Some Examples of Kernel-Hacker Informal Correctness Reasoning

Paul E. McKenney
paulmck@linux.vnet.ibm.com

June 17, 2015

1 Introduction

I presented an overview of read-copy update (RCU) [22, 17, 11] at the May 2015 Dagstuhl workshop on Compositional Verification Methods for Next-Generation Concurrency, and was pleasantly but profoundly surprised to learn that a number of the formal-verification researchers in attendance were disappointed to have not seen any RCU code. This document is an attempt to give them some degree of satisfaction.

This document therefore illustrates a few simple examples of concurrent data structures, each of which has roughly similar counterparts in the Linux kernel. Each of these examples should be short enough to be compatible with formal-verification techniques, and is accompanied by an overview of the reasoning process that a kernel hacker might use when deciding whether or not a given example is applicable to the situation at hand.

To that end, Section 2 presents a warm-up exercise involving split counters, Section 3 shows a toy implementation of the RCU infrastructure, Section 4 gives an RCU implementation of bags, Section 5 overviews an RCU implementation of bags that is suitable for very large sets, and Section 6 overviews one kernel hacker's view of RCU.

2 Warmup Exercise: Split Counters

Although RCU is conceptually quite simple, effective use of RCU often requires a subtle but profound change in thinking about concurrency. This document therefore starts not with RCU, but rather with a warmup exercise involving one of the simplest and most intuitive possible algorithms, namely that of integer addition, in the guise of the *split counter*. The semantics of a split counter are similar to those of an atomically manipulated single global counter, but with the addition of relaxed ordering semantics in conjunction with weakly ordered hardware and compilers.

The remainder of this section is organized as follows: Section 2.1 presents the implementation of a simple split counter, Section 2.2 gives an extreme elaboration of kernel-hacker thought process as to why this implementation is correct, Section 2.3 provides an error-injected version of the split-counter implementation for use in verifying verifiers, Section 2.4 lists the behaviors exhibited by split counters compared to an atomically manipulated global counter, and finally Section 2.5 gives the thought process that a kernel hacker might actually use when selecting split counters to solve a counting problem.

2.1 Split-Counter Implementation

Because the canonical implementation of split counters is trivial, this discussion will start with the implemntation.

Split counters are implemented by providing a separate counter for each thread. To update the counter, a given thread updates its per-thread counter using simple relaxed loads and stores. To read out the value of the counter, all threads' counters are accessed using relaxed loads, and the values loaded are summed.

```
DEFINE_SPLIT_COUNTER(mycount);

void add_split_counter(unsigned long v)
{
  WRITE_ONCE(__get_thread_var(mycount),
       READ_ONCE(__get_thread_var(mycount)) + v);
}

unsigned long read_split_counter(void)
{
  int t;
  unsigned long sum = 0;

  for_each_thread(t)
    sum += READ_ONCE(__get_thread_var(mycount));
  return sum;
}
```

Figure 1: Split Counter

There is no use of any sort of memory-ordering constraints or of any sort of read-modify-write atomic operations. These counters are bounded and exhibit the same wrap-around behavior that is exhibited by a simple C-language unsigned integer variable. Sample code is shown in Figure 1. Note that `READ_ONCE()` (`WRITE_ONCE()`) can be thought of as a C11 `memory_order_relaxed` load (respectively store) with volatile semantics.[1]

This trivial algorithm is used very heavily in kernels and server applications in order to provide extremely lightweight gathering of important statistical values, for but one example, the amount of network data transmitted and received by the system. The normal use case is to periodically read out and log the aggregate counter value. Rate information can be calculated from the logged values, and unusual rates can then be flagged to initiate troubleshooting. For example, an unexpected order-of-magnitude decrease in packet rate would normally indicate a problem that needs to be investigated.

2.2 Kernel-Hacker Split-Counter Correctness Argument

Split counters rely on the commutative and associative laws of modular addition. The effect of a split counter is to form the following sum:

[1] Within the Linux kernel, these primitives are implemented via casts to volatile.

$$S = \sum_{t \in T, 0 \leq i \leq N_t} A_{t,i} \tag{1}$$

Where S is the desired sum, T is the set of threads, N_t is the number of addition operations executed by thread t, and $A_{t,i}$ is the value of thread t's i^{th} addition operations.

