10 Conclusion and Future Work

This paper has presented our algorithm for providing recoverable, high-speed mutual exclusion without OS intervention and the associated overhead. It requires only a "lowest common denominator" atomic instruction, such as test-and-set or swap. Spin lock acquisition and release is very fast; with very high probability, recovery of dead processes is also very fast.

Spin locks based on test-and-set may cause an inordinate amount of bus activity in a shared-memory multiprocessor since any change in a spin lock's state invalidates a cache entry for all processes waiting for the spin lock. Recent research, notably [3] and [11], has provided methods in which the atomic instructions are used to construct a queue of processes waiting for the spin lock, and each process (and thus each processor) may busy-wait on a variable that no other processor is reading. We intend to develop a technique analogous to the one presented in this paper to recover these structures in case of process death.

Acknowledgements

We would like to thank Boris Lubachevsky and Mark Dilman for pointing out an error in an earlier version of the cleanup algorithm, and Phil Gibbons for discussions about consistency models, and for guiding us to the right sources of information on the topic. We would like to thank Lloyd Tucker for providing many helpful comments on an earlier draft, and Robert Wilhite for producing timing numbers on the HP 9000.

References

[1] A. Silberschatz and P. Galvin, *Operating System Concepts*. Addison-Wesley, 4 ed., 1993.

[2] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 6–16, Jan. 1990.

[3] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[4] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dali: A high performance main-memory storage manager," in *Procs. of the International Conf. on Very Large Databases*, 1994.

[5] P. Sindhu, J.-M. Frailong, and M. Cekleov, "Formal specification of memory models," Tech. Rep. CSL-91-11 [P91-00112], Xerox Corporation, Dec. 1991.

[6] D. E. Corporation, *The Alpha Architecture Handbook*, 1992.

[7] P. Bohannon, D. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava, "Recoverable user-level mutual exclusion," Tech. Rep. 950320-05, AT&T Bell Laboratories, Mar. 1995.

[8] M. Dilman and B. Lubachevsky, "Personal communication," 1994.

[9] B. N. Bershad, D. D. Redell, and J. R. Ellis, "Fast mutual exclusion for uniprocessors," in *Procs. of the International. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 223–233, Oct. 1992.

[10] G. Graunke and S. Thakkar, "Synchronization algorithms for shared memory multiprocessors," *IEEE Computer*, vol. 23, pp. 60–69, June 1990.

[11] T. S. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," Tech. Rep. 93-02-02, University of Washington, Feb. 1993.

[12] S. Khanna, M. Sebree, and J. Zolnowsky, "Real-time scheduling in SunOS 5.0," in *Winter Usenix Conference 1992*, 1992.

[13] M. A. Eisenberg and M. R. McGuire, "Further comments on Dijkstra's concurrent programming control problem," *Communications of the ACM*, vol. 15, Nov. 1972.

[14] M. Sullivan and M. Stonebreaker, "Using write protected data structures to improve software fault tolerance in highly available database management systems," in *Procs. of the International Conf. on Very Large Databases*, pp. 171–179, 1991.

HP-UX Semaphores on HP9000 712/60	19,000
SunOS 5.3 Semaphores on Sun SS20/61	20,000
Safe Spin Lock on Sun SS20/61	526,000
Spin Lock on Sun SS20/61	2,128,000

Figure 7: Uncontested Acquire and Release per Second

in a database scenario where the actions of the process can be rolled back.) This removes the assumption of progress from our claim that all processes will eventually leave ViewWants, at the cost of possibly killing a process which is making no progress in the spin lock acquisition code, even though the process may not hold the spin lock. It may be argued that killing such a process is not a bad idea any way, since it is not making progress.

It is fairly straightforward to extend the algorithms described here to the case of multiple spin locks, since each process can be *attempting* to acquire/release at most one spin lock at a time, though it may *hold* many at any given time.

8 Implementation

We have implemented whoOwns() in the context of Dali, a main memory storage manager [4]. This allows us to have "mostly trusted" processes which link with storage manager code and access the database through shared memory, yet which can die (for example be killed) without necessitating a complete recovery of the database.

An implementation of the code in C or C++ requires to use of volatile declarations to ensure that the code optimizer does not attempt to optimize away references to global variables. We also implement several optimizations, including one suggested by Mark Dilman [8] where the acquisition code first checks if the spin lock is busy and does not even raise the wants flag if it is busy.

We tested the performance of our implementation against operating system semaphores and against spin locks without the recoverability features. The results are shown in Figure 7. The timings are all in terms of uncontested acquire-release pairs per second. Timings for contested acquisition/release would depend on the backoff policy, which is needed for all spin locks and is orthogonal to our techniques. The performance benefits of our technique over system semaphores is clear.

