RCU Usage In the Linux Kernel: One Decade Later

Paul E. McKenney
Linux Technology Center
IBM Beaverton

Silas Boyd-Wickizer
MIT CSAIL

Jonathan Walpole
Computer Science Department
Portland State University

Abstract

Read-copy update (RCU) is a scalable high-performance synchronization mechanism implemented in the Linux kernel. RCU's novel properties include support for concurrent reading and writing, and highly optimized inter-CPU synchronization. Since RCU's introduction into the Linux kernel over a decade ago its usage has continued to expand. Today, most kernel subsystems use RCU. This paper discusses the requirements that drove the development of RCU, the design and API of the Linux RCU implementation, and how kernel developers apply RCU.

1 Introduction

The first Linux kernel to include multiprocessor support is not quite 20 years old. This kernel provided support for concurrently running applications, but serialized all execution in the kernel using a single lock. Concurrently executing applications that frequently invoked the kernel performed poorly.

Today the single kernel lock is gone, replaced by highly concurrent kernel subsystems. Kernel intensive applications that would have performed poorly on dual-processor machines 15 years ago, now scale and perform well on multicore machines with many processors [3].

Kernel developers have used a variety of techniques to improve concurrency, including fine-grained locks, lock-free data structures, per-CPU data structures, and read-copy-update (RCU), the topic of this paper. Uses of the RCU API have increased from none in 2002 to over 6500 in 2013 (see Figure 1). Most major Linux kernel subsystems use RCU as a synchronization mechanism. Linus Torvalds characterized a recent RCU-based patch to the virtual file system "as seriously good stuff" because developers were able to use RCU to remove bottlenecks affecting common workloads [25]. RCU is not unique to Linux (see [7, 13, 19] for other examples), but Linux's wide variety of RCU usage patterns is, as far as we know, unique among the commonly used kernels. Understanding RCU is now a prerequisite for understanding the Linux implementation and its performance.

The success of RCU is, in part, due to its high performance in the presence of concurrent readers and updaters. The RCU API facilitates this with two relatively simple primitives: readers access data structures within *RCU read-side critical sections*, while updaters use *RCU synchronization* to wait for all pre-existing RCU read-side critical sections to complete. When combined, these primitives allow threads to concurrently read data structures, even while other threads are updating them.

This paper describes the performance requirements that led to the development of RCU, gives an overview of the RCU API and implementation, and examines how kernel developers have used RCU to optimize kernel performance. The primary goal is to provide an understanding of the RCU API and how to apply it.

The remainder of the paper is organized as follows. Section 2 explains the important requirements for production-quality RCU implementations. Section 3 gives an overview of RCU's API and overall design. Section 4 introduces a set of common usage patterns that cover most uses of RCU, illustrating how RCU is used in Linux. In some cases its use as a replacement for existing synchronization mechanisms introduces subtle semantic problems. Section 5 discusses these problems and commonly applied solutions. Section 6 highlights the importance of RCU by documenting its use in Linux over time and by specific subsystems. Section 7 discusses related work and Section 8 presents conclusions.

2 RCU Requirements

RCU fulfills three requirements dictated by the kernel: (1) support for concurrent readers, even during updates; (2) low computation and storage overhead; and (3) deterministic completion time. The first two are performance-related requirements, while the third is important for real-

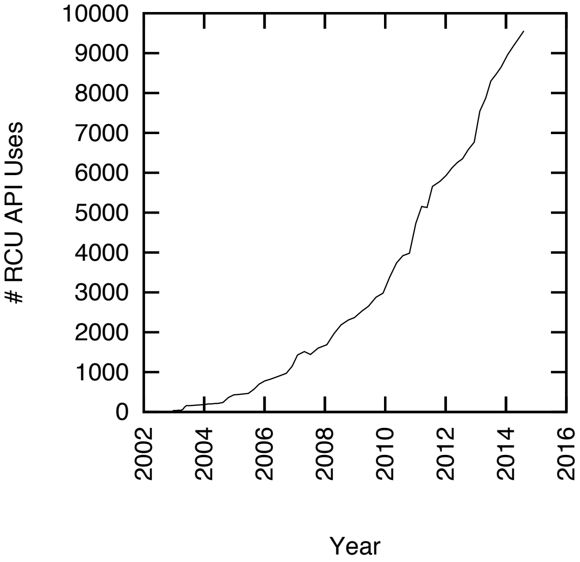

Figure 1: The number of uses of the RCU API in Linux kernel code from 2002 to 2014.

time response and software engineering reasons. This section describes the three requirements and the next two sections describe how RCU is designed to fulfill these requirements and how kernel developers use RCU.

The primary RCU requirement is support for concurrent reading of a data structure, even during updates. The Linux kernel uses many data structures that are read and updated intensively, especially in the virtual file system (VFS) and in networking. For example, the VFS caches directory entry metadata – each known as a `dentry` – for recently accessed files. Every time an application opens a file, the kernel walks the file path and reads the `dentry` for each path component out of the `dentry` cache. Since applications might access many files, some only once, the kernel is frequently loading `dentry`s into the cache and evicting unused `dentry`s. Ideally, threads reading from the cache would not interfere with each other or be impeded by threads performing updates.

The second RCU requirement is low space and execution overhead. Low space overhead is important because the kernel must synchronize access to millions of kernel objects. On a server in our lab, for example, the kernel caches roughly 8 million `dentry`s. An overhead of more than a few bytes per `dentry` is an unacceptable storage overhead.

