RCU Usage In the Linux Kernel: Eighteen Years Later

Paul E. McKenney
Facebook

Joel Fernandes
Google

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 forward progress for readers and writers as well as highly optimized inter-CPU synchronization. RCU was introduced into the Linux kernel eighteen years ago and most subsystems now 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 Linux kernel added multiprocessor support almost 25 years ago. This kernel provided support for concurrently running applications, but serialized all kernel execution 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 20 years ago now scale and perform well on multicore machines with many processors [11].

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. The number of uses of RCU API members has increased from none in 2002 to more than 16,000 in 2020 (see Figure 1).

Most major Linux kernel subsystems use RCU as a synchronization mechanism. Linus Torvalds once 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 [53]. RCU is not unique to Linux (see [15, 22, 28, 39, 44, 51] 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 perfor-

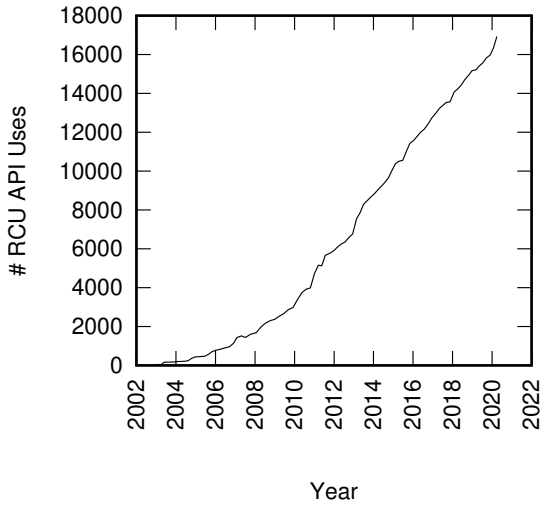

Figure 1: The number of uses of RCU API members in Linux kernel code from 2002 to 2020.

mance. 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) useful forward progress for concurrent readers, even during updates; (2) low computation and storage overhead; and (3) deterministic completion time [36, 35]. The first two are performance related requirements, while the third is important for real-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 memory and execution overhead. Low memory overhead is important because the kernel must synchronize access to millions of kernel objects. For example, on a ThinkPad laptop with 64GB of RAM, the kernel caches more than 4 million `dentry` objects and and 1 million `ext4 inodes`, to call out but two of many RCU-protected objects. Therefore, an overhead of more than a few bytes per object is unacceptable.

Low execution overhead is important because the kernel accesses data structures frequently using extremely short code paths. The SELinux access vector cache (AVC) [42] 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 [19], 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 an NMI migth interrupt a thread within a critical section. Using spinlocks 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 useful concurrent forward progress to both read and write operations acting on 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. In contrast, in the most aggressive RCU implementations (for example, those used by Linux-kernel servers), RCU readers might execute exactly the same sequence of machine instructions that would be executed by a single-threaded traversal of that same data structure.

The next section describes an RCU design and API that provides concurrent reads and writes, low memory 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

```
void rcu_read_lock()
{
  preempt_disable();
}

void rcu_read_unlock()
{
  preempt_enable();
}

void synchronize_rcu(void)
{
  int cpu;

  for_each_online_cpu(int cpu)
    run_on(cpu);
}
```

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

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 acquire 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 preexisting 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 [23]. 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 it, and these calls may be nested. `preempt_disable` and `preempt_enable` operate on the current CPU and in server-class non-preemptible kernels generate no machine code. Therefore, unlike read-write locks, RCU readers do not suffer from lock contention, even in the presence of updaters. 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 batches reader-to-writer communication by waiting for context switches. Writers can also use `call_rcu`, which asynchronously invokes a specified callback function after all CPUs have passed through at least one context switch. In the Linux kernel, `synchronize_rcu` is implemented in terms of `call_rcu`.

The Linux RCU implementation amortizes 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 [48].

For latency sensitive kernel subsystems, RCU provides expedited synchronization functions that send non-idle CPUs an inter-processor interrupt (IPI) to achieve order-of-magnitude reductions in latency compared to `synchronize_rcu`, but by incurring both greatly increased per-updater overhead and also IPI-induced degradations in real-time response latencies. In contrast, the

non-expedited synchronize_rcu avoids use of IPIs, even compared to sleeplocks, which often use IPIs to awaken sleeping threads.

