552

Appro XtremeServer™

2P or 4P Enterprise Server - Quad-Core Ready
Powered by AMD Opteron™ with DDR2

Kernel Korner - Using RCU in the Linux 2.5 Kernel

By **Paul McKenney** on Wed, 2003-10-01 01:00. **Software**

Read-copy update, a synchronization technique optimized for read-mostly data structures, is new with the 2.5/2.6 kernel and promises better SMP scalability.

The Linux hacker's toolbox already contains numerous symmetric multiprocessing (SMP) tools, so why bother with read-copy update (RCU)? Figure 1 answers this question, presenting hash-lookup performance with per-bucket locks on a four-CPU, 700MHz Pentium III system. Your mileage will vary with different workloads and on different hardware. For an excellent write-up on the use of other SMP techniques, see Robert Love's article in the August 2002 issue of *Linux Journal* [available at **www.linuxjournal.com/article/5833**].

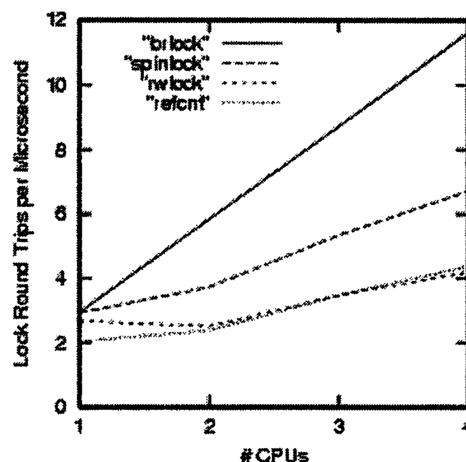

Figure 1. Hash-lookup performance scales poorly with number of CPUs.

All accesses are read-only, so one might expect rwlock to work as well as this system. However, one would be mistaken; rwlock actually scales negatively from one to two CPUs, partly because this

- **T-shirts, mugs & more**

Navigation

- **recent posts**
- **news aggregator**

variant of rwlock avoids starvation, thus incurring greater overhead. A much larger critical section is required for rwlock to be helpful. Although rwlock beats refcnt (a spinlock and reference counter) for small numbers of CPUs, even refcnt beats rwlock at four CPUs. In both cases, the scaling is atrocious; refcnt at four CPUs achieves only 54% of the ideal four-CPU performance, and rwlock achieves only 39%.

Simple spinlock incurs less overhead than either rwlock or refcnt, and it also scales somewhat better at 57%. But this scaling is still quite poor. Although some spinning occurs, due to CPUs attempting to access the same hash chain, such spinning accounts for less than one-quarter of the 43% degradation at four CPUs.

Only brlock scales linearly. However, brlock's single-CPU performance is subpar, requiring more than 300 nanoseconds to search a single-element hash chain with simple integer comparison. This process should not take much more than 100ns to complete.

Not Your Parents' Microprocessor

Figure 2 illustrates the past quarter century's progress in hardware performance. The features that make the new kids (brats) so proud, however, are double-edged swords in SMP systems.

