Journal of Computing and Information Technology - CIT 13, 2005, 1, 43–51

Process Synchronization with Readers and Writers Revisited

Jalal Kawash

Department of Computer Science, American University of Sharjah, Sharjah, UAE

The readers-writers problem is one of the very well known problems in concurrency theory. It was first introduced by Courtois *et.al.* in 1971 [1] and requires the synchronization of processes trying to read and write a shared resource. Several readers are allowed to access the resource simultaneously, but a writer must be given exclusive access to that resource. Courtois *et.al.* gave semaphore-based solutions to what they called the *first* and *second* readers-writers problems. Both of their solutions are prone to starvation. The first allows readers to indefinitely lock out writers and the second allows writers to indefinitely lock out readers. This paper presents and proves correct a third semaphore-based solution, which is starvation-free for both reader and writer processes.

Keywords: concurrency control, shared objects, mutual exclusion, formal verification, computing education.

1. Introduction

The readers-writers problem [1] requires the synchronization of concurrent processes simultaneously accessing a shared resource, such as a database object. This problem is different from the known mutual exclusion problem [9] in that it distinguishes between two categories of processes: those who only read the resource, called *readers*, and those who write it, called *writers*. Since reader processes only read the resource, it is more efficient to grant all such reader processes simultaneous access to the resource. However, a writer process is granted exclusive access to the resource. Thus, it is not acceptable to protect the resource using the traditional critical section [11] technique of mutual exclusion, allowing at most one process to access the resource at a time. The readers-writers requirements allow more concurrency and more efficient use of the resource.

Courtios *et.al.* [1] developed two solutions to two versions of the readers-writers problem, which are known as the *first* and the *second* readers-writers problems. Both of these solutions use Dijkstra's semaphore [2]. A (binary) semaphore `S` is an object that has an associated integer value (`val`) and a FIFO queue (`queue`) with the support of two *atomic* operations `wait(S)` and `signal(S)` defined as follows. Initially, `S.val` is 1 and `S.queue` is empty.

```
wait(S) {
      if S.val = 0 then
            wait on S.queue
            (block the process)
      else S.val ← 0
}

signal(S) {
      if S.queue is not empty then
            remove one process from S.queue
            and unblock it
      else S.val ← 1
}
```

These operations are atomic, which requires them to appear as if they are executed in a critical section. When a process is executing `wait(S)` or `signal(S)`, no other process can execute either of these two operations on the same semaphore `S`.

Most recent work on the readers-writers problem addresses building analytical models and studying performance implications (see [14,10, 7] and references therein). That work, however, does not propose solutions to the problem. The group mutual exclusion problem proposed by Joung [4] is a generalization of the readers-writers problem. A solution to group

Writer process

```
Repeat
      wait(resource);

      // write the resource

      signal(resource);
until done
```

Reader process

```
repeat
      wait(mutex);
      readers ← readers + 1;
      if readers = 1 then
            wait(resource);
      end-if
      signal(mutex);

      //read the resource

      wait(mutex);
      readers ← readers - 1;
      if (readers = 0) then
            signal(resource);
      end-if
      signal(mutex);
until done
```

Fig. 1. Solution to the first readers-writers problem.

exclusion implies a solution to the readers-writers problem. Joung's solution uses only read/write primitives of shared memory. It produces high processor-to-memory traffic, making it less scalable. Keane and Moir [5] provide a more efficient solution to group mutual exclusion than Joung's. Their solution depends on the pre-existence of a fair "classical" mutual exclusion algorithm to implement their *acquire* and *release* operations. The algorithm also makes use of explicit local spinning or busy waiting to force processes to wait. Finally, the solution depends on using an explicit queue for waiting processes.

