

The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux



D. Guniguntala
P. E. McKenney
J. Triplett
J. Walpole

Read-copy update (RCU) is a synchronization mechanism in the Linux™ kernel that provides significant improvements in multiprocessor scalability by eliminating the writer-delay problem of readers-writer locking. RCU implementations to date, however, have had the side effect of expanding non-preemptible regions of code, thereby degrading real-time response. We present here a variant of RCU that allows preemption of read-side critical sections and thus is better suited for real-time applications. We summarize priority-inversion issues with locking, present an overview of the RCU mechanism, discuss our counter-based adaptation of RCU for real-time use, describe an additional adaptation of RCU that permits general blocking in read-side critical sections, and present performance results. We also discuss an approach for replacing the readers-writer synchronization with RCU in existing implementations.

INTRODUCTION

In this paper we focus on environments in which real-time applications are running on shared-memory multiprocessor systems with the Linux** operating system. Such environments require both real-time response and multiprocessor scalability. Real-time response means that the hardware and the operating system perform within real-time constraints; that is, the response times to certain events are subject to operational deadlines. Multiprocessor scalability means that the system can process growing amounts of work when the level of multiprocessing is proportionally increased. Tech-

niques exist for meeting these requirements independently, but real-time response is often attained at the expense of multiprocessor scalability, and vice versa. Given the recent advent of low-cost multiprocessor systems, techniques that address both requirements are now needed.

©Copyright 2008 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/08/\$5.00 © 2008 IBM

Common sources of delay that affect real-time response in existing kernels include: (a) priority inversions due to locking; (b) writer delay due to concurrent readers in readers-writer locking; and (c) non-preemptible sections of code.

Priority inversion, first noted by Lampson and Redell,¹ occurs when a high-priority thread is blocked in lock acquisition by a lower-priority lock holder. It is further possible for the lower-priority lock holder to experience an indefinite scheduling delay due to the existence of a medium-priority thread executing outside the critical section (i.e., the section of code that accesses a shared resource that must not be concurrently accessed by more than one thread). Hence, the low-priority lock holder may never be scheduled to run and release the lock required by the high-priority thread. A common solution to the priority-inversion problem is to use a locking-aware scheduling approach, called priority inheritance. With priority inheritance, the scheduling priority of the lock holder is temporarily boosted to the priority of the highest-priority thread blocked in lock acquisition.

Priority inheritance works well for conventional locks, but is difficult to apply to readers-writer locks without negatively impacting concurrency.² Consider the case of a high-priority writer blocked by the existence of one or more low-priority readers. In this case, the priority of the existing readers must be boosted, but it is unclear how to handle the additional readers that arrive prior to the completion of the preexisting boosted readers. If their priority is not boosted, then the priority-inversion problem remains. If they are delayed until the writer completes, then concurrency is reduced. And if priority boosting is applied to them, this introduces the possibility of a nonvanishing queue of priority-boosted threads that can delay the writer indefinitely. This problem is inherent to readers-writer locking: either the writer is expedited at the expense of reader concurrency, or the writer risks indefinite delay.

Fair readers-writer locks that take requests in first-in, first-out (FIFO) order³ expedite writers at the expense of reader concurrency, and suffer especially low concurrency in the presence of interleaved reader and writer requests. This concurrency problem can be mitigated by a batch-fair readers-writer lock that services a limited number of read requests

out of order, but writers can still experience excessively long wait times, especially in the presence of readers with long-executing critical sections (execution of critical sections may include waiting for a resource). In short, no matter what scheduling policy is used, the presence of large numbers of readers can unduly delay writers, particularly if the readers are subject to preemption or blocking. Given that readers-writer locking is specifically designed to be used in scenarios with large numbers of readers, this potential for delay is a matter of serious concern.

An alternative to readers-writer synchronization is *read-copy update* (RCU).⁴ With RCU synchronization, writers create a new version of an object, while allowing readers to operate concurrently on the old version. Mechanisms resembling RCU have proven useful in a number of operating-system kernels, including the IBM VM/XA Hypervisor*,⁵ DYNIX*/PTX*,⁴ Tornado,⁶ K42,⁷ and, more recently, Linux.⁸ Because RCU allows readers and writers to run concurrently, writers may proceed even in the presence of an arbitrary number of higher-priority readers. Hence, RCU does not suffer from the aforementioned priority-inversion problems. Furthermore, RCU writers are largely immune to RCU reader delays.

However, existing implementations of RCU trade away real-time response for scalability. Because RCU writers create new versions of objects rather than updating them in place, RCU implementations must utilize a garbage collector to reclaim the old versions. For multiprocessor scalability reasons, existing implementations of RCU utilize a quiescent-state-based reclamation policy^{9,10} to keep track of old versions that are no longer in use. This approach has the advantage of minimizing the need for memory barriers and removing the need for expensive read-modify-write instructions. However, quiescent-state-based reclamation involves the suppression of preemption during read-side critical sections, which degrades the real-time response of the kernel. Subsequent sections describe in detail the mechanisms used by RCU implementations to reclaim memory and discuss their impact on real-time response. Several new variants of RCU that avoid suppressing preemption are then presented, and their performance is analyzed. In this paper we use RCU to denote the RCU synchronization

mechanism as well as its implementation—the context will make the meaning clear.

The RCU family of application programming interfaces (APIs) has grown over time in response to both technical and ease-of-programming issues. The earliest use of RCU-like mechanisms provided only reclamation primitives, similar to `synchronize_rcu()` or `call_rcu()` in Linux. However, ease-of-debugging concerns led the designers of DYNIX/PTX to add primitives similar to `rcu_read_lock()` and `rcu_read_unlock()` in Linux.

Early RCU implementations for Linux dispensed with `rcu_read_lock()` and `rcu_read_unlock()`, but the addition of preemptibility to the Linux kernel forced their introduction in order to prevent RCU read-side critical sections from being preempted, as required for correct operation of RCU. Objections to the use of explicit memory barriers within the Linux community led to the suggestion they be hidden in the list-traversing primitives of the kernel, the implementation of which greatly increased the popularity of RCU in the community. Similar concerns led in 2004 to the introduction of `rcu_dereference()` and `rcu_assign_pointer()`, which eliminated the need for coding explicit memory barriers when manipulating RCU-protected pointers.

By 2003, a number of network-based denial-of-service-attack scenarios exposed some weaknesses in the RCU implementation in Linux. These could not be fixed without adding overhead to the RCU read-side primitives, so the first alternate member of the RCU API family was added in the form of `rcu_read_lock_bh()`, `rcu_read_unlock_bh()`, and `call_rcu_bh()`. The RCU implementations that were vulnerable to such attacks were then migrated to this new API.

In 2004, work began in earnest on extending the real-time capability of Linux, with the introduction of Ingo Molnar's -rt patchset.¹¹ In the course of this work, it became apparent that Linux kernel developers had used RCU to obtain some side effects peculiar to the classic implementation, and that these side effects would not carry over to a real-time-friendly implementation of RCU. Therefore, we created a new RCU API family featuring the `synchronize_sched()` primitive, which waits for all code segments to complete over which preemption

has been disabled. Creating this API family enabled the implementation of the real-time variant of RCU described in this paper.

In 2006, the need arose for an implementation of RCU in which readers could block, resulting in the sleepable RCU (SRCU) member of the RCU API family.¹⁵ SRCU supports blocking readers, but cannot support priority boosting and cannot serve as a drop-in replacement for the other RCU API family members (i.e., generating a new family by simple name substitutions).

