Speeding Lamport's Fast Mutual Exclusion Algorithm*

Michael Merritt† Gadi Taubenfeld†

Abstract

A linked list is used to speed up a mutual exclusion algorithm. This optimization permits additional concurrency by allowing scans of the list to be concurrent with insertions and deletions of list entries.

1 Introduction

In [Lam87], Lamport presents a fast mutual exclusion algorithm, with time complexity — the number of accesses to the shared memory in order to enter and exit a critical section in the absence of contention — of only seven memory accesses. Figure 1 in Section 2 presents Lamport's algorithm. This algorithm assumes that each of n potentially contending processes has a unique identifier taken from $\{1, ..., n\}$, and that the only atomic operations are reads and writes. In particular, test-and-set operations are not supported.

Lamport's algorithm provides fast access in the absence of contention, however, in the presence of contention (of even just two processes), the winning process may have to check the status of all other n processes before it is allowed to enter its critical section. (In the **for** loop that scans the array $b[1..n]$.) That is, even if the winning process never has to busy-wait, it still may need to access the shared memory more than n times.

Since the other contending processes are waiting for the winner, it is particularly important to speed its entry to the critical section. That is, to design a *contention sensitive mutual exclusion* algorithm in which the time it takes for the winning process to enter (and exit) its critical section, since the last time some process exited its critical section, is a function of the actual number of contending processes.

Note that n is a constant in Lamport algorithm. In a typical implementation, the actual number of contending processes varies over time. Typically, tables such as the array $b[1..n]$ are statically allocated, and n is chosen as a pessimistic upper bound on the total number of potentially contending processes, although the expected number of such processes may be several orders of magnitude smaller.

This is the case in the TUXEDO[1] system [TUX90].[2] In this system the pessimistic upper bound on the total number of potentially contending processes is more than thirty thousand, while the expected number of such processes is less then a dozen. Lamport's algorithm is

*This paper is a revision of an AT&T Technical Memorandum, May 1991.

†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

[1]The TUXEDO Transaction Processing System provides applications designers and programmers with a framework for building on-line transaction processing applications in a distributed computing environment.

[2]TUXEDO and UNIX are registered trademarks of UNIX System Laboratories.

used when TUXEDO is run on machines – like the MIPS 3000 series processors – that do not support an atomic test-and-set operation [Mac91].

We describe a performance optimization of Lamport's algorithm, which exploits a distinction between *active* and *inactive* processes. Inactive processes may not contend for the shared resource. However, they can execute initialization code and become active. Similarly, active processes may become inactive by executing appropriate exit code. In our optimization, only the status of active processes must be examined, using a linked list. Processes become active by inserting themselves into the list, and become inactive by deleting themselves. (The notion of active process is made precise in Section 5.) While insertions and deletions cannot be done concurrently, the optimization permits additional concurrency by allowing scans of the list to be concurrent with insertions and deletions. The resulting algorithm is sensitive to the actual number of processes in the system and solves mutual exclusion efficiently in contexts in which process creation and destruction are rare.

The concluding section raises several open problems related to this optimization and mutual exclusion.

2 Lamport's algorithm

Figure 1 below presents Lamport's algorithm [Lam87]. The algorithm is deadlock-free but allows starvation of individual processes. (Lamport argues that deadlock-freedom is essential, but that starvation-freedom is not, since contention for a critical section should be rare in well designed systems.)

---

x, y: *integer*, initially $y = 0$
$b[1..n]$: *boolean array*, initially *false*
start: $b[i] := \mathit{true}$;
$x := i$;
if $y \neq 0$ **then** $b[i] := \mathit{false}$;
await $y = 0$;
goto *start* **fi**;
$y := i$;
if $x \neq i$ **then** $b[i] := \mathit{false}$;
for $j := 1$ **to** n **do await** $\neg b[j]$ **od**;
if $y \neq i$ **then await** $y = 0$;
goto *start* **fi fi**;
critical section;
$y := 0$;
$b[i] := \mathit{false}$

Figure 1: Lamport's Fast Mutual Exclusion Algorithm – process i's program.