The solution presented in this paper is simpler, mainly because it solves a special case (readers-writers) of the more general problem (group mutual exclusion). We do not make use of explicit spinning. Given that semaphore operations can be efficiently built into an operating system using blocking instead of spinning, spinning can be altogether avoided in our solution. In this paper, we do not address the complexity of our algorithm, but it is obvious that it largely depends on the implementation of the semaphore and the underlying memory architecture (such as cache coherent or non-uniform memory access). The most widely used operating system books (for example, see [11,12,13]) still refer to the original unfair solutions of Courtois *et.al.* [1], without explicitly detailing a fair alternative. Our algorithm can be of high educational value when it is used to complement the original solutions.

In Section 2, the original solutions to the first and second problems are restated. Section 3 introduces our third readers-writers problem and solution. In Section 3, we show that our algorithm is correct by automatically verifying the required properties using the SPIN formal verifier [3]. Finally, Section 4 concludes the paper.

2. Previous Solutions

Given a group of processes portioned into readers and writers, a solution to the readers-writers problem must satisfy the following two properties:

- *Safety*: if there are more than two processes using the resource at the same time, then all of these processes must be readers.
- *Progress*: if there is more than one process trying to access the resource, then at least one process succeeds.

The first, second, and our third problem require different fairness properties. Courtois *et.al.* [1] state:

"For the first problem it [is] possible that a writer could wait indefinitely while a stream of readers arrived."

Hence, the first problem requires:

- *Fairness-1*: if some *reader* process is trying to access the resource, then this process eventually succeeds.

This property obviously favors readers and in the first problem there is no guarantee that a writer process does not starve. Similarly, the second problem favors writers. Courtois *et.al.* [1] require:

"In [the second] problem we give priority to writers and allow readers to wait indefinitely while a stream of writers is working."

Hence, the fairness requirement of the second problem is as follows:

- *Fairness-2*: if some writer process is trying to access the resource, then this process eventually succeeds.

The original solutions [1] to the first and second readers-writers problems are given in Figure 1 and Figure 2, respectively.

In Figure 1, if the first reader progresses to read the resource, it will block any potential writers until it is done. However, if a stream of readers keep on arriving, they may all skip the if statement in the entry section. Therefore, it is possible that each such reader never waits for `resource` and writers can be locked out indefinitely. A similar argument applies to the solution in Figure 2, but here writers can lock out readers.

3. The Third Problem

For highly demanded resources both the first and second solutions could be undesirable in practice. In this section, we present a solution that gives the readers and writers equal priorities.

The fairness requirement for our third problem is stronger than that of both Fairness-1 and Fairness-2 since it does not restrict the eventual progress of any process by its type (reader or writer).

Writer process

```
Repeat
      wait(mutex2);
      writers ← writers + 1;
      if writers = 1 then
            wait(read);
      end-if
      signal(mutex2);
      wait(write);

      // write the resource

      signal(write);
      wait(mutex2);
      writers ← writers - 1;
      if (writers = 0) then
            signal(read);
      end-if
      signal(mutex2);
until done
```

Reader process

```
repeat
      wait(mutex3);
      wait(read);
      wait(mutex1);
      readers ← readers + 1;
      if readers = 1 then
            wait(write);
      end-if
      signal(mutex1);
      signal(read);
      signal(mutex3);

      // read the resource

      wait(mutex1);
      readers ← readers - 1;
      if (readers = 0) then
            signal(write);
      end-if
      signal(mutex1);
until done
```

Fig. 2. Solution to the second readers-writers problem.

- *Fairness-3*: if *some process* is trying to access the resource, then this process eventually succeeds.

That is, Fairness-3 is defined as Fairness-1 *and* Fairness-2.

Our solution is given in Figure 3. The solution uses two integer variables `readers` and `writers` to respectively count the number of reader and writer process trying to gain access to the resource. Both of these variables are initialized to 0. Before a writer process gains access to the shared resource, the process increments the variable `writers` (line `W2`). After releasing the resource, the writer process decrements the variable `writers` (line `W6`). The same applies to reader processes and the variable `readers`. The algorithm makes use of two semaphores `mutex`, which is used to guarantee mutual exclusive access to the variables `readers` and `writers`, and `resource`, which is used to synchronize access to the shared resource.