Our contributions in this paper include a description of how RCU may be used to solve a long-standing problem involving priority inheritance for readers-writer locking in real-time systems, a presentation of the informal semantics of RCU, a description of the priority-boosting technique of the RCU read-side critical sections, a presentation of SRCU, an implementation that permits general blocking in RCU read-side critical sections, and a presentation of read- and update-side micro-benchmark performance of the RCU API family members found in both real-time and non-real-time builds of the Linux kernel. We also summarize several conference papers that present real-time-safe RCU implementations that permit preemption of read-side critical sections, give an overview of system-level RCU performance results that are discussed in more detail elsewhere, and summarize methods for transforming algorithms using readers-writer locking to ones that use RCU instead.

The remainder of the paper is organized as follows. In the next section we present an overview of RCU, in which we cover the RCU API, informal semantics of RCU primitives, and the relation of RCU to priority inversion. Next, we show how RCU can be extended for use in real-time environments by describing how preemption may be permitted in RCU read-side critical sections, describing efficient implementations of real-time RCU read-side primitives, and discussing preventing low-priority processes from unduly delaying RCU reclamation. In the following section we describe variants of RCU that permit generalized blocking; such a variant is required to handle waiting for general events, such as timeouts and I/O completion. Then, we discuss RCU performance, which includes performance using micro-benchmarks and system-level workloads. We then discuss an approach to extend the

use of RCU by converting code that uses the readers-writer synchronization to code that uses RCU. The last section contains closing comments.

RCU: AN OVERVIEW

The multiple implementations of RCU correspond to a family of APIs for concurrent programming. Each implementation also includes a garbage collector suitable for use in a multiprocessor operating-system kernel. Each API distinguishes between readers and writers, and provides calls that support communication among readers, writers, and the garbage collector.

The basic strategy for writing an object is to create a copy of the object, modify the copy, and then replace the pointer to the old copy with a pointer to the new one. Hence, RCU uses an update-in-place strategy for modifying pointers to objects, but not for modifying the objects themselves, which normally remain immutable once published. This copy-based update strategy allows a writer to proceed immediately rather than wait until the completion of the tasks associated with concurrent readers.

Whereas this feature of RCU improves real-time response, it also introduces a new problem: reclaiming old copies of an object, that is, the copies used by readers' tasks when these copies are no longer needed because all the tasks have terminated.

The RCU API contains calls to coordinate pointer updates and dereference operations, to delineate the lifetime of pointers, and to trigger the reclaiming of old copies. The following subsections describe this RCU API in more detail.

Publishing and dereferencing pointers

Writers publish pointers using the `rcu_assign_pointer()` interface, which returns the new pointer after publishing it and after executing any memory barriers required for a given processor architecture. Such memory barriers are often required in order to ensure that initialization operations for the object pointed to are not reordered later than the assignment (publishing) of its pointer.¹³ Readers dereference pointers using the `rcu_dereference()` interface, which returns a pointer value that is safe to dereference and ensures correct memory ordering with respect to a corresponding `rcu_assign_pointer()` primitive. It is common coding practice to use `rcu_dereference()` to copy a shared pointer to a local variable, and then

to dereference this local variable, as shown in this example:

```
p = rcu_dereference(head.next);  
return p->data;
```

The `rcu_dereference()` interface is currently of no consequence on all processor architectures other than DEC Alpha¹¹ because the memory-ordering semantics of these other processors already guarantee the desired behavior.

Version collection primitives

The RCU API defines explicit calls for delineating the lifetime of pointers and for reclaiming memory. We refer to the process of reclaiming the memory associated with stale copies of an object as *version collection*. Although RCU uses explicit calls to allocate and free memory, many issues of version collection are similar to issues arising in automatic garbage collection. For example, to determine when it is *safe* to invoke `free()`, it is necessary to consider what objects a garbage detection algorithm would consider *live*.

RCU defines several distinct families of API for version collection. Each family of APIs is used to implement the same basic reclamation strategy, but in a slightly different kernel context. These different contexts will be discussed following an overview of the generic approach.

Following the creation of a new version, writers invoke the garbage collector either synchronously or asynchronously. The garbage collector, whose job is to reclaim the memory associated with old versions, must defer the freeing of memory until it is safe to do so. Safe collection points are determined synchronously using the `synchronize_rcu()` primitive, which blocks the caller until it is safe to free the old version. Freeing memory asynchronously in RCU is accomplished using the `call_rcu()` primitive. Instead of blocking, it registers a function to be invoked after it is safe to free the old version. This asynchronous variant is particularly useful in situations where it is illegal to block, such as in the context of an interrupt.

Determining a safe point for version collection is based on determining when the union of global and local root sets for the version is empty. The global root set consists of the global pointers to any of the

data structures in that version. The local root sets for a version consist of all thread-local memory that holds pointers to any of the data structures in that version. The global root set becomes empty when a version of the object is published using the `rcu_assign_pointer()` primitive, which updates the unique global pointer in place. Because garbage collection is not automatic in RCU, good programming practice involves placing a call to `synchronize_rcu()` or `call_rcu()` after the call to `rcu_assign_pointer()` in order to inform the collector that the global root set is now empty.

RCU read-side primitives: Efficiently tracking local root sets

The local root sets for a version are held in the local memory of all threads that have dereferenced the global pointer to that version. Keeping track of this information in an accurate way is difficult, and is the fundamental challenge when implementing version collection in a multiprocessor operating-system kernel.

One approach would be to explicitly keep track of all version pointers in thread-local memory, for example by forcing each thread to maintain a list of its currently active pointers, and ensuring that the collector check these lists prior to reclamation. This is the approach taken in the hazard pointers methodology.¹³ The problem with this approach in particular (and with non-blocking synchronization in general) is that it requires expensive operations in the read path.⁹ RCU takes a more conservative, but more scalable, approach; it keeps track of the lifetime of thread-local memory, and assumes that if a thread has local memory that was live at the time that the global root set of the version became empty, then that memory *might* contain a pointer to the version. Thus, RCU defines the local root set for a version to be the set of all thread-local memory whose lifetime overlaps the time during which the global root set for the version was non-empty. This is conservative and imprecise because the local root set of a thread really contains only those pointers that it actually dereferenced. However, this approach improves performance, especially in the read path, because it avoids the need to use expensive memory barriers or synchronization instructions in the implementation of the dereference primitive, a design decision that has fundamental performance implications for many read-mostly scenarios.

The start and end of lifetime for this thread-local memory are delimited explicitly by RCU readers invoking, respectively, the `rcu_read_lock()` and `rcu_read_unlock()` primitives. Despite their names, these primitives do not actually acquire or release any explicit locks; instead, they demark a region of code throughout which all accessible RCU-protected data structures are guaranteed to continue to exist. If, however, the `rcu_read_lock()` and `rcu_read_unlock()` primitives directly implemented communication with the collector, they would incur significant overhead. Instead, RCU uses a *quiescent-state-based* reclamation approach in which threads prevent the occurrence of *quiescent states* while their local memory is live. The collector indirectly observes quiescent states in all threads and uses this information to infer when certain versions can no longer be live.