Because RCU readers and updaters run concurrently, special consideration must be given to compiler and hardware memory-access re-ordering 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. The required ordering properties have been formalized [1].

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 [12].

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 the memory that contains the code for the NMI handler.

Figure 4 shows the pseudocode for the Linux NMI system. The nmi_list is a list of NMI handlers that requires a spinlock 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 a test that creates one thread per CPU. Each thread performs one operation per pass through a CPU-bound loop. In one experiment, this operation was an atomic increment

`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`.

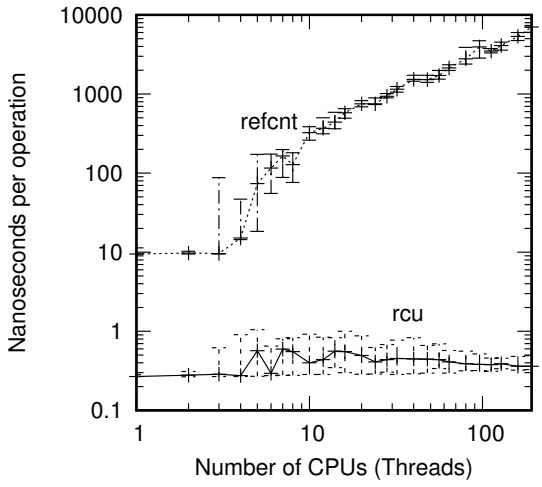

Figure 5: The overhead of RCU readers compared to that of a shared-integer reference count.

and decrement of a shared reference counter. In the other experiment, this operation was an empty RCU critical section. These two experiments are compared in Figure 5, where the y-axis shows the time required to execute one operation and the x-axis shows the number of threads.

It takes about 10 nanoseconds to increment and decrement a reference count on a single core, increasing to about 7,000 nanoseconds on 192 cores. In contrast, it takes only a few hundred *picoseconds* to execute an empty RCU critical section regardless of the number of cores doing so concurrently. RCU's advantage over reference counting therefore rises from more than an order of magnitude on a single core to more than four orders of magnitude on 192 cores.

The Linux networking stack uses RCU to implement high performance reference counting. Figure 6 lists example pseudocode showing the kernel networking stack's usage of RCU to hold a reference to IP options while the options are copied 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 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.

Using RCU read-side critical sections to hold a reference to an object is useful if the operation on the object is relatively short. However, if the thread executing the operation decides to sleep, then it must exit the RCU critical section prior to sleeping. To retain its reference to the object, a thread can increment a traditional reference counter before the RCU critical section ends. To free the

```
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.

```
struct pid
{
  refcount_t count;
  struct rcu_head rcu; ...
};

struct pid *find_get_pid(pid_t nr)
{
  struct pid *pid;
  rcu_read_lock();
  pid = find_vpid(nr);
  if (pid)
    refcount_inc(&pid->count);
  rcu_read_unlock();
  return pid;
}

void put_pid(struct pid *pid)
{
  if (!pid)
    return;
  if (refcount_dec_and_test(&pid->count)) {
    kmem_cache_free(ns->pid_cachep, pid);
  }
}

static void delayed_put_pid(struct rcu_head *rhp)
{
  struct pid *pid = container_of(rhp, struct pid);
  put_pid(pid);
}
```

Figure 7: Pseudocode for reference counters with RCU: Routines for acquiring and releasing references.

object, the kernel must wait for the reference counter to reach zero and synchronize with RCU critical sections currently in flight. Figure 7 and Figure 8 lists pseudocode showing usage of this pattern for management of the kernel's Process ID (PID) descriptor objects. Given a Process ID number, `find_vpid()` searches for the `struct pid` in a radix tree. During this search, the `rcu_read_lock` is held throughout, ensuring that any objects found will exist long enough for `find_get_pid` to increment the object's reference counter. Once incremented, the `rcu_read_lock` may be dropped because the object is now protected by the reference counter. When a process exits, its `struct pid` object is freed by calling `free_pid`. `free_pid` first removes the `struct pid` object from the radix tree making it inaccessible to callers of `find_get_pid`. It then uses `call_rcu` to wait for an RCU grace period before dropping a reference to the object via a call to `delayed_put_pid`. If the object's reference counter reaches 0, it can safely be freed since no new references to it are possible owing to the fact that the object was previously removed from the radix tree. Otherwise, the final call to `put_pid` from the last reference holder would take care of free'ing the object, which happens purely using reference counters without any RCU involvement.

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 spinlocks. 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 [4] code to provide existence guarantees using RCU instead of spinlocks.