Writers simply check the availability of the resource at line `W4`. If the resource is busy, the `wait(resource)` operation forces the writer to wait in the associated queue. A reader executes `wait(resource)` at line `R4` only if it is the first reader (`readers = 0`) trying to gain access to the shared resource, or if a writer process is trying to access the resource (`writers > 0` at line `R2`).

If the first reader is trying and the resource is available, the `wait(resource)` on line `R4` allows it to proceed locking out any following writers. All subsequent readers will skip the `if` statement (lines `R3` to `R5`) as long as there are no writers trying (`writers = 0`). Hence, the solution allows several readers to access the resource simultaneously. However, if a writer tries to use the resource, it will be forced to wait at line `W4`. The algorithm forces subsequent readers to execute the body of the if statement, forcing them to wait too (line `R4`). Eventually, all readers reading the shared resource will execute lines `R8` to `R12` and only the last such reader will execute line `R11`, allowing a waiting writer to proceed.

4. Proof of Correctness

In this section, we describe how we used the SPIN model checker [3] to verify the two properties of our algorithm: Safety and Fairness-3. Progress is implied by Fairness-3.

Writer process

```
Repeat
W1   wait(mutex);
W2   writers ← writers + 1;
W3   signal(mutex);
W4   wait(resource);

     // write the resource

W5   wait(mutex);
W6   writers ← writers - 1;
W7   signal(mutex);
W8   signal(resource);
until done
```

Reader process

```
Repeat
R1   wait(mutex);
R2   if writers > 0 or readers = 0 then
R3        signal(mutex);
R4        wait(resource);
R5        wait(mutex);
     end-if
R6   readers ← readers + 1;
R7   signal(mutex);

     // read the resource

R8   wait(mutex);
R9   readers ← readers - 1;
R10  if (readers = 0) then
R11       signal(resource);
     end-if
R12  signal(mutex);
until done
```

Fig. 3. Solution to the third readers-writers problem.

4.1. Assumptions

For the correctness of our algorithm, we assume the following:

- The execution is sequentially consistent. Lamport [6] requires for sequential consistency: “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”
- The execution either eventually terminates (the executing processes terminate and no new processes are admitted to the system) or, if it is infinite and there is at least one participating writer process, the execution continues indefinitely to have participating writer processes. That is, the Progress property requires that in an infinite execution with some participating writers, the execution does not come to a point where, from that point on, all the processes are indefinitely readers.

4.2. Formal Verification

Implementation of the `wait` and `signal` operations in Promela, SPIN’s programming language are given in Figure 5. Since Promela lacks constructs for blocking an active process, we must use busy waiting to delay the process. We choose to implement the `wait` and `signal` operations using Peterson’s n-process mutual exclusion algorithm [8], reproduced in Figure 4. That is, the `wait` operation is the code to enter a critical section and the `signal` is the exit code. The fairness of Peterson’s algorithm (a maximum fairness delay of $(n^2 - n)/2$) implies a fair semaphore implementation.

The Promela implementation of `wait(s,i)` in Figure 5 is an implementation of the `enter(i)` operation of Figure 4. The readers-writers solution of Figure 3, makes use of two semaphores, `mutex` and `resource`. The Promela integer constant `s` (0 or 1) identifies which semaphore the `wait(s,i)` is being invoked on. Hence, the variables `flag[i]`, `k`, `j`, and `turn[k]` of Figure 4 for process `i` and semaphore `s` are represented using the Promela variables `flag[s].val[i]`, `k[s].val[i]`, `j[s].val[i]`, and `turn[s].val[k[s].val[i]]`, respectively. The Promela `do-od` loop construct is used to represent for and while loops. The outer most do-od loop in Figure 5 corresponds to the for loop in Figure 4. The Promela statements

```
::  (k[s].val[i] == n-1) -> break
::  else ->
      flag[s].val[i] = k[s].val[i];
      turn[s].val[k[s].val[i]] = i;
      do ..
```

