Deterministic Synchronization in Multicore Systems: the Role of RCU

Paul E. McKenney
IBM Linux Technology Center
15350 SW Koll Parkway, Beaverton, OR 97006 USA
paulmck@linux.vnet.ibm.com

Abstract

Although real-time operating systems and applications have been available for multicore systems for some years, shared-memory parallel systems still pose some severe challenges for real-time algorithms, particularly as the number of CPUs increases. These challenges can take the form of lock contention, memory contention, conflicts/restarts for lockless algorithms, as well as many others. One technology that has been recently added to the real-time arsenal is a preemptable implementation of read-copy update (RCU), which permits deterministic read-side access to read-mostly data structures, even in the face of concurrent updates. In some cases, updates may also be carried out in a deterministic manner.

1 Introduction

RCU is a synchronization mechanism that was accepted into the Linux kernel in late 2002, though precursors to RCU date back to at least 1980 [4], with a more complete history available elsewhere [6, Section 2.2.20 and Table 3.2]. RCU improves scalability by permitting reads to run concurrently with updates. This is a departure from most prior synchronization primitives, for example, conventional locking primitives allow only one access at at time, and even reader-writer locking primitives, while allowing multiple concurrent readers, do not permit readers to run concurrently with updaters. More recent primitives, such as seqlock, appear to allow concurrent readers and updaters, but such readers are always retried. Research work on transactional memory permits concurrent readers and updaters, but conflicting accesses will result in rollback and retry. Although it is possible to construct real-time systems using blocking and rollback-retry primitives, primitives with better determinism would be desirable.

In contrast, RCU enables readers and updaters to run concurrently, with no blocking and no rollback-retry. Updaters handle conflicts with concurrent readers by maintaining multiple versions of data structures, and ensuring that old versions are not freed up (or, more generally, *reclaimed*) until all pre-existing readers (which might have references to the old versions) have completed. RCU also provides efficient and scalable mechanisms for publishing and subscribing to new data structures, and also for retracting previously published data, by deferring reclamation of old data structures that have since been rendered inaccessible to readers.

The RCU mechanisms distribute work among readers and updaters so as to make read paths extremely fast. In fact, in some cases, but unfortunately for neither real-time kernels nor user applications, RCU's read-side primitives can have zero overhead. In addition, as we will see, RCU is somewhat specialized, being intended primarily to protect read-mostly data structures.

Section 2 gives a brief overview of RCU, Section 3 recapitulates the progress made over the past five years towards a high-performance real-time implementation of RCU, Section 4 discusses where one might use RCU to meet real-time latency requirements, and finally, Section 5 presents concluding remarks.

2 What is RCU?

At its most basic level, RCU is a way of waiting for other things to get done. An RCU implementation must implement three fundamental operations [17]:

1. Updaters publish new data in presence of concurrent readers.

2. Readers subscribe to existing data in presence of concurrent updaters.

3. Updaters retract data previously published in presence of concurrent readers.

The retraction operation is particularly interesting, as it is not possible to retract data that a reader is currently subscribed to, given that readers cannot be blocked, rolled back, or aborted. Instead, updaters first make the data inaccessible to new readers, and then wait for a "grace period" to elapse, with the grace period being sufficiently long to permit all pre-existing readers to finish with their subscriptions. Once this grace period has completed, there will be no subscribers to the data, so that destructive operations (such as freeing) may safely be carried out.

Publication and subscription can be implemented deterministically, and are thus prime candidates for real-time use. Retraction's long grace periods often rules it out for use on real-time critical code paths, but it is possible to use asynchronous primitives in cases where critical code paths need to start a grace period, but need not wait for it to finish.

These fundamental operations can be used to implement a reader-writer locking replacement, a bulk reference counter, a "poor man's" garbage collector, and a form of existence guarantee, in many cases with extremely low and deterministic overhead [10]. Each of these three operations is described in the following sections.

