

What is RCU, Fundamentally?

[Editor's note: this is the first in a three-part series on how the read-copy-update mechanism works. Many thanks to Paul McKenney and Jonathan Walpole for allowing us to publish these articles. The remaining two sections will appear in future weeks.]

December 17, 2007

This article was contributed by
Paul McKenney

Part 1 of 3 of **What is RCU, Really?**

Paul E. McKenney, IBM Linux Technology Center

Jonathan Walpole, Portland State University Department of Computer Science

Introduction

Read-copy update (RCU) is a synchronization mechanism that was added to the Linux kernel in October of 2002. RCU achieves scalability improvements by allowing reads to occur concurrently with updates. In contrast with conventional locking primitives that ensure mutual exclusion among concurrent threads regardless of whether they be readers or updaters, or with reader-writer locks that allow concurrent reads but not in the presence of updates, RCU supports concurrency between a single updater and multiple readers. RCU ensures that reads are coherent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete. RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for deferring the collection of old versions. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast. In some cases (non-preemptable kernels), RCU's read-side primitives have zero overhead.

Quick Quiz 1: But doesn't seqlock also permit readers and updaters to get work done concurrently?

This leads to the question "what exactly is RCU?", and perhaps also to the question "how can RCU *possibly* work?" (or, not infrequently, the assertion that RCU cannot possibly work). This document addresses these questions from a fundamental viewpoint; later installments look at them from usage and from API viewpoints. This last installment also includes a list of references.

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. These mechanisms are described in the following sections, which focus on applying RCU to linked lists:

1. [Publish-Subscribe Mechanism](#) (for insertion)
2. [Wait For Pre-Existing RCU Readers to Complete](#) (for deletion)
3. [Maintain Multiple Versions of Recently Updated Objects](#) (for readers)

These sections are followed by [concluding remarks](#) and the [answers to the Quick Quizzes](#).

Publish-Subscribe Mechanism

One key attribute of RCU is the ability to safely scan data, even though that data is being modified concurrently. To provide this ability for concurrent insertion, RCU uses what can be thought of as a publish-subscribe mechanism. For example, consider an initially NULL global pointer `gp` that is to be modified to point to a newly allocated and initialized data structure. The following code fragment (with the addition of appropriate locking) might be used for this purpose:

```

1 struct foo {
2     int a;
3     int b;
4     int c;
5 };
6 struct foo *gp = NULL;
7
8 /* . . . */
9
10 p = kmalloc(sizeof(*p), GFP_KERNEL);
11 p->a = 1;
12 p->b = 2;
13 p->c = 3;
14 gp = p;

```

Unfortunately, there is nothing forcing the compiler and CPU to execute the last four assignment statements in order. If the assignment to `gp` happens before the initialization of `p`'s fields, then concurrent readers could see the uninitialized values. Memory barriers are required to keep things ordered, but memory barriers are notoriously difficult to use. We therefore encapsulate them into a primitive `rcu_assign_pointer()` that has publication semantics. The last four lines would then be as follows:

```

1 p->a = 1;
2 p->b = 2;
3 p->c = 3;
4 rcu_assign_pointer(gp, p);

```

The `rcu_assign_pointer()` would *publish* the new structure, forcing both the compiler and the CPU to execute the assignment to `gp` *after* the assignments to the fields referenced by `p`.

However, it is not sufficient to only enforce ordering at the updater, as the reader must enforce proper ordering as well. Consider for example the following code fragment:

```

1 p = gp;
2 if (p != NULL) {
3     do_something_with(p->a, p->b, p->c);
4 }

```

Although this code fragment might well seem immune to misordering, unfortunately, the [DEC Alpha CPU \[PDF\]](#) and value-speculation compiler optimizations can, believe it or not, cause the values of `p->a`, `p->b`, and `p->c` to be fetched before the value of `p`! This is perhaps easiest to see in the case of value-speculation compiler optimizations, where the compiler guesses the value of `p`, fetches `p->a`, `p->b`, and `p->c`, then fetches the actual value of `p` in order to check whether its guess was correct. This sort of optimization is quite aggressive, perhaps insanely so, but does actually occur in the context of profile-driven optimization.