Low execution overhead is important because the kernel accesses data structures frequently using extremely short code paths. The SELinux access vector cache (AVC) [21] is an example of a performance critical data structure that the kernel might access several times during a system call. In the absence of spinning to acquire a lock or waiting to fulfill cache misses, each read from the AVC takes several hundred cycles. Incurring a single cache miss, which can cost hundreds of cycles, would double the cost of accessing the AVC. RCU must coordinate readers and updaters in a way that provides low overhead synchronization in the common case.

A third requirement is deterministic completion times for read operations. This is critical to real-time response [11], but also has important software-engineering benefits, including the ability to use RCU within non-maskable interrupt (NMI) handlers. Accessing shared data within NMI handlers is tricky, because a thread might be interrupted within a critical section. Using spin locks can lead to deadlock, and lock-free techniques utilizing optimistic concurrency control lead to non-deterministic completion times if the operation must be retried many times.

Related synchronization primitives, such as read-write locks, Linux local-global locks, and transactional memory, do not fulfill the requirements discussed here. None of these primitives provide concurrent read and write operations to the same data. They all impose a storage overhead. Even the storage overhead for a read-write lock, which is a single integer, is unacceptable for some cases. Read-write locks and local-global locks use expensive atomic instructions or memory barriers during acquisition and release.

The next section describes an RCU design and API that provides concurrent reads and writes, low space and execution overhead, and deterministic execution times.

3 RCU Design

RCU is a library for the Linux kernel that allows kernel subsystems to synchronize access to shared data in an efficient manner. The core of RCU is based on two primitives: RCU read-side critical sections, which we will refer to simply as RCU critical sections, and RCU synchronization. A thread enters an RCU critical section by calling `rcu_read_lock` and completes an RCU critical section by calling `rcu_read_unlock`. A thread uses RCU synchronization by calling `synchronize_rcu`, which guarantees not to return until all the RCU critical sections executing when `synchronize_rcu` was called have completed. `synchronize_rcu` does not prevent new RCU critical sections from starting, nor does it wait for the RCU critical sections to finish that were started after `synchronize_rcu` was called.

Developers can use RCU critical sections and RCU synchronization to build data structures that allow concurrent reading, even during updates. To illustrate one possible use, consider how to safely free the memory associated with a `dentry` when an application removes a file. One way to implement this is for a thread to ac-

```
void rcu_read_lock()
{
  preempt_disable[cpu_id()]++;
}

void rcu_read_unlock()
{
  preempt_disable[cpu_id()]--;
}

void synchronize_rcu(void)
{
  for_each_cpu(int cpu)
    run_on(cpu);
}
```

Figure 2: A simplified version of the Linux RCU implementation.

quire a pointer to a dentry only from the directory cache and to always execute in an RCU critical section when manipulating a dentry. When an application removes a file, the kernel, possibly in parallel with dentry readers, removes the dentry from the directory cache, then calls synchronize_rcu to wait for all threads that might be accessing the dentry in an RCU critical section. When synchronize_rcu returns, the kernel can safely free the dentry. Memory reclamation is one use of RCU; we discuss others in Section 4.

RCU allows threads to wait for the completion of pre-existing RCU critical sections, but it does not provide synchronization among threads that update a data structure. These threads coordinate their activities using another mechanism, such as non-blocking synchronization, single updater thread, or transactional memory [15]. Most threads performing updates in the Linux kernel use locking.

The kernel requires RCU to provide low storage and execution overhead and provide deterministic RCU critical section completion times. RCU fulfills these requirements with a design based on scheduler context switches. If RCU critical sections disable thread preemption (which implies a thread cannot context switch in an RCU critical section), then synchronize_rcu need only wait until every CPU executes a context switch to guarantee all necessary RCU critical sections are complete. No additional explicit communication is required between RCU critical sections and synchronize_rcu.

Figure 2 presents a simplified version of the Linux RCU implementation. Calls to rcu_read_lock disable preemption and calls to rcu_read_unlock re-enable preemption. It is safe to nest RCU critical sections. preempt_disable is a CPU-local variable, so threads do not contend when modifying it. To ensure every CPU executes a context switch, the thread calling synchronize_rcu briefly executes on every CPU. Notice that the cost of synchronize_rcu is independent of the number of times threads execute rcu_read_lock and rcu_read_unlock.

In practice Linux implements synchronize_rcu by waiting for all CPUs in the system to pass through a context switch, instead of scheduling a thread on each CPU. This design optimizes the Linux RCU implementation for low-cost RCU critical sections, but at the cost of delaying synchronize_rcu callers longer than necessary. In principle, a writer waiting for a particular reader need only wait for that reader to complete an RCU critical section. The reader, however, must communicate to the writer that the RCU critical section is complete. The Linux RCU implementation essentially batches reader-to-writer communication by waiting for context switches. When possible, writers can use an asynchronous version of synchronize_rcu, call_rcu, that will asynchronously invokes a specified callback after all CPUs have passed through at least one context switch.

The Linux RCU implementation tries to amortize the cost of detecting context switches over many synchronize_rcu and call_rcu operations. Detecting context switches requires maintaining state shared between CPUs. A CPU must update state, which other CPUs read, that indicate it executed a context switch. Updating shared state can be costly, because it causes other CPUs to cache miss. RCU reduces this cost by reporting per-CPU state roughly once per scheduling clock tick. If the kernel calls synchronize_rcu and call_rcu many times in that period, RCU will have reduced the average cost of each call to synchronize_rcu and call_rcu at the cost of higher latency. Linux can satisfy more than 1,000 calls to synchronize_rcu and call_rcu in a single batch [22]. For latency sensitive kernel subsystems, RCU provides expedited synchronization functions that execute without waiting for all CPUs to execute multiple context switches.