read: `if (k == n-1)`, then break the for loop otherwise assign `k` to `flag[i]`, assign `i` to `turn[k]`, and hence forth. The local variable `busy` and the inner most `do-od` loop represent a for loop implementation of the condition `∃j≠i:flag[j]` $\geq$ `k`. So the statements

```
do
::  (j[s].val[i] == n) -> break
::  else ->
      if
        ::  (i != j[s].val[i]) ->
              if
                ::  (flag[s].val[j[s].val[i]]
                      >= k[s].val[i])
                    -> busy = true; break;
```

read: when `j` reaches the value `n`, break the loop; otherwise, if there is a `j` $\neq$ `i`, where `flag[j]`

```
Shared variables:
     flag[1 ..  n] values in {0 ..  n-1}
     turn[1 ..  n-1] values in {1 ..  n}

enter(i):
     for k ← 1 to n-1 do
          flag[i] ← k
          turn[k] ← i
          while (turn[k] = i) and ∃j≠i:flag[j] ≥ k do skip

exit(i):
     flag[i] ← 0
```

Fig. 4. Peterson’s n-process mutual exclusion algorithm.

$\geq$ `k`, then set `busy` to true and break the for loop. The variable `busy` is checked to break the outer `do-od` loop corresponding to the while loop in Figure 4. Now, the rest of the Promela code should be readable for readers with even little background in programming.

The code for the readers and writers in Promela is given in Figure 6 and it mimics the pseudocode given in Figure 3. The Safety property is verified using the `assert` statement in the protected sections for each reader and writer process. The extra variables `inr` and `inw` are introduced to verify the safety property. They respectively represent the number of readers and writers engaged in the critical section.

In the reader process, SPIN asserts that the number of writers writing the resource, while a reader is reading, is zero, indicated by `assert(inw == 0)` in Promela.

In the writer process, SPIN asserts that the number of writer processes is one and the number of reader processes is zero, when a writer process is writing the resource, indicated by `assert(inw == 1 && inr == 0)`. SPIN's results (Figure 7) indicating that these properties are never violated, establishing the Safety property.

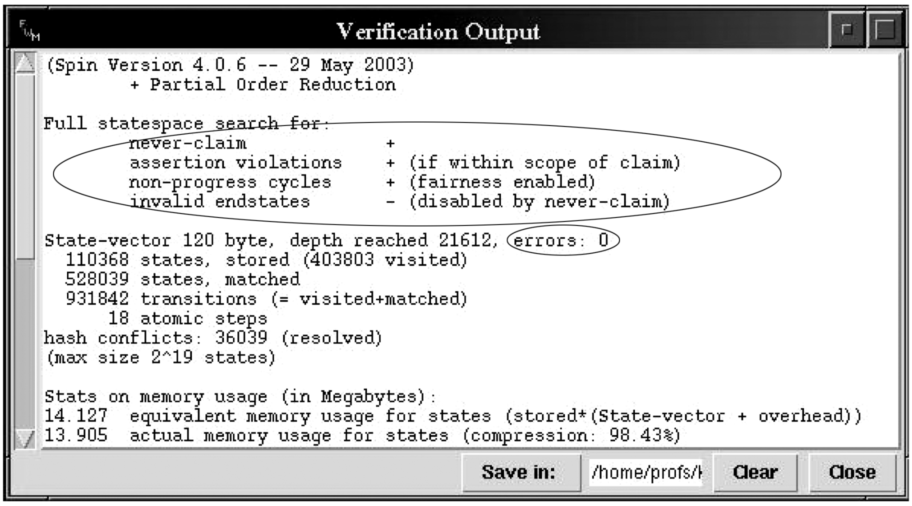
Fairness-3 is established using Promela's `progress` labels. SPIN checks for any scenario that violates the property that the progress-labeled instruction is always *eventually* reachable. There are two progress labels, one in the critical section of the writer and one in that of the reader process. SPIN's output indicates that both sections are always eventually reachable, establishing the Progress and Fairness-3 properties. Figure 7 shows a screen shot of SPIN's verification results.

```
inline wait(s,i){
   k[s].val[i] = 0;
   bool busy;
   do
      ::  (k[s].val[i] == n-1) -> break
      ::  else ->
         flag[s].val[i] = k[s].val[i];
         turn[s].val[k[s].val[i]] = i;
         do
            ::  (turn[s].val[k[s].val[i]] != i) -> break
            ::  else -> busy = false; j[s].val[i] = 0;
               do
                  ::  (j[s].val[i] == n) -> break
                  ::  else ->
                     if
                        ::  (i != j[s].val[i]) ->
                           if
                           ::  (flag[s].val[j[s].val[i]]
                                 >= k[s].val[i]) -> busy = true; break;
                           ::  else -> skip;
                           fi;
                        ::  else -> skip
                     fi;
                     j[s].val[i]++;
               od;
               if
                  ::  (!busy) -> break;
                  ::  else -> skip;
               fi;
         od;
         k[s].val[i]++;
   od;
}

inline signal(s,i){
      flag[s].val[i] = 0
}
```

