Listing 9.12: Sequence-Locked Pre-BSD Routing Table Add/Delete (BUGGY!!!)

```
1  int route_add(unsigned long addr, unsigned long interface)
2  {
3    struct route_entry *rep;
4
5    rep = malloc(sizeof(*rep));
6    if (!rep)
7      return -ENOMEM;
8    rep->addr = addr;
9    rep->iface = interface;
10   rep->re_freed = 0;
11   write_seqlock(&sl);
12   rep->re_next = route_list.re_next;
13   route_list.re_next = rep;
14   write_sequnlock(&sl);
15   return 0;
16 }
17
18 int route_del(unsigned long addr)
19 {
20   struct route_entry *rep;
21   struct route_entry **repp;
22
23   write_seqlock(&sl);
24   repp = &route_list.re_next;
25   for (;;) {
26     rep = *repp;
27     if (rep == NULL)
28       break;
29     if (rep->addr == addr) {
30       *repp = rep->re_next;
31       write_sequnlock(&sl);
32       smp_mb();
33       rep->re_freed = 1;
34       free(rep);
35       return 0;
36     }
37     repp = &rep->re_next;
38   }
39   write_sequnlock(&sl);
40   return -ENOENT;
41 }
```

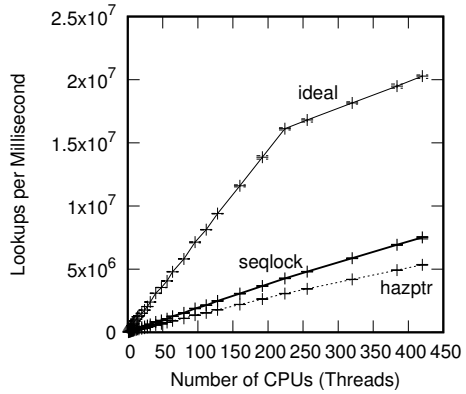

Figure 9.5: Pre-BSD Routing Table Protected by Sequence Locking

Quick Quiz 9.20: Can this bug be fixed? In other words, can you use sequence locks as the *only* synchronization mechanism protecting a linked list supporting concurrent addition, deletion, and lookup? ■

As hinted on page 129, both the read-side and write-side critical sections of a sequence lock can be thought of as transactions, and sequence locking therefore can be thought of as a limited form of transactional memory, which will be discussed in Section 17.2. The limitations of sequence locking are: (1) Sequence locking restricts updates and (2) Sequence locking does not permit traversal of pointers to objects that might be freed by updaters. These limitations are of course overcome by transactional memory, but can also be overcome by combining other synchronization primitives with sequence locking.

Sequence locks allow writers to defer readers, but not vice versa. This can result in unfairness and even starvation in writer-heavy workloads.[3] On the other hand, in the absence of writers, sequence-lock readers are reasonably fast and scale linearly. It is only human to want the best of both worlds: Fast readers without the possibility of read-side failure, let alone starvation. In addition, it would also be nice to overcome sequence locking's limitations with pointers. The following section presents a synchronization mechanism with exactly these properties.

9.5 Read-Copy Update (RCU)

"Free" is a very *good price!*

TOM PETERSON

All of the mechanisms discussed in the preceding sections used one of a number of approaches to defer specific actions until they may be carried out safely. The reference counters discussed in Section 9.2 use explicit counters to defer actions that could disturb readers, which results in read-side contention and thus poor scalability. The hazard pointers covered by Section 9.3 uses implicit counters in the guise of per-thread lists of pointer. This avoids read-side contention, but requires readers to do stores and conditional branches, as well as either full memory barriers in read-side primitives or real-time-unfriendly

[3] Dmitry Vyukov describes one way to reduce (but, sadly, not eliminate) reader starvation: `http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/improved-lock-free-seqlock`.

inter-processor interrupts in update-side primitives.[4] The sequence lock presented in Section 9.4 also avoids read-side contention, but does not protect pointer traversals and, like hazard pointers, requires either full memory barriers in read-side primitives, or inter-processor interrupts in update-side primitives. These schemes' shortcomings raise the question of whether it is possible to do better.

This section introduces *read-copy update* (RCU), which provides an API that allows readers to be associated with regions in the source code, rather than with expensive updates to frequently updated shared data. The remainder of this section examines RCU from a number of different perspectives. Section 9.5.1 provides the classic introduction to RCU, Section 9.5.2 covers fundamental RCU concepts, Section 9.5.3 presents the Linux-kernel API, Section 9.5.4 introduces some common RCU use cases, and finally Section 9.5.5 covers recent work related to RCU.

Although RCU has gained a reputation for being subtle and difficult, when used properly, it is quite straightforward. In fact, no less an authority than Butler Lampson classifies it as easy concurrency [AH22, Chapter 3].

9.5.1 Introduction to RCU

The approaches discussed in the preceding sections have provided good scalability but decidedly non-ideal performance for the Pre-BSD routing table. Therefore, in the spirit of "only those who have gone too far know how far you can go",[5] we will go all the way, looking into algorithms in which concurrent readers might well execute exactly the same sequence of assembly language instructions as would a single-threaded lookup, despite the presence of concurrent updates. Of course, this laudable goal might raise serious implementability questions, but we cannot possibly succeed if we don't even try!

And should we succeed, we will have uncovered yet another of the mysteries set forth on page 129.

9.5.1.1 Minimal Insertion and Deletion

To minimize implementability concerns, we focus on a minimal data structure, which consists of a single global pointer that is either `NULL` or references a single structure. Minimal though it might be, this data structure is heavily used in production [RH18]. A classic approach for insertion is shown in Figure 9.6, which shows four states with time advancing from top to bottom. The first row shows the initial state, with `gptr` equal to `NULL`. In the second row, we have allocated a structure which is uninitialized, as indicated by the question marks. In the third row, we have initialized the structure. Finally, in the fourth and final row, we have updated `gptr` to reference the newly allocated and initialized element.

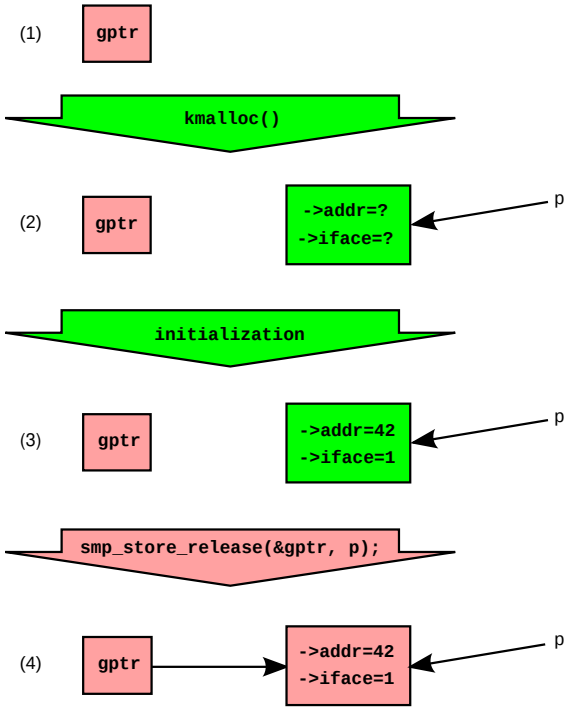

Figure 9.6: Insertion With Concurrent Readers

We might hope that this assignment to `gptr` could use a simple C-language assignment statement. Unfortunately, Section 4.3.4.1 dashes these hopes. Therefore, the updater cannot use a simple C-language assignment, but must instead use `smp_store_release()` as shown in the figure, or, as will be seen, `rcu_assign_pointer()`.