An additional consideration with the Linux RCU implementation is the handling of memory ordering. Since RCU readers and updaters run concurrently, special consideration must be given to compiler and memory reordering issues. Without proper care, a reader accessing a data item that a updater concurrently initialized and inserted could observe that item's pre-initialized value.

Therefore, RCU helps developers manage reordering with rcu_dereference and rcu_assign_pointer. Readers use rcu_dereference to signal their intent to read a pointer in a RCU critical section. Updaters use rcu_assign_pointer to mutate these pointers. These two primitives contain architecture-specific memory-barrier instructions and compiler directives to enforce correct ordering. Both primitives reduce to simple assignment statements on sequentially consistent systems. The rcu_dereference primitive is a volatile access except on DEC Alpha, which also requires a memory barrier [4].

Figure 3 summarizes the Linux RCU API. The next section describes how Linux developers have applied RCU.

4 Using RCU

A decade of experience using RCU in the Linux kernel has shown that RCU synchronization is powerful enough to support a wide variety of different usage patterns. This section outlines some of the most common patterns, explaining how to use RCU and what special considerations arise when using RCU to replace existing mechanisms.

We performed the experiments described in this section on a 16-CPU 3GHz Intel x86 system. The experiments were written as a kernel module and use the RCU implementation in Linux 2.6.23.

4.1 Wait for Completion

The simplest use of RCU is waiting for pre-existing activities to complete. In this use case, the waiters use synchronize_rcu, or its asynchronous counterpart call_rcu, and waitees delimit their activities with RCU read-side critical sections.

The Linux NMI system uses RCU to unregister NMI handlers. Before unregistering an NMI handler, the kernel must guarantee that no CPU is currently executing the handler. Otherwise, a CPU might attempt to execute code in invalid or free memory. For example, when the kernel unloads a module that registered an NMI handler, the kernel frees memory that contains the code for the NMI handler.

Figure 4 shows pseudocode for the Linux NMI system. The nmi_list is a list of NMI handlers that requires a spin lock to protect against concurrent updates, but allows lock-free reads. The rcu_list_t abstracts a common pattern for accessing linked lists. The function rcu_list_for_each calls rcu_dereference for every list element, and rcu_list_add and rcu_list_remove call rcu_assign_pointer when modifying the list. The NMI system executes every NMI handler within an RCU critical section. To remove a handler, the NMI system removes the handler from the list, then calls synchronize_rcu. When synchronize_rcu returns, every call to the handler must have returned.

Using RCU in the NMI system has three nice properties. One is that it is high performance. CPUs can execute NMI handlers frequently without causing cache misses on other CPUs. This is important for applications like Perf or OProfile which rely on frequent invocations of NMI handlers.

The second property, which is important for real time applications, is that entering and completing an RCU critical section always executes a deterministic number of instructions. Using a blocking synchronization primitive, like read-write locks, could cause handle_nmi to block for long periods of time.

The third property is that the implementation of the NMI system allows dynamically registering and unregistering NMI handlers. Previous kernels did not allow this because it was difficult to implement in a way that was performant and guaranteed absence of deadlock. Using a read-write lock is difficult because a CPU might be interrupted by an NMI while while holding the lock in unregister_nmi_handler. This would cause deadlock when the CPU tried to acquire the lock again in nmi_handler.

4.2 Reference Counting

RCU is a useful substitute for incrementing and decrementing reference counts. Rather than explicitly counting references to a particular data item, the data item's users execute in RCU critical sections. To free a data item, a thread must prevent other threads from obtaining a pointer to the data item, then use call_rcu to free the memory.

This style of reference counting is particularly efficient because it does not require updates, memory barriers, or atomic operations in the data-item usage path. Consequently, it can be orders of magnitude faster than reference counting that is implemented using atomic operations on a shared counter.

To demonstrate the performance difference we wrote an experiment that creates one thread per CPU. All threads loop and either increment and decrement a shared reference count, or execute an empty RCU critical section. Figure 5 presents the results of the experiment. The y-axis shows the time to either enter and complete an RCU critical section, or to increment and decrement a reference count. The number of CPUs using RCU or accessing the reference count varies along the x-axis.

On one core, it takes 6 nanoseconds to execute an empty RCU critical section, but it takes 66 nanoseconds to increment and decrement a reference count. The overhead of RCU is constant as the number of CPUs increases. The overhead of reference counting increases to 2819 nanoseconds on 16 cores, or more than 400× the cost of RCU.

The Linux networking stack uses RCU to implement high performance reference counting. Figure 6 shows an example that uses RCU to hold a reference to IP options while the kernel networking stack copies the IP options into a packet. udp_sendmsg calls rcu_read_lock before copying the IP options. Once the options are copied, udp_sendmsg calls rcu_read_unlock to complete the

`rcu_read_lock()`	Begin an RCU critical section.
`rcu_read_unlock()`	Complete an RCU critical section.
`synchronize_rcu()`	Wait for existing RCU critical sections to complete.
`call_rcu(callback, arguments...)`	Call the callback when existing RCU critical sections complete.
`rcu_dereference(pointer)`	Signal the intent to deference a pointer in an RCU critical section.
`rcu_assign_pointer(pointer_addr, pointer)`	Assign a value to a pointer that is read in RCU critical sections.