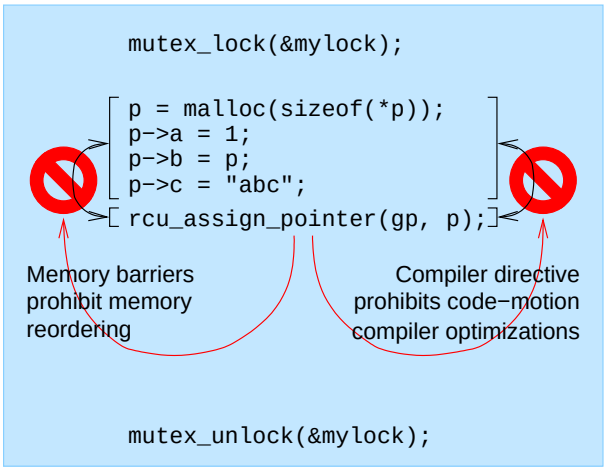

FIGURE 1: *RCU Publication Constraints*

The `rcu_assign_pointer()` primitive is a normal pointer assignment (hence the name), but possibly augmented with memory barriers and compiler directives. Such directives are often required in order to prohibit CPU or compiler optimizations that would otherwise cause previous initializations to be seen by RCU readers as occurring after the pointer assignment. An example of these prohibitions is shown in Figure 1. Please note that the need for such constraints is not specific to RCU. For example, locking primitives typically use memory barriers and compiler directives in order to prevent the CPU and the compiler from moving code out of the lock's critical section.

RCU may be used to publish into more complex data structures, including linked lists, trees, and hash tables. The insertion primitives for such RCU-protected data structures, for example, `list_add_rcu()`, contain carefully placed calls to `rcu_dereference()`.

Note that RCU coordinates among a single updater and concurrent readers. Some other synchronization primitive (typically locking) must be used to coordinate among concurrent updaters, as exemplified by the `mutex_lock()` and `mutex_unlock()` in Figure 1.

2.1 RCU Publication

RCU publication uses the `rcu_assign_pointer()` primitive. This primitives publishes a pointer to a structure, and guarantees that any initialization carried out prior to publication is seen by any RCU reader subscribing to the newly published structure.

2.2 RCU Subscription

RCU subscription uses the `rcu_dereference()` primitive to subscribe to structures published by `rcu_assign_pointer()`, returning a pointer to the specified structure. However, two additional primitives, `rcu_read_lock()` and `rcu_read_unlock()`,

are also required to bound the section of code throughout which the subscribed pointer may safely be dereferenced. Such sections of code are called "RCU read-side critical sections".

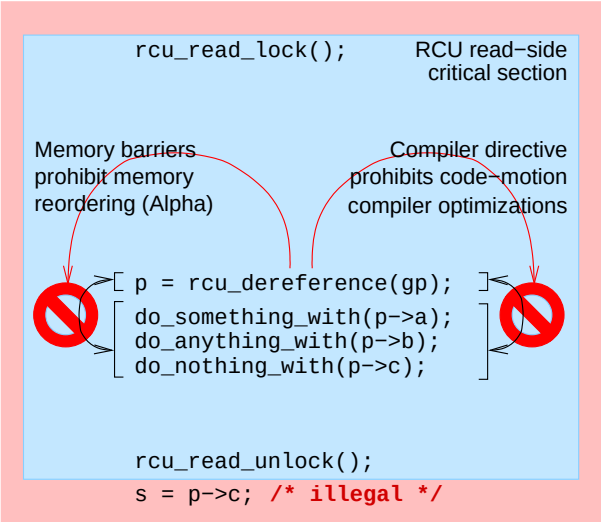

FIGURE 2: *RCU Subscription Constraints*

The `rcu_dereference()` primitive also acts as a normal pointer assignment, but again augmented with compiler directives and memory barriers. However, in contrast with `rcu_assign_pointer()`, `rcu_dereference()` requires memory barriers only when running on DEC Alpha. As with `rcu_assign_pointer()`, the directives and barriers associated with `rcu_dereference()` prohibit CPU and compiler optimiations that could otherwise permit the reader to see pre-initialed data, as shown in Figure 2.

Optimizations capable of disrupting users of `rcu_dereference()` can be quite counter-intuitive. An example optimization involves pointer-value speculation, which can result in the following sequence of events:

1. CPU 0 executes code that speculates the value that will be returned by `rcu_dereference()`, but incorrectly guesses a pointer that currently points into the freelist.