9 Related Work

Spin-locking implementations of mutual exclusion have been extensively studied in the parallel computation and operating system communities. It is widely recognized that spin locking (also referred to as busy waiting) is much faster than operating system provided mutual exclusion for parallel computations in shared memory multiprocessor systems [1]. Even in uni-processors, spin locking (with back-off) is recognized to be better than operating system spin locks in many applications, especially when the level of contention is low, which is borne out by the performance numbers in Figure 7. The scheme of [9] for implementing mutual exclusion is based on notifying the operating system that a particular section of code must be restarted from the beginning if interrupted by a context switch. Their scheme can be used to solve our problem, but requires a modification to the OS, and does not extend to multi-processors. Most recent work in high-speed mutual exclusion has centered on efficiency under various multiprocessor memory models [3], [10], [11].

Sun's Solaris operating system uses spin lock mutual exclusion within the kernel [12]. They track process ownership in order to avoid priority inversion — the current owner's priority is temporarily raised to that of the highest priority waiting thread, to allow it to progress and release the spin lock so the higher priority thread can proceed. As the only synchronization primitives on the SPARC architecture are a register to memory swap and a test-and-set, they face a very similar problem. Their solution is to reserve a hardware register to indicate if a process is attempting to acquire a spin lock. All interrupt handlers begin by checking this register, and if the owner needs to be filled in they fill it in before continuing with the interrupt. Though this takes a register completely out of use, it is quite fast; however, it is useless to us since we do not want to modify the kernel.

Older, software-based approaches to mutual exclusion that do not rely on atomic instructions are trivially recoverable. For example, the standard starvation free software mutual exclusion algorithm, [13], uses an array of per-process state information, thus the state of any process with respect to a given spin lock can be determined immediately by inspecting that variable.[4] Any interference with other processes can be removed by resetting the state to "uninterested."

Why, then, do we not simply use one of these algorithms? The primary reason is resources. All these algorithms require at least time proportional to the number of processes to acquire an *uncontested* spin lock, while solutions based on synchronization hardware, such as ours, typically require a small constant number of accesses. Furthermore, these systems require space proportional to the number of processes times the number of spin locks, as opposed to the sum of the two, as with our algorithm.

Process failure is often caused by software bugs, and is itself only one possible detrimental effect of these bugs. Our method provides recovery from this particular mode of failure, but is only one part of an overall fault detection and tolerance strategy. For example, unintended writes into shared memory can also be caused by bugs; this issue is discussed in [14]. Protection from such errors is orthogonal to the focus of our work.

[4]These algorithms were designed assuming a sequentially-consistent memory, and may need to be redesigned (perhaps by adding fence instructions) for a weakly-consistent memory system.

```
whoOwns(SafeSpinlock *L)

   Set_Of_ProcessID ViewWants;
   ProcessID owner;

C0:L→cleanup_in_progress = True;
   ⟨fence⟩ /* TIME: $t_{\text{start}}$ */
   ViewWants = ∅
   foreach process P do
      ⟨fence⟩
      if LockAccess(P)→wants == L then
C1:      ViewWants = ViewWants + {P};
C2:/* TIME: $t_{\text{view}}$ */
C3:while ViewWants != ∅
      owner = L→owner;
C4:   if owner != NO_PROCESS
            and not IsDead(owner) then
D1:      status = (HELD, owner, ALIVE);
         goto DONE;
D2:   if L→lock == 0 then
         status = (FREE, NO_PROCESS,_);
         goto DONE;
      †    /* Code for slow processes goes here */
C5:   sleep(A_SHORT_WHILE);
C6:   foreach P in ViewWants
         if LockAccess(P)→wants != L
               or IsDead(P) then
C7:         ViewWants = ViewWants - {P};
   endwhile
   ⟨fence⟩ /* Time:  $t_{\text{empty}}$ */
C8:owner = L->owner;
   /* Status of L now static.  May be NO_PROCESS */
C9:if (L->lock == 1) then
D3:   status = (HELD, owner, DEAD);
   else
D4:   status = (FREE, NO_PROCESS, _);
   DONE:
C10:  L →cleanup_in_progress = False;
   ⟨fence⟩/* TIME: $t_{\text{final}}$ */
   return status;
end whoOwns
```

Figure 6: The whoOwns() Procedure

1. Suppose the algorithm observes an "informative" state of a spin lock (free or owned by some process). If the lock is observed to be free, then this represents an accurate state of the lock based on the assumption that the spin lock itself works correctly. If it observes a registered owner which is subsequently seen to be alive, then that owner must have also been alive when it owned the lock, and any observed owner was indeed an owner, since this is an underestimation of ownership. (Note that by the time the observed state is reported, that state may no longer exist.)