Figure 3: Summary of the Linux RCU API.

```
rcu_list_t nmi_list;
spinlock_t nmi_list_lock;

void handle_nmi()
{
  rcu_read_lock();
  rcu_list_for_each(&nmi_list, handler_t cb)
    cb();
  rcu_read_unlock();
}

void register_nmi_handler(handler_t cb)
{
  spin_lock(&nmi_list_lock);
  rcu_list_add(&nmi_list, cb);
  spin_unlock(&nmi_list_lock);
}

void unregister_nmi_handler(handler_t cb)
{
  spin_lock(&nmi_list_lock);
  rcu_list_remove(cb);
  spin_unlock(&nmi_list_lock);
  synchronize_rcu();
}
```

Figure 4: Pseudocode for the Linux NMI system. The RCU list functions contain the necessary calls to `rcu_dereference` and `rcu_assign_pointer`.

critical section. An application can change the IP options on a per-socket basis by calling `sys_setsockopt`, which eventually causes the kernel to call `setsockopt`. `setsockopt` sets the new IP options, then uses `call_rcu` to asynchronously free the memory storing the old IP options. Using `call_rcu` ensures all threads that might be manipulating the old options will have released their reference by exiting the RCU critical section.

Holding a reference to an object using an RCU critical section is useful if the operation on the object is relatively short. However, if the thread executing the operation must sleep it must exit the RCU critical section. To retain its reference to the object, a thread can increment a traditional reference count. To free the object, the kernel must wait for the traditional reference count to reach zero and synchronize with RCU critical sections.

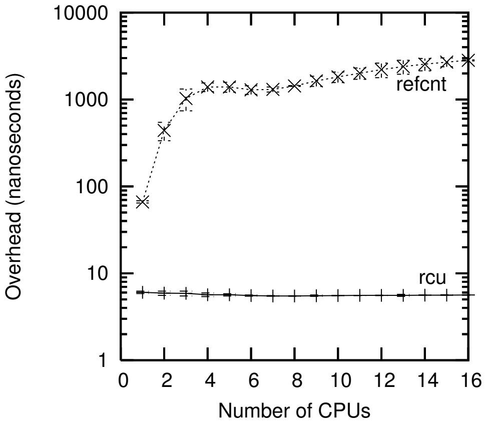

Figure 5: The overhead of entering using RCU as a reference count compared to the overhead of using a shared integer.

Developers use RCU to reference an object not only as a substitute for traditional reference counters, but also to enforce general existence guarantees in code that has never used traditional reference counters. To guarantee object existence simply means an object's memory will not be reclaimed until all references to it have been dropped.

In some situations, subsystems implement existence guarantees using synchronization mechanisms, like spin locks. In other scenarios it might be difficult to guarantee existence in a performant or practical manner, so implementations rely on other techniques, such as tagging the upper bits of pointers to avoid an ABA race in a lock-free algorithm. Linux developers have used RCU in many of these cases to provide high performance existence guarantees. For example, developers refactored the System-V IPC [1] code to provide existence guarantees using RCU instead of spin locks.

4.3 Type Safe Memory

Type safe memory is memory that retains its type after being deallocated. It is helpful in scenarios where a thread might deallocate an object while other threads still hold references to that object. If the object's memory is reallocated for a different purpose, but maintains the same type,

```
void udp_sendmsg(sock_t *sock, msg_t *msg)
{
  ip_options_t *opts;
  char packet[];

  copy_msg(packet, msg);
  rcu_read_lock();
  opts = rcu_dereference(sock->opts);
  if (opts != NULL)
    copy_opts(packet, opts);
  rcu_read_unlock();

  queue_packet(packet);
}

void setsockopt(sock_t *sock, int opt,
                void *arg)
{
  if (opt == IP_OPTIONS) {
      ip_options_t *old = sock->opts;
      ip_options_t *new = arg;

      rcu_assign_pointer(&sock->opts, new);
      if (old != NULL)
        call_rcu(kfree, old);
      return;
  }

  /* Handle other opt values */
}
```

Figure 6: Linux pseudocode for handling IP options using RCU to hold references.

threads can detect the reuse and roll back their operations. This approach is common in lock-free algorithms that use optimistic concurrency control [10]. In general, RCU can be used directly to remove the need for type safe memory, because its existence guarantees ensure that object memory is not reused while threads hold references. However, there are rare situations in which RCU can not be used directly to enforce existence guarantees. For example, when attempts to asynchronously free an object might block, using RCU runs the risk of stalling the thread that executes an entire batch of call_rcu invocations. In such situations, rather than using RCU to provide existence guarantees directly, it can be used to implement type safe memory. An example of this usage pattern occurs in Linux's slab allocators.

Linux slab allocators provide typed object memory. Each slab allocator uses pages of memory that are allocated by the Linux page allocator and splits them up into objects of a single type that can be individually allocated and freed. When a whole page of objects becomes free, the slab allocator returns it to the page allocator, at which point it may be reallocated to a slab allocator of a different type. If developers want to ensure that memory is not reused for objects of a different type while references to it are still active, they can set a special SLAB_DESTROY_BY_RCU flag in the slab. In this case, rcu synchronization is used prior to reallocating the slab to a different slab allocator. If the objects in the slab are only ever accessed in RCU critical sections, this approach has the effect of implementing type safe memory.

One example of using type safe memory is in the reverse page map, which is responsible for mapping a physical page to all the virtual address mappings that include that physical page. Each physical page in the Linux kernel is represented by a page_t and a virtual address mapping is represented by an anon_vma_t [1]. Each page_t contains a pointer to a list of anon_vma_ts that the reverse map allocates from type safe memory. To read an anon_vma_ts for a given page, the reverse map calls rcu_read_lock, reads the anon_vma_t, and calls rcu_read_unlock. A thread can unmap a page and deallocate an anon_vma_t without waiting for all threads to relinquish references to the anon_vma_t, because threads reading the anon_vma_t can tell if the object is being reused by a different object instance by checking bits in the object's memory.