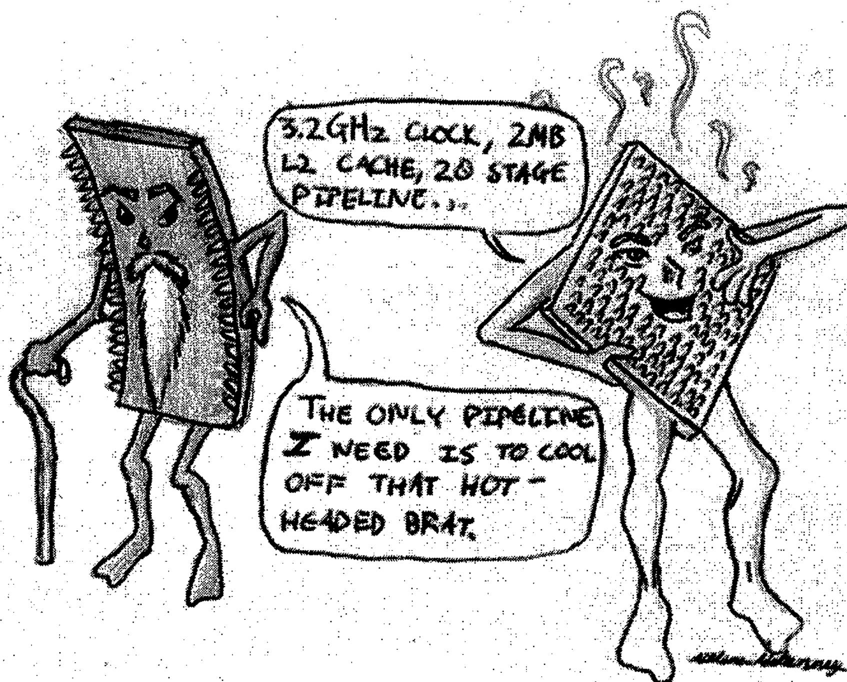

Figure 2. New, fast "brat" processors change the OS design rules.

Unfortunately, many algorithms fail to take advantage of the brat's strengths, because they were developed back when the old man was in his prime. Unless you like slow, stately computing, you need to work with the brat.

Impro
Learn h
out-of-
Read t

Key co
Secure
you in
Read,

The increase in CPU clock frequency has been astounding-where the old man might have been able to interfere with AM radio signals, the young brat might be able to synthesize them digitally. But memory speeds have not increased nearly as fast as CPU clock rates, so a single DRAM access can cost the brat up to a thousand instructions. Although the brat compensates for DRAM latency with large caches, these caches cannot help data bounced among CPUs. For example, when a given CPU acquires a lock, the lock has a 75% chance of being in another CPU's cache. The acquiring CPU stalls until the lock reaches its cache.

Cacheline bouncing explains much of the scaling shortfall in Figure 1, but it does not explain poor single-CPU performance. When there is only one CPU, no other caches are present in which the locks might hide. This is where the brat's 20-stage pipeline shows its dark side. SMP code must ensure that no critical section's instructions or memory operations bleed out into surrounding code. After all, the whole point of a lock is to prevent multiple CPUs from concurrently executing any of the critical section's operations.

Memory barriers prevent such bleeding. These memory barriers are included implicitly in atomic instructions on x86 CPUs, but they are separate instructions on most other CPUs. In either case, locking primitives must include memory barriers. But these barriers cause pipeline flushes and stalls, the overhead of which increases with pipeline length. This overhead is responsible for the single-CPU slowness shown in Figure 1.

Table 1 outlines the costs of basic operations on 700MHz Intel Pentium III machines, which can retire two integer instructions per clock. The atomic operation timings assume the data already resides in the CPU's cache. All of these timings can vary, depending on the cache state, bus loading and the exact sequence of operations.

Table 1. Time Required for Common Operations on a 700MHz Pentium III

Operation	Cost (ns)
Instruction	0.7
Clock cycle	1.4
L2 cache hit	12.9
Atomic increment	58.2
cmpxchg atomic increment	107.3
Main memory	162.4
CPU-local lock	163.7
Cache transfer	170.4-360.9

Practic

Find B
TotalVi
for thre
Get an

Lookin
System
Quick q
Monar

The overheads increase relative to instruction execution overhead. For example, on a 1.8GHz Pentium 4, atomic increment costs about 75ns-slower than the 700MHz Pentium III, despite having a more than twice as fast clock.

These overheads also explain rwlock's poor performance. The read-side critical section must contain hundreds of instructions for it to continue executing once some other CPU read acquires the lock, as illustrated in Figure 3. In this figure, the vertical arrows represent time passing on two pairs of CPUs, one pair using rwlock and the other using spinlock. The diagonal arrows represent data moving between the CPUs' caches. The rwlock critical sections do not overlap at all; the overhead of moving the lock from one CPU to the other rivals that of the critical section.