2. CPU 0 continues blithely executing `do_something_with()`, `do_anything_with()`, and even `do_nothing_with()`, making wild references into unallocated memory.

3. CPU 1 decides to publish a new data structure, and has the bad luck to have `malloc()` return exactly the value that CPU 0 guessed.

4. CPU 1 initializes the fields of the newly allocated structure, then uses `rcu_assign_pointer()` to publish it.

5. CPU 0 now validates the speculated value, finds that it matches, and therefore incorrectly concludes that its speculation was OK.

The `rcu_dereference()` primitive must prevent such speculation, as shown in Figure 2. In some (but not all!) RCU implementations, the `rcu_read_lock()` and `rcu_read_unlock()` primitives must also constrain the compiler and CPU.

As with `rcu_assign_pointer()`, `rcu_dereference()` is used in access primitives of more complex data structures, for example, the `list_for_each_entry_rcu()` list-traversal primitive.

2.3 RCU Retraction

RCU retraction is a three-step process that typically makes use of a data-element-removal primitive such as `list_del_rcu()`, the `synchronize_rcu()` primitive, and some reclamation primitive such as `free()`. This retraction process is shown in Figure 3, along with the relationship between retraction and subscription.

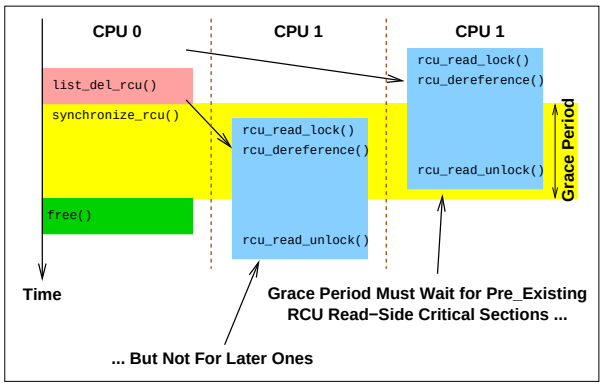

FIGURE 3: *RCU Retraction*

A key point about the retraction process is that only pre-existing readers can possibly hold a reference to the newly removed element. In the figure, the right-most reader might reference the data element, due to the fact that the reader started execution before the element was removed. In contrast, the left-most reader cannot possibly hold a reference, as it did not start execution until the removal step completed. RCU's `synchronize_rcu()` primitive takes

advantage of this distinction, waiting only for pre-existing RCU readers, not for any that start execution afterwards.

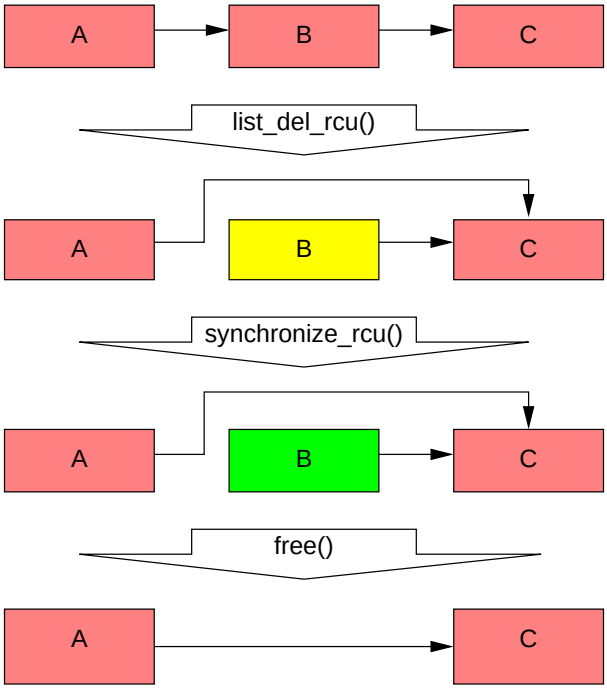

FIGURE 4: *RCU Retraction For Deletion From List*

Figure 4 shows how RCU retraction may be used to safely remove element B from a linked list containing elements A, B, and C. Initially, readers might have subscribed to (and thus might hold references to) each of these elements, signified by their red color. The `list_del_rcu()` primitive unlinks element B from the list, but leaves the element itself intact, including the pointer to element C, as shown by the second row of Figure 4. This permits readers holding concurrent references to element B to proceed normally, but prevents any new readers from acquiring such a reference, as signified by element B's yellow color. Therefore, pre-existing readers might still see the list as {A, B, C}, but because new readers have no way to gain a reference to element B, they will see only {A, C}.