Similarly, one might hope that readers could use a single C-language assignment to fetch the value of `gptr`, and be guaranteed to either get the old value of `NULL` or to get the newly installed pointer, but either way see a valid result. Unfortunately, Section 4.3.4.1 dashes these hopes as well. To obtain this guarantee, readers must instead use `READ_ONCE()`, or, as will be seen, `rcu_dereference()`. However, on most modern computer systems, each of

[4] In some important special cases, this extra work can be avoided by using link counting as exemplified by the UnboundedQueue and ConcurrentHashMap data structures implemented in Folly open-source library (`https://github.com/facebook/folly`).

[5] With apologies to T. S. Eliot.

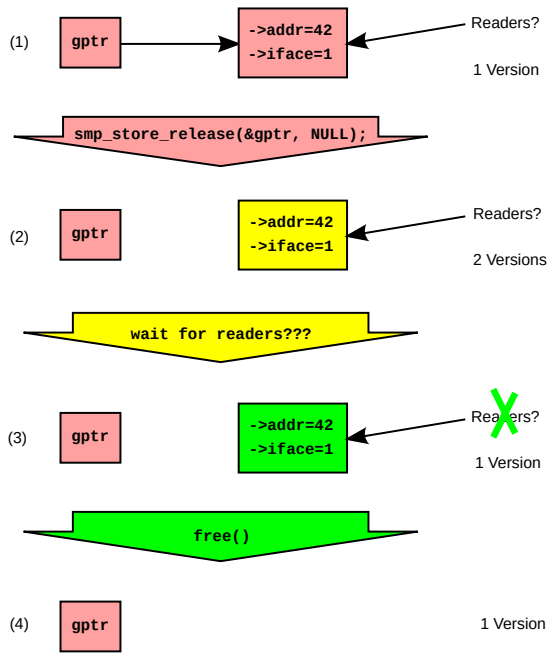

Figure 9.7: Deletion With Concurrent Readers

these read-side primitives can be implemented with a single load instruction, exactly the instruction that would normally be used in single-threaded code.

Reviewing Figure 9.6 from the viewpoint of readers, in the first three states all readers see `gptr` having the value `NULL`. Upon entering the fourth state, some readers might see `gptr` still having the value `NULL` while others might see it referencing the newly inserted element, but after some time, all readers will see this new element. At all times, all readers will see `gptr` as containing a valid pointer. Therefore, it really is possible to add new data to linked data structures while allowing concurrent readers to execute the same sequence of machine instructions that is normally used in single-threaded code. This no-cost approach to concurrent reading provides excellent performance and scalability, and also is eminently suitable for real-time use.

Insertion is of course quite useful, but sooner or later, it will also be necessary to delete data. As can be seen in Figure 9.7, the first step is easy. Again taking the lessons from Section 4.3.4.1 to heart, `smp_store_release()` is used to `NULL` the pointer, thus moving from the first row to the second in the figure. At this point, pre-existing readers see the old structure with `->addr` of 42 and `->iface` of 1, but new readers will see a `NULL` pointer, that is, concurrent readers can disagree on the state, as indicated by the "2 Versions" in the figure.

Quick Quiz 9.21: Why does Figure 9.7 use `smp_store_release()` given that it is storing a `NULL` pointer? Wouldn't `WRITE_ONCE()` work just as well in this case, given that there is no structure initialization to order against the store of the `NULL` pointer? ■

Quick Quiz 9.22: Readers running concurrently with each other and with the procedure outlined in Figure 9.7 can disagree on the value of `gptr`. Isn't that just a wee bit problematic??? ■

We get back to a single version simply by waiting for all the pre-existing readers to complete, as shown in row 3. At that point, all the pre-existing readers are done, and no later reader has a path to the old data item, so there can no longer be any readers referencing it. It may therefore be safely freed, as shown on row 4.

Thus, given a way to wait for pre-existing readers to complete, it is possible to both add data to and remove data from a linked data structure, despite the readers executing the same sequence of machine instructions that would be appropriate for single-threaded execution. So perhaps going all the way was not too far after all!

But how can we tell when all of the pre-existing readers have in fact completed? This question is the topic of Section 9.5.1.3. But first, the next section defines RCU's core API.

9.5.1.2 Core RCU API

The full Linux-kernel API is quite extensive, with more than one hundred API members. However, this section will confine itself to six core RCU API members, which suffices for the upcoming sections introducing RCU and covering its fundamentals. The full API is covered in Section 9.5.3.

Three members of the core APIs are used by readers. The `rcu_read_lock()` and `rcu_read_unlock()` functions delimit RCU read-side critical sections. These may be nested, so that one `rcu_read_lock()`–`rcu_read_unlock()` pair can be enclosed within another. In this case, the nested set of RCU read-side critical sections act as one large critical section covering the full extent of the nested set. The third read-side API member, `rcu_dereference()`, fetches an RCU-protected pointer. Conceptually, `rcu_dereference()` simply loads from memory, but we will see in Section 9.5.2.1 that `rcu_dereference()` must prevent the compiler and (in one

case) the CPU from reordering its load with later memory operations that dereference this pointer.

Quick Quiz 9.23: What is an RCU-protected pointer? ■

The other three members of the core APIs are used by updaters. The `synchronize_rcu()` function implements the "wait for readers" operation from Figure 9.7. The `call_rcu()` function is the asynchronous counterpart of `synchronize_rcu()` by invoking the specified function after all pre-existing RCU readers have completed. Finally, the `rcu_assign_pointer()` macro is used to update an RCU-protected pointer. Conceptually, this is simply an assignment statement, but we will see in Section 9.5.2.1 that `rcu_assign_pointer()` must prevent the compiler and the CPU from reordering this assignment to precede any prior assignments used to initialize the pointed-to structure.

Quick Quiz 9.24: What does `synchronize_rcu()` do if it starts at about the same time as an `rcu_read_lock()`? ■

The core RCU API is summarized in Table 9.1 for easy reference. With that, we are ready to continue this introduction to RCU with the key RCU operation, waiting for readers.

9.5.1.3 Waiting for Readers

It is tempting to base the reader-waiting functionality of `synchronize_rcu()` and `call_rcu()` on a reference counter updated by `rcu_read_lock()` and `rcu_read_unlock()`, but Figure 5.1 in Chapter 5 shows that concurrent reference counting results in extreme overhead. This extreme overhead was confirmed in the specific case of reference counters in Figure 9.2 on page 132. Hazard pointers profoundly reduce this overhead, but, as we saw in Figure 9.3 on page 136, not to zero. Nevertheless, many RCU implementations use counters with carefully controlled cache locality.

