

# Conflict Resolution for Readers and Writers

Gertrude Levine  
Computer Science Department  
Fairleigh Dickinson University  
levine@alpha.fdu.edu

## 1. Introduction

For those of us who have been teaching Operating Systems for years, you may remember with nostalgia the days when Operating Systems texts [ex: Deitel 1984] included Ada code to illustrate solutions for concurrency problems. I find concurrency the most difficult topic to teach in Operating Systems, and expressing solutions in Ada is a strong teaching aid. Ada's tasking construct is powerful and readable<sup>1</sup>. In addition, an Ada Programming Support Environment enables hands-on experimentation.

The Readers/Writers problem, a classic mutual exclusion synchronization problem, puzzled computer science professionals for years. Explaining these solutions, as presented in current Operating Systems texts, is daunting.

We contrast code expressed in C to that proposed by Burns [Burns1985] and Barnes [Barnes1995] for readability. We then suggest a slight modification to obtain what we consider to be a preferred solution, one that illuminates concurrency issues in general.

## 2. The Problem

The Readers/Writers Problem has multiple Readers and Writers accessing a shared database. A Writer must have mutually exclusive access to the database (be in the critical section by itself) to guarantee data consistency, while multiple Readers can overlap in their access to the data without any consistency issues. We are required to promote concurrency by allowing Readers to enter the critical section at overlapping time intervals with other Readers. Such an effort, however, raises the possibility that Readers will keep arriving and block the critical section from Writers, so that waiting Writers may be queued for unbounded periods of time.

On the other hand, solutions exist in which Writers have priority. New Readers are not accepted into the critical section if a Writer is waiting. Thus, waiting Readers can be indefinitely postponed while new Writers keep arriving and are given preference to the critical section.

The first solution, in which Readers are given higher priority, assumes that maximum throughput is the desired goal. Allowing multiple Readers rather than a single Writer is thus preferable. The second solution, in which Writers are given higher priority, assumes that Readers would prefer to get the most current information. Thus, throughput is sacrificed, since arriving Readers cannot overlap with current Readers if a Writer is queued for service. Newly arriving Writers are allowed to proceed into the database as soon as current Readers and earlier arriving Writers complete, and before waiting Readers.

No process should be subject to unbounded waits, however, if we agree that solutions to the critical-section problem should satisfy the following three requirements [Silberschatz 1998]:

- 1) Mutual Exclusion: at most a single writer is executing within a critical section that accesses a shared resource – i.e., processes' outputs to a shared resource cannot be interleaved with other

---

<sup>1</sup> Protected types are not used in this paper because we found other tasking constructs to be more appropriate for illustrating different mechanisms for resolving conflicts.

- writers or readers, thus guaranteeing resource consistency.
- 2) Progress: a process is not postponed from entering its critical section by processes that are neither waiting for nor are within their critical section (for a shared resource).
  - 3) Bounded Waiting: there exists a bound on the number of times a process is postponed from entering its critical section by the arrival of new processes competing for the shared resource.

Solutions to the Readers/Writers problem that satisfy the third criteria have also been developed. One, using the monitor construct [Hoare 1974], also requires that newly arriving Readers remain queued if there is a waiting Writer. As each Writer completes, however, it signals currently waiting Readers to allow them to be serviced in turn.

### 3. Solutions Giving Readers Preference

This section contains examples of solutions to the Readers/Writers problem in which Readers are given higher priority. A Writer cannot enter its critical section while any Readers are being serviced, yet Readers that may be continually arriving are accepted into the critical section as long as another Reader is in it. Thus, during high traffic of arriving Readers, Writers may be queued for an unbounded period.

In C-type code [Stallings2001 pp.249]:

```

/*program readersandwriters*/
int readcount;
semaphore x =1, wsem = 1;
void reader()
{
    while (true)
    {
        wait(x);
        readcount++;
        if (readcount== 1)
            wait (wsem);
        signal (x);
        READUNIT ();
        wait (x);
        readcount--;
        if (readcount== 0)
            signal (wsem);
        signal (x);
    }
}

void writer ()
{
    while (true)
    {
        wait (wsem);
        WRITEUNIT();
        signal (wsem);
    }
}

void main ()
{
    readcount = 0;
    parbegin (reader, writer);
}

