =

number of signals(reading)

So we finally conclude that

rr = number of running readers

and similarly for writers that

rw = number of running writers

Consider now correctness criteria 1 and 2. We assume that the assertions *W* and *X* hold immediately before a request by a reader. This is trivially true after initialization when

$$0 = rr = ar \;\&\; 0 = rw = aw$$

The increase of *ar* by one inside a request does not change the validity of *W* and *X*, so we have:

```
"reader request"
region v do
begin "W & X"
      ar:= ar + 1;                              (cont.)
```

```
        "$W$ & $X$"
        grant reading($v$, reading);
        "?"
end
```

The procedure *grant reading* either does nothing (when $aw \neq 0$ or $rr = ar$), in which case W and X still hold, or it increases the number of running readers by one until

$$0 < rr = ar \; \& \; 0 = rw = aw$$

holds. This implies that W and X still hold:

$$? \textbf{ implies } W \; \& \; X$$

Consider now a reader release. A release is only made by a running process, so we have $rr > 0$ immediately before. Assuming that W and X also hold initially, we have

```
"reader release"
region v do
begin "$W$ & $X$ & $rr > 0$"
        $rr := rr - 1$;
        $ar := ar - 1$;
        "??"
        grant writing($v$, writing);
        "???"
end
```

Now $W \; \& \; X \; \& \; rr > 0$ is equivalent to $0 < rr \leqslant ar \; \& \; 0 = rw \leqslant aw$ so

$$?? \equiv 0 \leqslant rr \leqslant ar \; \& \; 0 = rw \leqslant aw$$

which in turn implies W & X.

The procedure *grant writing* either does nothing (when $rr \neq 0$ or $rw = aw$), in which case W and X still hold, or it increases the number of running writers by one until

$$0 = rr \leqslant ar \; \& \; 0 < rw = aw$$

holds. This implies that W and X still hold:

$$??? \textbf{ implies } W \; \& \; X$$

By similar arguments, you can show that the invariance of W and X is maintained by a writer request and release.