2. Once the barricade, $\mathcal{L}\rightarrow$cleanup_in_progress has been raised, no process not in ViewWants can get the spin lock This eliminates scenarios involving an infinite stream of processes.

3. ViewWants is finite, and any process which makes progress or dies will leave the set. Since the algorithm can make a decision when ViewWants is empty, our assumption of progress guarantees that the algorithm will terminate.

If the algorithm doesn't terminate until ViewWants is empty, and the spin lock is still held, then it must be held by a dead process, since all the potential live owners were members of ViewWants. Further, if the registered owner is NO_PROCESS, the dead process either just acquired the spin lock and did not update any structures, or had finished its updates (if any) and was on the verge of releasing it. In either case it is safe to free the spin lock.

Also, Boris Lubachevsky and Mark Dilman have used a verification tool to generate an independent, mechanized proof of correctness for the case where only two processes are attempting to acquire the spin lock [8].

7 Extensions

The correctness of our algorithm depends on an a limited assumption of progress on the part of processes. It is conceivable that this will not be the case, and some process will not make progress during this system provided code, yet will not die and be cleaned up either. For example, the process could be getting no CPU time due to operating system scheduling policies, or perhaps it was transferred to a user interrupt handler with an infinite loop. We argue that it is reasonable to kill such a process and, in our case, roll back its transaction, as some mechanism would be required in any event to deal with loss of a resource for "an unreasonable amount of time". Note that processes waiting for a different lock (if multiple locks are allowed) will trivially not be part of ViewWants, as they will want a different lock.

To implement this in our algorithm, a timer must be introduced which is set upon creation of ViewWants, and reset whenever a process is removed from ViewWants. (The timer value itself should be a tunable parameter.) Code introduced at location † of whoOwns should, upon expiration of the timer, simply pick a member of ViewWants, and kill that process. (This may seem drastic, but is quite reasonable

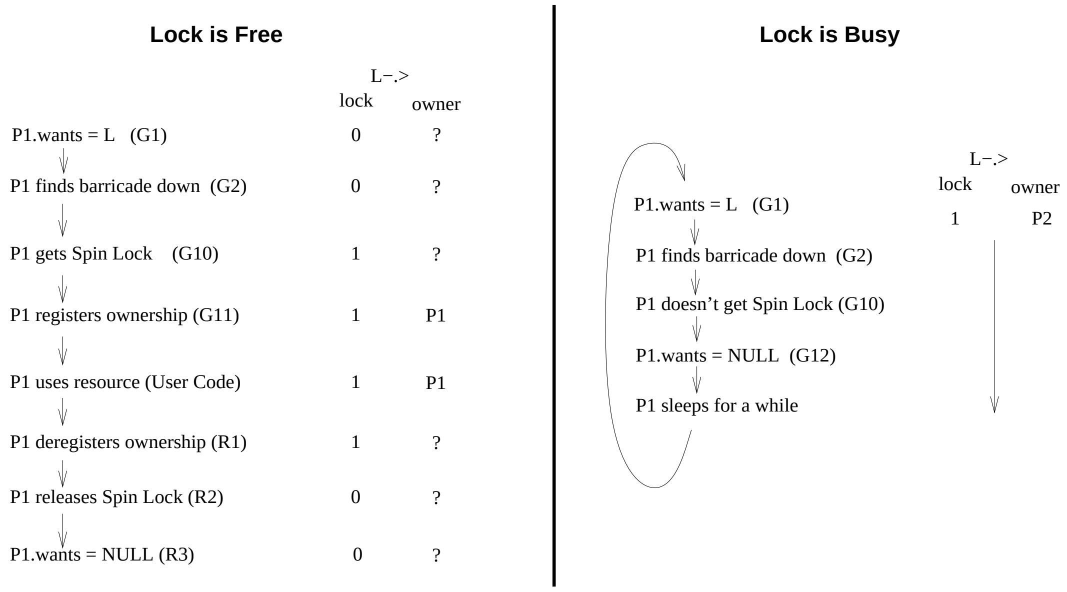
Lock is Free
L–.>
lock
owner
P1.wants = L (G1)
0
?
P1 finds barricade down (G2)
0
?
P1 gets Spin Lock (G10)
1
?
P1 registers ownership (G11)
1
P1
P1 uses resource (User Code)
1
P1
P1 deregisters ownership (R1)
1
?
P1 releases Spin Lock (R2)
0
?
P1.wants = NULL (R3)
0
?
Lock is Busy
P1.wants = L (G1)
P1 finds barricade down (G2)
P1 doesn't get Spin Lock (G10)
P1.wants = NULL (G12)
P1 sleeps for a while
L–.>
lock
owner
1
P2

Figure 5: Normal Operation of Spin Lock Acquisition and Release Code

(b) ViewWants is empty, i.e., no live process remains that could possibly own the spin lock.