In the algorithm, process i first sets x to i, and then checks the value of y. When it finds $y = 0$, it sets y to i and then checks the value of x. Let us say that process i enters its critical section along path α if it finds $x = i$ at this step. At most one process can enter

its critical section along path α. If a process finds $x \neq i$ then it delays itself by looping until it sees that all the bits in the arrary b are false. Checking these n bits plays two roles: (See [Lam87] for a complete correctness proof.)

1. Say that process i enters its critical section along path β if it finds $y = i$ after exiting the for-loop. A consequence of having observed all the bits to be *false* is that the value of y will not be changed thereafter until process i leaves the critical section. This follows because every other contending process either reads $y \neq 0$ in the third line and is looping in the first three statements, or reads $y = 0$ and then finished the assignment $y := j$ in the sixth line before setting $b[j]$ to *false*. Hence, once process i finds $y = i$ after the loop, no other process can change the value of y until process i sets y to 0 in its exit code. It follows that at most one process can enter along path β.

2. The for-loop ensures that if a process enters the critical section along path α, then any other process is prevented from entering along path β. To see this, observe that when a process j enters its critical section along path α, its bit $b[j]$ remains *true*. Thus, if another process tries to enter along path β it will find that $b[j]$= *true* and will have to wait until process j exits its critical section and sets $b[j]$ to *false*.

3 The modified algorithm

The optimization is to replace the scan through array $b[1..n]$ by maintaining a linked list of the active processes. Instead of checking the status of all processes, a process only checks the status of the processes in the list. The modified algorithm is shown in Figure 2.

```
start:  b[i] := true;
        x := i;
        if y ≠ 0 then b[i] := false;
                      await y = 0;
                      goto start fi;
        y := i;
        if x ≠ i then b[i] := false; next := list[0]; /* notice that next ≠ 0 */
                      repeat await ¬ b[next]; next := list[next]; /* scan the list */
                      until next = 0;
                      if y ≠ i then await y = 0;
                                    goto start fi fi;
        critical section;
        y := 0;
        b[i] := false
```

Figure 2: The modified Mutual Exclusion Algorithm – process i's program.

The linked list is implemented using a shared array *list*$[0..n]$, where the first entry *list*$[0]$ is the header of the list. The variable *next* is an additional *local* variable that is used as a pointer into the array. The range of *next* and of the entries of *list* is 0 through n, and all are initialized to 0.

As in Lamport's algorithm, the variable y is initialized to 0, and all entries in the boolean array $b[1..n]$ are initially *false*.

4 Maintaining the list

Of course, the linked list of active processes must be maintained correctly. At first examination, this would seem to require mutual exclusion between the different processes accessing the list, bringing the problem full circle. But we observe that if processes are inserted and deleted using the procedures in Figure 3, then processes may safely scan the list concurrently with inserts and deletes. These procedures presuppose a mechanism for maintaining mutual exclusion between inserts and deletes (the *new critical section*). This mechanism is discussed below.

```
Insert(i)
  enter new critical section:
        find predecessor;
        list[i] := list[predecessor];
        list[predecessor] := i;
  leave new critical section:

Delete(i)
  enter new critical section:
        find predecessor;
        list[predecessor] := list[i];
  leave new critical section:
```

Figure 3: List insertion and deletion for process i.

A process may choose its predecessor in various ways. To be *safe*, so that scans will not miss list entries, the insertion policy should not place any item later in the list than where it last appeared. Both insertion at the head of the list and by process *id* are safe. Insertion at the end of the list is not safe—a scan may access a node just as the node is deleted and then re-inserted, and the scan may jump to the end of the list, missing intervening entries.

A safe insertion policy ensures that a scanning process will observe all entries that are in the list *throughout the scan execution*. Processes that enter and/or leave the list during the scan may be missed. But process i is not active and $b[i]$=*false* while inserting or deleting itself. Hence, if a scanning process misses i during a scan, then $b[i]$=*false* at some point during the scan. Thus, we have the effect of reading $b[i]$, without the cost.