Clearly, we need to prevent this sort of skullduggery on the part of both the compiler and the CPU. The `rcu_dereference()` primitive uses whatever memory-barrier instructions and compiler directives are required for this purpose:

```

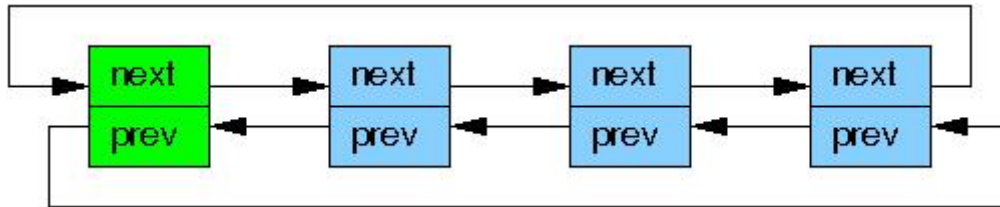
1 rcu_read_lock();
2 p = rcu_dereference(gp);
3 if (p != NULL) {
4     do_something_with(p->a, p->b, p->c);
5 }
6 rcu_read_unlock();

```

The `rcu_dereference()` primitive can thus be thought of as *subscribing* to a given value of the specified pointer, guaranteeing that subsequent dereference operations will see any initialization that occurred before the corresponding publish (`rcu_assign_pointer()`) operation. The `rcu_read_lock()` and `rcu_read_unlock()` calls are absolutely required: they define the extent of the RCU read-side critical section. Their purpose is explained in the [next section](#), however,

they never spin or block, nor do they prevent the `list_add_rcu()` from executing concurrently. In fact, in non-CONFIG_PREEMPT kernels, they generate absolutely no code.

Although `rcu_assign_pointer()` and `rcu_dereference()` can in theory be used to construct any conceivable RCU-protected data structure, in practice it is often better to use higher-level constructs. Therefore, the `rcu_assign_pointer()` and `rcu_dereference()` primitives have been embedded in special RCU variants of Linux's list-manipulation API. Linux has two variants of doubly linked list, the circular struct `list_head` and the linear struct `hlist_head`/struct `hlist_node` pair. The former is laid out as follows, where the green boxes represent the list header and the blue boxes represent the elements in the list.



Adapting the pointer-publish example for the linked list gives the following:

```

1 struct foo {
2     struct list_head list;
3     int a;
4     int b;
5     int c;
6 };
7 LIST_HEAD(head);
8
9 /* . . . */
10
11 p = kmalloc(sizeof(*p), GFP_KERNEL);
12 p->a = 1;
13 p->b = 2;
14 p->c = 3;
15 list_add_rcu(&p->list, &head);

```

Line 15 must be protected by some synchronization mechanism (most commonly some sort of lock) to prevent multiple `list_add()` instances from executing concurrently. However, such synchronization does not prevent this `list_add()` from executing concurrently with RCU readers.

Subscribing to an RCU-protected list is straightforward:

```

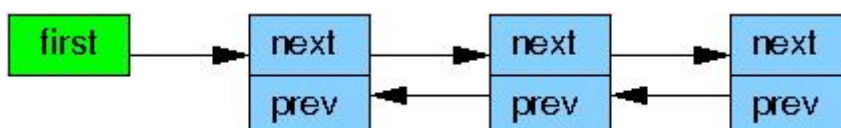
1 rcu_read_lock();
2 list_for_each_entry_rcu(p, head, list) {
3     do_something_with(p->a, p->b, p->c);
4 }
5 rcu_read_unlock();

```

The `list_add_rcu()` primitive publishes an entry into the specified list, guaranteeing that the corresponding `list_for_each_entry_rcu()` invocation will properly subscribe to this same entry.

Quick Quiz 2: What prevents the `list_for_each_entry_rcu()` from getting a segfault if it happens to execute at exactly the same time as the `list_add_rcu()`?

Linux's other doubly linked list, the `hlist`, is a linear list, which means that it needs only one pointer for the header rather than the two required for the circular list. Thus, use of `hlist` can halve the memory consumption for the hash-bucket arrays of large hash tables.



Publishing a new element to an RCU-protected hlist is quite similar to doing so for the circular list:

```
1 struct foo {
2     struct hlist_node *list;
3     int a;
4     int b;
5     int c;
6 };
7 HLIST_HEAD(head);
8
9 /* . . . */
10
11 p = kmalloc(sizeof(*p), GFP_KERNEL);
12 p->a = 1;
13 p->b = 2;
14 p->c = 3;
15 hlist_add_head_rcu(&p->list, &head);
```

As before, line 15 must be protected by some sort of synchronization mechanism, for example, a lock.

