On a “Theorem” of Peterson

Leslie Lamport
SRI International

31 October 1984

1 Introduction

In [2], Peterson considers the problem of allowing concurrent reading and writing of a data item by keeping multiple copies of the data. Theorem 3 of that paper asserts that with n readers and a single writer, $n+1$ copies of the data are needed if the writer need not wait for the readers and the readers wait only for the writer. In this paper, I present an algorithm that appears to contradict Peterson's theorem, using only two buffers for any number of readers. Whether this is a counterexample depends upon the interpretation of "readers wait only for the writer", which was not defined in [2] (hence the quotation marks of the title). This illustrates the need for rigor when reasoning about concurrent programs.

2 The Algorithm

Briefly stated, the problem addressed by Peterson is to allow concurrent reading and writing while assuring that a reader always obtains a correct value, meaning either the last value written by the writer before the read operation or a value written during the read. Moreover, if one read is completed before a second read is begun, then the first read cannot obtain a later value than the second. A future paper will describe the problem more rigorously, using the formalism of [1]. However, here I will be as informal as Peterson and pretend that my algorithm is also, in the words of [2], "sufficiently simple that there is no need to provide complicated ... formal proofs."

The algorithm uses Solution 2 of [2] in which, using one copy of the data plus a few flags, the writer never waits but the readers can be starved by repeated writing. For any data item x, let $\mathit{read.of}(x)$ and $\mathit{write.of}(x)$ denote the reading and writing operations in an instance of Peterson's Solution 2 for that item. The current value of the data is kept in one of the two buffers $\mathit{buff}[0]$ and $\mathit{buff}[1]$, where the value of the variable num indicates which. Initially, $\mathit{num} = 0$ and $\mathit{buff}[0]$ has the starting data value.

The basic idea of the algorithm is that a reader first reads num to see which buffer contains the current value, then reads that value. The presumed correctness of the *read.of* and *write.of* operations guarantees that if the reader ever terminates, then it will obtain a correct value—either one that was "current" when it started reading or else a more recent value written while it was reading. The writer keeps using the same buffer unless there is no other reader currently reading from the other buffer, in which case it switches to the other buffer. The writer changes *num* after writing to prevent any new read operation from using that other buffer before the new value is written.

The algorithms for the read and write operations are given below. I assume that there are n readers, numbered from one through n. Reader i uses the the Boolean flag $reading[i]$ to indicate that it is currently reading, and uses $rdbuf[i]$ to hold the value of the buffer that it is trying to read. The symbol $\oplus$ denotes addition modulo 2, and the writer's **if** condition has the value *true* if the writer finds that any reader is currently reading buffer number $num \oplus 1$.

Algorithm for the ith reader:

$rdbuf[i] := num;$
$reading[i] := true;$
$read.of(rdbuf[i]);$
$reading[i] := false$

Algorithm for the writer:

if $\bigvee_{j=1}^{n}(reading[j] \wedge (rdbuf[j] \neq num))$
    then $write.of(buff[num])$
    else $write.of(buff[num \oplus 1]);$
        $num := num \oplus 1$
fi

The argument given above shows that any read that terminates does so with a correct value. To see that every read does eventually terminate, assume to the contrary that reader i is performing a nonterminating *read.of* operation. Since the writer is never blocked, it must eventually be outside its write operation. Suppose that, at that time, $num \neq rdbuf[i]$. Then the writer will never attempt to read buffer $rdbuf[i]$, since it will always see that reader i is currently reading buffer $num \oplus 1$. By the assumed correctness of the *read.of* and *write.of* operations, this implies that reader i's *read.of* operation will eventually terminate, which is a contradiction.

Next, we consider the other possibility: $num = rdbuf[i]$. If the writer ever changes *num*, then the above argument shows that reader i will terminate. Hence, we may assume that *num* remains forever equal to $rdbuf[i]$. This means that any reader currently reading buffer $num \oplus 1$ must eventually terminate, and all newly initiated read operations will use buffer *num*. Hence, eventually there will never be any more reads being performed to buffer $num \oplus 1$. If the writer were ever to initiate a new write, it would change *num*. We therefore conclude that the writer never begins any new writes. The correctness of the *read.of* operation implies that reader i's operation must eventually terminate, which is the required contradiction.

3 Discussion

In my algorithm, a read will terminate if the writer waits long enough before initiating a new write, regardless of what other readers do. In this sense, a reader waits only for the writer. However, in the presence of repeated writes, a reader trying to read from a buffer may not be able to do so until other readers have finished reading from the other buffer. Thus, in the presence of repeated writes, a reader can be blocked if another reader fails to make progress. In this sense, the reader may be regarded as waiting for the other reader as well as for the writer.

Whether or not the algorithm contradicts Peterson's theorem is a question that could be debated by lawyers. The validity of a theorem in computer science should be a question of mathematics, not law. A more rigorous treatment of the subject is needed.

References

[1] Leslie Lamport. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. on Prog. Lang. and Systems 1*, 1 (July 1979), 84-97.

[2] Gary L. Peterson. Concurrent Reading While Writing. *ACM Trans. on Prog. Lang. and Systems 5*, 1 (January 1983), 46-55.