Type safe memory solves a challenge in implementing the reverse map, which is to avoid the race where a thread reads an anon_vma_t from a page_t, but the physical page is unmapped by another thread and the thread then frees the anon_vma_t. One option would be to add a spin lock to the page_t; however, the size of a page_t must be as small as possible, because there exists one for every physical page in the system. Using RCU to hold a reference and freeing the anon_vma_t using call_rcu would work, except that the function that frees an anon_vma_t might acquire a mutex, which would delay the RCU thread responsible for executing RCU callbacks. In principle it would be possible to remove the mutex, but it would require an extensive effort.

Type safe memory provides a practical solution for dealing with the complexities that arise in a large system like Linux. In the example of the reverse page map, it would be theoretically possible to rewrite the anon_vma_t to avoid blocking, but it would require changing all the code that depended on this behavior. Implementing type safety with RCU provided an easily implementable solution. In addition to the virtual memory systemâĂŹs reverse-mapping data structures, there are several other places in the Linux kernel where type-safe memory is used, including signal-handling data structures and networking.

[1] An anon_vma_t usually represents multiple physical pages.

```
syscall_t *table;
spinlock_t table_lock;

int invoke_syscall(int number, void *args...)
{
  syscall_t *local_table;
  int r = -1;

  rcu_read_lock();
  local_table = rcu_deference(table);
  if (local_table != NULL)
    r = local_table[number](args);
  rcu_read_unlock();

  return r;
}

void retract_table()
{
  syscall_t *local_table;

  spin_lock(&table_lock);
  local_table = table;
  rcu_assign_pointer(&table, NULL);
  spin_unlock(&table_lock);

  synchronize_rcu();
  kfree(local_table);
}
```

Figure 7: Pseudocode of the publish-subscribe pattern used to dynamically extend the system call table.

4.4 Publish-Subscribe

In the publish-subscribe usage pattern, a writer initializes a data item, then uses `rcu_assign_pointer` to publish a pointer to it. Concurrent readers use `rcu_dereference` to traverse the pointer to the item, The `rcu_assign_pointer` and `rcu_dereference` primitives contain the architecture-specific memory barrier instructions and compiler directives necessary to ensure that the data is initialized before the new pointer becomes visible, and that any dereferencing of the new pointer occurs after the data is initialized.

This pattern is often combined with existence guarantees in order to publish new versions and reclaim old versions of objects, while permitting concurrent access to those objects.

One example of using publish-subscribe in Linux is to dynamically replace system calls. The PowerPC Cell architecture, for example, appends to the system call table at run time. The kernel appends to the system call table by publishing a pointer to an extension table using `rcu_assign_pointer`. The kernel always calls `rcu_read_lock` before indexing into the extended portion of the table and executing a system call. The kernel uses `rcu_dereference` to read from the extended system call table. The combination of `rcu_assign_pointer` and `rcu_dereference` ensure that no threads will ever observe a partially initialized table extension. If the kernel needs to change the extended portion of the table, it retracts the extended table by setting the extended table pointer to `NULL` with `rcu_assign_pointer`, then uses the wait-for-completion pattern, calling `synchronize_rcu` to guarantee no threads are executing system calls contained in the extended table.

4.5 Read-Write Lock Alternative

The most common use of RCU in Linux is as an alternative to a read-write lock. Reading threads access a data structure in an RCU critical section, and writing threads synchronize with other writing threads using spin locks. The guarantees provided by this RCU usage pattern are different than the guarantees provided by read-write locks. Although many subsystems in the Linux kernel can tolerate this difference, not all can. The next section describes some complimentry techniques that developers can use with RCU to provide the same guarantees as read-write locking, but with better performance.

RCU provides higher performance than traditional read-write locking and can make it easier to reason about deadlock. Linux implements read-write locks using a single shared integer. To acquire the lock in read or write mode a thread must modify the integer using expensive atomic instructions and memory barriers. If another thread running on a different CPU acquires the lock next, that thread will stall while the CPU fulfills the cache miss on the shared integer. If the read operation being performed is relatively short, cache misses from acquiring the lock in read-mode essentially removes all read concurrency.

Figure 8 compares the overhead of two operations: entering and completing an RCU critical section; and acquiring a read-write lock in read-mode and releasing it. The x-axis shows the number of cores and y-axis shows the average time to complete an operation. On one core, entering and completing an RCU critical section takes 6 nanoseconds, while acquiring and releasing a read-write lock, which executes two atomic instructions on *x*86, takes 89 nanoseconds, almost 15× longer. The cost for acquiring the read-write increases with the number of cores, up to 6654 nanoseconds on 16 cores. The cost of RCU remains relatively constant.[2]

Another reason to choose RCU instead of a read-write lock is deadlock immunity. The only way for RCU to deadlock is if a thread blocks waiting for `synchronize_`

[2] Recent high-performance reader-writer-lock implementations either use RCU directly [23, 2] or a special-purpose RCU variant tailored to reader-writer locking [17].

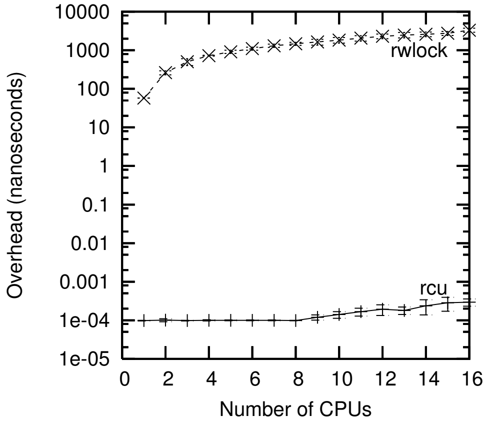

Figure 8: The overhead of entering and completing an RCU critical section, and acquiring and releasing a read-write lock.

rcu in a RCU critical section. RCU does not otherwise contribute to deadlock cycles. Developers do not need to consider rcu_read_lock ordering with respect to other locks to avoid deadlock, which would be necessary if a thread was holding a read-write lock.