Subscribing to an RCU-protected hlist is also similar to the circular list:

```
1 rcu_read_lock();
2 hlist_for_each_entry_rcu(p, q, head, list) {
3     do_something_with(p->a, p->b, p->c);
4 }
5 rcu_read_unlock();
```

Quick Quiz 3: Why do we need to pass two pointers into `hlist_for_each_entry_rcu()` when only one is needed for `list_for_each_entry_rcu()`?

The set of RCU publish and subscribe primitives are shown in the following table, along with additional primitives to "unpublish", or retract:

Category	Publish	Retract	Subscribe
Pointers	<code>rcu_assign_pointer()</code>	<code>rcu_assign_pointer(..., NULL)</code>	<code>rcu_dereference()</code>
Lists	<code>list_add_rcu()</code> <code>list_add_tail_rcu()</code> <code>list_replace_rcu()</code>	<code>list_del_rcu()</code>	<code>list_for_each_entry_rcu()</code>
Hlists	<code>hlist_add_after_rcu()</code> <code>hlist_add_before_rcu()</code> <code>hlist_add_head_rcu()</code> <code>hlist_replace_rcu()</code>	<code>hlist_del_rcu()</code>	<code>hlist_for_each_entry_rcu()</code>

Note that the `list_replace_rcu()`, `list_del_rcu()`, `hlist_replace_rcu()`, and `hlist_del_rcu()` APIs add a complication. When is it safe to free up the data element that was replaced or removed? In particular, how can we possibly know when all the readers have released their references to that data element?

These questions are addressed in the following section.

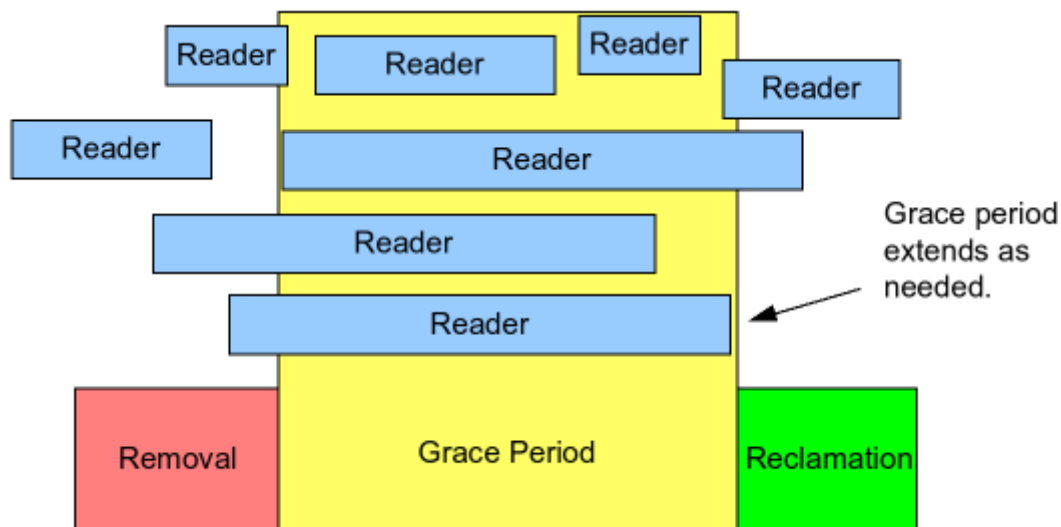
Wait For Pre-Existing RCU Readers to Complete

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, the things waited on are called "RCU read-side critical sections". 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 explicitly block or sleep (although a special form of RCU called "SRCU" does permit general sleeping in SRCU read-side critical sections). If you abide by these conventions, you can use RCU to wait for *any* desired piece of code to complete.

RCU accomplishes this feat by indirectly determining when these other things have finished, as has been described elsewhere for RCU Classic and realtime RCU.

In particular, as shown in the following figure, RCU is a way of waiting for pre-existing RCU read-side critical sections to completely finish, including memory operations executed by those critical sections.