The associative law of modular addition allows arbitrary grouping of the addition operations. When running on real hardware, a wise choice is to group by thread, which is implemented using the per-thread counters. These per-thread counters can then be summed to arrive at the correct value of S. The commutative law of addition further allows use of weak-memory ordering, because the order of addition has no effect on the sum.

However, this line of reasoning must face the additonal challenge of summation operations proceeding concurrently with addition operations. Bounds on the sum are needed.

Rough bounds can be provided by a mythical global counter that reflects the current aggregate value of the split counter. This global counter might take of the sequence of values that would be taken on by an atomically manipulated global counter, but the mythical status avoids the pathological levels of memory contention that would be experienced by a concrete implemmnetation running on a large system.

Let V be the set of values taken on by the mythical counter during the execution of a given instance of `read_split_counter()`. The the bounds on S are given by:

$$\min V \leq S \leq \max V \tag{2}$$

Where $\min V$ is the value of the smallest element of set V and, similarly, $\max V$ is the value of the larges element of V. Note that "$\leq$" must be defined so as to take overflow into account. This means that there are no constraints on S if the summation takes so long that the mythical counter ranges over all possible values, which is an intentional and accurate reflection of reality. The possibility of counter wrapping appears to pose a substantial challenge to a number of proof systems, however, this challenge cannot reasonably be considered to be a fault of the split counter.

Exact bounds require knowing the maximum time required for an update carried out by one CPU to be visible by another. Vendors are unfortunately reluctant to release this information, but an estimate could be derived empirically. Given such an estimate E, set V is collected over a time period starting E before the start of a given instance of `read_split_counter()` and ending E after that instance's completion.

In practice, these bounds are sufficiently tight for many use cases.

2.3 Potential Split-Counter Bugs

It is wise to maintain a healthy skepticism of a successful verification result. After all, a verification tool might simply unconditionally primt `VERIFICATION SUCCESSFUL`. If this tool included code that did nothing in sufficiently complex ways, we might be none the wiser even after examining its source code. It is therefore also wise to run the alleged verification tool on a program that contains an intentional bug.

Given that I have never observed a bug in a split-counters implmentation in a quarter century of parallel programming, it is safe to assume that split counters is a trivial software system. The bugs introduced in Figure 2 will therefore likely seem to be somewhat contrived. However, contrived or not, a formal verification run that fails to find the bugs introduced by any of the `FORCE_BUG_*` C-preprocessor symbols must be viewed with some suspicion.

```
DEFINE_SPLIT_COUNTER(mycount);

void add_split_counter(unsigned long v)
{
  unsigned long v1 = v;
  unsigned long oldcount;

#ifdef FORCE_BUG_RAND_ADD
  v1 = random();
#endif
#ifdef FORCE_BUG_WRONG_READ
  oldcount = READ_ONCE(per_thread(mycount,
                       my_smp_thread_id & ~0x1));
#else
  oldcount = READ_ONCE(__get_thread_var(mycount));
#endif
  oldcount += v;
#ifdef FORCE_BUG_WRONG_WRITE
  WRITE_ONCE(per_thread(mycount,
                        my_smp_thread_id & ~0x1), v1);
#else
  WRITE_ONCE(__get_thread_var(mycount),
             oldcount + v1);
#endif
}

unsigned long read_split_counter(void)
{
  int t;
  unsigned long sum = 0;

  for_each_thread(t) {
#ifdef FORCE_BUG_DOUBLE_READ
    sum += READ_ONCE(per_thread(mycount,
                                my_smp_thread_id & ~0x1));
#elif defined(FORCE_BUG_OMIT_READ
    if (t & 0x1)
      sum += READ_ONCE(__get_thread_var(mycount));
#else
    sum += READ_ONCE(__get_thread_var(mycount));
#endif
  }
  return sum;
}
```