A second approach observes that memory synchronization is expensive, and therefore uses registers instead, namely each CPU's or thread's program counter (PC), thus imposing no overhead on readers, at least in the absence of concurrent updates. The updater polls each relevant PC, and if that PC is not within read-side code, then the corresponding CPU or thread is within a quiescent state, in turn signaling the completion of any reader that might have access to the newly removed data element. Once all CPU's or thread's PCs have been observed to be outside of any reader, the grace period has completed. Please note that this approach poses some serious challenges, including memory ordering, functions that are *sometimes* invoked from readers, and ever-exciting code-motion optimizations. Nevertheless, this approach is said to be used in production [Ash15].

A third approach is to simply wait for a fixed period of time that is long enough to comfortably exceed the lifetime of any reasonable reader [Jac93, Joh95]. This can work quite well in hard real-time systems [RLPB18], but in less exotic settings, Murphy says that it is critically important to be prepared even for unreasonably long-lived readers. To see this, consider the consequences of failing do so: A data item will be freed while the unreasonable reader is still referencing it, and that item might well be immediately reallocated, possibly even as a data item of some other type. The unreasonable reader and the unwitting reallocator would then be attempting to use the same memory for two very different purposes. The ensuing mess will be exceedingly difficult to debug.

A fourth approach is to wait forever, secure in the knowledge that doing so will accommodate even the most unreasonable reader. This approach is also called "leaking memory", and has a bad reputation due to the fact that memory leaks often require untimely and inconvenient reboots. Nevertheless, this is a viable strategy when the update rate and the uptime are both sharply bounded. For example, this approach could work well in a high-availability cluster where systems were periodically crashed in order to ensure that cluster really remained highly available.[6] Leaking the memory is also a viable strategy in environments having garbage collectors, in which case the garbage collector can be thought of as plugging the leak [KL80]. However, if your environment lacks a garbage collector, read on!

A fifth approach avoids the period crashes in favor of periodically "stopping the world", as exemplified by the traditional stop-the-world garbage collector. This approach was also heavily used during the decades before ubiquitous connectivity, when it was common practice to power systems off at the end of each working day. However, in today's always-connected always-on world, stopping the world can gravely degrade response times, which has been one motivation for the development of concurrent garbage collectors [BCR03]. Furthermore, although we need all pre-existing readers to complete, we do not need them all to complete at the same time.

[6] The program that forces the periodic crashing is sometimes known as a "chaos monkey": `https://netflix.github.io/chaosmonkey/`. However, it might also be a mistake to neglect chaos caused by systems running for too long.

Table 9.1: Core RCU API

	Primitive	Purpose
Readers	`rcu_read_lock()`	Start an RCU read-side critical section.
	`rcu_read_unlock()`	End an RCU read-side critical section.
	`rcu_dereference()`	Safely load an RCU-protected pointer.
Updaters	`synchronize_rcu()`	Wait for all pre-existing RCU read-side critical sections to complete.
	`call_rcu()`	Invoke the specified function after all pre-existing RCU read-side critical sections complete.
	`rcu_assign_pointer()`	Safely update an RCU-protected pointer.

This observation leads to the sixth approach, which is stopping one CPU or thread at a time. This approach has the advantage of not degrading reader response times at all, let alone gravely. Furthermore, numerous applications already have states (termed *quiescent states*) that can be reached only after all pre-existing readers are done. In transaction-processing systems, the time between a pair of successive transactions might be a quiescent state. In reactive systems, the state between a pair of successive events might be a quiescent state. Within non-preemptive operating-systems kernels, a context switch can be a quiescent state [MS98a]. Either way, once all CPUs and/or threads have passed through a quiescent state, the system is said to have completed a *grace period*, at which point all readers in existence at the start of that grace period are guaranteed to have completed. As a result, it is also guaranteed to be safe to free any removed data items that were removed prior to the start of that grace period.[7]

Within a non-preemptive operating-system kernel, for context switch to be a valid quiescent state, readers must be prohibited from blocking while referencing a given instance data structure obtained via the `gptr` pointer shown in Figures 9.6 and 9.7. This no-blocking constraint is consistent with similar constraints on pure spinlocks, where a CPU is forbidden from blocking while holding a spinlock. Without this constraint, all CPUs might be consumed by threads spinning attempting to acquire a spinlock held by a blocked thread. The spinning threads will not relinquish their CPUs until they acquire the lock, but the thread holding the lock cannot possibly release it until one of the spinning threads relinquishes a CPU. This is a classic deadlock situation, and this deadlock is avoided by forbidding blocking while holding a spinlock.

Again, this same constraint is imposed on reader threads dereferencing `gptr`: Such threads are not allowed to block until after they are done using the pointed-to data item. Returning to the second row of Figure 9.7, where the updater has just completed executing the `smp_store_release()`, imagine that CPU 0 executes a context switch. Because readers are not permitted to block while traversing the linked list, we are guaranteed that all prior readers that might have been running on CPU 0 will have completed. Extending this line of reasoning to the other CPUs, once each CPU has been observed executing a context switch, we are guaranteed that all prior readers have completed, and that there are no longer any reader threads referencing the newly removed data element. The updater can then safely free that data element, resulting in the state shown at the bottom of Figure 9.7.

This approach is termed *quiescent-state-based reclamation* (QSBR) [HMB06]. A QSBR schematic is shown in Figure 9.8, with time advancing from the top of the figure to the bottom. The cyan-colored boxes depict RCU read-side critical sections, each of which begins with `rcu_read_lock()` and ends with `rcu_read_unlock()`. CPU 1 does the `WRITE_ONCE()` that removes the current data item (presumably having previously read the pointer value and availed itself of appropriate synchronization), then waits for readers. This wait operation results in an immediate context switch, which is a quiescent state (denoted by the pink circle), which in turn means that all prior reads on CPU 1 have completed. Next, CPU 2 does a context switch, so that all readers on CPUs 1 and 2 are now known to have completed. Finally, CPU 3 does a context switch. At this point, all readers throughout the

[7] It is possible to do much more with RCU than simply defer reclamation of memory, but deferred reclamation is RCU's most common use case, and is therefore an excellent place to start. For an example of the more general case of deferred execution, please see phased state change in Section 9.5.4.3.

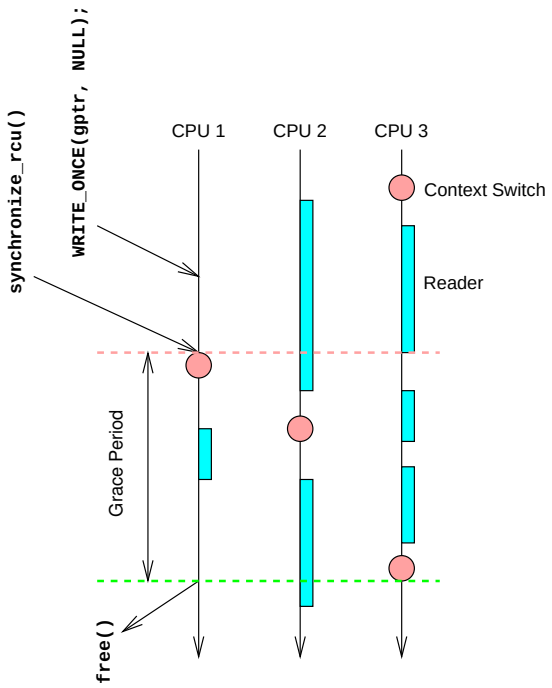

Figure 9.8: QSBR: Waiting for Pre-Existing Readers

entire system are known to have completed, so the grace period ends, permitting `synchronize_rcu()` to return to its caller, in turn permitting CPU 1 to free the old data item.