One example of using RCU as a read-write lock is to synchronize access to the PID hash table. The Linux PID table maps PIDs to sessions, process groups, and individual processes. To access a process in the PID, a thread calls rcu_read_lock, looks up the process in the table, manipulates the process, then calls rcu_read_unlock. To remove a process from the table, a thread hashes the PID, acquires a per-bucket spin lock, adds the process to the bucket, and releases the lock. Figure 9 presents pseudocode for the Linux PID table.

A key difference between RCU and read-write locking is that RCU supports concurrent reading and writing of the same data while read-write locking enforces mutual exclusion. As a result, concurrent operations on an RCU protected data structure can yield results that a read-write lock would prevent. In the example above, suppose two threads simultaneously add processes A and B to different buckets in the table. A concurrently executing reading thread searching for process A then process B, might find process A, but not process B. Another concurrently executing reader searching for process B then A, might find process B, but not process A. This outcome is valid, but could not occur if the PID table used read-write locks.

Developers considering using RCU must reason about requirements of their application to decide if the additional orderings allowed by RCU, but disallowed by read-write locks, are correct. In addition to the PID table, other important kernel subsystems, such as the directory cache, networking routing tables, the SELinux access vector cache, and the System V IPC implementation, use RCU as an alternative to read-write locks. A tentative conclusion to draw from RCU's widespread use in Linux is that many kernel subsystems are either able to tolerate additional orderings allowed by RCU or use the techniques described in the next section to avoid problematic orderings.

```
pid_table_entry_t pid_table[];

process_t *pid_lookup(int pid)
{
  process_t *p

  rcu_read_lock();
  p = pid_table[pid_hash(pid)].process;
  if (p)
    atomic_inc(&p->ref);
  rcu_read_unlock();
  return p;
}

void pid_free(process *p)
{
  if (atomic_dec(&p->ref))
    free(p);
}

void pid_remove(int pid)
{
  process_t **p;

  spin_lock(&pid_table[pid_hash(pid)].lock);
  p = &pid_table[pid_hash(pid)].process;
  rcu_assign_pointer(p, NULL);
  spin_unlock(&pid_table[pid_hash(pid)].lock);

  if (*p)
    call_rcu(pid_free, *p);
}
```

Figure 9: Pseudocode for the Linux PID table implemented using RCU as an alternative to read-write locks. After calling pid_lookup, a thread calls pid_free to release it's reference to the process.

5 Algorithmic Transformations

Since RCU does not force mutual exclusion between readers and updaters, mechanical substitution of RCU for reader-writer locking can change the application's semantics. Whether this change violates correctness depends on the specific correctness properties required.

Experience in the Linux kernel has uncovered a few common scenarios in which the changes in semantics are problematic, but are handled by the techniques described below. The following subsections discuss three commonly used techniques, explaining why they are needed, how they are applied, and where they are used.

5.1 Impose Level of Indirection

Some uses of reader-writer locking depend on the property that all of a given write-side critical section's updates become visible atomically to readers. This property is provided by mutual exclusion between readers and updaters. However, since RCU allows concurrent reads and updates, a mechanical conversion to RCU could allow RCU readers to see intermediate states of updaters. For example, consider the errors that might arise if the PID table stored `process_t`s directly, instead of pointers to `process_t`s. It would be possible for `pid_lookup` to manipulate and return a partially initialized `process_t`.

This problem is often solved by imposing a level of indirection, such that the values to be updated are all reached via a single pointer which can be published atomically by the writer. In this case, readers traverse the RCU-protected pointer in order to reach the data. The PID table, for example, stores pointers to `process_t`s instead of `process_t`s. This approach ensures that updates appear atomic to RCU readers. The indirection required for this approach to work is often inherent in the linked data structures that are widely used in the Linux kernel, such as linked lists, hash tables, and various search trees.

5.2 Mark Obsolete Objects

The solution discussed in the previous section ensures that updates appear atomic to readers, but it does not prevent readers from seeing obsolete versions that updaters have removed. The RCU approach has the advantage of allowing expedited updates, but in some cases reader-writer locking applications depend on the property that reads not access obsolete versions. One solution is to use a flag with each object that indicates if the object is obsolete. Updaters set the flag when the object becomes obsolete and readers are responsible for checking the flag.

The System V semaphore implementation uses this technique [1]. Each `semaphore_t` has an obsolete flag that the kernel sets when an application deletes the semaphore. The kernel resolves a semaphore ID provided by an application into a `semaphore_t` by looking up the semaphore ID in a hash table protected by `rcu_read_lock`. If the kernel finds a `semaphore_t` with the obsolete flag set, it acts as if the lookup failed.

5.3 Retry Readers

In some cases the kernel might replace an obsolete object with an updated version. In these cases a thread using RCU should retry the operation when it detects an obsolete object, instead of failing. If updates are rare, this technique provides high performance and scalability.

One technique for detecting when a new version of an object is available is to use a Linux sequence lock in conjunction with `rcu_read_lock`. Before modifying an object, an updater thread acquires the sequence lock in write mode, which increments an internal counter from an even value to an odd value. When done modifying the object, the thread releases the sequence lock by incrementing the value by one. Before accessing an object a reader thread reads the value of the sequence lock. If the value is odd, the reader knows that an updater is modifying the object, and spins waiting for the updater to finish. Then the thread calls `rcu_read_lock`, reads the object, and calls `rcu_read_unlock` to complete the RCU critical section. The thread then must read the sequence lock again and check that the value is the same. If the value changed, the thread retries the operation.