If (a) occurs, a decision can be made immediately and an appropriate status returned. Otherwise, the cleanup routine waits a little, and retries after eliminating some candidates from ViewWants using simple tests. If (a) does not occur, (b) eventually occurs (based on our limited assumption of progress, see Section 3). Once the second condition occurs, if the spin lock is held, it must be held by a dead process. The ownership may or may not be determined at this point, and the procedure returns an appropriate status.

The driver code of whoOwns takes appropriate action based on the return status, and is discussed in Section 6.1.

Performance of the above code can be improved in several ways. First, a test to determine if the owner of the spin lock is known[3] should be done before executing the main code of the function. Second, a check could be added to perform a wait in line C5 only after eliminating candidates as in C6. We omit these optimizations from the above code for simplicity of exposition and proof.

6.1 Driver Code and O/S Interaction

To avoid any problems with multiple or concurrent cleanup processes, we assume a single cleanup process which calls whoOwns() on the death of a process P for which LockAccess[P]$\rightarrow$wants is non-null, or upon the complaint of a process which has timed out attempting to acquire a spin lock.

We assume that a normally exiting process deallocates its lock access record at a point when it has finished all spin lock accesses. The cleanup process will eventually visit the lock access record of any dead process, ensuring that every record is eventually deallocated. For example, if one run of whoOwns() decides the spin lock is alive because it is held by a process which does not seem dead at D2, but has in fact died immediately after the cleanup process called IsDead(), this access record will not be cleaned up until the next run of the cleanup routine. We also assume that no access records are reallocated during a run of whoOwns(), though this restriction is not difficult to remove.

The full version of the paper, [7], gives an example of a loop which polls for dead processes, combined with an example of how the information returned by whoOwns() can be used to return data structures guarded by the spin lock to use.

6.2 Correctness

Given that the state of a lock held by a dead process will not change, the following theorem allows useful recovery routines to be built around whoOwns.

Theorem 6.1 *Procedure* whoOwns(L) *terminates and reports an ownership status of spin lock* L *which accurately reflects the state of* $\mathcal{L}$ *at some point in time after* whoOwns *is called.* □

We present the proof in the full version of the paper ([7]), and merely present the intuition here.

The correctness of the cleanup algorithm follows from three main points:

[3]This test must be done carefully since reading the "owner" field and the call to IsDead() cannot be done as an atomic action. This can be handled by a second reading of owner if IsDead() returns True.

```
getLockAttempt(int myPid, SafeSpinlock *L)

    Register R;
    LockAccessRecord *ma = LockAccess(myPid);
    Boolean cleanup;

G1:ma→wants = L;
    ⟨fence⟩
G2:cleanup = L→cleanup_in_progress;
G3:if cleanup then
G4:    ma→wants = NULL;
G5     ⟨fence⟩
G6:    while L→cleanup_in_progress do
           sleep a while;
       endwhile;
G7:    return BUSY;
G10R = test-and-set(L→lock);
    if R == 0 then // We have the mutex
G11:   L→owner = pid;
       ⟨fence⟩
       return ACQUIRED;
    else
G12:   ma→wants = NULL;
       ⟨fence⟩
       return BUSY;
end getLockAttempt
```

Figure 3: Spin Lock Acquisition Code

instructions denoted using ⟨fence⟩. These instructions are not required if the architecture supports sequential consistency, but are required under the weaker consistency model that we assume.

A process sets its wants variable (ma→wants) to indicate a spin lock that it wishes to acquire, checks to make sure the barricade (L→cleanup_in_progress) is down, and then tries to acquire the lock. If the attempt is successful, the process records its new ownership (L→owner = pid) and returns. If it fails, the process clears its wants variable, and returns to the enveloping routine getLock. If the barricade was found to be up, then it waits until it is down, and again returns failure to getLock. The routine getLock (not shown) repeatedly calls getLockAttempt() until it succeeds, though it can easily be rewritten to time out and fail after some number of attempts, or to implement a backoff strategy. Similarly, getLockAttempt() may be augmented with a finite inner loop which "spins" more tightly than getLock, for multi-processor systems.

5.3 Spin Lock Release

The spin lock release code, given in Figure 4 is simple, and clearly demonstrates the underestimation and overestimation of ownership by S→owner and ma→wants respectively.