However, note that RCU read-side critical sections that begin after the beginning of a given grace period can and will extend beyond the end of that grace period.

The following pseudocode shows the basic form of algorithms that use RCU to wait for readers:

1. Make a change, for example, replace an element in a linked list.
2. Wait for all pre-existing RCU read-side critical sections to completely finish (for example, by using the `synchronize_rcu()` primitive). The key observation here is that subsequent RCU read-side critical sections have no way to gain a reference to the newly removed element.
3. Clean up, for example, free the element that was replaced above.

The following code fragment, adapted from those in the previous section, demonstrates this process, with field `a` being the search key:

```

1 struct foo {
2     struct list_head list;
3     int a;
4     int b;
5     int c;
6 };
7 LIST_HEAD(head);
8
9 /* . . . */
10
11 p = search(head, key);
12 if (p == NULL) {
13     /* Take appropriate action, unlock, and return. */
14 }

```

```

15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);

```

Lines 19, 20, and 21 implement the three steps called out above. Lines 16-19 gives RCU ("read-copy update") its name: while permitting concurrent *reads*, line 16 *copies* and lines 17-19 do an *update*.

The `synchronize_rcu()` primitive might seem a bit mysterious at first. After all, it must wait for all RCU read-side critical sections to complete, and, as we saw earlier, the `rcu_read_lock()` and `rcu_read_unlock()` primitives that delimit RCU read-side critical sections don't even generate any code in non-CONFIG_PREEMPT kernels!

There is a trick, and the trick is that RCU Classic read-side critical sections delimited by `rcu_read_lock()` and `rcu_read_unlock()` are not permitted to block or sleep. Therefore, when a given CPU executes a context switch, we are guaranteed that any prior RCU read-side critical sections will have completed. This means that as soon as each CPU has executed at least one context switch, *all* prior RCU read-side critical sections are guaranteed to have completed, meaning that `synchronize_rcu()` can safely return.

Thus, RCU Classic's `synchronize_rcu()` can conceptually be as simple as the following:

```

1 for_each_online_cpu(cpu)
2   run_on(cpu);

```

Here, `run_on()` switches the current thread to the specified CPU, which forces a context switch on that CPU. The `for_each_online_cpu()` loop therefore forces a context switch on each CPU, thereby guaranteeing that all prior RCU read-side critical sections have completed, as required. Although this simple approach works for kernels in which preemption is disabled across RCU read-side critical sections, in other words, for non-CONFIG_PREEMPT and CONFIG_PREEMPT kernels, it does *not* work for CONFIG_PREEMPT_RT realtime (-rt) kernels. Therefore, realtime RCU uses a different approach based loosely on reference counters.

Of course, the actual implementation in the Linux kernel is much more complex, as it is required to handle interrupts, NMIs, CPU hotplug, and other hazards of production-capable kernels, but while also maintaining good performance and scalability. Realtime implementations of RCU must additionally help provide good realtime response, which rules out implementations (like the simple two-liner above) that rely on disabling preemption.

Although it is good to know that there is a simple conceptual implementation of `synchronize_rcu()`, other questions remain. For example, what exactly do RCU readers see when traversing a concurrently updated list? This question is addressed in the following section.

Maintain Multiple Versions of Recently Updated Objects

This section demonstrates how RCU maintains multiple versions of lists to accommodate synchronization-free readers. Two examples are presented showing how an element that might be referenced by a given reader must remain intact while that reader remains in its RCU read-side critical section. The first example demonstrates deletion of a list element, and the second example demonstrates replacement of an element.