The `synchronize_rcu()` primitive then waits for all pre-existing concurrent readers to exit their RCU read-side critical sections, in other words, `synchronize_rcu()` waits for a grace period to complete. Because RCU readers' subscriptions are not valid outside of an RCU read-side critical section, once all such pre-existing RCU read-side critical sections have completed, there can be no readers holding references to element B, as signified by its green color in the third row.

Because no readers can possibly hold a reference to element B, it is now safe for the updater to carry out destructive operations, in this case, `free()`. This leaves the list containing only elements A and C, as shown in the last row of Figure 4.

Retraction as described above is normally not used in critical real-time code paths due to the fact that grace periods can extend for many milliseconds. However, there is an asynchronous counterpart to the `synchronize_rcu()` primitive named `call_rcu()`. The `call_rcu()` primitive registers a function and argument to be invoked at the end of a subsequent grace period, but incurs only queuing overhead on the critical code path. In addition, the advent of expedited RCU grace periods might enable some real-time applications to place expedited grace periods on critical code paths using the new `synchronize_rcu_expedited()` primitive [11].

2.4 Discussion

A large number of concurrent algorithms may be constructed using RCU's publication, subscription, and retraction primitives. However, such algorithms are useful only if they provide adequate performance, scalability, and real-time response. Because RCU is designed primarily for read-mostly (or subscription-mostly) situations, the next section will give a rough feel for the overhead of RCU's read-side primitives.

3 Towards a High-Performance Real-Time RCU Implementation

This section gives a quick overview of some Classic RCU and Preemptable RCU read-side primitives, followed by some references to work on user-level RCU implementations. This section is only intended to give a rough feel for the overhead of these primitives, but citations are given to direct the reader to additional information.

3.1 Classic RCU

Server-class (`!CONFIG_PREEMPT`) builds of the Linux kernel use the extremely efficient implementations for `rcu_read_lock()` and `rcu_read_unlock()` shown in Table 1. It is hard to imagine an implementation with less overhead.

Unfortunately, `!CONFIG_PREEMPT` kernel builds

```
1 static inline void rcu_read_lock(void)
2 {
3 }
4
5 static inline void rcu_read_unlock(void)
6 {
7 }
```

Table 1: Server-Class RCU Read-Side Primitives

```
1 static inline void rcu_read_lock(void)
2 {
3   preempt_disable();
4 }
5
6 static inline void rcu_read_unlock(void)
7 {
8   preempt_enable();
9 }
```

Table 2: Desktop-Class RCU Read-Side Primitives

have unimpressive real-time capabilities, so `CONFIG_PREEMPT` desktop-class real-time support was added during the Linux 2.5 kernel development effort. This adds some overhead to the RCU read-side primitives, as can be seen in Table 2. These primitives are still very light-weight, updating a field in the local `thread_info` structure, and, in the case of `rcu_read_unlock()`, also checking to see if the scheduler needs to be invoked, and invoking it if so. However, the fact that preemption is disabled for the duration of each RCU read-side critical section does not help real-time response.

This situation lead to the development of Preemptable RCU.

3.2 Preemptable RCU

The read-side primitives for preemptable RCU are shown in Table 3. These primitives have been described fully elsewhere [15], so we will instead simply count the number of atomic instructions and memory barriers, which can be as many as two each for both `rcu_read_lock()` and `rcu_read_unlock()`, for a total of up to four for a given read-side critical section. To add insult to injury, both primitives also disable interrupts.

Table 4 shows the current mainline implementation of the RCU read-side primitives. Again, these have been described in detail elsewhere [7], so we will instead count expensive operations. There are no atomic instructions or memory barriers, but these primitives still disable interrupts. In addition, there are nine primitives that constrain the compiler and quite a bit more code. So while the current mainline preemptable-RCU implementation is a decided improvement over the early implementation, it is still quite heavyweight compared to the non-preemptable implementations shown above.