Figure 2: Split Counter Bugs

2.4 Split-Counter Behaviors

The behaviors of a split counter are those of an atomically updated global counter, but with the addition of weak-memory behaviors. For an example of weak-memory behavior, consider the following set of concurrent operations:

1. Thread 0: `add_split_counter(1)`
2. Thread 1: `add_split_counter(2)`
3. Thread 2: `r1 = read_split_counter()`

4. Thread 3: `r2 = read_split_counter()`

Given a global atomically manipulated counter, the following outcomes are allowed:

1. `r1 == 0 && r2 == 0`
2. `r1 == 0 && r2 == 1`
3. `r1 == 0 && r2 == 2`
4. `r1 == 0 && r2 == 3`
5. `r1 == 1 && r2 == 0`
6. `r1 == 1 && r2 == 1`
7. `r1 == 1 && r2 == 3`
8. `r1 == 2 && r2 == 0`
9. `r1 == 2 && r2 == 2`
10. `r1 == 2 && r2 == 3`
11. `r1 == 3 && r2 == 0`
12. `r1 == 3 && r2 == 1`
13. `r1 == 3 && r2 == 2`
14. `r1 == 3 && r2 == 3`

Split counters allow the following outcomes in addition to those listed above:

1. `r1 == 1 && r2 == 2`
2. `r1 == 2 && r2 == 1`

The challenge is thus to construct a formal specification of split counters, verify a correct implementation, and locate bugs in erroneous implementations, as exemplified by those in Figure 2.

2.5 How Kernel Hackers Really Model Split Counters

The actual thought process is extremely simple: A split counter is a mechanism with extremely lightweight updates and approximate reads, where the approximation is many orders of magnitude more than good enough for the intended use cases. There is therefore absolutely no need to do any sort of error analysis in practice. It would typically take less time for the kernel hacker to decide whether or not to use a split counter than it took you to read this paragraph. Actually implementing the split counter would take a similar amount of time.

A counter update is expected to have roughly the same performance characteristics as that of a simple addition operation on a private variable, possibly with an additional small constant cost to compute the location of the running thread's per-thread counter. Reading out the aggregate value of the counter is expected to take up to $N - 1$ additions and to incur up to $2N - 2$ cache misses, where N is the number of threads. One might instead expect $N - 1$ cache misses, but that fails to account for the cache misses incurred by each other thread the next time it attempts to update its per-thread counter.

Please note that this split-counter is merely the simplest member of a large set of concurrent counter algorithms[13, Chapter 5]. The more complex members pose less trivial verification challenges.

3 RCU Infrastructure

There is a surprisingly large number of independent inventions of mechanisms vaguely resembling RCU [10, 24, 7, 23, 8, 9, 4, 22, 6, 25]. The distinctive feature of RCU in the Linux kernel is the fact that a large number of developers has successfully used it. Recently, there has also been a surprisingly large number of proofs of correctness (both formal and informal of RCU algorithms and implementations [1, 5, 26].[2] That said, these proofs focus on

[2] There is rumored to be an additional Czech-language dissertation proving correctness of RCU, but I do not have a citation.

```
 1 #define rcu_read_lock()
 2 #define rcu_read_unlock()
 3 #define rcu_dereference(p) \
 4 ({ \
 5         typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
 6   smp_read_barrier_depends(); \
 7   _p1; \
 8 })
 9 #define rcu_assign_pointer(p, v) \
10 ({ \
11         smp_wmb(); \
12   (p) = (v); \
13 })
14 void synchronize_rcu(void)
15 {
16   int cpu;
17
18   for_each_online_cpu(cpu)
19     run_on(cpu);
20 }
```

Figure 3: Toy Implementation of RCU Infrastructure

RCU's safety properties, leaving much unsaid about RCU's ordering properties. The rest of this section therefore fills in this gap.

This section uses very imprecise terminology to denote ordering properties. More precise determination of the ordering properties is left as an exercise for the reader. This is typical when kernel hacking: The kernel hacker does not care what ordering a given algorithm exhibits in any formal sense, but rather whether or not it is strong enough to do the job at hand.