```
void free_pid(struct pid *pid)
{
  spin_lock_irqsave(&pidmap_lock, flags);
  idr_remove(&ns->idr, upid->nr);
  spin_unlock_irqrestore(&pidmap_lock, flags);
  call_rcu(&pid->rcu, delayed_put_pid);
}

struct pid *alloc_pid(struct pid_namespace *ns,
    pid_t *set_tid, size_t set_tid_size)
{
  struct pid *pid;
  pid = kmem_cache_alloc(ns->pid_cachep);
  if (!pid)
    return ERR_PTR(retval);
  spin_lock_irq(&pidmap_lock);
  refcount_set(&pid->count, 1);
  pid->nr = idr_alloc();
  idr_add(pid, pid->nr);
  spin_unlock_irq(&pidmap_lock);
  return pid;
}
```

Figure 8: Pseudocode for reference counters with RCU: Routines for allocation and free.

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, threads can detect the reuse and roll back their operations. This approach is common in lock-free algorithms that use optimistic concurrency control [18]. 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_t` structures that the reverse map allocates from type safe memory. To read the `anon_vma_t` for a given page, the reverse map calls `rcu_read_lock`, reads the `anon_vma_t`, and calls `rcu_read_unlock`. Another thread can concurrently unmap a page and deallocate its `anon_vma_t` without waiting for all threads to relinquish references to that `anon_vma_t`. This is because threads reading the `anon_vma_t` can tell if it has been reused (thus possibly representing a different mapping) by checking bits within the `anon_vma_t`. This works because as long as the readers of `anon_vma_t` structures remain within their RCU critical sections, the object will remain of type `anon_vma_t` even after it has been deallocated: The page containing the `anon_vma_t` will not be released from the slab allocator until after at least one full RCU grace periods elapses, which cannot happen until after all pre-existing RCU readers have exited their critical sections.

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 spinlock 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

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

with RCU provided an easily implementable solution. In addition to the virtual memory system's 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.

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 spinlocks. 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 complimentary 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 10 compares the overhead of an empty RCU read-side critical section to that of an empty read-write spinlock critical section. The x-axis shows the number of cores and y-axis shows the average time to complete one critical section. The overhead of an empty spinlock read-side critical section ranges from about 11 nanoseconds on a single core to more than 7,000 nanoseconds on 192 cores. In contrast, the overhead of the RCU read-side critical section is a few hundred picoseconds regardless of the number of cores.

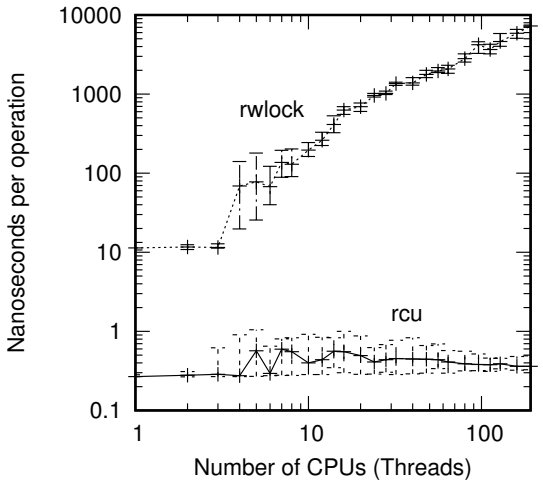

Figure 10: The overhead of an empty RCU read-side critical section compared to that of read-write locking.

Section 4.6 discusses how to use RCU to create faster reader-writer locks.

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 within a read-side critical section blocks waiting for all RCU read-side critical sections to end, for example, by calling `synchronize_rcu` inside an RCU read-side critical section. In contrast with (say) reader-writer locks, developers need not consider `rcu_read_lock` ordering with respect to other locks.

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 spinlock, adds the process to the bucket, and releases the lock. Figure 11 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.