5.4 Example of Normal Operation

In Figure 5 we illustrate on the left the sequence of actions a process will undertake while successfully acquiring a spin lock, and on the right the sequence involved in attempts which fail due to contention. Shown in boxes are the associated changes in the state of the lock itself.

```
releaseLock(LockAccessRecord *ma)

    SafeSpinlock *L = ma→wants;

R1:L→owner = NO_PROCESS;
       ⟨fence⟩
R2:L→lock = 0;
       ⟨fence⟩
R3:ma→wants = NULL;
end releaseLock
```

Figure 4: Spin Lock Release Code

On a successful acquisition, illustrated on the left, the process begins by registering its interest in the spin lock $\mathcal{L}$ at point G1, and checking the status of the barricade at G2. After the test-and-set at G10, it finds that it has the spin lock, and it registers its ownership by setting the owner value of the spin lock to its own process id at G11. At that point, control is returned to the user code, which may access the protected resource. Upon release, the reverse process is carried out, deregistering ownership, followed by release of the spin lock, followed by deregistering interest.

In a failed attempt to get the lock, illustrated to the right, it is important to note that a check of the barricade at point G2 or G6 is always made after registering interest and before attempting to get the spin lock. Also, note that the process deregisters its interest in the mutex before sleeping, thus decreasing the chance that it will need to be considered by the cleanup routine.

6 The Cleanup Process

The cleanup process executes procedure whoOwns, which is shown in detail in Figure 6, to determine ownership of a spin lock . At a high level, the procedure whoOwns proceeds as follows.

1. Raise the $\mathcal{L}$→cleanup_in_progress barricade, preventing processes which don't currently "want" the spin lock from getting it while we are cleaning things up (C0).

2. Take an "overestimation snapshot" of processes which could have, or could get, the spin lock during the cleanup period. Call this snapshot ViewWants (C1).

3. The main loop of the cleanup routine, C3–C7, waits for one of two conditions:

    (a) The state of the spin lock becomes observable, either because no one has it, or because a live owner is registered (D1–D2).

spin lock acquisition or release code, it receives some CPU time to execute, and if interrupted, it returns to the spin lock code within a finite amount of time.)

Since processes may violate this assumption, for example by having a very low priority and getting no CPU time from the operating system, we will present a simple extension of our algorithm in Section 7 to kill these processes if the fate of the spin lock cannot be resolved in a "reasonable" amount of time.

4 Overview of Approach

Consider an atomic test-and-set based implementation of a spin lock. The first and most obvious step in tracking ownership of such a spin lock is to require that a successful attempt to acquire the test-and-set latch be immediately followed by a write which stores the new owner's identifier (process or thread identifier, which we abbreviate to *process id*) in an "owner" field associated with the spin lock. Clearly, if these two steps were atomic, we could always find out which process currently owns the spin lock. However, as discussed in Section 2, many common architectures cannot implement these two steps atomicly.

As a first step toward solving this problem, we require that all processes that are trying to acquire the spin lock note the name of the spin lock in which they are interested in a per-process shared location. We call this location the process's "wants" field. The collection of all processes' "wants" fields provides us with an overestimate of the set of possible owners of the spin lock (there are zero or one owners, but an arbitrary number of "interested" parties). This helps establish a set of all processes that might hold the spin lock.

The set of processes that want the spin lock may, however, change even as the cleanup process attempts to determine which processes have set their "wants" field. To solve this problem, we introduce a flag associated with the spin lock called "cleanup-in-progress," and require that processes do not attempt to get the spin lock if this flag is set. This flag provides a barrier which, when "raised", prevents any new processes from entering the set of potential owners deduced from the "wants" field. The cleanup-in-progress flag for a particular spin lock is set by the cleanup process while it attempts to resolve the ownership of that spin lock. Without this "barricade," the (remote) possibility exists that one or more processes can repeatedly acquire and release the spin lock, always leaving the spin lock acquired but unregistered while its status is tested by the cleanup process, declaring ownership only between tests by the cleanup process. We cannot distinguish between this case and the death of a single process in an indeterminate state. (We explored the use of a counter that is incremented on each spin lock acquisition to distinguish between the two cases. However it complicated the proof of correctness considerably, and we abandoned the approach.)

Given these additional tools, how do we determine whether a dead process holds a spin lock? We start by setting the cleanup-in-progress flag, then gathering a list of potential owners from the "wants" information. (We gather this list to avoid certain pathological scenarios with streams of new processes.) Now it becomes reasonable to wait until the situation resolves itself, as we must only wait for a finite number of processes to give up their interest in, or register their ownership of, the lock. In all cases, a process must only advance by a few instructions to either register ownership, or notice that the cleanup-in-progress flag is set, and relinquish its interest in the spin lock. A method for handling the case where these processes fail to make progress is described in Section 7.

```
struct SafeSpinlock {
   int lock;
   ProcessID owner;
   int cleanup_in_progress;
};

struct LockAccessRecord {
   SafeSpinlock *wants;
};
```

Figure 2: System Data Structures

5 Acquisition and Release Protocol

After detailing shared data needed for our scheme, the spin lock acquisition and release protocols are presented. The procedure for determining ownership in case of failure is given in the Section 6.