Fig. 5. Wait and signal implementation in Promela.

```
proctype writer(int i) {
         do
            ::
               skip;
               wait(mutex,i);
               writers++;
               signal(mutex,i);
               wait(resource,i);

                  progress:  inw++;
                           assert(inr == 0 && inw == 1);
                           inw--;

               wait(mutex,i);
               writers--;
               signal(mutex,i);
               signal(resource,i)
         od
}
proctype reader(int i) {
         do
            ::
               skip;
               wait(mutex,i);
               if
               ::  ((writers > 0) || (readers == 0)) ->
                           signal(mutex,i);
                           wait(resource,i);
                           wait(mutex,i)
                  ::  else -> skip;
               fi;
               readers++;
               signal(mutex,i);

                  progress:  inr++;
                           assert(inw == 0);
                           inr--;

               wait(mutex,i);
               readers--;
               if
               ::  (readers == 0) -> signal(resource,i)
               ::  else -> skip;
               fi;
               signal(mutex,i);
         od
}
```

Fig. 6. The reader and writer processes of the third problem in Promela.



```
(Spin Version 4.0.6 -- 29 May 2003)
        + Partial Order Reduction

Full statespace search for:
        never-claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness enabled)
        invalid endstates       - (disabled by never-claim)

State-vector 120 byte, depth reached 21612, errors: 0
  110368 states, stored (403803 visited)
  528039 states, matched
  931842 transitions (= visited+matched)
      18 atomic steps
hash conflicts: 36039 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
14.127  equivalent memory usage for states (stored*(State-vector + overhead))
13.905  actual memory usage for states (compression: 98.43%)
```

Fig. 7. SPIN's verification output for the third problem.

5. Conclusion

This paper introduced a new semaphore-based solution to the readers-writers concurrency problem. Previous specialized solutions either (a) did not permit more than one reader to simultaneously access the resource, (b) permitted readers to indefinitely lock out writers, (c) or permitted writers to indefinitely lock out readers. None of these solutions is practically appealing and our solution answers all of their limitations. There are, however, recent solutions to a more general problem, the group mutual exclusion problem. Our solution is a simpler solution to a simpler problem (the readers-writers problem versus the group mutual exclusion problem). It also has an educational value if the widely quoted unfair solutions in famous operating systems text books are supplemented with it.