```
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 11: 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.

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.

```
1 struct percpu_rw_semaphore {
2   struct rcu_sync         rss;
3   unsigned int __percpu   *read_count;
4   struct rw_semaphore     rw_sem;
5   struct rcuwait          writer;
6   int                     readers_block;
7 };
```

Figure 12: percpu_rw_semaphore structure definition

4.6 RCU-Mediated Fast Paths

A heavily used design pattern provides optimized *fast paths* for common cases of operations. RCU itself is an example of this pattern, for example, providing a fast path for readers traversing a linked data structure, along with a corresponding *slow path* for updaters.

But RCU also mediates fast paths for other algorithms, for example, for reader-writer locks [8, 32, 49]. This section focuses on the Linux kernel's `percpu_rw_semaphore` locking primitive. This primitive supports highly concurrent readers with orders of magnitude performance and scalability over that of the traditional reader-writer semaphore (`rw_semaphore`). Although `rw_semaphore` readers do not block each other, they nevertheless incur substantial memory contention due to their atomic operations on the `rw_semaphore` data structure. These atomic operations result in expensive communications cache misses, degrading both performance and scalability, even for a read-only workload. The `percpu_rw_semaphore` primitive avoids this degradation by making use of a per-CPU read counter on its read-only fast path. It thereby achieves read-side performance and scalability advantages similar to those of RCU in Figure 10, with these advantages increasing with increasing numbers of CPUs.

Figure 12 shows the `percpu_rw_semaphore` data structure. The `read_count` field references a per-CPU counter tracking the number of readers for the corresponding CPU. A writer cannot enter its critical section until all CPU's `read_count` values become zero. The `rw_sem` field is a traditional `rw_semaphore` that is used in the presence of writers. The `readers_block` field is used to communicate the presence of any writers to the readers. The `rss` and `writer` fields mediate the transition between readers and writers.

The following sections describe the fast and slow paths for read-lock acquisition, followed by the procedure writers use to switch readers from their fast path to their slow path. Additional details may be found in the Linux-kernel source code.

```
 1 void percpu_down_read(sem)
 2 {
 3   rcu_read_lock();
 4   if (sem->rss->state == 0) {
 5     this_cpu_inc(sem->read_count);
 6     rcu_read_unlock();
 7   } else {
 8     rcu_read_unlock();
 9     down_read(&sem->rw_sem);
10     this_cpu_inc(sem->read_count);
11     up_read(sem->rw_sem);
12   }
13 }
14
15 void percpu_down_write(sem)
16 {
17   sem->rss->state = 1;
18   synchronize_rcu();
19   down_write(&sem->rw_sem);
20   wait_for_all_read_count_zero(&sem->rw_sem);
21 }
```

Figure 13: percpu_rw_semaphore read and write lock pseudo-code

4.6.1 Read-Lock Acquisition Fast Path

If a reader acquires the lock in the absence of writers, the reader need only enter an RCU read-side critical section, check the (read-mostly) `->rss` state, increment its per-CPU `->read_count` field, and finally exit the RCU read-side critical section, as shown in the rough pseudocode on lines 3-6 of Figure 13. The per-CPU and read-mostly nature of the two accesses on lines 4 and 5 means that reader acquisition will normally not incur any communications cache misses, so that read acquisition in the absence of writers is fast and scalable.

4.6.2 Read-Lock Acquisition Slow Path

In the presence of writers, the reader check of the `->rss` state on line 4 of Figure 13 will fail, and the reader will therefore execute its slow path. This slow path exits the RCU read-side critical section (line 8), read-acquires the `->rwsem` reader-writer semaphore (line 9), increments the per-CPU `->read_count` field (line 10), and finally read-releases the `->rwsem` reader-writer semaphore (line 11).