It turns out to be possible to further reduce read-side overhead, as shown in Table 5. This implementation does not use atomic instructions, memory barriers, or interrupt disabling, and has only five constraints on the compiler. Furthermore, this experimental implementation executes only four lines of C code in the common case, many fewer than the earlier implementations. The uncommon case is quite heavyweight, but is invoked only in exceptional circumstances, such as when the read-side critical section actually did block or was preempted—in which case the overhead of the resulting context switches dominates the overhead [12]. The overhead of this variant of `rcu_read_lock()` and `rcu_read_unlock()` can be expected to be roughly that of the desktop-class variant shown in Table 2.

3.3 Catalog of User-Level RCU Implementations

Of course, a real-time RCU implementation in an operating-system kernel is all well and good, but it does not provide much help to real-time applications.

```
 1 void rcu_read_lock(void)
 2 {
 3   int f;
 4   unsigned long oldirq;
 5   struct task_struct *t = current;
 6
 7   raw_local_irq_save(oldirq);
 8   if (t->rcu_read_lock_nesting++ == 0) {
 9     f = rcu_ctrlblk.completed & 1;
10     smp_read_barrier_depends();
11     t->rcu_flipctr1 =
12       &(__get_cpu_var(rcu_flipctr)[f]);
13     atomic_inc(t->rcu_flipctr1);
14     smp_mb__after_atomic_inc();
15     if (f != (rcu_ctrlblk.completed & 1)) {
16       t->rcu_flipctr2 =
17         &(__get_cpu_var(rcu_flipctr)[!f]);
18       atomic_inc(t->rcu_flipctr2);
19       smp_mb__after_atomic_inc();
20     }
21   }
22   raw_local_irq_restore(oldirq);
23 }
```

```
 1 void rcu_read_unlock(void)
 2 {
 3   unsigned long oldirq;
 4   struct task_struct *t = current;
 5
 6   raw_local_irq_save(oldirq);
 7   if (--t->rcu_read_lock_nesting == 0) {
 8     smp_mb__before_atomic_dec();
 9     atomic_dec(t->rcu_flipctr1);
10     t->rcu_flipctr1 = NULL;
11     if (t->rcu_flipctr2 != NULL) {
12       atomic_dec(t->rcu_flipctr2);
13       t->rcu_flipctr2 = NULL;
14     }
15   }
16   raw_local_irq_restore(oldirq);
17 }
```

Table 3: Early Preemptable RCU Read-Side Primitives

```
 1 void __rcu_read_lock(void)
 2 {
 3   int idx;
 4   struct task_struct *t = current;
 5   int nesting;
 6
 7   nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
 8   if (nesting != 0) {
 9     t->rcu_read_lock_nesting = nesting + 1;
10   } else {
11     unsigned long flags;
12
13     local_irq_save(flags);
14     idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1;
15     ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])++;
16     ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1;
17     ACCESS_ONCE(t->rcu_flipctr_idx) = idx;
18     local_irq_restore(flags);
19   }
20 }
```

```
 1 void __rcu_read_unlock(void)
 2 {
 3   int idx;
 4   struct task_struct *t = current;
 5   int nesting;
 6
 7   nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
 8   if (nesting > 1) {
 9     t->rcu_read_lock_nesting = nesting - 1;
10   } else {
11     unsigned long flags;
12
13     local_irq_save(flags);
14     idx = ACCESS_ONCE(t->rcu_flipctr_idx);
15     ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
16     ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])--;
17     local_irq_restore(flags);
18   }
19 }
```

Table 4: Current Mainline Preemptable RCU Read-Side Primitives

```
 1 void __rcu_read_lock(void)
 2 {
 3   ACCESS_ONCE(current->rcu_read_lock_nesting)++;
 4   barrier();
 5 }
 6
 7 void __rcu_read_unlock(void)
 8 {
 9   struct task_struct *t = current;
10
11   barrier();
12   if (--ACCESS_ONCE(t->rcu_read_lock_nesting) == 0 &&
13       unlikely(ACCESS_ONCE(t->rcu_read_unlock_special)))
14     rcu_read_unlock_special(t);
15 }
```

Table 5: Experimental Preemptable RCU Read-Side Primitives

Fortunately, there have been a number of user-level RCU algorithms put forward over the past five years.

