ERRATA
Lock-Free Linked Lists Using Compare-and-Swap

John D. Valois*

November 7, 1995

1 Introduction

In [4], we describe algorithms that implement a singly-linked list data structure in a *lock-free* manner; i.e., the list can be manipulated concurrently by many processes without the use of critical sections. Maged Michael of the University of Rochester has discovered an error in these algorithms [2] which can cause memory to be reclaimed incorrectly.

2 The Error

The error is not with the list manipulation algorithms per se, but rather with the algorithms that manage the memory used for the linked list nodes.

Each node contains two fields: a reference count and a *claim bit*. The reference count is used to ensure that a node that is deleted from the list is not returned to the free memory pool while other processes still have pointers to the node. The reference counts are maintained by two algorithms, SafeRead and Release, which are used when pointers in the list are read and when a process is finished using a pointer, respectively.

With concurrency, it is possible that two or more processes may see the reference count in a node become zero. This can happen because the reading of a pointer and the incrementing of the reference count are not atomic. Since errors would result if more than one process simultaneously tried to return the node to the free memory pool, the claim bit is used to ensure that only one process will reclaim the node.

*Dept. of Comp. Sci., Rensselaer Polytechnic Institute, Troy, NY 12180. Email: valoisj@cs.rpi.edu

This part of the algorithm is incorrect. The original algorithm the RELEASE operation is reproduced below; at line 4, the claim bit is used to prevent more than one process from reclaiming the a node with no more pointers to it.

Suppose two processes concurrently see the reference count go to zero (i.e., the test at line 2 fails), and that one of the processes, say process p, is stalled before executing line 4, while the other process executes the remainder of the algorithm and reclaims the node. The node is now in the free memory pool, and can be reallocated; when it is, its claim bit is reset to zero as part of the allocation algorithm. If process p now wakes up and executes line 4 of the RELEASE algorithm, is will incorrectly return the node (which is in use) to the free pool.

```
RELEASE(p : pointer)

1    c ← FETCH&ADD(p^.refct, −1)
2    if c > 1 then
3       return
4    c ← TEST&SET(p^.claim)
5    if c = 1 then
6       return
     else
7       RECLAIM(p)
```

3 A Corrected Algorithm

Several solutions to this problem are possible. A simple solution is to modify the SAFEREAD algorithm so that it does not increment a reference count that is already zero (this is easily done using the COMPARE&SWAP primitive). This change eliminates the possibility of more than one process seeing the reference count become zero, and thus no claim bit is needed. However, the SAFEREAD and RELEASE algorithms are performance sensitive, and the above solution introduces a loop which, especially with high levels of concurrency, can degrade performance.[1]

Another solution is suggested in [2]. In this solution the reference count and claim bit

[1] On some architectures (e.g., MIPS) the FETCH&ADD primitive must be simulated using a loop anyway, and so the performance degradation may not be as apparent.

are combined into the same word of memory, and the RELEASE algorithm is modified so that the decrementing of the reference count and the test-and-setting of the claim bit are done atomically. While this eliminates the error, it also introduces a loop into the algorithm.

A third solution is possible which does not introduce any loops. Like the second solution above, the claim bit is combined with reference count (the claim bit will be the low-order bit of the reference count).

```
RELEASE(p : pointer)

1    c ← FETCH&ADD(p^.refct, −2)
2    if c ≠ 3 then
3      return
4    if COMPARE&SWAP(p^.refct, 1, 0) then
5      RECLAIM(p)
```

Note: a corresponding change is needed in the SAFEREAD algorithm, so that it increments the reference count by 2 rather than 1, and small changes are needed in the code that resets the claim bit when a node is allocated from the free pool.

In addition to being combined with the reference count, the meaning of the claim bit has also changed. In the original algorithm, a value of zero for the claim bit indicated that the node could be reclaimed by any process that had seen a reference count of zero for the node, while a value of one indicated that the node was not currently in use. In the new algorithm, a value of one for the claim bit indicates the node is in use while a value of zero indicates it is not.

During a RELEASE operation, the combined reference count/claim bit just before it is decremented at line 1 is in one of three states:

1. $refct > 3$ — there is at least one other pointer to the node.
2. $refct = 3$ — there is no other pointer to the cell, and the node is in use. It should be returned to the free memory pool.
3. $refct = 2$ — there is no other pointer to the cell, and the node is not in use.

A process should try to return the node to the free memory pool only in case 2 above. (Note that more than one process can see this simultaneously). To ensure that only one

process can do this, COMPARE&SWAP is used at line 4 in the algorithm to turn off the claim bit.

4 Conclusion

Although the original algorithms were formally verified as well as actually implemented and tested [3], the error described here was not detected. Formal verification did not identify the error because the assumption was made, in order to simplify the proofs, that memory was not reused; hence, the error could not occur. The error was not discovered during implementation because, like many bugs in concurrent programs, it occurs extremely rarely. The error was finally discovered during experiments involving several billion executions of the algorithms [1].

References

[1] M. M. Michael. Private communication, November 1995.

[2] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report 599, University of Rochester, December 1995.

[3] J. D. Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[4] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Symposium on Principles of Distributed Computing*, pages 214–222, 1995.