Executing this last read release during acquisition of the `percpu_rw_semaphore` might seem unconventional, but it is not necessary for a reader to exclude writers for the full duration of its read-side critical section. As will be shown in the next section, it instead suffices to ensure that any subsequent writer that write-acquires `->rw_sem` sees the increase in the `->read_count` per-CPU counter.

4.6.3 Read-Lock Fast/Slow-Path Switch

Before a writer acquires a lock, it must first notify any future read-side acquisitions that they must use their slow path. Rough pseudocode for this notification is shown on lines 15-21 of Figure 13.

The first step of this notification is on line 17, which sets the `->rss` state, which will cause future readers' checks on line 4 to choose the slow path on lines 8-11. Unfortunately, there might be a large number of readers who have already executed their fast paths (lines 3-6), but who have not yet exited their critical sections. There might be additional readers who have not yet fetched the `->rss` state, but who, due to weak memory ordering, are nevertheless destined to see the old value. The writer clearly must wait for all of these readers before entering its write-side critical section. This write-side waiting process has several phases.

The first phase is shown on line 18. The `synchronize_rcu()` on this line will wait for any readers currently executing within the RCU read-side critical section spanning lines 3-8 to exit that critical section. Therefore, once this `synchronize_rcu()`, all future read acquisitions will see the new `->rss` state, and will therefore choose the slowpath.

The second phase is line 18's `down_write()`, which will wait for any readers read-holding the `->rw_sem` (lines 9-11) to release it. It will also prevent any future readers reaching line 9 from acquiring `->rw_sem`. Because the readers decrement their per-CPU `->read_count` counters when exiting their read-side critical sections, the sum of these counters can no longer increase, but will instead decrease monotonically towards zero.

This sets the stage for the third and final phase on line 20, which waits for this sum to reach zero.

This phased approach is encapsulated within the Linux kernel's RCU-sync framework [43].

Note that this `percpu_rw_semaphore` is helpful only if write acquisitions are quite rare. To see this, consider that the latency incurred by each writer's call to `synchronize_rcu` will be several milliseconds at a minimum. The `percpu_rw_semaphore` nevertheless sees significant use in the Linux kernel, for example, for serial lines, where connects and disconnects are quite rare compared to character I/O. However, it is also possible to configure `percpu_rw_semaphore` to avoid `synchronize_rcu` at the expense of a full memory barrier in the read path. This configuration is used by the Linux kernel's control groups ("cgroups") `cgroup_threadgroup_rwsem` lock [54].

This example shows how RCU can be used to effect a safe and controlled switch between fast and slow paths.

5 Algorithmic Transformations

Since RCU does not force mutual exclusion between readers and updaters, the 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 [4]. 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 [38]. 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. This analysis demonstrates that developers use RCU in many kernel subsystems and that RCU usage has increased dramatically over time.

Figure 1 on page 1 shows the increase in the usage of RCU in the Linux kernel over time, where the y-axis indicates the number of occurrences of RCU primitives in the kernel source. Although this increase has been quite dramatic, there are almost 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 16,000, reader-writer locking went from about 3000 uses to almost 4000 [34]. In that same time, the Linux kernel source code more than quadrupled. This slow growth of read-write lock usage is due in part to conversions to RCU.

Figure 14 shows the usage of RCU in each kernel subsystem. These counts exclude indirect uses of RCU via wrapper functions or macros. Lines of code are computed over all the .c and .h files in the Linux source tree that potentially contribute to the Linux-kernel object code.

Linux's networking stack contains more than 40% 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, early uses of DYNIX/ptx's RCU equivalent [39] also involved networking.

System V inter-process communications (IPC) uses RCU most intensively, with almost 1% of its lines of code invoking RCU. The IPC code contains many RCU-protected read-mostly data structures, for example, those mapping from user identifiers to the corresponding in-kernel data structures.