In 2006, Hart et al. [3] used a user-level variant of Classic RCU for benchmarking purposes. This implementation is sufficient for real-time applications whose architectures guarantee that each thread will periodically pass through a quiescent state.

In 2008, McKenney put forward a number of user-level RCU implementations, primarily for the purpose of debugging Linux kernel code at user level and to serve as an introduction to the more-complex implementations in the Linux kernel [8, 13]. These algorithms, though simple, all either impose restrictions on the application or suffer from high read-side overhead.

In 2009, Desnoyers adapted preemptable RCU for user-level use by substituting POSIX signals for the scheduling-clock interrupt [2]. This approach allows good read-side overhead combined with minimal restrictions on the application (namely, that it be willing to give up a signal).

Given this recent progress, it seems safe to say that RCU is ready to take on user-level applications, including real-time applications.

4 Where to Use RCU

Details on the use of RCU has been discussed at length [1, 9, 10], although continued progress indicates that there is still much more to be learned. This section will instead focus on the situations to which RCU is best applied, as summarized by Figure 5.

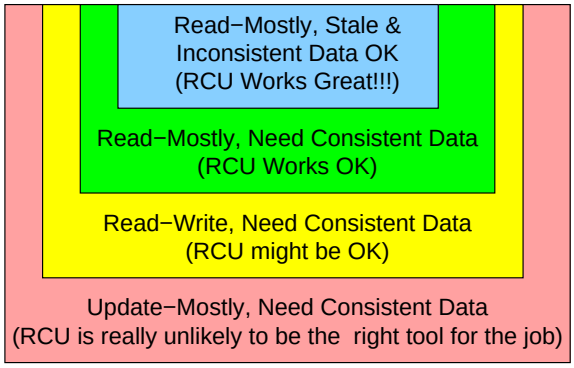

FIGURE 5: *RCU Areas of Applicability*

The situation most favorable to RCU is for read-mostly data structures where stale and inconsistent data can be tolerated. The prototypical use is routing tables [14], which maintain in-memory state that represents outside-of-system connectivity. Changes in outside-of-system connectivity result in routing-table updates, but these changes often take significant time to propagate to the system. This propagation delay means that the system has been misrouting for some tens of seconds or even minutes, so that the additional few milliseconds required to retract the old data is not significant. The benefit is that readers gain deterministic access to the data.

RCU can also be quite helpful for read-mostly data structures where consistent and fresh data is required. For example, in the Linux kernel's System V semaphore implementation, RCU protects the data structures mapping from the user-mode semid to the in-kernel `sem_array` structure, but the structure itself is protected by a spinlock in the enclosed

kern_ipc_perm structure [1]. This permits deterministic mapping from semid to sem_array, while the spinlock reintroduces freshness and consistency, but at the level of an individual sem_array structure rather than globally.

RCU is less likely to be helpful for read-write data structures where consistent and fresh data is required. That said, the dcache system is one such example, where RCU is used to map from a pathname segment and parent dentry to the child dentry. Per-dentry locking and a global seqlock are used to impose freshness and consistency to pathname lookups [5, 16].

Finally, if the data structure is updated more often than it is read, and if consistent and fresh data are required, RCU is quite unlikely to be the right tool for the job.