For example, consider a context switch (preemption by the scheduler or voluntary blocking) as a quiescent state. In this case, `rcu_read_lock()` disables preemption, `rcu_read_unlock()` enables preemption, and it is illegal for a thread to voluntarily block within a critical section delimited by `rcu_read_lock()` and `rcu_read_unlock()`. If the collector observes all processors pass through a context switch, it can safely collect any versions whose global root set became empty prior to the first of those context switches because no new reader can have obtained a reference to those versions. In RCU terminology, a period during which all processors have passed through a quiescent state is called a *grace period*. Reclamation of an object must be deferred by at least a grace period to ensure that concurrent readers cannot access freed or potentially reused memory, which would result in undefined behavior. This case also provides a trivial conceptual implementation of `synchronize_rcu()`:

```
void synchronize_rcu(void)
{
    int cpu;

    for_each_cpu(cpu)
        run_on(cpu);
}
```

By running on each CPU in turn, this implementation guarantees that each CPU has passed through a voluntary context switch, and thus through a quiescent state. Therefore, once this function has

completed, all prior RCU read-side critical sections will be guaranteed to have completed; in other words, a grace period will have elapsed.

This choice of context switch as a quiescent state is especially interesting because it allows the collector to ignore all threads that are not executing. The fact that they are not executing means that they are in the middle of a context switch event, and hence cannot have live references. Therefore, the union of all local root sets for non-active threads is guaranteed to be empty. The choice of context switch also allows the signaling that occurs conceptually between threads and collectors to be piggybacked on the pre-existing context-switch events. However, these efficiency advances come at the price of delaying collection for longer than is strictly necessary, and more significantly for real-time systems, introducing non-preemptible sections of code in the RCU read-path. Hence, the RCU generic version collection strategy is safe, but it is imprecise and conservative in a number of dimensions, and degrades real-time response.

A family of APIs for version collection

As mentioned earlier, RCU defines a family of APIs for use in different kernel contexts, with different trade-offs between overhead and reclamation delay. The specific choice of kernel event utilized as a quiescent state varies among these APIs. Example quiescent states used by existing variants of RCU include: voluntary context switch, completion of a softirq (soft interrupt request) handler invocation, and preemption by scheduler. For each choice of quiescent state there is an associated set of API primitives that mirror the generic RCU API primitives discussed above. The following paragraphs give a very brief summary of a few of the RCU APIs in use in the Linux kernel.

When the quiescent state is *voluntary context switch*, the start of thread-local memory lifetime is delimited by the `rcu_read_lock()` primitive, which essentially does nothing. The end of thread-local memory lifetime is signaled by the `rcu_read_unlock()` primitive, which essentially does nothing. The programmer follows the convention of not yielding the processor while executing between a pair of `rcu_read_lock()` and `rcu_read_unlock()` primitives. Either `synchronize_rcu()` or `call_rcu()` may be used to delay the actual freeing of memory until readers can no longer hold references to it. In this

case, a grace period is defined as a time period during which all processors have undergone a voluntary context switch. Note that in a non-preemptive kernel the RCU read-side critical sections described here can delay real-time response due to their refusal to yield the processor.

When the quiescent state is *completion of a softirq handler* (or “bottom half”), the start of thread-local memory lifetime is delimited by the `rcu_read_lock_bh()` primitive. This primitive disables preemption by softirqs, which can be thought of as a software-interrupt environment, similar to that found in a number of operating systems. The end of thread-local memory lifetime is delimited by `rcu_read_unlock_bh()`, which enables preemption by softirqs. The `call_rcu_bh()` primitive, which counts softirq completions, may be used to delay freeing of memory until readers can no longer hold references to it. The definition of the corresponding grace period is any time period during which all processors have experienced a softirq completion. In this case, real-time response is affected by the disabling of preemption by softirqs during RCU read-side critical sections.

When the quiescent state is *preemption by the scheduler* (or, equivalently, context switch, whether voluntary or involuntary), the start of thread-local memory lifetime is delimited by the `preempt_disable()` primitive, which disables scheduler preemption. The end of thread-local memory lifetime is delimited by the `preempt_enable()` primitive, which enables scheduler preemption. Memory is freed using the `synchronize_sched()` primitive, which counts scheduler preemptions and may be used to delay the actual freeing of memory until readers can no longer hold references to it. The corresponding grace-period definition is any time period during which all processors have experienced a context-switch event. This definition is also used by the `rcu_read_lock()` and `rcu_read_unlock()` primitives in preemptible-kernel builds of the Linux kernel. In this case, real-time response is affected by the disabling of preemption during RCU read-side critical sections.

Informal semantics of RCU primitives

The RCU primitives partition code into distinct sets, as shown in *Figure 1*. Read-side primitives, such as `rcu_read_lock()` and `rcu_read_unlock()` (shown at the top of the figure), partition code into sets R_0

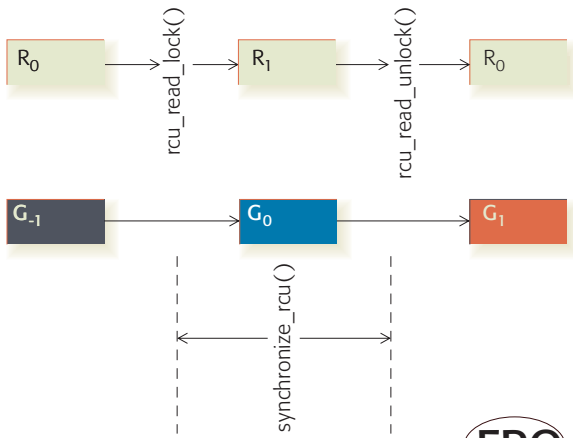


Figure 1
 RCU primitives partition code into sets of statements

outside of the critical section and R_1 within it. Update-side primitives, such as `synchronize_rcu()` (shown at the bottom of the figure), partition code into sets G_{-1} preceding the grace period, G_0 concurrent with the grace period, and G_1 following the grace period. This partitioning applies globally to all code in all threads. As shown in the diagram, the set G_0 extends from the call to `synchronize_rcu()` through the return. In the case of the `call_rcu()` primitive, the set G_0 extends from the invocation of `call_rcu()` to the invocation of the callback function passed to the `call_rcu()` primitive.

Given these sets defined with respect to a given RCU grace period, the fundamental definition of RCU states that if any statement in a given RCU read-side critical section's set R_1 precedes any statement in a given grace period's G_{-1} , then all statements in that critical section's R_1 must precede any statement in that grace period's G_1 .

Of course, both compiler optimizations and weakly ordered processors¹³ can reorder code. It is the responsibility of the RCU primitives either to prevent such reordering or to define the sets so as to make any reordering harmless, which means that users of RCU almost never need to specify compiler or memory barriers explicitly. For example, the set R_1 may often be extended to subsume portions of set R_0 while still maintaining correctness. Similarly, the set G_0 may be extended to cover a greater time duration; however, it clearly cannot be allowed to subsume

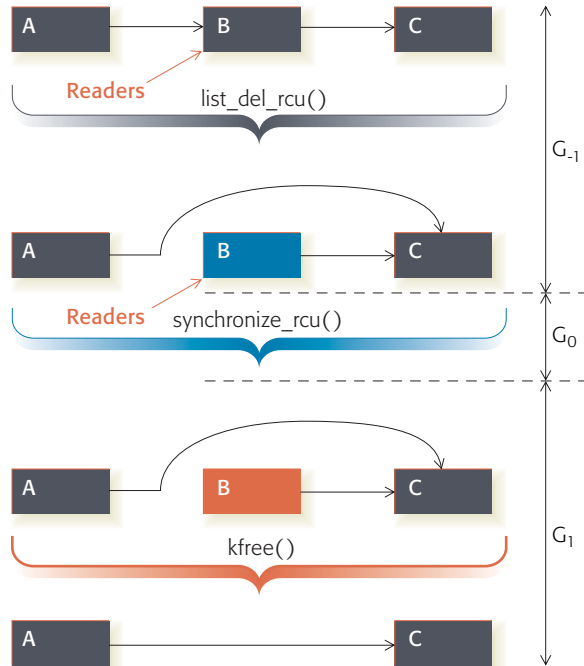


Figure 2
 RCU semantics applied to deletion from a linked list

the portion of set G_1 that immediately follows the `synchronize_rcu()` primitive.