5.1 System Data Structures

Our spin lock protocol involves additional information associated with each process as well as additional information associated with each spin lock. The former is stored in a per-process "Lock Access Structure." For process P, we refer to the structure as LockAccess[P]. The latter is stored with the spin lock itself.

Figure 2 is an example declaration of these data structures in a C-like syntax. In the structure SafeSpinlock, lock refers to the actual test-and-set target variable, while owner provides the "safe underestimate" of ownership. That is, owner is set by a process to its own process id immediately *after* it has gained access to the spin lock, and back to NO_PROCESS immediately *before* releasing it. Finally, cleanup_in_progress is a special variable that is written only by the cleanup process. It is used to form the barricade against new processes as described earlier.

As for LockAccessRecord, the only per-process information we require is the variable, wants, which is set by a process to point to a spin lock before trying to acquire it, and reset to NULL only after releasing it, or after a failed attempt to acquire. Thus, it is a "safe overestimate" of spin lock ownership.

5.2 Spin Lock Acquisition

The spin lock acquisition attempt routine getLockAttempt(), is shown in pseudo-code in Figure 3. Indentation indicates nesting. The labels at the left are for ease of reference. Our code shows explicit use of fence

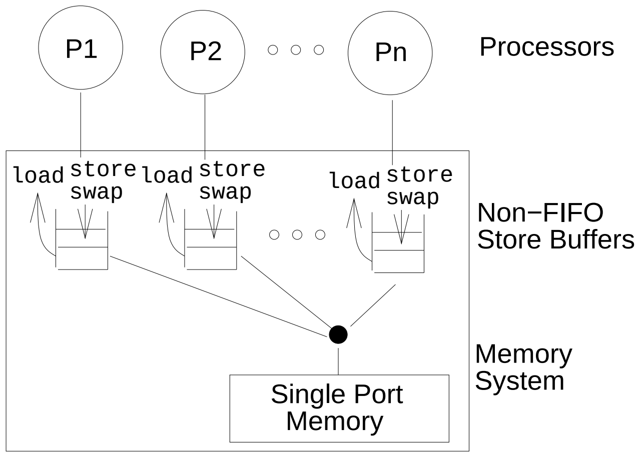

Figure 1: SPARC Memory Consistency Model

another processor. In a sequentially consistent system, at least one of the two reads will return 1. With weaker levels of consistency, it may be possible for both reads to return 0 — they could have read locally cached values for the variables, and the writes may take some time to propagate to the other processor. Systems providing weaker levels of consistency also provide explicit synchronization instructions, for example instructions that ensure that all pending writes (or cache invalidate requests) are propagated to all processors.

Our proofs of correctness are based on the *Partial Store Ordering* model of memory used in the SPARC architecture, which is shown in Figure 1 and described below. Most current generation shared memory multiprocessor systems are built on roughly the same memory model. We believe that our algorithms work in some weaker memory models as well, and will explore the issue in the full version of this paper.

Figure 1, adapted from [5], illustrates several aspects of the SPARC memory model. There is a store buffer for each processor, that buffers writes that have been issued by the processor. The writes are propagated to the memory one at a time (the memory is treated as if it were single ported), but may be propagated in a different order from the the order in which they were issued by the processor (that is, the store buffers are not FIFO). In addition to loads and store, the model also supports `swap` instructions, which atomically swap the values in a specified register with a specified memory location.

Formally, the memory model can be defined in terms of partial orderings of load, store and swap operations as described in [5]. There is a partial ordering of memory operations generated by processor i (denoted by $\preceq^i$), and a partial ordering of memory operations executed at the memory (denoted by $\leq$). Informally, the rules defining the partial store ordering model are as follows.

Total order The store operations in memory are totally ordered.

Atomic swap No other write to a memory location is allowed between the load and store parts of a swap.

Termination Buffered writes are eventually carried out in memory.

Value The value returned by a load in processor i is the last in the $\leq$ order of stores that are before the load either in the $\leq$ order or in $\preceq^i$.

Load ordering $L^i_a \preceq^i Op^i_b \Rightarrow L^i_a \leq Op^i_b$ where Op is any memory operation, and L^i_a denotes a load from location a by processor i.

Storage barrier Store operations from a processor that are separated by a 'storage barrier' or 'fence' instruction (`fence`) appear in the same order in $\leq$.