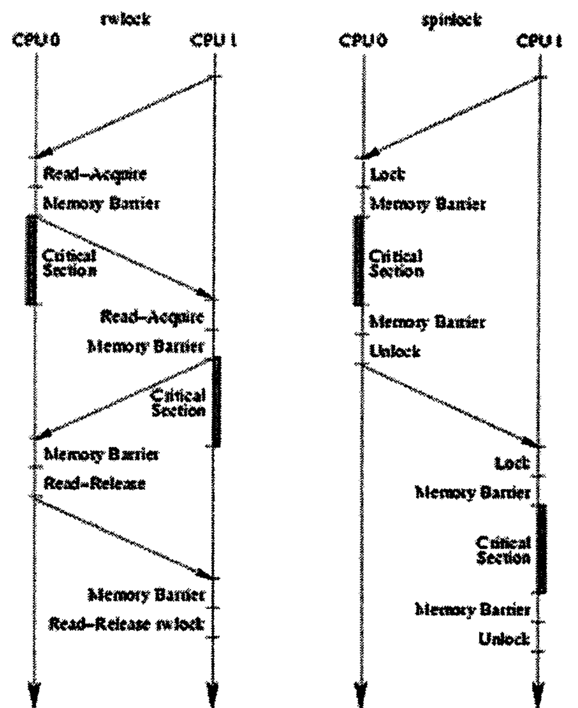

Figure 3. Timelines for rwlock and Spinlock on Two-CPU Systems

Lesson: Avoid Expensive Operations

If you care about performance, you want to avoid these expensive operations. Avoiding them is precisely what RCU does, at least for read-only accesses to read-mostly data structures, although the DEC Alpha still requires some read-side memory barriers. As seen in Figure 4, RCU scales well and has good single-CPU performance for the hash-table-search benchmarklet.

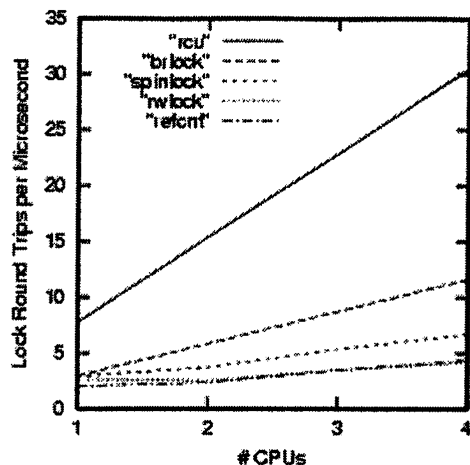

Figure 4. RCU Read Performance by Number of Processors

Of course, updates do slow down RCU, as shown in Figure 5. This graph illustrates the relative performance of these synchronization primitives as the workload varies from read-only (left-hand side) to write-only (right-hand side). RCU is better than brlock across the board. In fact, RCU has replaced brlock in the 2.5 kernel, thanks to Steve Hemminger of OSDL and a number of Linux's networking luminaries. RCU is the best option overall as long as fewer than about one-third of the accesses are updates. Again, your mileage will vary depending on your workload and hardware. In particular, workloads with greater per-element local processing-for example, more complex comparisons-would scale better. As always, use the right tool for the job.

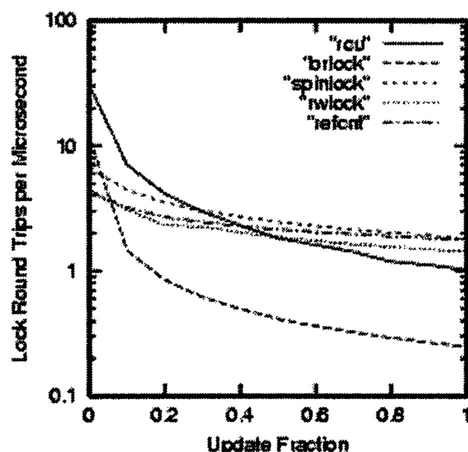

Figure 5. RCU Performance by Fraction of Accesses That Are Updates

How Does RCU Work?

If reading CPUs never make their presence known, how can updating CPUs avoid messing up readers? With locks, the updating CPU examines the lock state to determine when it is safe to carry out the update. With RCU, the updating CPU must make this determination indirectly.

The trick is RCUs reading CPUs are not permitted to block while traversing the data structure, the same as when CPUs holding a spinlock or rwlock are not permitted to block. This means that once an element is unlinked from a list, any CPU that subsequently performs a context switch cannot possibly gain a reference to this element. Context switch is a quiescent state: CPUs undergoing context switches cannot hold references to RCU-protected data structures. Any time period during which all CPUs pass through a quiescent state is a grace period. A CPU may therefore free up an element after a grace period has elapsed from the time that it unlinked the element from the list.

Thus, a simple, though inefficient, RCU-based deletion algorithm could perform the following steps in a non-preemptive Linux kernel:

- Unlink element B from the list, but do not free it-the state of the list as shown in Step 2 of Figure 6.
- Run on each CPU in turn. At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B, as shown in Step 3 (Figure 6).
- Free up element B, as shown in Step 4 (Figure 6).