Figure 2 illustrates the steps associated with deleting an element B from a linked list in an environment with concurrent readers and RCU. Initially, the list contains elements A, B, and C, as shown at the top. Element B is removed from the list by causing A's "next" pointer to reference C rather than B. Given that A's "next" pointer is properly aligned so that loads and stores of that pointer are atomic, concurrent readers will fetch a pointer either to B or to C, and not some bitwise combination of the two pointers.

The updating thread then executes `synchronize_rcu()`, which blocks until all current readers finish, ensuring that there are no remaining references to element B. To infer this, note that any RCU read-side critical section that obtained a pointer to element B has at least one statement that precedes the removal (via the `list_del_rcu()` primitive), which is in set G_{-1} . Therefore, from the definition of RCU, all statements in that read-side critical section must precede any statement in G_1 , so that the entire critical section must have completed before invocation of the `free()` statement that reclaims element

B. In short, any RCU read-side critical section that obtains a reference to element B is guaranteed that element B will still exist throughout that critical section.

RCU does in fact implement a variant of mutual exclusion, albeit a temporal variant. Unlike readers-writer locking, in which the mere presence of a reader prevents an updater (writer) from running, in RCU only sufficiently old readers are permitted to block updaters via the grace-period computation, and even then, these readers are only permitted to block the reclamation phase of the update.

RCU and priority inversion

Given this background on RCU, it is time to revisit priority inheritance, but this time using RCU in conjunction with an exclusive updater lock in place of the readers-writer lock. In this revised example, readers use RCU read-side primitives while writers acquire the exclusive lock. As before, there are an arbitrary number of low-priority readers and a single high-priority real-time writer. Because the readers no longer acquire the lock, the writer can acquire it immediately, regardless of the number of readers and regardless of whether or when the readers might have been preempted. Use of RCU can thus greatly reduce the worst-case real-time latencies of the updater, comparing favorably both to locking and to retry-based schemes such as the Linux kernel sequence locks, both of which can impose large latencies on readers. However, the RCU read-side primitives may be used both by real-time and by low-priority readers. If any of the low-priority readers have been indefinitely delayed while executing within their RCU read-side critical sections, for example, due to being preempted by real-time processes, the corresponding grace period will extend indefinitely. Any invocation of `synchronize_rcu()` will therefore block indefinitely, ruining the real-time response of the writing thread. However, the solution is simply to use the asynchronous form `call_rcu()`. This primitive posts a *callback* function that is invoked after the end of a grace period, permitting the writing thread to continue processing without having to wait for this possibly indefinite grace period to end. Of course, the callback cannot be invoked until the grace period ends, and therefore the corresponding memory cannot be reclaimed until the grace period ends. Therefore, an indefinite grace period might eventually block the attempt of a writing thread to

allocate memory once memory has been exhausted (assuming finite memory).

This discussion assumes first, that it is possible to substitute RCU for standard uses of readers-writer locking, and second, that it is possible to prevent preempted RCU read-side critical sections from blocking by extending grace periods indefinitely. Such indefinite blocking would prevent any elements removed from RCU-protected lists from ever being reclaimed, eventually exhausting the available memory. We examine both of these assumptions in greater depth later in this paper.

ADAPTING RCU FOR REAL-TIME KERNELS

Real-time systems require low response times for high-priority threads, which means that such threads must be able to preempt even those low-priority threads that are executing critical-section code. Unfortunately, preemption can greatly extend the duration of critical-section execution, and therefore it is no longer appropriate to guard them with a spinlock (i.e., the thread waits in a loop until the lock is available) because, as critical-section duration increases, spinlock acquisition consumes an increasing number of processor cycles. Instead, real-time systems use blocking locks, which often must be acquired within RCU read-side critical sections (pure spinlocks are still required in some special cases, for which the Linux kernel provides “raw” spinlocks). Thus, RCU read-side critical sections must be permitted to block while acquiring locks. The only problem remaining, then, is to produce an RCU implementation that allows RCU readers to block on locks (or to be preempted) while still allowing timely reclamation of memory released by RCU updaters.

Permitting preemption of RCU read-side critical sections

If RCU read-side critical sections are to be preempted, then it is necessary to track threads in such critical sections so that it is possible to determine when a grace period can end. In principle, this can be as simple as a single global counter that is atomically incremented by `rcu_read_lock()` and atomically decremented by `rcu_read_unlock()`, so that any statement outside the RCU read-side critical section can be a quiescent state. Once this single global counter reaches zero, all prior RCU read-side critical sections are guaranteed to have completed. In practice, as the number of processors increases, there

is a point beyond which at least one processor can be expected to be executing in an RCU read-side critical section at any given time, and the counter might never reach zero and grace periods will never terminate.

This situation can be addressed as described by Gamsa et al.,⁶ by providing a two-element array of counters along with an index indicating which of the two should be atomically incremented by the `rcu_read_lock()` primitive. The `rcu_read_unlock()` primitive then atomically decrements the same counter that was incremented by the corresponding `rcu_read_lock()` invocation. At the beginning of a grace period, the sense of the index is inverted, at which point the old counter (that is, the counter no longer referenced by the index value) will no longer be incremented, but will continue to be decremented by `rcu_read_unlock()`. The value of this old counter must therefore eventually reach zero, marking the end of the grace period.

Unfortunately, use of a single pair of counters leads to high overhead due to memory contention and the resulting cache misses. Although this overhead does not affect the underlying real-time properties of the RCU read-side primitives, as their latency remains bounded (deterministic), low overhead is important because a number of RCU-based algorithms were designed with the zero-overhead non-real-time RCU read-side primitives in mind.

Reducing overhead of RCU read-side primitives

We describe now a mechanism for avoiding the cache-miss overhead. The mechanism is based on a $2 \times N$ array of counters, where N is the number of threads. Thus, there are two counters for each thread. In addition, a global index points to one of the two columns of the array, as illustrated in *Figure 3*. The `rcu_read_lock()` primitive increments the current counter from the pair corresponding to the processor on which the `rcu_read_lock()` is running, and also retains the value of the global index used, 0 or 1. The `rcu_read_unlock()` primitive then uses this value to select the counter to decrement from the pair corresponding to the processor on which the `rcu_read_unlock()` is running. This mechanism avoids cache misses, atomic instructions, and memory barriers, but requires that the grace-period detection mechanism be a state machine that determines when each processor has observed a change to the global index, that forces each