Same-location ordering Writes to the same location from the same processor are carried out in the order in which they were generated (formally, $S^i_a \preceq^i S'^i_a \Rightarrow S^i_a \leq S'^i_a$, where S^i_a denotes a store to location a from processor i).

Note that in the above model, reading a word and writing a word are each atomic. The swap instruction easily simulates an atomic test-and-set instruction.[2]

We use the term `fence` to denote the generic storage barrier instruction. All shared memory systems with non sequential consistency that we are aware of (e.g. the Alpha [6]) provide such storage barrier instructions. We use the term "integer" synonymously with the term "word".

Our entire description is in terms of processes, but could equally well apply to threads or light-weight processes. We assume that the identifier of a process is a single word, so it can be written atomically. We assume that processes are fail-safe, i.e., they do not modify spin lock control information except through the provided interface code. We assume that our interface code is able to maintain and manipulate shared information other than the spin lock itself.

The main example of this additional shared information is a table with a slot allocated to each process that may want to acquire a spin lock, where the slot has space for a process to note what spin lock it is currently trying to acquire. The allocation of a slot in the table itself requires mutual exclusion on the table, which would cause a circularity if handled using our spin locks, so a more epxensive mechanism is required for this once-per-process task.

For simplicity of presentation, we assume that a process may hold only one spin lock at one time. Our implementation, however, allows a process to hold multiple spin locks.

Finally, we assume that processes make progress while running spin lock acquisition and release code. (We are not assuming that our concurrency mechanism is starvation free, just that once a process enters

[2]The SPARC assembly language for the version of the architecture which we currently use provides a test-and-set instruction (`ldstub`), and a swap instruction which also acts as a `fence` instruction.

which share data between processes.

Operating system semaphores maintain ownership information, allowing one to find at any time which process, if any, has acquired the semaphore and not (yet) released it. The information can be used for recovery purposes in the case when software faults result in a process failing (halting) after having acquired a semaphore but before releasing it. Determining ownership of semaphores is the critical first step to recovery from process failure. Given the knowledge that a dead process holds the semaphore, the software application could potentially use that information along with other application specific information to carry out appropriate recovery actions on resources guarded by the semaphore, and then release the semaphore, allowing other processes in the software application to continue normal operation. Such recovery from process failure is particularly important in systems with high availability requirements, since the only alternative is to shut down all processes, restore to consistency all shared resources that may be left in an inconsistent state, reinitialize all semaphores, and then restart the system. An example of a system that provides recovery from process failure is the Dali main-memory storage manager system [4].

Unfortunately, it is difficult to determine ownership of spin locks based on the commonly used atomic test-and-set or atomic swap instructions. (See Section 2.) Providing a correct and efficient mechanism to determine ownership for spin locks is the central issue addressed by this paper.

In this paper, we present a scheme for effectively dealing with the ownership problem in an environment that supports the atomic test-and-set (or, equivalently, atomic swap) instruction. We do so by recording information in shared memory as part of the acquisition and release code. In the case of an uncontested acquisition, we add very little overhead to the path length — two writes and a read.[1] However, our implementation has the property that we can always detect ownership of a spin lock. In very rare cases, this may involve killing processes that fail to make progress for a long time while executing semaphore acquisition/release code.

The basic idea behind our algorithm is to take a global picture rather than a local one – instead of just examining a failed process, we examine *all* processes that may have wanted to acquire a spin lock, which gives us enough information to determine ownership of a spin lock. This is the critical novel feature of our algorithm. Several difficulties arise in the context of a system where a new process may be spawned at any time and subsequently attempt to acquire a spin lock. Handling such situations constitutes the bulk of the technical challenges in carrying out the basic idea. Further complications are introduced by the weak-memory-consistency models that most multi-processor architectures today implement, and we prove our algorithms works under the assumptions of a representative weak memory model.

The remainder of the paper is organized as follows. Section 2 describes the basic problem, and drawbacks of current solutions. Our system model is presented in Section 3, and an overview of our approach is presented in Section 4. Section 5 describes the system control information used to support crash safety and the spin lock acquisition and release protocols. Section 6 presents our cleanup algorithm, along with some intuition about the proof of correctness. Section 7 discusses extensions of our algorithm. Section 8 describes our implementation and presents some timing numbers. Section 9 discusses related work.