Linux's drivers contain the second-greatest number of uses of RCU, but also have low intensity, in part because drivers have more recently started using RCU. From an intuitive standpoint, device drivers often need exclusive locking to safely access device state. However, it is also the case that many of them must also track external state, a task well-suited to RCU. For example, the wireless drivers' use of RCU has increased from 339 in 3.16 to 719 in 5.6, with ethernet and GPU drivers also showing large increases. Furthermore, there are several drivers with more than one RCU use per hundred lines of code, led by the recently added wireguard drivers with more than one RCU use per fifty lines of code.

Aside from sound, which whose use of RCU is relatively recent, 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. Instead, the kernel must update the hardware state in place, which often requires exclusive locking. Nevetheless, the RCU-usage intensity of this code has doubled from the time of Linux-kernel version 3.16 to version 5.6.

Figure 15 breaks down RCU usage into types of RCU API calls. Focusing first on the 5.6 column, RCU critical sections are used most frequently, with 7,617 uses in Linux kernel version 5.6. These critical sections access RCU-protected data using `rcu_dereference` on a per-pointer basis or by using RCU list functions that use `rcu_deference` internally. Taken together, RCU pointer and list traversal functions account for 4,140 RCU

Subsystem	5.6 Uses	5.6 LoC	5.6 Uses/KLoC	3.16 Uses	3.16 LoC	3.16 Uses/KLoC	Delta Uses 3.16-5.6
ipc	91	9,550	9.53	92	9,094	10.12	-1
net	6,959	1,116,949	6.23	4,519	839,441	5.38	+2440
security	449	99,352	4.52	289	73,134	3.95	+160
kernel	1,407	361,593	3.89	885	224,471	3.94	+522
virt	85	26,624	3.19	82	10,037	8.17	+3
block	126	58,148	2.17	76	37,118	2.05	+50
mm	287	145,251	1.98	204	103,612	1.97	+83
init	8	4,352	1.84	2	3,616	0.55	+6
lib	263	172,995	1.52	75	94,008	0.80	+188
fs	1,220	1,336,854	0.91	792	1,131,589	0.70	+428
include	758	1,031,768	0.73	331	642,722	0.51	+427
drivers	4,806	16,928,299	0.28	1,949	10,375,284	0.19	+2857
arch	445	2,197,420	0.20	249	2,494,395	0.10	+196
crypto	6	101,679	0.06	12	74,794	0.16	-6
sound	21	1,192,625	0.02				+21
Total	16,931	24,783,459	0.68	9,559	16,257,496	0.59	+7372

Figure 14: Linux 5.6 and 3.16 RCU usage by subsystem.

API uses (2,865 and 1,275, respectively). Updates to RCU-protected data (RCU synchronization, RCU list update, and RCU pointer update) account for 3,470 uses of the RCU API (1,143, 1,156, and 871, respectively). 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.

Qualitatively comparing the 5.6 and 3.16 columns, the overall usage profile of the RCU API members has remained steady. The largest outliers are RCU pointer annotation (2.3x) and other forms of RCU validation (4.4x), both of which might be interpreted as signalling an increased focus on Linux-kernel robustness.

RCU was accepted into the Linux kernel about 18 years ago, and each subsequent kernel has used RCU more heavily. Today RCU pervades the kernel source, with about one of every 1,500 lines of code being an RCU primitive. RCU usage ranges from about one of every 60,000 lines of code (sound) to almost one out of every 100 lines of code (ipc). RCU use will likely continue to increase with the increasing size of the kernel, but as the virt subsystem shows, adding lines of code does not necessarily result in corresponding increases in RCU usage. Nevertheless, during the past 18 years, RCU has proven to be quite valuable within the confines of the Linux kernel. The next section looks further afield, presenting other uses of RCU and at other deferred-reclamation mechanisms.

Usage Category	API Usage 5.6	API Usage 3.16	Delta
RCU critical sections	7,617	4,431	3,168
RCU pointer traversal	2,865	1,365	1,500
RCU synchronization	1,443	855	855
RCU list traversal	1,275	813	462
RCU list update	1,156	745	411
RCU pointer annotation	914	393	521
RCU pointer update	871	454	417
Initialization and cleanup	626	341	285
RCU validation	164	37	127
Total	16,931	9,434	7,497

Figure 15: Linux 5.6 and 3.16 RCU usage by category.