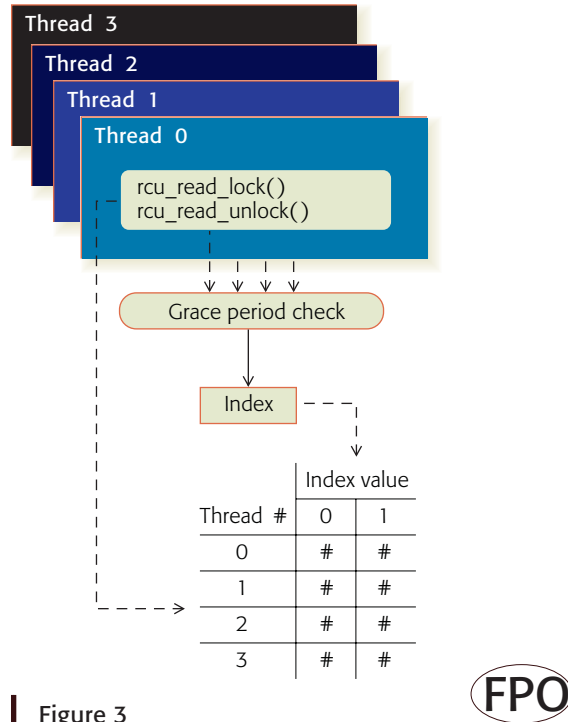


Figure 3
 Real-time RCU avoiding cache-miss overhead

processor to periodically execute memory barriers, and that sums the non-current counters from each CPU to determine when the global index may safely be incremented. See Reference 16 for a full description of this approach.

As of this writing (July 2007), the `-rt` patchset¹¹ contains an older implementation that requires readers to execute expensive atomic instructions and memory barriers. We expect this older implementation to be replaced once the new version has proven itself trustworthy on all relevant CPU architectures. Of course, we also expect the number of relevant CPU architectures to increase as porting of the `-rt` patchset continues. However, because the new implementation has passed the “RCU-torture” test suite on Intel 32-bit Xeon** (i386), AMD 64-bit Opteron** (x86_64), and IBM POWER* (PowerPC*) processor architectures, and because the new implementation is, just like the old one, architecture independent, we do not anticipate major problems from additional CPU architectures.

Preventing indefinite delay within RCU read-side critical sections

As noted earlier, indefinite delay within a thread executing an RCU read-side critical section will

indefinitely extend grace periods, which can exhaust the available memory, which will in turn delay indefinitely any updaters attempting to allocate memory. It is therefore necessary to prevent low-priority threads from being indefinitely preempted when executing within RCU read-side critical sections.

A straightforward solution would be to boost the priority of RCU readers when the available memory has been exhausted. However, this must be done carefully in order to preserve the performance of RCU read-side primitives. In particular, maintaining a list of all tasks in RCU read-side critical sections would not be advisable, even if the list were very cleverly implemented. Instead, we note that RCU read-side critical sections tend to be short and rarely preempted, especially if the system has been appropriately provisioned and there is significant idle time, as is typical for systems that must maintain bounded response times (otherwise, response times can be degraded due to queuing delays in the scheduler). Furthermore, if a given RCU read-side critical section has not been preempted, boosting its priority will not make the processor that it is running on go any faster. Instead, this suggests having the scheduler check for preemption or blocking within RCU read-side critical sections. Because such preemption or blocking is relatively rare, heavyweight linked-list operations can be tolerated in this case.

Because most preempted or blocked threads in RCU read-side critical sections can be expected to become runnable in short order, and because priority boosting is a relatively expensive operation, it makes sense to defer priority boosting for a short time. Such deferral should normally allow the threads to become runnable and complete their critical sections without the need for expensive priority boosting. Although preemption of RCU critical sections is rare, that does not mean that we may simply choose never to preempt them, as such a choice would degrade worst-case scheduling latencies.

Figures 4A and *4B* show a pair of snapshots of a data structure designed for this purpose. Each processor has a pair of four-element arrays, one of which is labeled “Boosting Old Aging New” in the upper half of each snapshot and the other of which is labeled “Boosted” in the lower half of each

snapshot. Each entry of these per-CPU array pairs is a linked list of tasks that might soon require priority boosting. In Figure 4A, element 3 of the upper array accepts tasks that block or are preempted while executing in an RCU read-side critical section, as indicated by the “Index=3.” Figure 4B shows a later point in time when the index has been incremented modulo the number of elements in the array, so that element 0 of the upper array accepts tasks that block or are preempted while executing in an RCU read-side critical section—and the other elements of the array also change roles as indicated in the two figures. If a task remains in its RCU read-side critical section for three such time periods, it advances through the “Aging,” “Old,” and finally “Boosting” roles, at which point it is eligible for priority boosting. Tasks eligible for boosting are periodically moved from the “Boosting” element of the upper array to the “Local List.” Tasks on the “Local List” that are successfully boosted are moved to the “Boosted” element of the lower list, while tasks for which boosting has been unsuccessful are placed back onto the “Boosting” element of the upper array, and, if the boosting is successful, it is moved into the lower array of the set. When a task exits its RCU read-side critical section, it removes itself from whatever list it is on at the time, as indicated by the arrows pointing to the “Tasks Removed at End of RCU Critical Section” labels. Each processor maintains four locking domains, one for each element of its pair of per-CPU arrays. Each task, once enqueued on a given array element for a given CPU, remains enqueued either on that array element or on the corresponding element of the other array of that CPU. This therefore means that a task, once enqueued, is always protected by the same lock, which minimizes lock contention while eliminating races that would otherwise occur if a given task were to be moved among different locking domains.

GENERAL BLOCKING IN RCU READ-SIDE CRITICAL SECTIONS

The preceding sections discussed a counter-based RCU implementation that permits preemption and blocking during lock acquisition. Although this is sufficient in many situations, it is sometimes necessary to block for other reasons, such as completion of I/O, memory allocation, or for fixed time durations. Unfortunately, software-engineering considerations prevent direct use of the counter-based real-time RCU implementations in this manner.

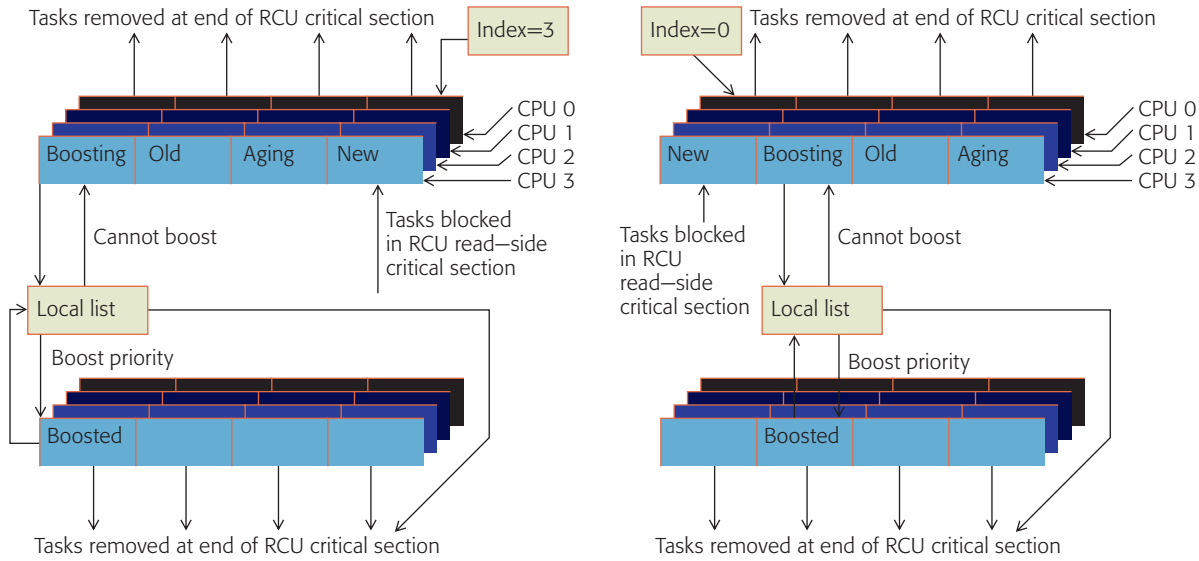


Figure 4
 Tracking RCU priority-inheritance candidates



The problem is that the RCU API provides a global set of grace periods, so that a single thread blocking for an excessive time in an RCU read-side critical section will delay reclamation for all updaters, even if completely unrelated to the blocked reader. The solution to this problem is to provide separate sets of grace periods by introducing the concept of an RCU control block. All related invocations of a blocking variant of RCU would then share a single RCU control block, which must be passed to each such invocation. Updaters can then be delayed only by readers using the same RCU control block, containing the effects of excessive delays by RCU readers.