The Linux kernel uses RCU with sequence locks throughout the VFS subsystem [18]. For example, each `dentry` has a sequence lock that a thread acquires in write mode when it modifies a `dentry` (*e.g.* to move or to rename it). When an application opens a file, the kernel walks the file path by looking up each `dentry` in the path. To prevent inconsistent lookup results, like opening a file for which the path never existed, the path lookup code reads the sequence lock, and retries if necessary.

6 RCU Usage Statistics

This section examines the usage of RCU in the Linux kernel over time, by subsystem within the kernel, and by type of RCU primitive. The purpose of this analysis is to demonstrate that developers use RCU in many kernel subsystems and that it's likely RCU usage will increase in future Linux kernels.

Figure 1 shows how the usage of RCU in the Linux kernel has increased over time, where the number of RCU API uses counts the number of occurrences of RCU primitives in the kernel source. Although this increase has been quite large, there are more than ten times as many uses of locking (of all types) as there are of RCU. However, in the time that RCU went from zero to more than 9000, reader-writer locking went from about 3000 uses to only about 4000. In that same time, the Linux kernel source code more than tripled. The slow growth of read-write lock usage is due in part to conversions to RCU.

Figure 10 shows the usage of RCU in each kernel subsystem. These counts exclude indirect uses of RCU via

Subsystem	Uses	LoC	Uses / KLoC
ipc	92	9,094	10.12
virt	82	10,037	8.17
net	4519	839,441	5.38
security	289	73,134	3.95
kernel	885	224,471	3.94
block	76	37,118	2.05
mm	204	103,612	1.97
lib	75	94,008	0.80
fs	792	1,131,589	0.70
init	2	3,616	0.55
include	331	642,722	0.51
drivers	1949	10,375,284	0.19
crypto	12	74,794	0.16
arch	249	2,494,395	0.10
tools	2	144,181	0.01
Total	9,559	16,257,496	0.59

Figure 10: Linux 3.16 RCU usage by subsystem.

Type of Usage	API Usage
RCU critical sections	4,431
RCU dereference	1,365
RCU synchronization	855
RCU list traversal	813
RCU list update	745
RCU assign	454
Annotation of RCU-protected pointers	393
Initialization and cleanup	341
RCU lockdep assertion	37
Total	9,434

Figure 11: Linux 3.16 RCU usage by RCU API function.

wrapper functions or macros. Lines of code are computed over all the .c and .h files in the Linux source tree.

Linux's networking stack contains almost half of the uses of RCU, despite comprising only about 5% of the kernel. Networking is well-suited to RCU due to its large proportion of read-mostly data describing network hardware and software configuration. Interestingly enough, the first uses of DYNIX/ptx's RCU equivalent [19] also involved networking.

System V inter-process communications (ipc) uses RCU most intensively, with 1% of its lines of code invoking RCU. The ipc code contains many read-mostly data structure, for example, those mapping from user identifiers to in-kernel data structures that benefit from using RCU. The Linux networking achieves similar performance benefits by using RCU critical sections to access read-mostly data, like device configuration and routing tables. Linux's drivers contain the second-greatest number of uses of RCU, but also have low intensity: drivers have only recently started using RCU.

Aside from tools, which contain large quantities of scripting, the architecture specific code (arch) uses RCU the least intensively. One possible reason is that manipulating the hardware state (*e.g.* programming interrupt controllers or writing to MSRs) does not usually support the approach of updating by creating a new version of hardware state while threads might concurrently read an older version. The kernel must update the hardware state in place. Another possible reason is that much of the architecture code is used during boot and to periodically re-configuring hardware, but is not invoked frequently enough to be a performance concern.

Figure 11 breaks down RCU usage into types of RCU API calls. RCU critical sections are used most frequently, with 4431 uses. RCU critical sections access RCU-protected data using `rcu_dereference` or using RCU list functions that automatically call `rcu_deference` when accessing a list. Together, `rcu_dereference` and RCU list traversal functions account for 2178 RCU API uses. Updates to RCU-protected data (RCU synchronization, RCU list update, and RCU pointer assignment) account for 2054 uses of the RCU API. The remaining uses of the RCU API help analyze correctness (annotating RCU-protected pointers for Linux Sparse and RCU lock dependency assertions) and initialize and cleanup RCU data structures.

RCU was accepted into the Linux kernel more than a decade ago. Each subsequent kernel has used RCU more heavily. Today RCU pervades the kernel source, with about one of every 2,000 lines of code being an RCU primitive. Developers use RCU critical sections more frequently than other RCU API calls. RCU usage ranges from about one of every 70,000 lines of code (tools) to more than one out of every 100 lines of code (ipc). RCU use will likely increase in future kernels as developers re-implement subsystems and implement new features that rely on RCU for high performance.

7 Related Work

McKenney and Slingwine first implemented and documented RCU in the Sequent DYNIX/ptx OS [19], and later in the Linux kernel [24]. Numerous other RCU-like mechanisms have been independently invented [16, 13]. The Tornado and K42 kernels used an RCU-like mechanism to enforce existence guarantees [7]. In these cases, the reclamation of object memory was deferred until the number of active threads reached zero on each CPU, rather than waiting for each CPU to context switch, as is the case with RCU. Michael's Hazard Pointers technique [20] is similar to RCU in that it can be used to defer collection of memory until no references remain. However, unlike RCU, it requires readers to write state (hazard pointers) for each reference they hold. Writing this state requires memory barriers too on architectures with weak memory consistency semantics. Fraser solved the same deferred reclamation problem using an epoch-based approach [6]. Recently, Gotsman used a combination of separation logic and temporal logic to verify RCU, hazard pointers and epoch-based reclamation, and showed that all three rely on the same implicit synchronization pattern based on the concept of a grace-period [8, 9]. The performance of the three approaches has also been compared [12].