There is still the question of maintaining mutual exclusion between inserts and deletes. A natural alternative is to use Lamport's original algorithm to implement mutual exclusion for this purpose. Scanning the entire array happens, at worst, only when processes are inserted or deleted. Assuming that this is rare in comparison to the number of times processes request shared resources, a net savings results. Using this approach, processes that do not expect to be active contenders for the resource may voluntarily delete themselves from the

list, and similarly insert themselves only upon first contending for the shared resource.

In practice, it may be more efficient (and simpler) to insert and delete processes from this list when they are first created and finally destroyed–events which typically involve updates to system data structures that take place in mutual exclusion. Including the code for insertion and deletion from the list within existing critical sections would entail negligible overhead.

5 Scan termination

There can be a limited kind of interference between processes that scan the list in the modified algorithm (Figure 2), and processes that insert or delete elements from the list, depending on how processes choose their predecessor (Figure 3).

For example, if processes insert themselves at the head of the list (i.e., $predecessor = 0$), then in principle a reader may find itself scanning the list forever, as entries are deleted from late in the list and reinserted at the beginning. Under the assumption that inserts and deletes are rare, this may not be a problem in practice.

Even this problem may be prevented, by insertion in order of process *id*. Since readers would always move from larger to smaller entries, no reader will scan more than n entries before returning. Insertion becomes slightly more expensive, since a process must find its position before insertion.

6 Correctness

The correctness of the modified algorithm, assuming a safe insertion policy, follows easily from the correctness of the original [Lam87], and the following observations. Assume that each process does an insertion, some number of scans through the list (in the modified mutual exclusion algorithm), and then a delete, possibly repeating this sequence. If a process has finished an insertion operation but not yet begun a deletion, call it *active*, and if it has finished a deletion operation but not yet begun an insertion, call it *inactive*. Processes executing insert or delete operations are *in transition.*

Consider the array $list[0..n]$ as a directed graph of $n + 1$ nodes, such that there is an edge from node i to node j if $list[i] = j$.

It is straightforward to verify the following invariants:

- *There is a path from every node to node* 0. *This and uniform outdegree of* 1 *implies there is a unique cycle in the graph, which includes node* 0.
- *Active processes are on the cycle and inactive processes are not on the cycle.*
- *If j is active and there is a directed path from node i to j not going through node* 0, *then such a path exists for as long as j is active.*

From this, it follows that if a scan of the list terminates, it will visit all the nodes that are active throughout the entire scan, and perhaps some nodes in transition. However, no node that is inactive throughout the scan will be visited. These properties are sufficient to

maintain the correctness of the mutual exclusion algorithm. Specifically, if a process i is not active, then $b[i]$ is *false*. Any process i missed during a scan of the list has implicitly been observed to have $b[i]$=*false*. (Formally, such read steps can be inserted into runs of the modified algorithm, resulting in a run of Lamport's original algorithm.) The correctness of the modified algorithm now follows from that of the original.

7 Parallel sub-lists

Parallel processors could advantageously exploit greater concurrency, if the list is maintained as a collection of distinct sub-lists, such as one per processor. (Here we assume that processes resident at separate processors must still maintain mutual exclusion via a shared memory.) Processes could be inserted and deleted from only one sub-list, while scanners would concurrently scan all of them. Insertions and deletions would require mutual exclusion only with similar operations on the same sub-list. More complex quorum schemes are possible, in which nodes are inserted to more than one sub-list and scanners need only access enough sub-lists, where "enough" requires the scan of at least one sub-list used by each potential insert operation. (All insert and scan sets must intersect.)

8 Discussion

The scan of the linked list could be pipelined with parallel awaits for the individual entries of b[1..n]. This might be profitably exploited in machines with fine grains of parallelism.