For example, the sleepable RCU (SRCU)¹⁵ implementation uses the following two primitives to delimit the start and end of thread-local memory lifetime respectively:

```
idx = srcu_read_lock(&scb)
/* SRCU read-side critical section. */
srcu_read_unlock(&scb, idx)
```

The `scb` variable is the SRCU control block; different users would each specify their own control block. The `synchronize_srcu(&scb)` primitive blocks until the end of a subsequent per-`scb`-variable grace period, permitting the following idiom to delay the actual freeing of memory until the counter values indicate that all threads have subsequently passed through a safe collection point:

```
list_del_rcu(p);
synchronize_srcu(&scb);
free(p);
```

Because SRCU is optimized for minimal read-side overhead, it places responsibility for memory ordering onto the `synchronize_srcu()` primitive, resulting in that primitive having significant latency. Nevertheless, with slight modifications to the RCU API, it is now possible to permit general blocking within RCU read-side critical sections. This modification to the RCU API is necessary in order to address the need for multiple RCU “domains,” preventing an RCU reader in one domain from blocking reclamation in some other domain. Software environments permitting dynamically allocated thread-local storage may be able to provide multiple RCU domains within the confines of the classic RCU API.

RCU PERFORMANCE

In this section we first describe results of evaluating the overhead of individual RCU primitives using micro-benchmarks. Then we briefly discuss RCU performance results obtained with system-level workloads.

Micro-benchmarks

For RCU to be generally useful, its read-side performance must not only exceed that of uncontended locking, but it must also be competitive with

Table 1 Performance of RCU variants on 4-processor, 1.8-GHz Opteron 844 system

RCU variant	Build	Reads (ns)			Updates			Quiescent state
		1 CPU	2 CPUs	3 CPUs	1 CPU	2 CPUs	3 CPUs	
Rcu	S	1.1	1.1	1.1	12.0 ms	16.0 ms	16.0 ms	Voluntary context switch
rcu-bh	S	11.2	11.2	11.2	N/a	N/a	N/a	Completion of softirq handler
rcu-preempt	r	17.3	17.3	17.3	12.0 ms	12.0 ms	16.0 ms	Context switch
Sched-rt	R	27.4	27.4	27.4	28.2 us	28.7 us	141.2 us	Context switch
rcu-rt	R	48.0	48.0	48.1	2.0 ms	2.0 ms	2.3 ms	Any unprotected
rcu-rt (new)	R	33.0	33.0	33.0	79.3 ms	79.4 ms	79.4 ms	Any unprotected
rcu-rt (nested)	R	8.4	8.4	8.4	79.3 ms	79.4 ms	79.4 ms	Any unprotected
srcu	A	10.0	10.0	10.1	47.9 ms	68.0 ms	83.8 ms	Any unprotected

that of best-case cache-local compare-and-swap (CAS). RCU does achieve this level of performance, as shown in **Table 1** for a number of implementations in different builds of the 2.6.21 version of the Linux kernel, running on a 4-processor, 1.8-GHz AMD Opteron 844 system.

The first column of this table identifies the RCU variant. The second column identifies the type of Linux kernel build: *S* for server, *r* for the less-aggressive CONFIG_PREEMPT real-time, *R* for the more-aggressive -rt real-time patch set, and *A* for all builds. The three columns headed by *Reads* list the overhead in nanoseconds of read-side primitives on single-processor, two-processor, and four-processor runs, and the three columns headed by *Updates* list the latency of update-side primitives, also on single-processor, two-processor, and four-processor runs. The repeatability of these measurements is quite good, with the maximum standard deviation being about 6 percent of the mean, and most being orders of magnitude below the mean. The final column lists the quiescent state used by the corresponding RCU implementation. The RCU primitives map to the rows of the table as follows:

1. `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()` map to “rcu” in a server build, to “rcu-preempt” in a CONFIG_PREEMPT build, and to “rcu-rt” in a -rt aggressive real-time build. In the future, we expect that these primitives will map to “rcu-rt (new)” in a -rt build. The “rcu-rt (new)” implementation differs from the “rcu-rt” implementation in that it avoids use of atomic instructions and memory barriers in read-side RCU primitives.

2. `rcu_read_lock_bh()` and `rcu_read_unlock_bh()` map to “rcu-bh” in both the server and the CONFIG_PREEMPT builds, and map to `rcu_read_lock()` and `rcu_read_unlock()` in a -rt aggressive real-time build.
3. `preempt_disable()`, `preempt_enable()`, and `synchronize_sched()` map to “rcu” in a server build, “rcu-preempt” in a CONFIG_PREEMPT build, and to “sched-rt” in a -rt aggressive real-time build.
4. `srcu_read_lock()`, `srcu_read_unlock()`, and `synchronize_srcu()` map to “srcu” in all builds.

In a perfect world, we would present the worst-case overheads of the read-side primitives, as worst-case overheads are relevant to real-time workloads. Unfortunately, the extremely low overheads of these primitives render such an approach infeasible. In addition, even given an extremely fine-grained and low-access-overhead clock, it would be necessary to reject measurements within which interrupts occurred, which in this case was infeasible. We therefore measured the overhead of long repeating sequences of these operations, and calculated the average overhead. However, the fact that the read-side primitives are implemented using a short, fixed sequence of instructions means that execution time has a definite upper bound.

The read-side performance of all RCU implementations is independent of the number of processors, all differences being within the bounds of statistical error, as is to be expected, given that these primitives have deterministic overhead and do not access any shared variables. All are well below the 65.6-ns contention-free zero-cache-miss combined

overhead of lock acquisition and release, and, with the exception of the “rcu-rt” variant on four processors, all are well below the 37.9-ns best-case zero-cache-miss CAS overhead. For purposes of comparison, the overhead of a single cache miss on this system is 139.5 ns, and the overhead of the read-then-modify pair of cache misses resulting from a load followed by a CAS of a given variable is 306.0 ns. These overheads will of course increase when additional processors are concurrently modifying the same variable, due to the resulting increase in memory contention.