7 Related Work

McKenney and Slingwine first implemented and documented RCU in the Sequent DYNIX/ptx OS [39], and later in the Linux kernel [52]. Numerous other RCU-like mechanisms have been independently invented [22, 24, 25, 30]. The Tornado and K42 kernels used an RCU-like mechanism to enforce existence guarantees [15], where 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 the original DYNIX/ptx RCU implementation. Fraser solved the same deferred-reclamation problem using an epoch-based RCU implementation [14]. 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 [16, 17]. More recently, three different teams applied two different mechanical formal verification tools to significant portions of Linux-kernel RCU [29, 31, 47].

The performance of a number of deferred-reclamation approaches has also been compared [21, 37]. 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 [19].

Furthermore, RCU is by no means restricted to the Linux kernel, having been applied to a number of additional kernels [28, 51]. RCU is now also available to userspace applications via Desnoyers's open source library [13]. More recently, a number of other user-level RCU-like implementations have appeared [2, 3, 5, 6, 9, 27, 32, 33, 45, 46, 50].

Michael's Hazard Pointers technique [41] 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 often requires memory barriers on architectures with weak memory consistency semantics, but on the other hand, hazard pointers often enjoys a much smaller memory footprint. Hazard pointers is also widely used [7, 10, 20, 26, 40].

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 [30].

More detail on deferred reclamation may be found elsewhere [37, Chapter 9].

8 Conclusions

Linux-kernel RCU was developed to meet performance and programmability requirements that could not be accommodated by existing synchronization primitives. Over the last eighteen years 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, type safe memory, publish-subscribe, read-write locking, and RCU-mediated fast paths. It also described three design patterns (impose level of indirection, mark obsolete objects, and retry readers) that allow developers 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 and scalability.

References

[1] ALGLAVE, J., MARANGET, L., MCKENNEY, P. E., PARRI, A., AND STERN, A. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2018), ASPLOS '18, ACM, pp. 405–418.

[2] ARBEL, M., AND ATTIYA, H. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2014), PODC '14, ACM, pp. ???–???

[3] ARBEL, M., AND MORRISON, A. Predicate RCU: An RCU for scalable concurrent updates. *SIGPLAN Not. 50*, 8 (Jan. 2015), 21–30.

[4] 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.

[5] ASH, M. Concurrent memory deallocation in the objective-c runtime. mikeash.com: just this guy, you know?, May 2015.

[6] BAHRA, S. A. ck_epoch: Support per-object destructors. `https://github.com/concurrencykit/ck/commit/10ffb2e6f1737a30e2dcf3862d105ad45fcd60a4`, October 2011.

[7] BAHRA, S. A. ck_hp.c. Hazard pointers: `https://github.com/concurrencykit/ck/blob/master/src/ck_hp.c`, February 2011.

[8] BHAT, S. S. percpu_rwlock: Implement the core design of per-CPU reader-writer locks. `https://patchwork.kernel.org/patch/2157401/`, February 2014.

[9] BONZINI, P., AND DAY, M. RCU implementation for Qemu. `http://lists.gnu.org/archive/html/qemu-devel/2013-08/msg02055.html`, August 2013.

[10] BOSTIC, K. Switch lockless programming style from epoch to hazard references. `https://github.com/wiredtiger/wiredtiger/commit/dddc21014fc494a956778360a14d96c762495e09`, January 2010.

[11] 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.

[12] COMPAQ COMPUTER CORPORATION. Shared memory, threads, interprocess communication. Available: http://h71000.www7.hp.com/wizard/wiz_2637.html, August 2001.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] GWYNNE, D. introduce srp, which according to the manpage i wrote is short for "shared reference pointers". https://github.com/openbsd/src/blob/HEAD/sys/kern/kern_srp.c, July 2015.

[21] 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.

[22] HENNESSY, J. P., OSISEK, D. L., AND SEIGH II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168, Assigned to International Business Machines Corp, Washington, DC, February 1989.

[23] 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.

[24] JACOBSON, V. Avoid read-side locking via delayed free. private communication, September 1993.

