< PREVIOUS NEXT >

Parallel or Concurrent Programming Programming M. Ben-Ari Addison-Wesley Principles of Concurrent and Distributed Programming, Second Edition

7.6. The Problem of the Readers and Writers

The problem of the readers and writers is similar to the mutual exclusion problem in that several processes are competing for access to a critical section. In this problem, however, we divide the processes into two classes:

Readers Processes which are required to exclude writers but not other readers.

Writers Processes which are required to exclude both readers and other writers.

The problem is an abstraction of access to databases, where there is no danger in having several processes read data concurrently, but writing or modifying data must be done under mutual exclusion to ensure consistency of the data. The concept database must be understood in the widest possible sense; even a couple of memory words in a real-time system can be considered a database that needs mutual exclusion when being written to, while several processes can read it simultaneously.

Algorithm 7.4 is a solution to the problem using monitors. We will use the term "reader" for any process executing the statements of `reader` and "writer" for any process executing the statements of `writer`.

Algorithm 7.4. Readers and writers with a monitor

```
monitor RW
  integer readers ⟵ 0
  integer writers ⟵ 0
  condition OKtoRead, OKtoWrite

  operation StartRead
    if writers ≠ 0 or not empty(OKtoWrite)
       waitC(OKtoRead)
    readers ⟵ readers + 1
    signalC(OKtoRead)

operation EndRead
  readers ⟵ readers - 1
  if readers = 0
     signalC(OKtoWrite)

operation StartWrite
  if writers ≠ 0 or readers ≠ 0
     waitC(OKtoWrite)
  writers ⟵ writers + 1

operation EndWrite
   writers ⟵ writers - 1
   if empty(OKtoRead)
      then signalC(OKtoWrite)
      else signalC(OKtoRead)
```

reader	writer
p1: RW.StartRead p2: read the database p3: RW.EndRead	q1: RW.StartWrite q2: write to the database q3: RW.EndWrite

The monitor uses four variables:

readers The number of readers currently reading the database after successfully executing `StartRead` but before executing `EndRead`.

writers The number of writers currently writing to the database after successfully executing `StartWrite` but before executing `EndWrite`.

OKtoRead A condition variable for blocking readers until it is "OK to read."

OKtoWrite A condition variable for blocking writers until it is "OK to write."

The general form of the monitor code is not difficult to follow. The variables `readers` and `writers` are incremented in the `Start` operations and decremented in the `End` operations to reflect the natural invariants expressed in the definitions above. At the beginning of the `Start` operations, a boolean expression is checked to see if the process should be blocked, and at the end of the `End` operations, another boolean expression is checked to see if some condition should be signaled to unblock a process.

A reader is suspended if some process is currently writing (*writers* $\neq$ 0) or if some process is waiting to write (¬*empty*(*OKtoWrite*)). The first condition is obviously required by the specification of the problem. The second condition is a decision to give the first blocked writer precedence over waiting readers. A writer is blocked only if there are processes currently reading (*readers* $\neq$ 0) or writing (*writers* $\neq$ 0). `StartWrite` does not check the condition queues.

`EndRead` executes `signalC(OKtoWrite)` if there are no more readers. If there are blocked writers, one will be unblocked and allowed to complete `StartWrite`. `EndWrite` gives precedence to the first blocked reader, if any; otherwise it unblocks a blocked writer, if any.

Finally, what is the function of `signalC(OKtoRead)` in the operation `StartRead`? This statement performs a *cascaded unblocking* of the blocked readers. Upon termination of a writer, the algorithm gives precedence to unblocking a blocked reader over a blocked writer. However, if one reader is allowed to commence reading, we might as well unblock them all. When the first reader completes `StartRead`, it will signal the next reader and so on, until this cascade of signals unblocks all the blocked readers.

What about readers that attempt to start reading during the cascaded unblocking? Will they have precedence over blocked writers? By the immediate resumption requirement, the cascaded unblocking will run to completion before any new reader is allowed to commence execution of a monitor operation. When the last `signalC(OKtoRead)` is executed (and does nothing because the condition queue is empty), the monitor will be released and a new reader may enter. However, it is subject to the check in `StartRead` that will cause it to block if there are waiting writers.

These rules ensure that there is no starvation of either readers or writers. If there are blocked writers, a new reader is required to wait until the termination of (at least) the first write. If there are blocked readers, they will (all) be unblocked before the next write.

‹ PREVIOUS NEXT ›

‹ PREVIOUS NEXT ›

Parallel or Concurrent Programming Programming M. Ben-Ari Addison-Wesley Principles of Concurrent and Distributed Programming, Second Edition

7.7. Correctness of the Readers and Writers Algorithm[A]

Proving monitors can be relatively succinct, because monitor invariants are only required to hold outside the monitor procedures themselves. Furthermore, the immediate resumption