The lowest-overhead read-side primitive is that of the “rcu” implementation, which is expected given that the `rcu_read_lock()` and `rcu_read_unlock()` primitives for this implementation generate zero machine instructions, so that the 1.1 ns is a measure of loop overhead. Free is a very good price, but this implementation operates only in non-preemptible kernels, which are completely unfit for aggressive real-time use. Only the “rcu-rt” variants and “srcu” permit preemption of RCU read-side critical sections, and of these, only the “rcu-rt (new)” implementation permits the read-side primitives to be invoked from non-maskable interrupt (NMI) handlers. This variant is more expensive, and further reducing its overhead is important future work. That said, the overhead of “rcu-rt (new)” read-side primitives when nested in an enclosing RCU read-side critical section are second only to “rcu” in speed, at 8.4-ns overhead per critical section. Furthermore, even the full 33.0-ns latency for the outermost RCU read-side critical section is better than that of uncontended locking and best-case CAS. Finally, the overheads of all of the read-side primitives exhibit perfect scaling up to four CPUs. These results give hope that significant system-level performance benefits might result from use of RCU, a topic that is taken up in the following section.

The update latencies for RCU are quite large, typically many milliseconds, due to the grace-period-detection state machine that is driven by the scheduling-clock interrupt, which in turn allows concurrent readers to take few if any precautions against concurrent updates. It is important to note that updaters block rather than busy-wait. In fact, for all the four-updater tests other than “sched-rt,” CPU utilization is negligible, as is contention for locks used by the state machine on systems with up

to many tens of CPUs. Finally, latency-sensitive updaters can make use of asynchronous interfaces such as `call_rcu()`, which have sub-microsecond overheads. That said, these large update latencies are one motivation for reserving RCU for read-mostly situations, or in situations where the need for read-side determinism outweighs the need for low update-side latency. However, several of the update-side implementations have excellent scalability (assuming fixed read-side critical-section duration), so that RCU-based software retains its performance advantages as the number of processors increases. The “rcu” and “rcu-preempt” variants show some increase in latency with number of CPUs due to the batching algorithm used: if a grace period is already in progress, a subsequent request must wait for both this grace period and another to complete. The “sched-rt” algorithm is known to have poor scalability because it visits each CPU in turn, and will be replaced once real-time systems gain sufficient CPUs for this to be an issue. Finally, the “srcu” algorithm contains multiple serial uses of the “rcu-preempt” update-side algorithm, and in addition uses a single per-SRCU-domain lock.

More detailed performance measurements of RCU may be found in Hart et al.,⁹ showing that RCU enjoys substantial performance advantages over other deferred-reclamation mechanisms over a wide variety of operating parameters.

System-level workloads

Ideally, the system-level performance of RCU would be evaluated by first replacing RCU with some other synchronization primitive in a recent version of the Linux kernel, and then comparing system-level performance using the two versions of the kernel. Unfortunately, most uses of RCU in the Linux kernel have come to rely heavily on a number of properties unique to RCU, in particular, its read-side immunity to deadlock, so that all attempts thus far to map RCU to other synchronization primitives have failed.⁹

Therefore, the best available way to evaluate the system-level performance of RCU is to review comparisons made when RCU-based algorithms were proposed for inclusion into Linux. A survey of such comparisons¹⁰ has shown performance improvements ranging from several percentage points to orders of magnitude on benchmarks up to and including transactional database workloads.⁸

TRANSFORMING READERS-WRITER LOCKING TO RCU

The performance and determinism benefits of RCU are of immediate value only if it can reasonably be incorporated into existing operating-system kernels. Although the fact that there are more than 1000 invocations of the RCU API in the 2.6.21 version of the Linux kernel⁸ gives empirical evidence that RCU can in fact replace the readers-writer lock as the synchronization mechanism in an operating-system kernel, a conceptual framework is needed to help determine when readers-writers lock may most readily be replaced by RCU, as well as how to handle more difficult conversions from readers-writer lock to RCU. This section provides such a framework, giving a method of classifying readers-writer locking uses to determine which may be trivially converted, and also providing methods for converting more difficult cases.

Two major issues arise when converting the readers-writer lock to RCU: consistency and freshness. Here, *consistency* means that readers “see” a single version throughout any given read-side critical section, and *freshness* means that readers always “see” that version of each referenced data structure installed by its most recent writer. Readers-writer locking guarantees that readers see data that is both consistent and fresh by excluding (and thus delaying) writers. However, taking this approach would destroy an extremely important property of RCU for real-time systems, namely, deterministic readers (bounded overhead).

A key insight drawn from past uses of RCU is that a surprising number of algorithms can dispense with consistency, freshness, or even both. The canonical example of this is Internet routing—because the routing algorithms take significant time to propagate updates across Internet, and because there is little or no synchronization between changes in different regions of the Internet, routing updates may be both stale and inconsistent upon arrival at a given network node. Nor is this an isolated case: consistency and freshness are unavailable in many situations in which an external physical state at a remote location must be tracked by internal data structures, particularly in those situations similar to routing, where updates either add an element to or remove an element from a larger data structure, but never modify an element in place. Such algorithms can often be converted from readers-writer locking

to RCU by mechanical substitution of RCU API members for those of readers-writer locking.¹⁰

Providing data consistency with RCU

If updates modify preexisting elements in a data structure, and if readers must observe consistent values from a given update, then more care is required. The standard approach is to hide complex updates behind an atomic pointer assignment, so that an update can be carried out through use of the following steps:

1. Allocate a new version of the data structure that is to be updated.
2. Copy the contents of the old data structure to the new version.
3. Update the new version. Note that this new version is not yet accessible to readers.
4. Replace any pointers to the old version with pointers to the new version. (Note that the `rcu_assign_pointer()` primitive that is normally used to perform this step includes any memory barrier instructions that might be required when running on weakly ordered systems.¹³)

This approach works well, and is used repeatedly within the Linux kernel. However, in some cases, copying introduces excessive overhead. In cases where the data elements are large, but the number of updated fields is small, the updated fields may be moved to a separate, small data element linked from the main data structure. This small data structure may then be updated by copying it, as described above. This approach also works well in cases where there are a large number of pointers to the enclosing data structure.

Another approach is simply to introduce for each data element a lock that both readers and updaters must acquire. This approach has the disadvantage of introducing nondeterminism to readers, but for data structures containing a very large number of data elements, lock contention may be acceptably low for many workloads. In this case, use of RCU reduces the domain of contention from the full data structure to the individual data elements comprising that structure.

Providing data consistency and freshness with RCU

Some algorithms require both data consistency and freshness, one example being the System V inter-

process communication (IPC) implementation, which was the first use of RCU in the Linux kernel.⁸ Such algorithms start with the per-data-element lock described in the previous section, but with the addition of a flag indicating stale data. RCU read-side critical sections can then be structured as follows:

1. Enter an RCU read-side critical section (for example, using `rcu_read_lock()`).
2. Traverse the data structure, locating the desired element. If the desired element cannot be located, exit the RCU read-side critical section and indicate failure.
3. Acquire the lock of that element (or other mutual-exclusion mechanism).
4. Check the stale-data flag. If set, release the lock, exit the RCU read-side critical section, and indicate failure.
5. Access the element.
6. Exit the RCU read-side critical section.
7. Indicate success.

Updaters, upon removing a given data element, set that element's flag while holding that element's lock. These methods for providing consistency and freshness with use of RCU have been used in a number of operating-system kernels, though more work is needed to enable conversion of additional readers-writer locking cases to be converted gracefully to RCU.

CONCLUSION

In this paper, we showed how RCU can be adapted for use in a real-time operating-system kernel for solving long-standing problems with readers-writer synchronization. RCU read-side primitives offer deterministic overhead as well as better performance than either uncontended locking or best-case CAS. There is a growing body of experience with conversion from readers-writer lock to RCU, and we have identified a number of methods to ease such conversions.