Quick Quiz 9.25: In Figure 9.8, the last of CPU 3's readers that could possibly have access to the old data item ended before the grace period even started! So why would anyone bother waiting until CPU 3's later context switch??? ■

9.5.1.4 Toy Implementation

Although production-quality QSBR implementations can be quite complex, a toy non-preemptive Linux-kernel implementation is quite simple:

```
1 void synchronize_rcu(void)
2 {
3   int cpu;
4
5   for_each_online_cpu(cpu)
6     sched_setaffinity(current->pid, cpumask_of(cpu));
7 }
```

The `for_each_online_cpu()` primitive iterates over all CPUs, and the `sched_setaffinity()` function causes the current thread to execute on the specified CPU, which forces the destination CPU to execute a context switch. Therefore, once the `for_each_online_cpu()` has completed, each CPU has executed a context switch, which in turn guarantees that all pre-existing reader threads have completed.

Listing 9.13: Insertion and Deletion With Concurrent Readers

```
1  struct route *gptr;
2
3  int access_route(int (*f)(struct route *rp))
4  {
5    int ret = -1;
6    struct route *rp;
7
8    rcu_read_lock();
9    rp = rcu_dereference(gptr);
10   if (rp)
11     ret = f(rp);
12   rcu_read_unlock();
13   return ret;
14 }
15
16 struct route *ins_route(struct route *rp)
17 {
18   struct route *old_rp;
19
20   spin_lock(&route_lock);
21   old_rp = gptr;
22   rcu_assign_pointer(gptr, rp);
23   spin_unlock(&route_lock);
24   return old_rp;
25 }
26
27 int del_route(void)
28 {
29   struct route *old_rp;
30
31   spin_lock(&route_lock);
32   old_rp = gptr;
33   RCU_INIT_POINTER(gptr, NULL);
34   spin_unlock(&route_lock);
35   synchronize_rcu();
36   free(old_rp);
37   return !!old_rp;
38 }
```

Please note that this approach is *not* production quality. Correct handling of a number of corner cases and the need for a number of powerful optimizations mean that production-quality implementations are quite complex. In addition, RCU implementations for preemptible environments require that readers actually do something, which in non-real-time Linux-kernel environments can be as simple as defining `rcu_read_lock()` and `rcu_read_unlock()` as `preempt_disable()` and `preempt_enable()`, respectively.[8] However, this simple non-preemptible approach is conceptually complete, and demonstrates that it really is possible to provide read-side synchronization at zero cost, even in the face of concurrent updates. In fact, Listing 9.13 shows how

[8] Some toy RCU implementations that handle preempted read-side critical sections are shown in Appendix B.

reading (access_route()), Figure 9.6's insertion (ins_route()) and Figure 9.7's deletion (del_route()) can be implemented. (A slightly more capable routing table is shown in Section 9.5.4.1.)

Quick Quiz 9.26: What is the point of rcu_read_lock() and rcu_read_unlock() in Listing 9.13? Why not just let the quiescent states speak for themselves? ■

Quick Quiz 9.27: What is the point of rcu_dereference(), rcu_assign_pointer() and RCU_INIT_POINTER() in Listing 9.13? Why not just use READ_ONCE(), smp_store_release(), and WRITE_ONCE(), respectively? ■

Referring back to Listing 9.13, note that route_lock is used to synchronize between concurrent updaters invoking ins_route() and del_route(). However, this lock is not acquired by readers invoking access_route(): Readers are instead protected by the QSBR techniques described in Section 9.5.1.3.

Note that ins_route() simply returns the old value of gptr, which Figure 9.6 assumed would always be NULL. This means that it is the caller's responsibility to figure out what to do with a non-NULL value, a task complicated by the fact that readers might still be referencing it for an indeterminate period of time. Callers might use one of the following approaches:

1. Use synchronize_rcu() to safely free the pointed-to structure. Although this approach is correct from an RCU perspective, it arguably has software-engineering leaky-API problems.

2. Trip an assertion if the returned pointer is non-NULL.

3. Pass the returned pointer to a later invocation of ins_route() to restore the earlier value.

In contrast, del_route() uses synchronize_rcu() and free() to safely free the newly deleted data item.

Quick Quiz 9.28: But what if the old structure needs to be freed, but the caller of ins_route() cannot block, perhaps due to performance considerations or perhaps because the caller is executing within an RCU read-side critical section? ■

This example shows one general approach to reading and updating RCU-protected data structures, however, there is quite a variety of use cases, several of which are covered in Section 9.5.4.

In summary, it is in fact possible to create concurrent linked data structures that can be traversed by readers executing the same sequence of machine instructions that would be executed by single-threaded readers. The next section summarizes RCU's high-level properties.

9.5.1.5 RCU Properties

A key RCU property is that reads need not wait for updates. This property enables RCU implementations to provide low-cost or even no-cost readers, resulting in low overhead and excellent scalability. This property also allows RCU readers and updaters to make useful concurrent forward progress. In contrast, conventional synchronization primitives must enforce strict mutual exclusion using expensive instructions, thus increasing overhead and degrading scalability, but also typically prohibiting readers and updaters from making useful concurrent forward progress.

Quick Quiz 9.29: Doesn't Section 9.4's seqlock also permit readers and updaters to make useful concurrent forward progress? ■

As noted earlier, RCU delimits readers with rcu_read_lock() and rcu_read_unlock(), and ensures that each reader has a coherent view of each object (see Figure 9.7) by maintaining multiple versions of objects and using update-side primitives such as synchronize_rcu() to ensure that objects are not freed until after the completion of all readers that might be using them. RCU uses rcu_assign_pointer() and rcu_dereference() to provide efficient and scalable mechanisms for publishing and reading new versions of an object, respectively. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast, using replication and weakening optimizations in a manner similar to hazard pointers, but without the need for read-side retries. In some cases, including CONFIG_PREEMPT=n Linux kernels, RCU's read-side primitives have zero overhead.

But are these properties actually useful in practice? This question is taken up by the next section.

9.5.1.6 Practical Applicability

RCU has been used in the Linux kernel since October 2002 [Tor02]. Use of the RCU API has increased substantially since that time, as can be seen in Figure 9.9. RCU has enjoyed heavy use both prior to and since its acceptance in the Linux kernel, as discussed in Section 9.5.5. In short, RCU enjoys wide practical applicability.