Example 1: Maintaining Multiple Versions During Deletion

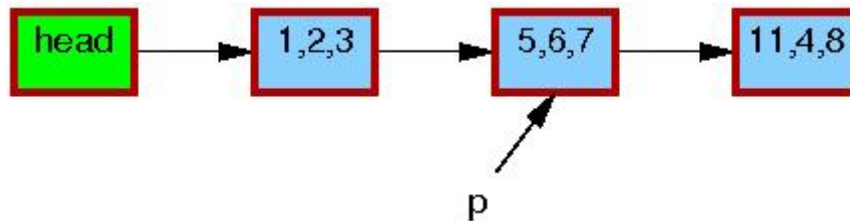
To start the "deletion" example, we will modify lines 11-21 in the example in the previous section as follows:

```

1 p = search(head, key);
2 if (p != NULL) {
3     list_del_rcu(&p->list);
4     synchronize_rcu();
5     kfree(p);
6 }

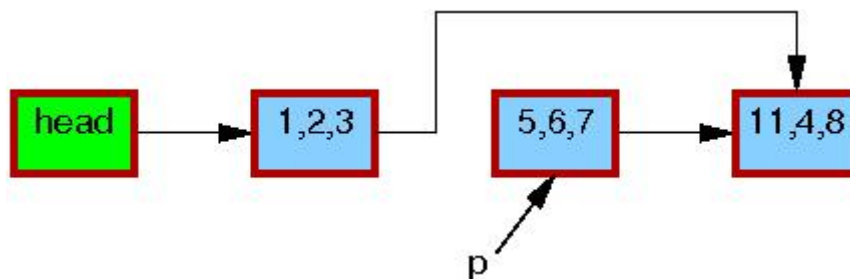
```

The initial state of the list, including the pointer p , is as follows.

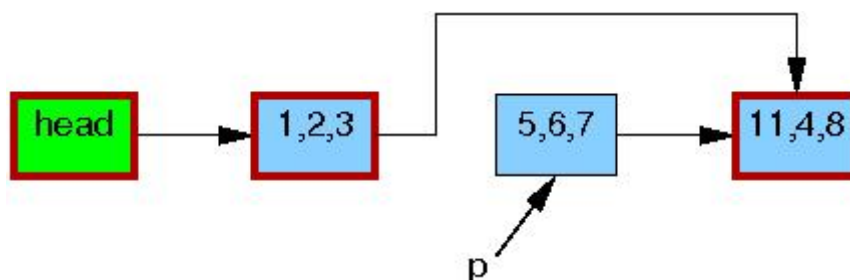


The triples in each element represent the values of fields a , b , and c , respectively. The red borders on each element indicate that readers might be holding references to them, and because readers do not synchronize directly with updaters, readers might run concurrently with this entire replacement process. Please note that we have omitted the backwards pointers and the link from the tail of the list to the head for clarity.

After the `list_del_rcu()` on line 3 has completed, the $5,6,7$ element has been removed from the list, as shown below. Since readers do not synchronize directly with updaters, readers might be concurrently scanning this list. These concurrent readers might or might not see the newly removed element, depending on timing. However, readers that were delayed (e.g., due to interrupts, ECC memory errors, or, in `CONFIG_PREEMPT_RT` kernels, preemption) just after fetching a pointer to the newly removed element might see the old version of the list for quite some time after the removal. Therefore, we now have two versions of the list, one with element $5,6,7$ and one without. The border of the $5,6,7$ element is still red, indicating that readers might be referencing it.



Please note that readers are not permitted to maintain references to element $5,6,7$ after exiting from their RCU read-side critical sections. Therefore, once the `synchronize_rcu()` on line 4 completes, so that all pre-existing readers are guaranteed to have completed, there can be no more readers referencing this element, as indicated by its black border below. We are thus back to a single version of the list.



At this point, the $5,6,7$ element may safely be freed, as shown below:



At this point, we have completed the deletion of element 5,6,7. The following section covers replacement.

Example 2: Maintaining Multiple Versions During Replacement

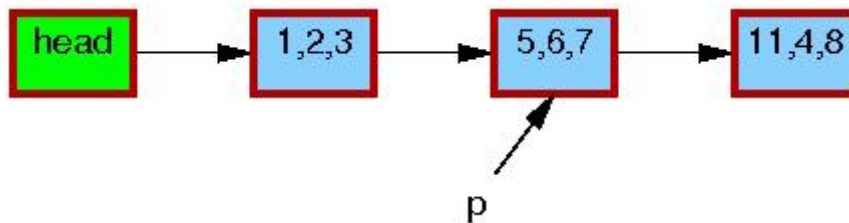
To start the replacement example, here are the last few lines of the [example in the previous section](#):

```

1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;
3 q->b = 2;
4 q->c = 3;
5 list_replace_rcu(&p->list, &q->list);
6 synchronize_rcu();
7 kfree(p);

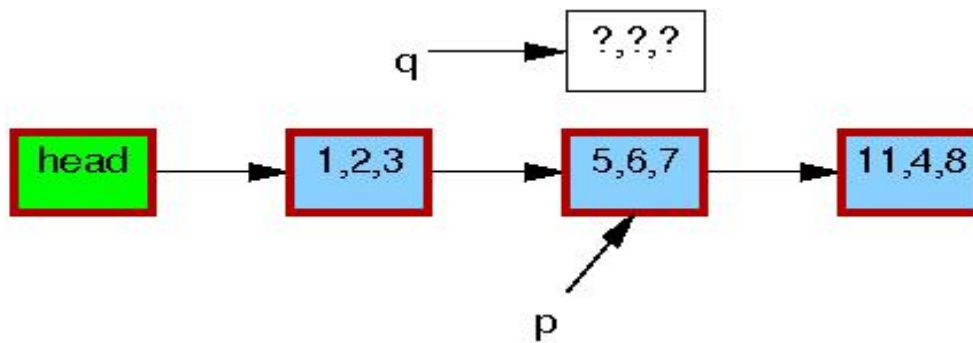
```