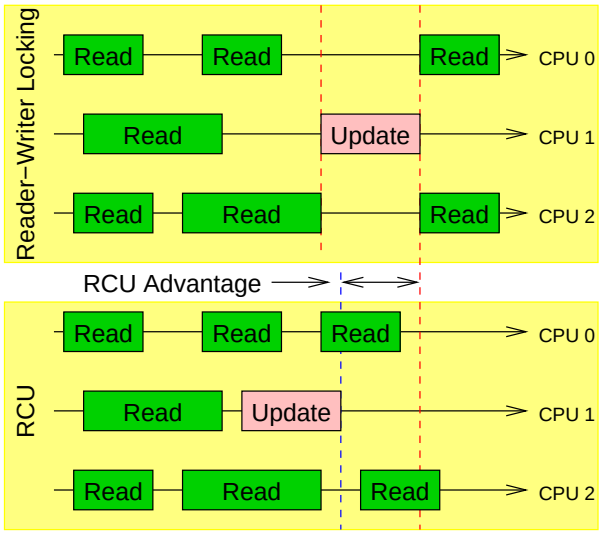

FIGURE 6: *Staleness for Reader-Writer Locking and RCU*

A surprising number of algorithms tolerate access to stale data. A key question to ask is "what defines stale?" It is natural to assume that the data is by definition fresh when it first arrives at the system, however, communication latencies often invalidate this assumption. When data is stale upon arrival, as it was in the routing-table example above, RCU can often provide greater freshness than can other synchronization primitives such as reader-writer locking. This can be seen in Figure 6, where reader-writer locking delays the update, thus resulting in the update happening later than for RCU. Therefore, if the freshness of the data is defined externally, use of reader-writer locking can result in data being more stale than for RCU.

Additional information on uses of RCU from the Linux kernel may be found in McKenney's dissertation [6, Chapter 6].

5 Conclusions

The past year has seen a number of user-level RCU implementations that promise to be useful in real-time applications, and also the beginnings of a much simpler and faster implementation of in-kernel preemptable (real-time) RCU. These developments promise to bring new options for real-time applications running on multi-core hardware.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Dietrich, K.R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. We all owe Darren Hart a debt of gratitude for his efforts to render this document human-readable. Finally, I owe thanks to Kathy Bennett for her support of this effort.

Legal Statement

References

[1] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)* (June 2003), USENIX Association, pp. 297–310. Available: http://www.rdrop.com/users/paulmck/RCU/rcu.FREENIX.2003.06.14.pdf [Viewed November 21, 2007].

[2] Desnoyers, M. [RFC git tree] userspace RCU (urcu) for Linux. Available: http://lkml.org/lkml/2009/2/5/572 git://lttng.

org/userspace-rcu.git [Viewed February 20, 2009], February 2009.

[3] Hart, T. E., McKenney, P. E., and Brown, A. D. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium* (Rhodes, Greece, April 2006). Available: http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf [Viewed April 28, 2008].

[4] Kung, H. T., and Lehman, Q. Concurrent maintenance of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382. Available: http://portal.acm.org/citation.cfm?id=320619&dl=GUIDE, [Viewed December 3, 2007].

[5] Linder, H., Sarma, D., and Soni, M. Scalability of the directory entry cache. In *Ottawa Linux Symposium* (June 2002), pp. 289–300.

[6] McKenney, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed October 15, 2004].

[7] McKenney, P. E. The design of preemptible read-copy-update. Available: http://lwn.net/Articles/253651/ [Viewed October 25, 2007], October 2007.

[8] McKenney, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2008. Available: git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git [Viewed August 15, 2009].

[9] McKenney, P. E. RCU part 3: the RCU API. Available: http://lwn.net/Articles/264090/ [Viewed January 10, 2008], January 2008.

[10] McKenney, P. E. What is RCU? part 2: Usage. Available: http://lwn.net/Articles/263130/ [Viewed January 4, 2008], January 2008.

[11] McKenney, P. E. [PATCH -tip 0/3] expedited 'big hammer' RCU grace periods. Available: http://lkml.org/lkml/2009/6/25/306 [Viewed August 16, 2009], June 2009.

[12] McKenney, P. E. [PATCH RFC -tip 0/4] RCU cleanups and simplified preemptable RCU. Available: http://lkml.org/lkml/2009/7/23/294 [Viewed August 15, 2009], July 2009.

[13] McKenney, P. E. Using a malicious user-level RCU to torture RCU-based algorithms. In *linux.conf.au 2009* (Hobart, Australia, January 2009). Available: http://www.rdrop.com/users/paulmck/RCU/urcutorture.2009.01.22a.pdf [Viewed February 2, 2009].

[14] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: http://www.linuxsymposium.org/2001/abstracts/readcopy.php http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf [Viewed June 23, 2004].

[15] McKenney, P. E., Sarma, D., Molnar, I., and Bhattacharya, S. Extending rcu for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184 http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf [Viewed January 1, 2007].

[16] McKenney, P. E., Sarma, D., and Soni, M. Scaling dcache with RCU. *Linux Journal 1*, 118 (January 2004), 38–46.

[17] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: http://lwn.net/Articles/262464/ [Viewed December 27, 2007], December 2007.