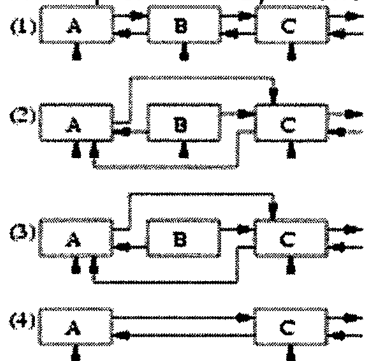

Figure 6. Steps of a Simple RCU-Based Deletion Algorithm

Andrea Arcangeli created a more efficient algorithm that boasts extremely short grace periods, which was the first Linux RCU implementation shipped. Dipankar Sarma coded up an even more efficient RCU implementation that maintains callback cache locality and permits a grace period to service any number of concurrent updates. Dipankar's algorithm is included in the 2.5 kernel and was described in detail at the Ottawa Linux Symposium in 2002.

RCU API

Listing 1 shows the external API for RCU. The synchronize_kernel() function blocks for a full grace period. This is an easy-to-use function, but it incurs expensive context-switch overhead. It also cannot be called with locks held or from interrupt context. However, it does allow concurrent callers to share a grace period.

Listing 1. The RCU API

```
void synchronize_kernel(void);
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
struct rcu_head {
        struct list_head list;
        void (*func)(void *obj);
        void *arg;
};
void rcu_read_lock(void);
void rcu_read_unlock(void);
```

In contrast, call_rcu() schedules a callback function func(arg) after a full grace period. Because call_rcu() never sleeps, it may be called with locks held and from interrupt context. The call_rcu() function uses its struct rcu_head argument to remember its callback function and argument during the grace period. An rcu_head is often placed within the structure being protected by RCU, eliminating the need to allocate it separately.

The primitives rcu_read_lock() and rcu_read_unlock() demark a read-side RCU critical section but generate no code in non-preemptive kernels. In preemptive kernels, they disable preemption, thereby preventing preemption from prematurely ending a grace period. RCU-like mechanisms also may be used in preemptive kernels without suppressing preemption, as demonstrated by the K42 research OS.

Most modern microprocessors feature weak memory-consistency models, which require special memory-barrier instructions. However, these instructions are difficult to understand and even more difficult to test, so the Linux list-manipulation API is extended to include memory barriers, as suggested by Manfred Spraul and as shown in Listing 2. The hlist primitives recently were added by Andi Kleen in order to reduce memory requirements for large hash tables.

Listing 2. RCU Extensions to the Linux List-Manipulation API

```
list_add_rcu(struct list_head *new,
             struct list_head *head);
list_add_rcu_tail(struct list_head *new,
                  struct list_head *head);
list_del_rcu(struct list_head *entry);
list_for_each_rcu(struct list_head *pos,
                  struct list_head *head);
list_for_each_safe_rcu(struct list_head *pos,
                       struct list_head *n,
                       struct list_head *head);
list_for_each_entry_rcu(struct list_head *pos,
                        struct list_head *n,
                        struct list_head *head);
list_for_each_continue_rcu(struct list_head *pos,
                           struct list_head *head);
hlist_del_rcu(struct hlist_node *n);
void hlist_del_init(struct hlist_node *n);
```

```
hlist_add_head_rcu(struct hlist_node *n,
                   struct hlist_head *h);
```

When RCU is applied to data structures other than lists, memory-barrier instructions must be specified explicitly. For an example, see Mingming Cao's RCU-based modifications to System V IPC.

How to Use RCU

Although RCU has been used in many interesting and surprising ways, one of the most straightforward uses is as a replacement for reader-writer locking. In fact, RCU may be thought of as the next step in a progression. The rwlock primitives allow readers to run in parallel with each other, but not in parallel with updaters. RCU, on the other hand, allows readers to run in parallel both with each other and with updaters.

This section presents the analogy between rwlock and RCU, protecting the simple doubly linked list data structure shown in Listing 3 with reader-writer locks and then with RCU. This structure has a struct list_head, a search key, a single integer for data and a struct rcu_head.

Listing 3. A Data Structure Protected by RCU

```
struct el {
     struct list_head list;
     long key;
     long data;
     struct rcu_head my_rcu_head;
};
```

The reader-writer-lock/RCU analogy substitutes primitives as shown in Table 2. The asterisked primitives are no-ops in non-preemptible kernels; in preemptible kernels, they suppress preemption, which is an extremely cheap operation on the local task structure. Because rcu_read_lock() does not block interrupt contexts, it is necessary to add primitives for this purpose where needed. For example, read_lock_irqsave must become rcu_read_lock(), followed by local_irq_save().

