A detailed Solution to the Readers and Writers Problem

Algorithm (2nd solution):

When a writer process requests access to the shared resource, any subsequent reader process must wait for the writer to gain access to the shared resource and then release it.

- This method still allows a stream of readers to enter the critical region until a writer arrives.
- Writer then takes priority over all subsequent readers, except those already accessing the shared resource.

```
WRITER()
{
  while (TRUE)
  {  ------
     P(mutex2);
        W2 = W2 + 1;
        if (W2 == 1)          // first writer blocks
           P(readBlock);      // all readers
     V(mutex2);
     P(writeBlock);           // Then it blocks
        access resource;      // other writers
     V(writeBlock);
     P(mutex2);
        W2 = W2 - 1;
        if(W2==0)             // If no other writer
           V(readBlock)       // is using the resource,
                              // then allow readers to
     V(mutex2);               // access the resource
  }
}
```

```
READER( )
{
  while (TRUE)
  {
    .....
    P(writePending);     // only the current 1st reader
                         // can reset writePending
      P(readBlock);      // it then blocks other readers
        P(mutex1);
          R2 = R2 + 1;
          if (R2 == 1)       // first reader also blocks
              P(writeBlock); // other writers.
        V(mutex1);
    V(readBlock);        // Now other readers can read
  V(writePending);       // as writePending = 1 again
    access resource;
  P(mutex1);
    R2 = R2 - 1;
    if (R2 == 0)         // If no other readers are
        V(writeBlock);   // still reading, then allow
                         // possible writing.
  V(mutex1);
  }
}
```

Algorithm

```
int  R2 = 0, W2 = 0;

Semaphore  mutex1 = 1, mutex2 = 1;
Semaphore  readBlock = 1,
           writePending = 1,
           writeBlock = 1;

COBEGIN
    repeat until no more data READER();
    repeat until no more data WRITER();
COEND
```

3rd Solution
(by C.A.R. Hoare)
CACM, 1974

writers to wait on:

```
OKtoread, OKtowrite: condition;
```

The following annotation is relevant:

```
OKtoread ≡ ¬ busy
OKtowrite ≡ ¬ busy & readercount = 0
invariant: busy ⟹ readercount = 0
```

```
class readers and writers: monitor
  begin readercount: integer;
      busy: Boolean;
      OKtoread, OKtowrite: condition;
    procedure startread;
      begin if busy ∨ OKtowrite.queue then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        comment Once one reader can start, they all can;
      end startread;
    procedure endread;
      begin readercount := readercount − 1;
        if readercount = 0 then OKtowrite.signal
      end endread;
    procedure startwrite;
      begin
        if readercount ≠ 0 ∨ busy then OKtowrite.wait
        busy := true
      end startwrite;
    procedure endwrite;
      begin busy := false;
        if OKtoread.queue then OKtoread.signal
                          else OKtowrite.signal
      end endwrite;
    readercount := 0;
    busy := false;
  end readers and writers;
```

I am grateful to Dave Gorman for assisting in the discovery of this solution.

6.3 Readers and Writers

As a more significant example, we take a problem which arises in on-line real-time applications such as airspace control. Suppose that each aircraft is represented by a record, and that this record is kept up to date by a number of "writer" processes and accessed by a number of "reader" processes. Any number of "reader" processes may simultaneously access the same record, but obviously any process which is updating (writing) the individual components of the record must have exclusive access to it, or chaos will ensue. Thus we need a class of monitors; an instance of this class local to *each* individual aircraft record will enforce the required discipline for that record. If there are many aircraft, there is a strong motivation for minimizing local data of the monitor; and if each read or write operation is brief, we should also minimize the time taken by each monitor entry.

When many readers are interested in a single aircraft record, there is a danger that a writer will be indefinitely prevented from keeping that record up to date. We therefore decide that a new reader should not be permitted to start if there is a writer waiting. Similarly, to avoid the danger of indefinite exclusion of readers, all readers waiting at the end of a write should have priority over the next writer. Note that this is a very different scheduling rule from that propounded in [4], and does not seem to require such subtlety in implementation. Nevertheless, it may be more suited to this kind of application, where it is better to read stale information than to wait indefinitely!

The monitor obviously requires four local procedures:

startread	entered by reader who wishes to read.
endread	entered by reader who has finished reading.
startwrite	entered by writer who wishes to write.
endwrite	entered by writer who has finished writing.

We need to keep a count of the number of users who are reading, so that the last reader to finish will know this fact:

```
readercount: integer
```

We also need a *Boolean* to indicate that someone is actually writing:

```
busy: Boolean;
```

We introduce separate conditions for readers and

7. Conclusion

This paper suggests that an appropriate structure for a module of an operating system, which schedules resources for parallel user processes, is very similar to that of a data representation used by a sequential program. However, in the case of monitors, the bodies of the procedures must be protected against re-entrance by being implemented as critical regions. The textual grouping of critical regions together with the data which they update seems much superior to critical regions scattered through the user program, as described in [7, 12]. It also corresponds to the traditional practice of the writers of operating system supervisors. It can be recommended without reservation.

However, it is much more difficult to be confident about the condition concept as a synchronizing primitive. The synchronizing facility which is easiest to use is probably the conditional *wait* [2, 12]:

```
wait(B);
```

where B is a general Boolean expression (it causes the given process to wait until B becomes true); but this may be too inefficient for general use in operating systems,