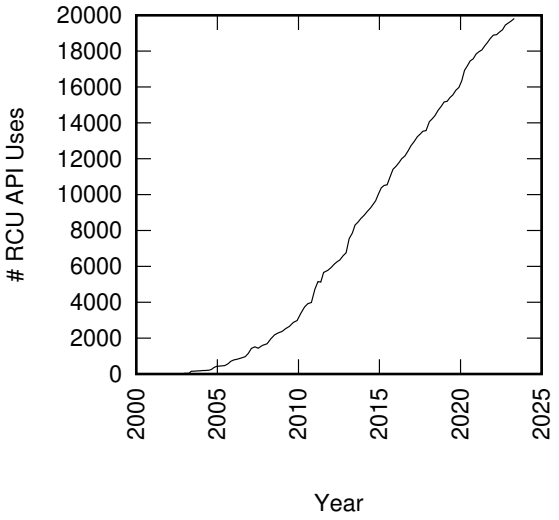

Figure 9.9: RCU Usage in the Linux Kernel

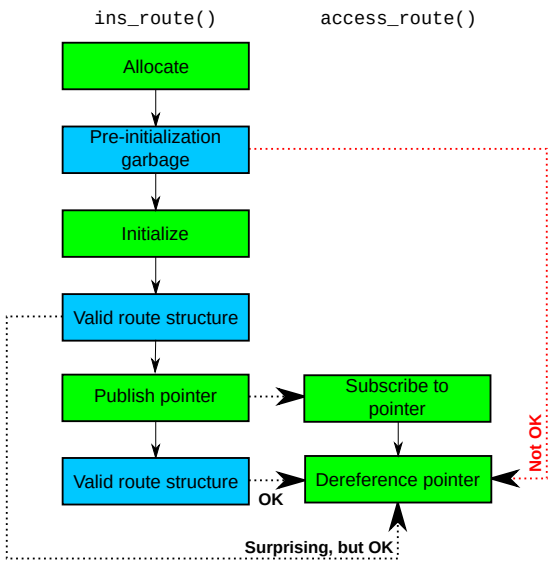

Figure 9.10: Publication/Subscription Constraints

The minimal example discussed in this section is a good introduction to RCU. However, effective use of RCU often requires that you think differently about your problem. It is therefore useful to examine RCU's fundamentals, a task taken up by the following section.

9.5.2 RCU Fundamentals

This section re-examines the ground covered in the previous section, but independent of any particular example or use case. People who prefer to live their lives very close to the actual code may wish to skip the underlying fundamentals presented in this section.

RCU is made up of three fundamental mechanisms, the first being used for insertion, the second being used for deletion, and the third being used to allow readers to tolerate concurrent insertions and deletions. Section 9.5.2.1 describes the publish-subscribe mechanism used for insertion, Section 9.5.2.2 describes how waiting for pre-existing RCU readers enables deletion, and Section 9.5.2.3 discusses how maintaining multiple versions of recently updated objects permits concurrent insertions and deletions. Finally, Section 9.5.2.4 summarizes RCU fundamentals.

9.5.2.1 Publish-Subscribe Mechanism

Because RCU readers are not excluded by RCU updaters, an RCU-protected data structure might change while a reader accesses it. The accessed data item might be moved, removed, or replaced. Because the data structure does not "hold still" for the reader, each reader's access can be thought of as subscribing to the current version of the RCU-protected data item. For their part, updaters can be thought of as publishing new versions.

Unfortunately, as laid out in Section 4.3.4.1 and reiterated in Section 9.5.1.1, it is unwise to use plain accesses for these publication and subscription operations. It is instead necessary to inform both the compiler and the CPU of the need for care, as can be seen from Figure 9.10, which illustrates interactions between concurrent executions of `ins_route()` (and its caller) and `access_route()` from Listing 9.13.

The `ins_route()` column from Figure 9.10 shows `ins_route()`'s caller allocating a new `route` structure, which then contains pre-initialization garbage. The caller then initializes the newly allocated structure, and then invokes `ins_route()` to publish a pointer to the new `route` structure. Publication does not affect the contents of the structure, which therefore remain valid after publication.

The `access_route()` column from this same figure shows the pointer being subscribed to and dereferenced. This dereference operation absolutely must see a valid `route` structure rather than pre-initialization garbage because referencing garbage could result in memory corruption, crashes, and hangs. As noted earlier, avoiding such garbage means that the publish and subscribe operations must inform both the compiler and the CPU of the need to maintain the needed ordering.

Publication is carried out by `rcu_assign_pointer()`, which ensures that `ins_route()`'s caller's initialization is ordered before the actual publication operation's store of the pointer. In addition, `rcu_assign_pointer()` must be atomic in the sense that concurrent readers see either the old value of the pointer or the new value of the pointer, but not some mash-up of these two values. These requirements are met by the C11 store-release operation, and in fact in the Linux kernel, `rcu_assign_pointer()` is defined in terms of `smp_store_release()`, which is similar to C11 store-release.

Note that if concurrent updates are required, some sort of synchronization mechanism will be required to mediate among multiple concurrent `rcu_assign_pointer()` calls on the same pointer. In the Linux kernel, locking is the mechanism of choice, but pretty much any synchronization mechanism may be used. An example of a particularly lightweight synchronization mechanism is Chapter 8's data ownership: If each pointer is owned by a particular thread, then that thread may execute `rcu_assign_pointer()` on that pointer with no additional synchronization overhead.

Quick Quiz 9.30: Wouldn't use of data ownership for RCU updaters mean that the updates could use exactly the same sequence of instructions as would the corresponding single-threaded code? ■

Subscription is carried out by `rcu_dereference()`, which orders the subscription operation's load from the pointer is before the dereference. Similar to `rcu_assign_pointer()`, `rcu_dereference()` must be atomic in the sense that the value loaded must be that from a single store, for example, the compiler must not tear the load.[9] Unfortunately, compiler support for `rcu_dereference()` is at best a work in progress [MWB+17, MRP+17, BM18]. In the meantime, the Linux kernel relies on volatile loads, the details of the various CPU architectures, coding restrictions [McK14e], and, on DEC Alpha [Cor02], a memory-barrier instruction. However, on other architectures, `rcu_dereference()` typically emits a single load instruction, just as would the equivalent single-threaded code. The coding restrictions are described in more detail in Section 15.3.2, however, the common case of field selection ("`->`") works quite well. Software that does not require the ultimate in read-side performance can instead use C11 acquire loads, which provide the needed ordering and more, albeit at a cost. It is hoped that lighter-weight compiler support for `rcu_dereference()` will appear in due course.

In short, use of `rcu_assign_pointer()` for publishing pointers and use of `rcu_dereference()` for subscribing to them successfully avoids the "Not OK" garbage loads depicted in Figure 9.10. These two primitives can therefore be used to add new data to linked structures without disrupting concurrent readers.

Quick Quiz 9.31: But suppose that updaters are adding and removing multiple data items from a linked list while a reader is iterating over that same list. Specifically, suppose that a list initially contains elements A, B, and C, and that an updater removes element A and then adds a new element D at the end of the list. The reader might well see {A, B, C, D}, when that sequence of elements never actually ever existed! In what alternate universe would that qualify as "not disrupting concurrent readers"??? ■

Adding data to a linked structure without disrupting readers is a good thing, as are the cases where this can be done with no added read-side cost compared to single-threaded readers. However, in most cases it is also necessary to remove data, and this is the subject of the next section.