Table 2. Reader-Writer Lock and RCU Primitives

Reader-Writer Lock	Read-Copy Update
rwlock_t	spinlock_t
read_lock()	rcu_read_lock() *
read_unlock()	rcu_read_unlock() *
write_lock()	spin_lock()

write_unlock()	spin_unlock()
list_add()	list_add_rcu()
list_add_tail()	list_add_tail_rcu()
list_del()	list_del_rcu()
list_for_each()	list_for_each_rcu()

Table 3 illustrates this transformation for some simple linked-list operations. Notice that line 10 of the rwlock delete() function is replaced with a call_rcu() that delays the invocation of my_free() until the end of a grace period. The rest of the functions are transformed in a straightforward fashion, as indicated in Table 2. Garrick making table 3.

Although this analogy can be quite compelling and useful-in Dipankar Sarma's and Maneesh Soni's use of RCU in dcache, for example-there are some caveats:

- Read-side critical sections may see stale data that has been removed from the list but not yet freed. There are some situations where this is not a problem, such as routing tables for best-effort protocols. In other situations, such stale data may be detected and ignored, as happens in the 2.5 kernel's System V IPC implementation.
- Read-side critical sections may run concurrently with write-side critical sections.
- The grace period delays the freeing of memory, which means the memory footprint is somewhat larger when using RCU than it is when using reader-writer locking.
- When changing to RCU, write-side reader-writer locking code that modifies list elements in place often must be restructured to prevent read-side RCU code from seeing the data in an inconsistent state. In many cases, this restructuring is quite straightforward; for example, you could create a new list element with the desired state, then replace the old element with the new one.

Where it applies, this analogy can deliver full parallelism with hardly any increase in complexity.

RCU Synchronizing with NMIs

Retrofitting existing code with RCU as shown above can produce significant performance gains. The best results, of course, are obtained by designing RCU into the algorithms and code from the start.

The i386 oprofile code contains an excellent example of designed-in RCU. This code can use NMIs (nonmaskable interrupts) to handle profiling independently of the normal clock interrupt, which permits profiling of the clock interrupt handler. Synchronizing with NMIs traditionally has been difficult; by definition, no way exists to block an NMI. Straightforward locking designs therefore are subject to

deadlock, where the CPU holding the lock receives an NMI, and the NMI handler spins forever on this same lock. Another approach is to mask NMIs in software using things like spin_trylock(). This method, however, produces cache bouncing and memory-barrier overhead, and the NMIs thus masked are lost. The solution in nmi_timer_int.c is as shown in Listing 4.

Listing 4. Using RCU in nmi_timer_int.c

```
static void timer_stop(void)
{
        enable_timer_nmi_watchdog();
        unset_nmi_callback();
        synchronize_kernel();
}
static struct oprofile_operations nmi_timer_ops = {
        .start   = timer_start,
        .stop    = timer_stop,
        .cpu_type = "timer"
};
```

The synchronize_kernel() ensures that any NMI handlers executing the old NMI callback upon entry to timer_stop() have completed before timer_stop() returns. The code for oprofile_stop() and oprofile_shutdown() shown in Listing 5 illustrates why this is important. Notice that oprofile_ops->stop() invokes timer_stop(). Therefore, if oprofile_stop() and oprofile_shutdown() were called in quick succession, the newly freed CPU buffers could be accessed by an ongoing NMI. This action could surprise any code quickly reallocating this memory.

Listing 5. More Code from nmi_timer_int.c

```
void oprofile_stop(void)
{
        down(&start_sem);
        if (!oprofile_started)
                goto out;
        oprofile_ops->stop();
        oprofile_started = 0;
        /* wake up the daemon to read remainder */
        wake_up_buffer_waiter();
out:
        up(&start_sem);
}
void oprofile_shutdown(void)
{
        down(&start_sem);
        sync_stop();
        if (oprofile_ops->shutdown)
                oprofile_ops->shutdown();
        is_setup = 0;
        free_event_buffer();
        free_cpu_buffers();
        up(&start_sem);
```

```
}
```

Use of RCU eliminates this race naturally, without incurring any locking or memory-barrier overhead.