Although the discussion in this paper has focused on one particular implementation of RCU, several other specialized RCU implementations exist within the Linux kernel. These implementations make use of other system events, besides context switches, and allow RCU critical sections to be preempted and to sleep [11]. RCU is also no longer restricted to the domain of kernel programming. Desnoyers has produced an open source, user-level implementation of RCU, making RCU available to application programmers [5].

Although the discussion in this paper has focused on a Linux kernel implementation of RCU, RCU is no longer restricted to Linux kernel programming. Desnoyers has produced an open source, user-level implementation of RCU, making RCU available to application programmers [5].

The challenges and opportunities presented by RCU's concurrent reading and writing are the focus of ongoing research on relativistic programming (RP). Triplett presented a causal ordering model that simplifies the construction of scalable concurrent data structures using RCU-like primitives [26]. This work has led to the development of numerous highly scalable concurrent data structures, including hash tables [27] and Red-Black Trees [14]. Howard has extended this work to show how relativistic programming constructs such as RCU can be combined with transactional memory [15] in order to support automatic disjoint access parallelism for writes concurrent with relativistic reads.

The use of RCU to defer memory reclamation has led to comparisons to garbage collectors. RCU is not a complete garbage collector. It automatically determines *when* an item can be collected, but it does not automatically determine *which* items can be collected. The programmer must indicate which data structures are eligible for collection, and must enclose accesses in RCU read-side critical sections. However, a garbage collector can be used to implement something resembling RCU [16].

8 Conclusions

RCU for Linux was developed to meet performance and programmability requirements imposed by the kernel and not fulfilled by existing kernel synchronization primitives. Over the last decade developers have applied RCU to most subsytems in the Linux kernel, making RCU an essential component of the Linux kernel. This paper described some of the notable usage patterns of RCU, including wait for completion, reference counting, publish-subscribe, type safe memory, and read-write locking. It also described three design patterns, impose level of indirection, mark obsolete objects, and retry readers, that developers can use to transform incompatible applications to a form suitable for RCU use. Given the trend of increasing RCU usage, it is likely RCU will continue to play an important role in Linux performance.

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.

[2] BHAT, S. S. percpu_rwlock: Implement the core design of per-CPU reader-writer locks. `https://patchwork.kernel.org/patch/2157401/`, February 2013.

[3] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation* (Vancouver, BC, Canada, October 2010), USENIX, pp. 1–16.

[4] COMPAQ COMPUTER CORPORATION. Shared memory, threads, interprocess communication. Available: `http://www.openvms.compaq.com/wizard/wiz_2637.html`, August 2001.

[5] DESNOYERS, M., MCKENNEY, P. E., STERN, A., DAGENAIS, M. R., AND WALPOLE, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[6] FRASER, K., AND HARRIS, T. Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*, 2 (2007), 1–61.

[7] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100.

[8] Gotsman, A., Rinetzky, N., and Yang, H. Verifying highly concurrent algorithms with grace (extended version). http://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf, July 2012.

[9] Gotsman, A., Rinetzky, N., and Yang, H. Verifying concurrent memory reclamation algorithms with grace. In *ESOP'13: European Symposium on Programming* (Rome, Italy, 2013), Springer, pp. 249–269.

[10] Greenwald, M., and Cheriton, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.

[11] Guniguntala, D., McKenney, P. E., Triplett, J., and Walpole, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal 47*, 2 (May 2008), 221–236.

[12] Hart, T. E., McKenney, P. E., Brown, A. D., and Walpole, J. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput. 67*, 12 (2007), 1270–1285.

[13] Hennessy, J. P., Osisek, D. L., and Seigh II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989.

[14] Howard, P. *Extending Relativistic Programming to Multiple Writers*. PhD thesis, Portland State University, 2012.

[15] Howard, P. W., and Walpole, J. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, pp. 1–6.

[16] Kung, H. T., and Lehman, Q. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[17] Liu, R., Zhang, H., and Chen, H. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 219–230.

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

[19] McKenney, P. E., and Slingwine, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.

[20] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June 2004), 491–504.

[21] Morris, J. [PATCH 2/3] SELinux scalability - convert AVC to RCU. http://marc.theaimsgroup.com/?l=linux-kernel&m=110054979416004&w=2, November 2004.

[22] Sarma, D., and McKenney, P. E. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)* (June 2004), USENIX Association, pp. 182–191.

[23] Shenoy, G. R. [patch 4/5] lock_cpu_hotplug: Redesign - lightweight implementation of lock_cpu_hotplug. Available: http://lkml.org/lkml/2006/10/26/73 [Viewed January 26, 2009], October 2006.

[24] Torvalds, L. Linux 2.5.43. Available: http://lkml.org/lkml/2002/10/15/425 [Viewed March 30, 2008], October 2002.

[25] Torvalds, L. Linux 2.6.38-rc1. Available: https://lkml.org/lkml/2011/1/18/322 [Viewed March 4, 2011], January 2011.

[26] Triplett, J. *Relativistic Causal Ordering: A Memory Model for Scalable Concurrent Data Structures*. PhD thesis, Portland State University, 2012.

[27] Triplett, J., McKenney, P. E., and Walpole, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR USA, June 2011), The USENIX Association, pp. 145–158.