```

Solutions in Ada are found in Burns [Burns 1985 pp. 116-117] and Barnes [Barnes 1995 pp.414-415]. The solutions are essentially the same, and we combine them below:

```
generic
  type ITEM is range <>;
package READERS_WRITERS is
  procedure READ (I : out ITEM);
  procedure WRITE (I : in ITEM);
end READERS_WRITERS;

package body READERS_WRITERS is

  SHARED_ITEM: ITEM;      -- Should be generalized for a database of values.

task CONTROL is
  entry START_READ;
  entry STOP_READ;
  entry WRITE (I: in ITEM);
end CONTROL;

task body CONTROL is
  READERS: NATURAL := 0;
begin
  accept WRITE (I: in ITEM) do      -- Data is written before it is read.
    SHARED_ITEM := I;
  end WRITE;
  loop
    select
      accept START_READ;
      READERS := READERS + 1;
    or
      accept STOP_READ;
      READERS := READERS - 1;
    or
      when READERS = 0 =>
        accept WRITE (I: in ITEM) do
          SHARED_ITEM := I;
        end WRITE;
    or
      terminate;
    end select;
  end loop;
end CONTROL;

procedure READ (I: out ITEM) is
begin
  CONTROL.START_READ;
  I := SHARED_ITEM;
  CONTROL.STOP_READ;
end READ;

procedure WRITE (I: in ITEM) is
begin
  CONTROL.WRITE(I);
end WRITE;
```

```
end READERS_WRITERS;
```

Barnes also recommends changing the READ procedure into a function, and placing it, together with the WRITE procedure, within a protected unit. Calls to task CONTROL, together with the overhead of context switching, will then be eliminated. Indeed, if we need to give Readers higher priority, this method would be preferred.

In contrasting the C code with Ada, we see, of course, that Ada's high level tasking construct guarantees correct access to the critical section. The correct use of semaphores, on the other hand, is left up to the programmer and can thus result in various types of misuse. For this paper, however, we are concerned with the clarity of the solution for educational purposes. Note that the Ada solution frees the reader from low-level details of mutual exclusion synchronization.

#### 4. Solutions in Which Writers are Given Preference:

Newly arriving Readers can be postponed until there are no waiting Writers, by the use of a flag that is flipped when a Writer arrives. Any Readers in the critical section will complete, but no new Readers will be accepted if a Writer is waiting. If Writers keep arriving, then Readers' waits may be unbounded.

This solution is given in C-like code [Stallings 2001 p. 251].

```
/*program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader ()
{
    while (true)
    {
        wait (z);
        wait (rsem);
        wait (x);
        readcount++;
        if (readcount == 1)
        {
            wait (wsem);
        }
        signal (x);
        signal (rsem);
        signal (z);
        READUNIT ();
        wait (x);
        readcount--;
        if (readcount == 0)
            signal (wsem);
        signal (x);
    }
}
void writer()
{
    while (true)
    {
        wait (y);
        writecount++;
        if (writecount == 1)
```

```

        wait (rsem);
        signal (y);
        wait (wsem);
        WRITEUNIT();
        signal (wsem);
        wait (y);
        writecount--;
        if (writecount == 0)
            signal (rsem);
        signal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}

```

We tried to explain the “C” solution to our top students. We faltered on several issues, including correctness.

In Ada, again combining versions of Barnes [Barnes 1995 p.416] and Burns [Burns 1985 pp.116-117]:

```

-- Precondition: callers to the START_READ entry cannot abort their calls.
-- This requirement is necessary because we rely on the count attribute (as
-- do Barnes and Deitel), which is not reevaluated if a waiting Reader aborts
-- its call.

```

```

generic
    type ITEM is range <>;
package READERS_WRITERS2 is
    procedure READ (I : out ITEM);
    procedure WRITE (I : in ITEM);
end READERS_WRITERS2;

package body READERS_WRITERS2 is
    SHARED_ITEM: ITEM;
    task CONTROL is
        entry START_READ;
        entry STOP_READ;
        entry WRITE (I: in ITEM);
    end CONTROL;

    task body CONTROL is
        READERS: NATURAL := 0;
    begin
        accept WRITE (I: in ITEM) do
            SHARED_ITEM := I;
        end WRITE:
        loop
            select
                when WRITE'COUNT = 0 =>
                    accept START_READ;
                    READERS := READERS + 1;
            or
                when READERS = 0 =>

```

```

        accept WRITE (I: in ITEM) do
            SHARED_ITEM := I;
        end WRITE;
    or
        accept STOP_READ;
        READERS := READERS - 1;
    or
        terminate;
    end select;
end loop;
end CONTROL;

procedure READ (I: out ITEM) is
begin
    CONTROL.START_READ;
    I := SHARED_ITEM;
    CONTROL.STOP_READ;
end READ;

procedure WRITE (I: in ITEM) is
begin
    CONTROL.WRITE (I);
end WRITE;

end READERS_WRITERS2;

```

We chose the count attribute for reasons of simplicity, readability, and its specificity in signaling conflict. Burns' method uses a flag that is set by queued Writers requesting service.