2 Problem Definition

Determining the ownership of a spin lock requires that the process that acquired a spin lock also register itself as the owner (by writing its process identifier to a known location). Unfortunately, the act of acquiring a spin lock using the basic hardware instruction test-and-set (or atomic swap) cannot be used to also atomically register ownership. At best, the atomic instruction can be followed by a conditional branch testing for a successful acquisition, which can be followed by an instruction writing the process id of the new owner. If the process that is trying to acquire a spin lock is interrupted between the test-and-set and the write, the ownership of a spin lock is left in doubt until the process gets to execute the write. If the process fails in this interval, the ownership of the spin lock will never become clear. Worse still, it is impossible to distinguish between a process that has failed at this step and a process that has not failed, but has not yet carried out the write, either because it is servicing an interrupt, or because it has not been allocated CPU cycles. A symmetric problem can also arise when releasing the semaphore, since the deregistration and release may have to be accomplished using separate instructions (depending on the exact atomic instruction used).

3 System Model

We first present our hardware model. We assume one or more processors sharing memory. In multi-processor systems, read and write requests may originate from different processors, and the memory system processes the requests. Memory systems used in earlier generations of multi-processor systems provided a *sequential consistency* model. Under this model, all the reads and writes handled by the memory system can be ordered in such a fashion that the values returned by a read on a memory location is exactly the last value written to the memory location, and the operations generated by each processor appear in the same order in which they were generated.

However, current generation multi-processor systems provide weaker levels of memory consistency, in order to improve the speed of memory accesses and allow more efficient caching of data in each processor. In particular, suppose A and B were initially 0, and we have the requests $Write(A, 1), Read(B)$ from one processor, and the requests $Write(B, 1), Read(A)$ from

[1] On systems supporting only a *weak consistency* model, these two writes also require the use of special synchronization instructions with a somewhat higher overhead.

Recoverable User-Level Mutual Exclusion

Philip Bohannon*
Daniel Lieuwen
Avi Silberschatz
S. Sudarshan
Jacques Gava

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

Abstract

Mutual exclusion primitives based on user-level atomic instructions (often called spin locks) have proven to be much more efficient than operating-system semaphores in situations where the contention on the semaphore is low. However, many of these spin lock schemes do not permit registration of ownership to be carried out atomically with acquisition, potentially leaving the ownership undetermined if a process dies (or makes very slow progress) at a critical point in the registration code. We present an algorithm which can ensure the successful registration of ownership of a spin lock, regardless of where processes fail. Thus, our spin lock implementation is 'recoverable'. The determination of a spin lock's ownership can potentially be used to restore resources protected by the spin lock to consistency and then release the spin lock. Other processes using the lock can then continue to function normally, improving fault resiliency for the application. Our algorithm provides very fast lock acquisition when the acquisition is uncontested (comparable in speed to a simple test-and-set based spin lock), and we prove it works even on the weak memory consistency models implemented by many modern multiprocessor computer systems. Other implementations of a recoverable user-level mutual exclusion primitive are either dependent on special instructions such as compare-and-swap that are not supported on many architectures, or are implemented using (variants of) the Baker's Algorithm, which is quite costly even in the case of uncontested acquisition.

1 Introduction

Current day computing environments provide support for concurrent accesses to shared memory by multiple processes (threads), possibly running on multiple processors. Such systems (hardware and/or operating system) must provide synchronization constructs to allow processes to access shared data in a mutually exclusive manner. One mechanism for achieving this goal is the use of semaphores (see, e.g., [1]). In a traditional semaphore implementation, as provided on Unix systems, semaphore operations (*wait* and *signal*) are implemented as operating system kernel calls, which allows the system to take the operations into consideration for CPU scheduling. However, system calls require a context switch, which is usually quite expensive (equivalent to thousands of instructions) on current generation processors.

In a situation where contention for a shared resource seldom occurs, one would like to avoid the cost of context switching for the purpose of providing mutual exclusive access to data. This can be done through the use of binary semaphores implemented as *spin locks* (see e.g. [2, 3]). When requesting a spin lock, a process uses an atomic read-and-update hardware instruction (e.g. test-and-set or register-memory-swap), to perform the following operations as a single unit: check if the semaphore is free, and update its status to not-free. If the semaphore was free, the process has now acquired the semaphore; the acquisition cost is very low — a single instruction — compared to a kernel call. If the semaphore was busy, the process retries the acquisition repeatedly until the semaphore is acquired. In a uniprocessor, between acquisition attempts, the process indicates to the operating system that another process may be scheduled on the CPU, typically by executing a *sleep* operation. On a multiprocessor system, if the semaphore is held by a process on a different processor, the acquisition code may not even need to perform the sleep operation, but simply keep trying the acquisition until it succeeds. This activity is called "busy waiting" or "spinning." For semaphores with a low degree of contention, spin locks have been shown to offer significant performance benefits over operating system semaphores. (A semaphore acquisition is "uncontested" if, during the acquisition attempt, no other process holds or tries to acquire the semaphore.) Low contention semaphores which are repeatedly acquired and released are common in many software systems

*A Ph.D. candidate in the Department of Computer Science at Rutgers University.