The initial state of the list, including the pointer p, is the same as for the deletion example:

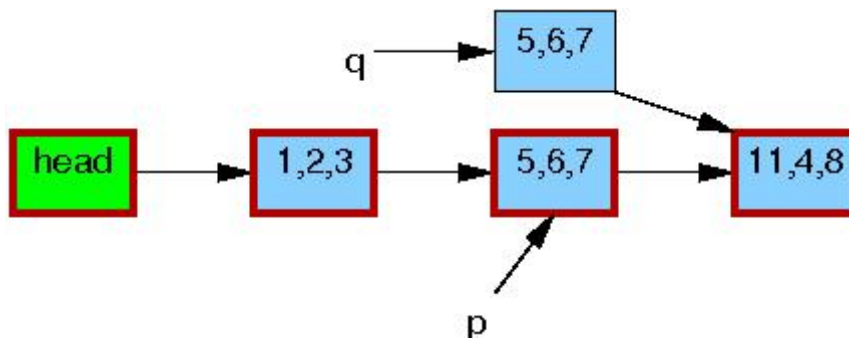


As before, the triples in each element represent the values of fields a, b, and c, respectively. The red borders on each element indicate that readers might be holding references to them, and because readers do not synchronize directly with updaters, readers might run concurrently with this entire replacement process. Please note that we again omit the backwards pointers and the link from the tail of the list to the head for clarity.

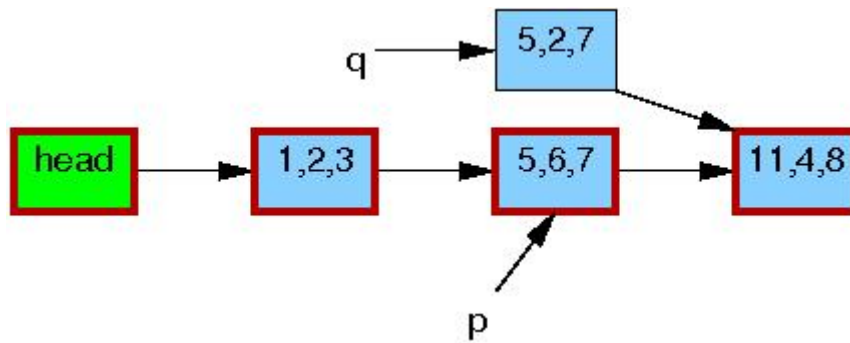
Line 1 `kmalloc()`s a replacement element, as follows:



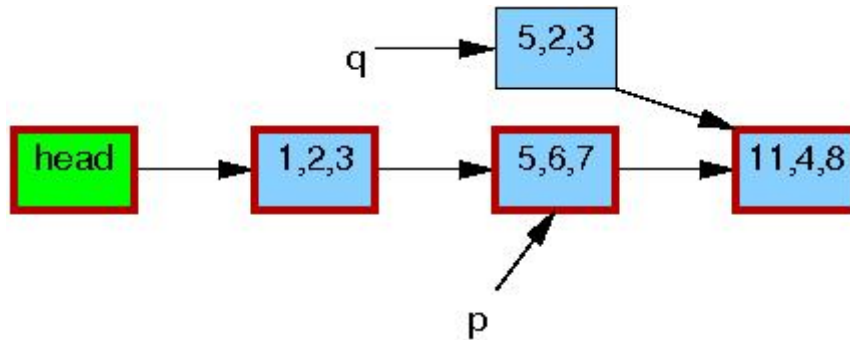
Line 2 copies the old element to the new one:



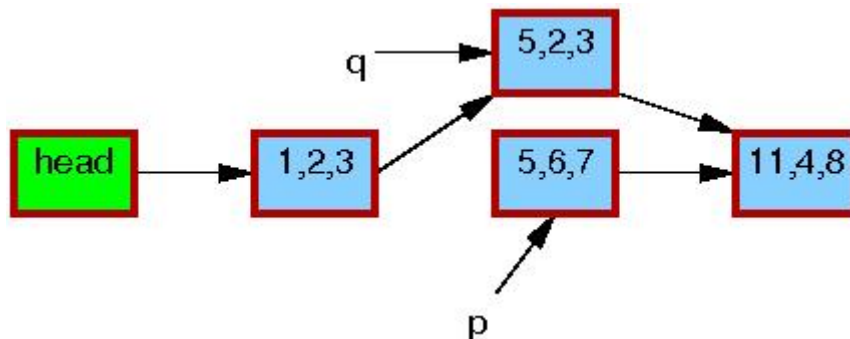
Line 3 updates q->b to the value "2":



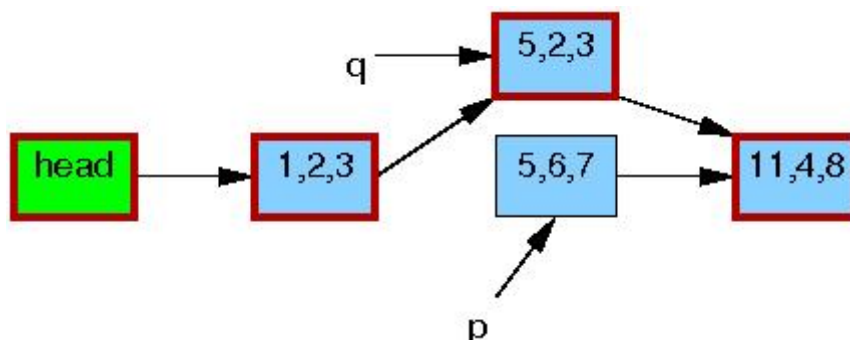
Line 4 updates q->c to the value "3":



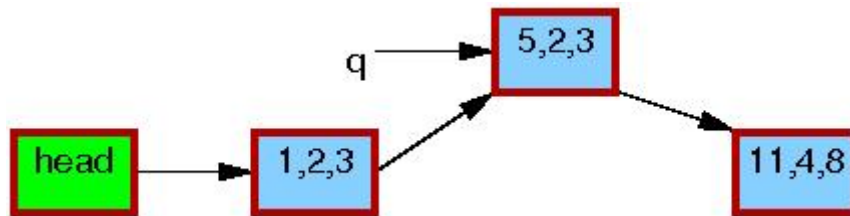
Now, line 5 does the replacement, so that the new element is finally visible to readers. At this point, as shown below, we have two versions of the list. Pre-existing readers might see the 5,6,7 element, but new readers will instead see the 5,2,3 element. But any given reader is guaranteed to see some well-defined list.



After the `synchronize_rcu()` on line 6 returns, a grace period will have elapsed, and so all reads that started before the `list_replace_rcu()` will have completed. In particular, any readers that might have been holding references to the 5,6,7 element are guaranteed to have exited their RCU read-side critical sections, and are thus prohibited from continuing to hold a reference. Therefore, there can no longer be any readers holding references to the old element, as indicated by the thin black border around the 5,6,7 element below. As far as the readers are concerned, we are back to having a single version of the list, but with the new element in place of the old.



After the `kfree()` on line 7 completes, the list will appear as follows:



Despite the fact that RCU was named after the replacement case, the vast majority of RCU usage within the Linux kernel relies on the simple deletion case shown in the [previous section](#).

Discussion

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 4: How would you modify the deletion example to permit more than two versions of the list to be active?

Quick Quiz 5: How many RCU versions of a given list can be active at any given time?