[25] JOHN, A. Dynamic vnodes – design and implementation. In *USENIX Winter 1995* (New Orleans, LA, January 1995), USENIX Association, pp. 11–23. Available: https://www.usenix.org/publications/library/proceedings/neworl/full_papers/john.a [Viewed October 1, 2010].

[26] KHIZHINSKY, M. Memory management schemes. https://kukuruku.co/post/lock-free-data-structures-the-inside-memory-management-schemes/, June 2014.

[27] KIM, J., MATHEW, A., KASHYAP, S., RAMANATHAN, M. K., AND MIN, C. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 779–792.

[28] KIVITY, A. rcu: add basic read-copy-update implementation. https://github.com/cloudius-systems/osv/commit/94b69794fb9e6c99d78ca9a58ddaee1c31256b43, August 2013.

[29] KOKOLOGIANNAKIS, M., AND SAGONAS, K. Stateless model checking of the linux kernel's hierarchical read-copy update (Tree RCU). Tech. rep., National Technical University of Athens, January 2017. https://github.com/michalis-/rcu/blob/master/rcupaper.pdf.

[30] KUNG, H. T., AND LEHMAN, P. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems 5*, 3 (September 1980), 354–382.

[31] LIANG, L., MCKENNEY, P. E., KROENING, D., AND MELHAM, T. Verification of the tree-based hierarchical read-copy update in the Linux kernel. Tech. rep., Cornell University Library, October 2016. https://arxiv.org/abs/1610.03052.

[32] 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.

[33] MATVEEV, A., SHAVIT, N., FELBER, P., AND MARLIER, P. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 168–183.

[34] MCKENNEY, P. E. RCU Linux usage. Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html [Viewed January 14, 2007], October 2006.

[35] MCKENNEY, P. E. RCU requirements part 2 — parallelism and software engineering. http://lwn.net/Articles/652677/, August 2015.

[36] MCKENNEY, P. E. Requirements for RCU part 1: the fundamentals. http://lwn.net/Articles/652156/, July 2015.

[37] MCKENNEY, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (2019.12.22a Release)*. kernel.org, Corvallis, OR, USA, 2019.

[38] MCKENNEY, P. E., SARMA, D., AND SONI, M. Scaling dcache with RCU. *Linux Journal 1*, 118 (January 2004), 38–46.

[39] 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.

[40] MICHAEL, M. Rewrite from experimental, use of deterministic schedule, improvements. Hazard pointers: https://github.com/facebook/folly/commit/d42832d2a529156275543c7fa7183e1321df605d, June 2018.

[41] 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.

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

[43] NESTEROV, O., AND ZIJLSTRA, P. rcu: Create rcu_sync infrastructure. https://lore.kernel.org/lkml/20131002150518.675931976@infradead.org/, October 2013.

[44] PODZIMEK, A. Read-copy-update for opensolaris. Master's thesis, Charles University in Prague, 2010. Available: https://andrej.podzimek.org/thesis.pdf [Viewed January 31, 2011].

[45] RAMALHETE, P., AND CORREIA, A. Poor man's rcu. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/poormanurcu-2015.pdf, August 2015.

[46] ROMER, G., AND HUNTER, A. An RAII interface for deferred reclamation. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0561r0.html [Viewed May 29, 2017], February 2017.

[47] ROY, L. rcutorture: Add CBMC-based formal verification for SRCU. URL: https://www.spinics.net/lists/kernel/msg2421833.html, January 2017.

[48] 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.

[49] 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.

[50] SIVARAMAKRISHNAN, K., ZIAREK, L., AND JAGANNATHAN, S. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management* (New York, NY, USA, 2012), ISMM '12, ACM, pp. 49–60.

[51] THE NETBSD FOUNDATION. pserialize(9). http://netbsd.gw.com/cgi-bin/man-cgi?pserialize+9+NetBSD-current, October 2012.

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

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

[54] ZIJLSTRA, P. [PATCH] cgroup: avoid synchronize_sched in __cgroup_procswrite. https://lore.kernel.org/lkml/20160811165413.GA22807@redhat.com/, August 2016.