### 5. A Solution in Which Sets of Readers Alternate With a Writer During Contention

We propose a minor modification of the above code in order to guarantee that all Readers and Writers are supplied access to their critical sections within a bounded interval of time (obviously dependent upon the volume of traffic).

```

-- Precondition: callers to the START_READ entry cannot abort their calls.
generic
    type ITEM is range <>;
package READERS_WRITERS3 is
    procedure READ (I : out ITEM);
    procedure WRITE (I: in ITEM);
end READERS_WRITERS3;

package body READERS_WRITERS3 is
    SHARED_ITEM: ITEM;
    task CONTROL is
        entry START_READ;
        entry STOP_READ;
        entry WRITE (I: in ITEM);
    end CONTROL;

    task body CONTROL is
        type TURNS is (READING, WRITING);
        TURN: TURNS := WRITING;
        READERS: NATURAL := 0;
    begin

```

```

accept WRITE (I: in ITEM) do
    SHARED_ITEM := I;
end WRITE;
loop
    select
        when WRITE'COUNT = 0 or TURN = READING =>
-- If Readers keep arriving without end and implementation keeps accepting
-- this entry, neither Readers nor Writers will complete, but that was true
-- in the second solution as well.
            accept START_READ;
            READERS := READERS + 1;
        or
            when READERS = 0 =>
                accept WRITE (I: in ITEM) do
                    SHARED_ITEM := I;
                end WRITE;
                TURN := READING;
            or
                accept STOP_READ;
                READERS := READERS - 1;
                TURN := WRITING;
            or
                terminate;
        end select;
    end loop;
end CONTROL;

procedure READ (I: out ITEM) is
begin
    CONTROL.START_READ;
    I := SHARED_ITEM;
    CONTROL.STOP_READ;
end READ;

procedure WRITE (I: in ITEM) is
begin
    CONTROL.WRITE (I);
end WRITE;

end READERS_WRITERS3;

```

As before, a Writer can be accepted into the critical section if there are no Readers in that section at that time. Once completed, it signals the Readers' turn. A Reader can be accepted either if there are no waiting Writers or if it is the Reader's turn. Note that the first Reader completing (and typically all waiting Readers, since most implementations accept all tasks queued on the same entry before checking other entries) flips the turn back for Writers, so that Writers are not postponed indefinitely.

The suggested solution uses Dekker's logic [Dijkstra 1965] for the critical section problem. During light traffic, a Reader(s) or a Writer is accepted into the critical section as it arrives. When conflict does occur, Readers and a Writer take turns accessing the shared data. Thus the third solution can be used to illuminate other concurrency problems.

We recognize that all multiple-access (scheduling, routing, search, etc.) protocols can be classified as of three types [Levine 1989] based on the order that they follow to supply resources.

- 1) Time controlled orders, such as followed by random access, sequential access, FIFO, and FCFS schemes, are appropriate for light access traffic.
- 2) Process/resource controlled orders, such as used by FDM and resource pre-allocation mechanisms, are appropriate for uniformly high traffic. So are TDM, round robin, and traffic lights, although they modify the process/resource order by turns.
- 3) For non-uniform traffic distribution, conflict (as signaled in Ada by the count attribute) is used to alternate between ordering schemes. We can choose to provide preferential treatment (as the WRITE'COUNT attribute does to halt Readers, similar to the use of stop signs) or fair treatment (as the use of a flag does to assist Readers during heavy traffic, similar to the use of traffic lights). In the latter case, when the WRITE'COUNT attribute evaluates to 0, signaling no contention, Readers do not have to wait for the flag to signal their turn (similar to Dekker's and a Right Turn on Red).

Conflict resolution for cooperation synchronization clearly is based on only the first two types of mechanisms. This type of conflict requires the abortion of one of the request sets, not an ordering to resolve the contention.

## 6. Summary

We have presented several versions of the Readers/Writers problems, chiefly for the purpose of championing the use of Ada for the illustration of concurrency issues. We have suggested a minor modification of previously published solutions to satisfy requirements for the use of a critical section. This suggestion is not original. The implementation with Ada's count attribute, however, is seen to be particularly useful in illuminating conflict resolution in mutual exclusion concurrency control.

## REFERENCES:

- Barnes, J. *Programming in Ada95*, Addison-Wesley Publishing Co., Wokingham, 1995.
- Burns, A., *Concurrent Programming in Ada*, Cambridge University Press, Cambridge, 1985.
- Dijkstra, E. W., "Cooperating Sequential Processes," Technological University, Eindhoven, Netherlands, 1965.
- Deitel, H.M., *An Introduction to Operating Systems*, Addison-Wesley Publishing Co., Reading, 1984.
- Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *CACM*, 17(10) October 1974, 549-57.
- Levine, G. "The Control of Starvation," *International Journal of General Systems*, 15, 1989, 113-127.
- Silberschatz, A. and Galvin, P. *Operating System Concepts, 5<sup>th</sup> edition*, Addison-Wesley, Reading, Mass. 1998.
- Stallings, W. *Operating Systems*, Prentice Hall, Upper Saddle River, NJ, 2001.