This sequence of events shows how RCU updates use multiple versions to safely carry out changes in presence of concurrent readers. Of course, some algorithms cannot gracefully handle multiple versions. There are [techniques \[PDF\]](#) for adapting such algorithms to RCU, but these are beyond the scope of this article.

Conclusion

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

1. a publish-subscribe mechanism for adding new data,
2. a way of waiting for pre-existing RCU readers to finish, and
3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

Quick Quiz 6: How can RCU updaters possibly delay RCU readers, given that the `rcu_read_lock()` and `rcu_read_unlock()` primitives neither spin nor block?

These three RCU components allow data to be updated in face of concurrent readers, and can be combined in different ways to implement a surprising variety of different types of RCU-based algorithms, some of which will be the topic of the next installment in this "What is RCU, Really?" series.

Acknowledgements

We are all indebted to Andy Whitcroft, Gautham Shenoy, and Mike Fulton, whose review of an early draft of this document greatly improved it. We owe thanks to the members of the Relativistic Programming project and to members of PNW TEC for many valuable discussions. We are grateful to Dan Frye for his support of this effort. Finally, this material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

This work represents the view of the authors and does not necessarily represent the view of IBM or of Portland State University.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.

Answers to Quick Quizzes

Quick Quiz 1: But doesn't seqlock also permit readers and updaters to get work done concurrently?

Answer: Yes and no. Although seqlock readers can run concurrently with seqlock writers, whenever this happens, the `read_seqretry()` primitive will force the reader to retry. This means that any work done by a seqlock reader running concurrently with a seqlock updater will be discarded and redone. So seqlock readers can *run* concurrently with updaters, but they cannot actually get any work done in this case.

In contrast, RCU readers can perform useful work even in presence of concurrent RCU updaters.

Quick Quiz 2: What prevents the `list_for_each_entry_rcu()` from getting a segfault if it happens to execute at exactly the same time as the `list_add_rcu()`?

Answer: On all systems running Linux, loads from and stores to pointers are atomic, that is, if a store to a pointer occurs at the same time as a load from that same pointer, the load will return either the initial value or the value stored, never some bitwise mashup of the two. In addition, the `list_for_each_entry_rcu()` always proceeds forward through the list, never looking back. Therefore, the `list_for_each_entry_rcu()` will either see the element being added by `list_add_rcu()`, or it will not, but either way, it will see a valid well-formed list.

Back to Quick Quiz 2.

Quick Quiz 3: Why do we need to pass two pointers into `hlist_for_each_entry_rcu()` when only one is needed for `list_for_each_entry_rcu()`?

Answer: Because in an hlist it is necessary to check for NULL rather than for encountering the head. (Try coding up a single-pointer `hlist_for_each_entry_rcu()`. If you come up with a nice solution, it would be a very good thing!)

Back to Quick Quiz 3.

Quick Quiz 4: How would you modify the deletion example to permit more than two versions of the list to be active?

Answer: One way of accomplishing this is as follows:

```
spin_lock(&mylock);
p = search(head, key);
if (p == NULL)
    spin_unlock(&mylock);
else {
    list_del_rcu(&p->list);
    spin_unlock(&mylock);
    synchronize_rcu();
    kfree(p);
}
```

Note that this means that multiple concurrent deletions might be waiting in `synchronize_rcu()`.

Back to Quick Quiz 4.

Quick Quiz 5: How many RCU versions of a given list can be active at any given time?

Answer: That depends on the synchronization design. If a semaphore protecting the update is held across the grace period, then there can be at most two versions, the old and the new.

However, if only the search, the update, and the `list_replace_rcu()` were protected by a lock, then there could be an arbitrary number of versions active, limited only by memory and by how many updates could be completed within a grace period. But please note that data structures that are updated so frequently probably are not good candidates for RCU. That said, RCU can handle high update rates when necessary.

[Back to Quick Quiz 5.](#)

Quick Quiz 6: How can RCU updaters possibly delay RCU readers, given that the `rcu_read_lock()` and `rcu_read_unlock()` primitives neither spin nor block?

Answer: The modifications undertaken by a given RCU updater will cause the corresponding CPU to invalidate cache lines containing the data, forcing the CPUs running concurrent RCU readers to incur expensive cache misses. (Can you design an algorithm that changes a data structure *without* inflicting expensive cache misses on concurrent readers? On subsequent readers?)

[Back to Quick Quiz 6.](#)