Of course, RCU is not a panacea—it is instead another tool in the synchronization toolbox with a specific area of applicability: low-overhead, deterministic read-side primitives for read-mostly data structures. In other situations, use of other synchronization mechanisms may be preferable. However, RCU-like mechanisms have proven useful in a number of operating-system kernels, including the

IBM VM/XA Hypervisor, DYNIX/PTX, Tornado/K42, and, more recently, Linux. Future work includes further reducing the overhead of the read-side primitives of real-time RCU, improving tooling used to validate RCU usage, and developing additional methods of converting readers-writer locking to RCU, perhaps including a group of “universal” methods that permit all uses of readers-writer locking to be converted to RCU. We expect that these enhancements will establish RCU as the mechanism of choice for read-mostly situations, particularly in general-purpose operating systems providing real-time response.

ACKNOWLEDGMENTS

We owe a debt of gratitude to Ingo Molnar, Thomas Gleixner, Sven Dietrich, K. R. Foley, Gene Heskett, Bill Huey, Esben Nielsen, Nick Piggin, Lee Revell, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karten Wiese, whose remarkable efforts have made real-time Linux what it is today, and to Andrew Morton and Linus Torvalds for their efforts to bring real-time functionality into mainstream Linux. We are grateful to Dipankar Sarma, Oleg Nesterov, and Jens Axboe for many fruitful discussions and for many contributions to RCU in Linux. We thank Premalatha Nair, Daniel Frye, Ray Harney, and Darren Hart for their unstinting support of this effort.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Linus Torvalds, Intel Corp., or Advanced Micro Devices, Inc., in the United States, other countries, or both.

CITED REFERENCES

1. B. W. Lampson and D. D. Redell, “Experience with Processes and Monitors in Mesa,” *Communications of the ACM* **23**, No. 2, 105–117 (1980).
2. U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall, Upper Saddle River, NJ, 1996.
3. J. M. Mellor-Crummey and M. L. Scott, “Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors,” *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, ACM, New York (1991), pp. 106–113.
4. P. E. McKenney and J. D. Slingwine, “Read-Copy Update: Using Execution History to Solve Concurrency Problems,” *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, IASTED, Anaheim, CA (1998), pp. 508–518.
5. J. P. Hennessy, D. L. Osisek, and J. W. Seigh II, *Passive Serialization in a Multitasking Environment*, U.S. Patent No. 4,809,168 (1989).

6. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI)*, New Orleans, USENIX, Berkeley, CA (1999), pp. 87–100.
7. J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling Autonomic Behavior in Systems Software with Hot Swapping," *IBM Systems Journal* **42**, No. 1, 60–76 (2003).
8. A. Arcangeli, M. Cao, P. E. McKenney, and D. Sarma, "Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel," *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, San Antonio, TX, USENIX, Berkeley, CA (2003), pp. 297–310.
9. T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole, "Performance of Memory Reclamation for Lockless Synchronization," *Journal of Parallel and Distributed Computing* **67**, No. 12, 1270–1285, DOI: 10.1016/j.jpdc.2007.04.010 (2007).
10. P. E. McKenney, *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating Systems Kernels*, Ph.D. Dissertation, OGI School of Science and Engineering at Oregon Health & Science University, Beaverton, OR (2004).
11. I. Molnar, *Realtime-Preempt Patch Set*, Red Hat, Inc., Raleigh, NC (2007), <http://people.redhat.com/mingo/realtime-preempt/>.
12. P. E. McKenney, "Sleepable RCU," *Linux Weekly News* (Oct. 9, 2006), <http://lwn.net/Articles/202847/>.
13. K. Gharachorloo, *Memory Consistency Models for Shared-Memory Multiprocessors*, Ph.D. Dissertation, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA (1995).
14. P. E. McKenney, "Memory Ordering in Modern Microprocessors, Part I," *Linux Journal* **X**, No. 136, 52–57 (2005), <http://www.linuxjournal.com/article/8211>.
15. M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Transactions on Parallel and Distributed Systems* **15**, No. 6, 491–504 (2004).
16. P. E. McKenney, D. Sarma, I. Molnar, and S. Bhattacharya, "Extending RCU for Realtime and Embedded Workloads," *Proceedings of the Linux Symposium* **2**, Ottawa, Canada; Linux Symposium Inc., Ottawa, Canada (2006), pp. 123–138.

Accepted for publication September 7, 2007.

Dinakar Guniguntala

IBM India Systems and Technology Laboratory, EGL D Block, Indiranagar-Koramangala Intermediate Ring Road, Bangalore KA 560071 India (dino@in.ibm.com). Mr. Guniguntala is a software engineer with the Linux Technology Center in Bangalore, India, working on real-time Linux. He has been with IBM India for over nine years, working primarily on operating systems. His areas of interest include real-time Linux (particularly in the areas of real-time scheduling and priority inheritance), dynamic scheduler domains (particularly the CPUSET abstraction), NPTL threading, and embedded Linux. He graduated from NIT Surat, Gujarat, India, with a B.E. degree in computer engineering in 1995.

Paul E. McKenney

IBM Beaverton Laboratory, 15350 SW Koll Parkway, Beaverton, OR 97006 (paulmck@linux.vnet.ibm.com). Dr. McKenney, an IBM Distinguished Engineer in the Open Systems Development Department, works on synchronization primitives within the Linux kernel for real-time multi-core/multi-threaded systems. His previous work at IBM included SMP, NUMA, and RCU algorithms for Linux and AIX®. Prior to joining IBM he worked on DYNIX®/PTX® at Sequent Computer Systems, on packet-radio and Internet protocols at SRI International, and on real-time systems and business applications as a self-employed contract programmer. He received B.S. degrees in computer science and in mechanical engineering in 1981 from Oregon State University, an M.S. degree in computer science in 1988, also from Oregon State University, and a Ph.D. degree in computer science and engineering in 2004 from Oregon Health & Science University. He holds more than 20 patents and has published more than 30 papers, and is a member of the IBM Academy of Technology.

Josh Triplett

Portland State University, 3890 NE Jackson School Road, Hillsboro, OR 97124 (josh@kernel.org). Mr. Triplett is a Ph.D. degree candidate at Portland State University, in Oregon, where he is involved in research on relativistic programming and advanced synchronization techniques for highly parallel systems. He received a B.Sc. degree (summa cum laude) in computer science from Portland State University in 2005. He has interned for the IBM Linux Technology Center three times, most recently working on real-time Linux and read-copy update (RCU). He maintains the Sparse static analysis tool for C, originally written by Linus Torvalds, and co-maintains the X Window System C Binding (XCB).

Jonathan Walpole

Portland State University, 11160 SW Goldfinch Terrace, Beaverton, OR 97007 (walpole@cs.pdx.edu). Professor Walpole received his Ph.D. degree in computer science from Lancaster University, U.K. He is a Professor in the Computer Science Department at Portland State University. Prior to joining PSU he was a Professor and Director of the Systems Software Laboratory at the OGI School of Science and Engineering at Oregon Health & Science University. His research interests are in operating systems, networking, distributed systems, and multimedia computing. He has pioneered research in adaptive resource management and the integration of application and system-level quality-of-service management. He has also done leading-edge research on dynamic specialization for enhanced performance, survivability and evolvability of large software systems. His research on distributed multimedia systems began in 1988, and in the early 1990s he led the development of one of the first QoS-adaptive Internet streaming video players. ■