9.5.2.2 Wait For Pre-Existing RCU Readers

In its most basic form, RCU is a way of waiting for things to finish. Of course, there are a great many other ways of waiting for things to finish, including reference counts, reader-writer locks, events, and so on. The great advantage of RCU is that it can wait for each of (say) 20,000 different things without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking.

In RCU's case, each of the things waited on is called an *RCU read-side critical section*. As noted in Table 9.1, an RCU read-side critical section starts with an `rcu_read_lock()` primitive, and ends with a corresponding `rcu_read_unlock()` primitive. RCU read-side critical sections can be nested, and may contain pretty much any code, as long as that code does not contain a quiescent state. For example, within the Linux kernel, it is illegal to sleep within an RCU read-side critical section because a context switch is a quiescent state.[10] If you abide by these conventions, you can use RCU to wait for *any*

[9] That is, the compiler must not break the load into multiple smaller loads, as described under "load tearing" in Section 4.3.4.1.

[10] However, a special form of RCU called SRCU [McK06] does permit general sleeping in SRCU read-side critical sections.

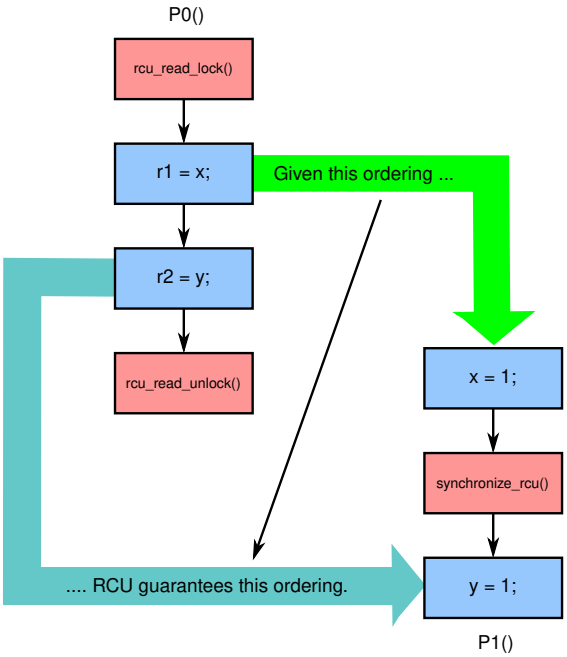

Figure 9.11: RCU Reader and Later Grace Period

pre-existing RCU read-side critical section to complete, and `synchronize_rcu()` uses indirect means to do the actual waiting [DMS+12, McK13].

The relationship between an RCU read-side critical section and a later RCU grace period is an if-then relationship, as illustrated by Figure 9.11. If any portion of a given critical section precedes the beginning of a given grace period, then RCU guarantees that all of that critical section will precede the end of that grace period. In the figure, `P0()`'s access to `x` precedes `P1()`'s access to this same variable, and thus also precedes the grace period generated by `P1()`'s call to `synchronize_rcu()`. It is therefore guaranteed that `P0()`'s access to `y` will precede `P1()`'s access. In this case, if `r1`'s final value is 0, then `r2`'s final value is guaranteed to also be 0.

Quick Quiz 9.32: What other final values of `r1` and `r2` are possible in Figure 9.11? ■

The relationship between an RCU read-side critical section and an earlier RCU grace period is also an if-then relationship, as illustrated by Figure 9.12. If any portion of a given critical section follows the end of a given grace period, then RCU guarantees that all of that critical section will follow the beginning of that grace period. In the figure, `P0()`'s access to `y` follows `P1()`'s access

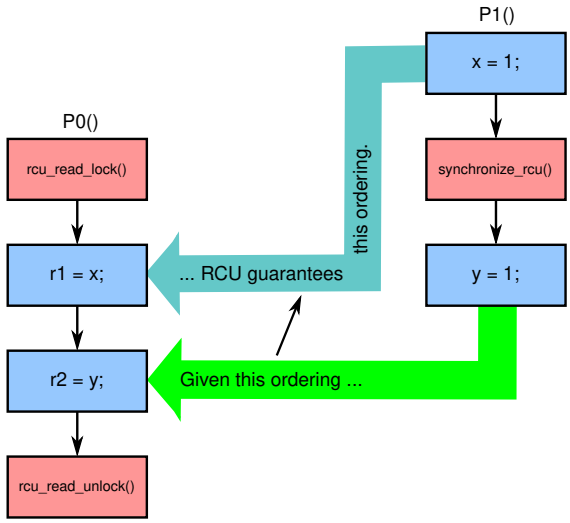

Figure 9.12: RCU Reader and Earlier Grace Period

to this same variable, and thus follows the grace period generated by `P1()`'s call to `synchronize_rcu()`. It is therefore guaranteed that `P0()`'s access to `x` will follow `P1()`'s access. In this case, if `r2`'s final value is 1, then `r1`'s final value is guaranteed to also be 1.

Quick Quiz 9.33: What would happen if the order of `P0()`'s two accesses was reversed in Figure 9.12? ■

Finally, as shown in Figure 9.13, an RCU read-side critical section can be completely overlapped by an RCU grace period. In this case, `r1`'s final value is 1 and `r2`'s final value is 0.

However, it cannot be the case that `r1`'s final value is 0 and `r2`'s final value is 1. This would mean that an RCU read-side critical section had completely overlapped a grace period, which is forbidden (or at the very least constitutes a bug in RCU). RCU's wait-for-readers guarantee therefore has two parts: (1) If any part of a given RCU read-side critical section precedes the beginning of a given grace period, then the entirety of that critical section precedes the end of that grace period. (2) If any part of a given RCU read-side critical section follows the end of a given grace period, then the entirety of that critical section follows the beginning of that grace period. This definition is sufficient for almost all RCU-based algorithms, but for those wanting more, simple executable formal models of RCU are available as part of Linux kernel v4.17 and later, as discussed in Section 12.3.2. In addition, RCU's ordering properties are examined in much greater detail in Section 15.4.3.

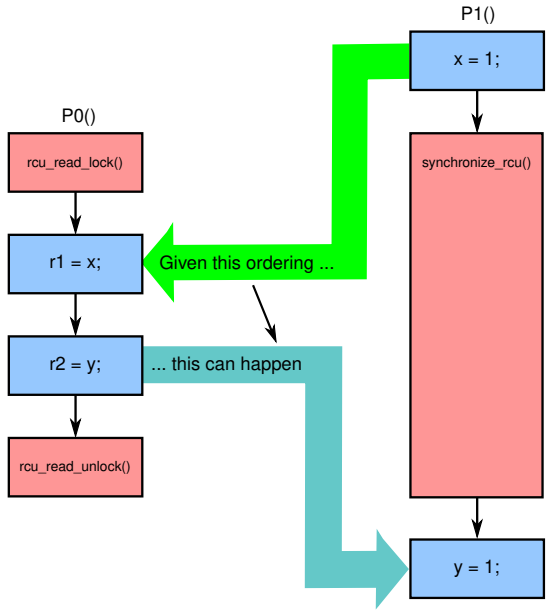

Figure 9.13: RCU Reader Within Grace Period

Quick Quiz 9.34: What would happen if `P0()`'s accesses in Figures 9.11–9.13 were stores? ■

Although RCU's wait-for-readers capability really is sometimes used to order the assignment of values to variables as shown in Figures 9.11–9.13, it is more frequently used to safely free data elements removed from a linked structure, as was done in Section 9.5.1. The general process is illustrated by the following pseudocode:

1. Make a change, for example, remove an element from a linked list.
2. Wait for all pre-existing RCU read-side critical sections to completely finish (for example, by using `synchronize_rcu()`).
3. Clean up, for example, free the element that was replaced above.