We followed an automatic verification approach to prove the correctness of our algorithm, using the state-of-the-art SPIN model checker with the Promela programming language. We believe that the use of SPIN to establish the correctness of our algorithm is of an independent interest and deserves the attention given in this paper. This also can serve teaching purposes, especially at the undergraduate level, where students studying operating systems typically do not have the necessary background to construct formal proofs of correctness for concurrent algorithms.

Because our solution is extremely fair, it is possible, under certain circumstances, that only one process at a time is allowed to access the resource. This can take place when both readers and writers are lining up to use the resource. Precisely, if streams of writers and readers exist, the readers and the writers will be forced to wait on semaphore `resource`. When a process (reader or writer) exits `signal(resource)` must be executed. (In the case of a reader process, the stream of readers will be blocked in entry because `writers` $>$ `0` and the last reader exiting the protected section will execute the `signal(resource)` operation). The next waiting process will be allowed to proceed, regardless of its type. If such a process is a reader, it will be the only reader process accessing the resource at that time, even if the next waiting process is also a reader. It may be more efficient to allow more than one reader to proceed with simultaneous reading. However, it is not clear to us how this could be achieved without indefinitely locking writers out. We are currently investigating if it is possible to optimize the algorithm to behave more efficiently in such a situation. Furthermore, we would like to consider the complexity implications of our algorithm.

6. Acknowledgments

We are thankful to the anonymous reviewers for their comments, which helped us improve the paper.

References

[1] P. J., Courtois, F. Heymans, and D. L. Parnas, Concurrent Control with ‘Readers’ and ‘Writers’, *Communications of the ACM* 14(10):667–668, 1971.

[2] E. Dijkstra, Cooperating Sequential Processes, in F. Genuys, editor, *Programming Languages*, Academic Press, 1968.

[3] G. J. Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering* 23(5):1–5, 1997.

[4] Y. J. Joung, Asynchronous Group Mutual Exclusion, in *Proc. 17thACM Symp. Principles of Distributed Computing*, pp. 51–60, 1998.

[5] P. Keane and M. Moir, A Simple Local-Spin Group Mutual Exclusion Algorithm, *IEEETtrans. Parallel and Distributed Systems* 12(7):673–685, 2001.

[6] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess programs, *IEEE Transactions on Computers* C-28(9):690–691, 1979.

[7] C. Langris and E. Moutzoukis, A Batch Arrival Reader-Writer Queue with Retrial Writers, *Commun. Statist, Stochast. Models*, 13(3):523–545, 1997.

[8] G. Peterson, Myths About the Mutual Exclusion Problem, *Parallel Processing Letters* 12(3):115–116, 1981.

[9] M. Raynal, *Algorithms for Mutual Exclusion*, The MIT Press, 1986.

[10] T. Sanli and V. Kulkarni, Optimal Admission to Reader-Writer Systems with no Queuing, *Operations Research Letters* 25:213–218, 1999.

[11] A. SILBERSCHATZ AND P. GALVIN, *Operating System Concepts 5th ed.*, Wiley, 1999.

[12] W. STALLINGS, *Operating Systems 4th ed.*, Prentice Hall, 2001.

[13] A. S. TANENBAUM AND A. S. WOODHULL, *Operating Systems Design and Implementation 2nd ed.*, Prentice Hall, 1997.

[14] E. XU AND A. S. ALFA, A Vacation Model for the Non-saturated Readers and Writers with a Threshold Policy, *Performance Evaluation* 50:233–244, 2002.

Received: November, 2003
Accepted: September, 2004

Contact address:
Jalal Kawash
Department of Computer Science
American University of Sharjah
P.O.Box 2666
Sharjah
UAE
e-mail: `jkawash@ausharjah.edu`

JALAL KAWASH received his Ph.D. from The University of Calgary, Canada in 2000. He is currently an assistant professor of computer science at the American University of Sharjah, UAE and an adjunct assistant professor at The University of Calgary, Canada. His research interests are in distributed systems and algorithms, Internet computing, and computing education.