Alternatives to (or elaborations of) the linked list data structure are possible. For performance in practice, the key issue is most likely to be which operations must take place in mutual exclusion. In particular, alternative data structures should allow scan operations to proceed concurrently with inserts and deletes. If the linked list is unordered (as discussed in Section 4), in the absence of contention inserts at the front take constant time, while deletes could take time linear in the number of list elements (the cost is scanning to find the predecessor node), and scans may not terminate (deletions and insertions may continually move the scanner back to the front of the list). If the list is doubly-linked, deletes are also constant time, but scans still may not terminate. (Scans in mutual exclusion would terminate but sacrifice concurrency.) Double links on an ordered list guarantee scan termination in time linear in n and deletes remain constant time, but insertions could take time linear in the number of list elements (due to the cost of finding the correct insertion point). Are there alternative data structures that reduce both inserts and deletes to constant time in the absence of contention, still guaranteeing termination of scans in linear time, and preserving concurrency?

Abstractly, the linked list supports three logical operations, insert, delete and scan, where the scan returns a set of *id*'s that contains all the processes that are active throughout the scan, and none that are inactive. In the specific implementation, the scan is very inexpensive relative to insert and delete, since it does not require exclusive access to the list. Are there other applications for such a shared primitive?

The modified algorithm solves mutual exclusion efficiently in contexts in which process creation and destruction are rare. The cost of inserting or deleting a process from the list

is then amortized over the more frequent mutually exclusion operations. With Lamport's original algorithm implementing mutual exclusive access to the list, in the presence of even two contending processes, the winning process may still have to check the status of all other n processes. (i.e., even if the winning process never has to busy-wait, it still may need to access the shared memory n times.) It is interesting to speculate whether there exists a deterministic mutual exclusion algorithm in which the number of accesses to the shared memory by the winning process is a function of actual number of contending processes.

This question is complicated by the issue of measuring time (or memory accesses) in systems which involve waiting. For example, it is possible to show that in any mutual exclusion algorithm, when there is contention of two or more processes, there is no bound on the number of steps taken by the winning process since the last time some process exited its critical section [AT92]. That is, whenever there is contention, the adversary can schedule the contending processes in such a way that each of them will have to busy-wait.

One possibility is to measure time under the assumption that every memory access takes *at most* unit time. For example, if one process busy-waits on a bit (reads repeatedly until the value returned is 1), and simultaneously a second process is executing a write to set the bit to 1, the the writer's single operation dominates: although any number of reads may have taken place, the single write operation takes time at most one, and within at most two more time units, the busy-wait terminates. (An unsuccessful read may partially overlap the write. The first read operation that occurs entirely after the write is the earliest read that must return 1.)

In Lamport's algorithm (and ours), until the critical section is occupied, one process waits for another only long enough for the latter to execute a constant number of memory accesses. Under the assumption above, the waiting process can only execute a constant number of reads during this busy wait. Hence, under this measure, we could argue that the winning process in Lamport's algorithm can take $O(n)$ time, under contention of two or more processes, while our algorithm takes time proportional to the number of active processes.

Using this efficiency measure, some preliminary results indicate that contention sensitive mutual exclusion may be possible: There exists an algorithm where the time it takes a the winning process to enter the critical section is of order $\min\{k, \log\ n\}$, where k is the number of contenders and n is the total number of processes [AGMT92]. Unfortunately, this algorithm uses an unbounded number of shared registers. The algorithm uses a building block which solves the leader election problem with a *bounded* number of shared registers and with time complexity of order $\min\{k, \log\ n\}$.

References

[AGMT91] Y. Afek, E. Gafni, M. Merritt, and G. Taubenfeld. Contention sensitive mutual exclusion. Presented at the rump session of the 10th Annual ACM Symp. on Principles of Distributed Computing, Montreal, Quebec, August 1991.

[AT92] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. A preliminary version appeared in the *Proceedings of the 13th IEEE Real-Time Sys-*

tems Symposium, pages 12 – 21, December 1992. (Full version available from *gadi@research.att.com.*)

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[TUX90] *TUXEDO System Release 4.0 – Product Overview.* AT&T, 1990.

[Mac91] M. R. MacBlane. Source level atomic test-and-set for the TUXEDO System source product. UNIX System Laboratories, May 14, 1991.