This more abstract procedure requires a more abstract diagram than Figures 9.11–9.13, which are specific to a particular litmus test. After all, an RCU implementation must work correctly regardless of the form of the RCU updates and the RCU read-side critical sections. Figure 9.14 fills this need, showing the four possible scenarios, with time advancing from top to bottom within each scenario. Within each scenario, an RCU reader is represented by

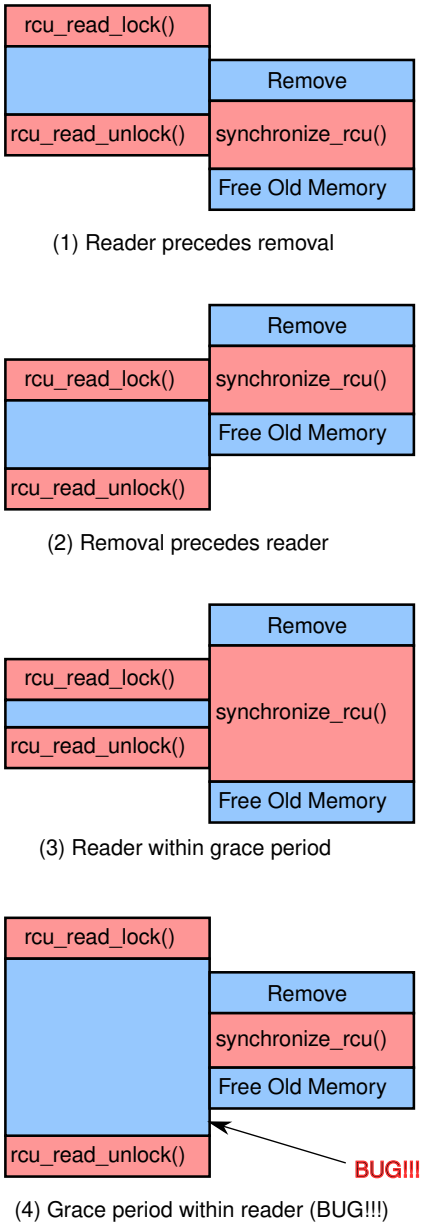

Figure 9.14: Summary of RCU Grace-Period Ordering Guarantees

the left-hand stack of boxes and RCU updater by the right-hand stack.

In the first scenario, the reader starts execution before the updater starts the removal, so it is possible that this reader has a reference to the removed data element. Therefore, the updater must not free this element until after the reader completes. In the second scenario, the reader does not start execution until after the removal has completed. The reader cannot possibly obtain a reference to the already-removed data element, so this element may be freed before the reader completes. The third scenario is like the second, but illustrates that even when the reader cannot possibly obtain a reference to an element, it is still permissible to defer the freeing of that element until after the reader completes. In the fourth and final scenario, the reader starts execution before the updater starts removing the data element, but this element is (incorrectly) freed before the reader completed. A correct RCU implementation will not allow this fourth scenario to occur. This diagram thus illustrates RCU's wait-for-readers functionality: Given a grace period, each reader ends before the end of that grace period, starts after the beginning of that grace period, or both, in which case it is wholly contained within that grace period.

Because RCU readers can make forward progress while updates are in progress, different readers might disagree about the state of the data structure, a topic taken up by the next section.

9.5.2.3 Maintain Multiple Versions of Recently Updated Objects

This section discusses how RCU accommodates synchronization-free readers by maintaining multiple versions of data. Because these synchronization-free readers provide very weak temporal synchronization, RCU users compensate via spatial synchronization. Spatial synchronization was discussed in Chapter 6, and is heavily used in practice to obtain good performance and scalability. In this section, spatial synchronization will be used to attain a weak (but useful) form of correctness as well as excellent performance and scalability.

Figure 9.7 in Section 9.5.1.1 showed a simple variant of spatial synchronization, in which different readers running concurrently with `del_route()` (see Listing 9.13) might see the old `route` structure or an empty list, but either way get a valid result. Of course, a closer look at Figure 9.6 shows that calls to `ins_route()` can also result in concurrent readers seeing different versions: Either the initial empty list or the newly inserted `route` structure.

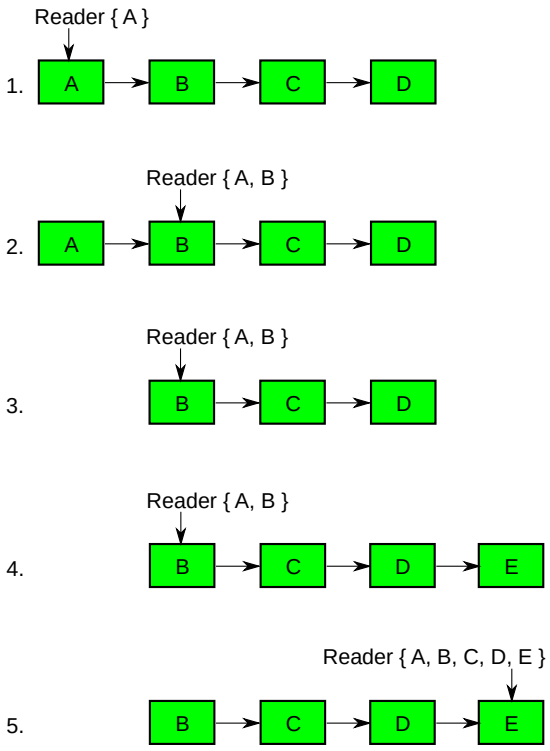

Figure 9.15: Multiple RCU Data-Structure Versions

Note that both reference counting (Section 9.2) and hazard pointers (Section 9.3) can also cause concurrent readers to see different versions, but RCU's lightweight readers make this more likely.

However, maintaining multiple weakly consistent versions can provide some surprises. For example, consider Figure 9.15, in which a reader is traversing a linked list that is concurrently updated.[11] In the first row of the figure, the reader is referencing data item A, and in the second row, it advances to B, having thus far seen A followed by B. In the third row, an updater removes element A and in the fourth row an updater adds element E to the end of the list. In the fifth and final row, the reader completes its traversal, having seeing elements A through E.

Except that there was no time at which such a list existed. This situation might be even more surprising than that shown in Figure 9.7, in which different concurrent readers see different versions. In contrast, in Figure 9.15 the reader sees a version that never actually existed!

[11] RCU linked-list APIs may be found in Section 9.5.3.

One way to resolve this strange situation is via weaker semanitics. A reader traversal must encounter any data item that was present during the full traversal (B, C, and D), and might or might not encounter data items that were present for only part of the traversal (A and E). Therefore, in this particular case, it is perfectly legitimate for the reader traversal to encounter all five elements. If this outcome is problematic, another way to resolve this situation is through use of stronger synchronization mechanisms, such as reader-writer locking, or clever use of timestamps and versioning, as discussed in Section 9.5.4.11. Of course, stronger mechanisms will be more expensive, but then again the engineering life is all about choices and tradeoffs.