Incremental Use of RCU

Garrick, please kern the double underscores in the list below. Using RCU is not an all-or-nothing affair. It may be applied incrementally to particular code paths as needed. A good example of this is a patch coded by Dipankar Sarma that prevents ls /proc from blocking fork(). The changes are as follows:

1. The read_lock() and read_unlock() of tasklist_lock in get_pid_list() are replaced by rcu_read_lock() and rcu_read_unlock(), respectively.
2. A struct rcu_head is added to task_struct in order to track the task structures waiting for a grace period to expire.
3. The put_task_struct() macro invokes __put_task_struct() via call_rcu() rather than directly. This ensures that all concurrently executing get_pid_list() invocations complete before any task structures that they might have been referencing are freed.
4. The SET_LINKS() and REMOVE_LINKS() macros use the _rcu form of the list-manipulation primitives.
5. The for_each_process() macro gets a read_barrier_depends() to make this code safe for the DEC Alpha.
   The problem is get_pid_list() traverses the entire tasklist in order to build the PID list needed by ls /proc. It read-holds tasklist_lock during this traversal and blocks updates to the tasklist, such as those performed by fork(). On machines with large numbers of tasks, this can cause severe difficulties, particularly given multiple instances of certain performance-monitoring tools.

Dipankar's modifications are shown in Table 4, changing only two files, adding 13 lines and deleting seven for a six-line net addition to the kernel. This patch does delete a pair of tasklist_lock uses, but none of the other 249 uses of tasklist_lock are modified. This example demonstrates use of RCU for a late-in-cycle optimization.

Garrick making Table 4.

Where Do We Go from Here?

RCU will become more important as CPU architecture continues to evolve. Nonetheless, other primitives always will be needed. It is quite likely that Rusty Russell's implementation of RCU (the call_rcu () and synchronize_kernel() primitives themselves) can be modified to be entirely free of locks, memory barriers and atomic instructions. This implementation might run faster than the current 2.5 kernel implementation.

Numerous people are looking at new uses of RCU in the VFS layer, VM code, filesystems and networking code. I look forward to

continuing to learn about RCU and its uses and am grateful to the many people who have tried it out.

Paul E. McKenney has worked on SMP and NUMA algorithms for longer than he cares to admit. Prior to that, he worked on packet-radio and Internet protocols (but long before the Internet became popular). His hobbies include running and the usual house-wife-and-kids habit. This work represents the view of the author and does not necessarily represent the view of IBM. 6993aa.jpg

» **add new comment** | **email this page** | **printer friendly version**

Comment viewing options

Threaded list - expanded | Date - newest first

50 comments per page | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

Can you explain a little more about smp_wmb?

Submitted by Anonymous on Mon, 2004-02-23 02:00.

It seems that smp memory barrier is tightly linked with RCU. Codes taking advantage of RCU also use smp_wmb to deal with weak memory-consistency processes, for example, in the routing cache. So, I think understanding this thing is key to the "USEING" of RCU. Thanks.

» **reply** | **email this page**

Re: Can you explain a little more about smp_wmb?

Submitted by Anonymous on Tue, 2004-04-06 01:00.

When you are using the linux/list.h _rcu macros along with normal locking to protect against concurrent updates, the memory barriers are taken care of for you. Many of the places where one can use RCU do involve linked lists, so this works much of the time. However, if you are in a situation where you cannnot use these macros, then you are quite right that an understanding of memory barriers is required.

Modern CPUs are within their rights to reorder operations unless explicitly told not to. Therefore, locking primitives on many CPUs contain memory barriers that prevent (for example) the contents of the critical section from "bleeding

out" into the surrounding code. Any such "bleeding" would mean that part of the critical section was no longer protected by the lock, which would result in failure. Hence the memory barriers in locking primitives.

This situation means that lock-free operations require explicit memory barriers. A full explanation is beyond the scope of this comment, but the LKML thread starting with **this message** and this **web page** are a place to start. If you want a full treatment, **Gharachorloo's Ph.D. thesis** is a place to finish.

» **reply** | **email this page**

Magazine | Advertise | Community | Contact Us | Privacy Statement
Copyright © 1994 - 2006 SSC Media Corporation. All rights reserved.
Visit SSC's other sites: TUX Magazine | Doc's IT Garage | RSS Links
Read "News Feeds" from SSC's other sites: News Feeds | Report Problems