Strange though this situation might seem, it is entirely consistent with the real world. As we saw in Section 3.2, the finite speed of light cannot be ignored within a computer system, and it most certainly cannot be ignored outside of this system. This in turn means that any data within the system representing state in the real world outside of the system is always and forever outdated, and thus inconsistent with the real world. Therefore, it is quite possible that the sequence {A, B, C, D, E} occurred in the real world, but due to speed-of-light delays was never represented in the computer system's memory. In this case, the reader's surprising traversal would correctly reflect reality.

As a result, algorithms operating on real-world data must account for inconsistent data, either by tolerating inconsistencies or by taking steps to exclude or reject them. In many cases, these algorithms are also perfectly capable of dealing with inconsistencies within the system.

The pre-BSD packet routing example laid out in Section 9.1 is a case in point. The contents of a routing list is set by routing protocols, and these protocols feature significant delays (seconds or even minutes) to avoid routing instabilities. Therefore, once a routing update reaches a given system, it might well have been sending packets the wrong way for quite some time. Sending a few more packets the wrong way for the few microseconds during which the update is in flight is clearly not a problem because the same higher-level protocol actions that deal with delayed routing updates will also deal with internal inconsistencies.

Nor is Internet routing the only situation tolerating inconsistencies. To repeat, any algorithm in which data within a system tracks outside-of-system state must tolerate inconsistencies, which includes security policies (often set by committees of humans), storage configuration, and WiFi access points, to say nothing of removable hardware such as microphones, headsets, cameras, mice, printers, and much else besides. Furthermore, the large number of Linux-kernel RCU API uses shown in Figure 9.9, combined with the Linux kernel's heavy use of reference counting and with increasing use of hazard pointers in other projects, demonstrates that tolerance for such inconsistencies is more common than one might imagine.

One root cause of this common-case tolerance of inconsistencies is that single-item lookups are much more common in practice than are full-data-structure traversals. After all, full-data-structure traversals are much more expensive than single-item lookups, so developers are motivated to avoid such traversals. Not only are concurrent updates less likely to affect a single-item lookup than they are a full traversal, but it is also the case that an isolated single-item lookup has no way of detecting such inconsistencies. As a result, in the common case, such inconsistencies are not just tolerable, they are in fact invisible.

In such cases, RCU readers can be considered to be fully ordered with updaters, despite the fact that these readers might be executing the exact same sequence of machine instructions that would be executed by a single-threaded program, as hinted on page 129. For example, referring back to Listing 9.13 on page 144, suppose that each reader thread invokes `access_route()` exactly once during its lifetime, and that there is no other communication among reader and updater threads. Then each invocation of `access_route()` can be ordered after the `ins_route()` invocation that produced the `route` structure accessed by line 11 of the listing in `access_route()` and ordered before any subsequent `ins_route()` or `del_route()` invocation.

In summary, maintaining multiple versions is exactly what enables the extremely low overheads of RCU readers, and as noted earlier, many algorithms are unfazed by multiple versions. However, there are algorithms that absolutely cannot handle multiple versions. There are techniques for adapting such algorithms to RCU [McK04], for example, the use of sequence locking described in Section 13.4.2.

Exercises These examples assumed that a mutex was held across the entire update operation, which would mean that there could be at most two versions of the list active at a given time.

Quick Quiz 9.35: How would you modify the deletion example to permit more than two versions of the list to be active? ■

Quick Quiz 9.36: How many RCU versions of a given list can be active at any given time? ■

Quick Quiz 9.37: How can the per-update overhead of RCU be reduced? ■

9.5.2.4 Summary of RCU Fundamentals

This section has described the three fundamental components of RCU-based algorithms:

1. A publish-subscribe mechanism for adding new data featuring `rcu_assign_pointer()` for update-side publication and `rcu_dereference()` for read-side subscription,
2. A way of waiting for pre-existing RCU readers to finish based on readers being delimited by `rcu_read_lock()` and `rcu_read_unlock()` on the one hand and updaters waiting via `synchronize_rcu()` or `call_rcu()` on the other (see Section 15.4.3 for a formal description), and
3. A discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

Quick Quiz 9.38: How can RCU updaters possibly delay RCU readers, given that neither `rcu_read_lock()` nor `rcu_read_unlock()` spin or block? ■

These three RCU components allow data to be updated in the face of concurrent readers that might be executing the same sequence of machine instructions that would be used by a reader in a single-threaded implementation. These RCU components can be combined in different ways to implement a surprising variety of different types of RCU-based algorithms, a number of which are presented in Section 9.5.4. However, it is usually better to work at higher levels of abstraction. To this end, the next section describes the Linux-kernel API, which includes simple data structures such as lists.

9.5.3 RCU Linux-Kernel API

This section looks at RCU from the viewpoint of its Linux-kernel API.[12] Section 9.5.3.2 presents RCU's wait-to-finish APIs, Section 9.5.3.3 presents RCU's publish-subscribe and version-maintenance APIs, Section 9.5.3.4 presents RCU's list-processing APIs, Section 9.5.3.5 presents RCU's diagnostic APIs, and Section 9.5.3.6 describes in which contexts RCU's various APIs may be used. Finally, Section 9.5.3.7 presents concluding remarks.

Readers who are not excited about kernel internals may wish to skip ahead to Section 9.5.4 on page 162, but preferably after reviewing the next section covering software-engineering considerations.

9.5.3.1 RCU API and Software Engineering

Readers who have looked ahead to Tables 9.2, 9.3, 9.4, and 9.5 might have noted that the full list of Linux-kernel APIs sports more than 100 members. This is in sharp (and perhaps dismaying) contrast to the mere six API members shown in Table 9.1. This situation clearly raises the question "Why so many???"

This question is answered more thoroughly in the following sections, but in the meantime the rest of this section summarizes the motivations.

There is a wise old saying to the effect of "To err is human." This means that purpose of a significant fraction of the RCU API is to provide diagnostics, most notably in Table 9.5, but elsewhere as well.

Important causes of human error are the limits of the human brain, for example, the limited capacity of short-term memory. The toy examples shown in this book do not stress these limits. This is out of necessity: Many readers push their cognitive limits while learning new material, so the examples need to be kept simple.

These examples therefore keep `rcu_dereference()` invocations in the same function as the enclosing `rcu_read_lock()` and `rcu_read_unlock()` calls. In contrast, real-world software must frequently invoke these API members from different functions, and even from different translation units. The Linux kernel RCU API has therefore expanded to accommodate lockdep, which allows `rcu_dereference()` and friends to complain if it is not protected by `rcu_read_lock()`. Linux-kernel RCU also checks for some double-free errors, infinite loops in RCU read-side critical sections, and attempts to invoke quiescent states within RCU read-side critical sections.

Another way that real-world software accommodates the limits of human cognition is through abstraction. The Linux-kernel API therefore includes members that operate on lists in addition to the pointer-oriented core API of

[12] Userspace RCU's API is documented elsewhere [MDJ13f].