3.1 Toy Implementation of RCU Infrastructure

Figure 3 shows a "toy" implementation of a fragment of the RCU API for a non-preemptive environment.[3] This implementation provides full read-side performance, but suffers from a number of problems:

1. It has abysmal update-side scalability and energy efficiency.

2. The prohibition against blocking while in RCU read-side critical sections degrades real-time response.

3. Updaters can degrade real-time response for readers (or, alternatively, high-priority readers can starve updaters).

4. Updaters can fail in the presence of CPU-hotplug operations.

That said, this implementation has the advantage of abject simplicity. In addition, an early implementation of RCU within the DYNIX/ptx clusters implementation used an approach quite similar to this.

The `rcu_read_lock()` and `rcu_read_unlock()` primitives on lines 1 and 2 of the figure generate no code and in fact don't even reach the compiler backend. The purpose of these primitives is not to affect machine state, but rather to remind the developer of the need to avoid blocking within the resulting RCU read-side critical section.

The `rcu_dereference()` primitives on lines 3-8 of the figure simply applies volatile semantics to the load [3].[4] In conjunction with suitable restrictions on use of the returned value [21], these volatile semantics ensure that any dereferences of the pointer returned from `rcu_dereference()` will be ordered after the `rcu_dereference()` itself.

The `rcu_assign_pointer()` primitive on lines 9-13 of the figure is slightly weaker than a `memory_order_release()` store. (Recent versions of the Linux kernel actually use a `memory_order_release()` store.)

Finally, the `synchronize_rcu()` primitive shown on lines 14-20 of the figure simply schedules on each online CPU in turn. Because this is a non-preemptive environment and because RCU read-side critical section are not permitted to block, if a given CPU is executing within an RCU read-side critical section, the `run_on()` on line 19 will delay until that critical section completes. Therefore, once `synchronize_rcu()` completes, all pre-existing RCU read-side critical sections will have completed, as required.

Please note that any formal verification of this code should be able to detect an incomplete scan of the CPUs by `synchronize_rcu()`, for example, as shown in Figure 4, which ignores all even-numbered CPUs.

[3] The full implementation has many tens of API members [15].

[4] `READ_ONCE()` is the new name for read-side use cases of the `ACCESS_ONCE()` primitive described in this document.

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

  for_each_online_cpu(cpu)
    if (cpu & 0x1)
      run_on(cpu);
}
```

Figure 4: Toy Implementation of RCU Infrastructure With Injected Bug

```
int x, y;

void t0(void)
{
  rcu_read_lock();
  r2 = READ_ONCE(y);
  rcu_read_unlock();
  rcu_read_lock();
  r1 = READ_ONCE(x);
  rcu_read_unlock();
}

void t1(void)
{
  rcu_read_lock();
  WRITE_ONCE(x, 1);
  rcu_read_unlock();
  rcu_read_lock();
  WRITE_ONCE(y, 1);
  rcu_read_unlock();
}
```

Figure 5: No RCU Read-Side Critical Section Ordering

Any verification that fails to locate this bug will of course not be taken seriously.

Thus, a toy implementation of RCU requires only 20 lines of code. However, people still insist on asserting that RCU is complicated.

3.2 RCU Readers Are Weakly Ordered

RCU read-side critical sections begin with `rcu_read_lock()`, end with `rcu_read_unlock()`, and can contain RCU iterators and RCU pointer-traversal primitives. Important safety tip: In and of themselves, `rcu_read_lock()` and `rcu_read_unlock()` have absolutely no ordering properties whatsoever. For example, in the litmus test shown in Figure 5, all four combinations of the final values of `r1` and `r2` are allowed.

The primitives demarking RCU read-side critical sections can therefore be considered to be maximally weak.

The pointer-access primitive `rcu_dereference()` has semantics similar to a C11 `memory_order_consume` load. Roughly speaking, subsequent operations whose address or data depends on the value returned by `rcu_dereference()` will be ordered after the `rcu_dereference()`. A more precise definition of these ordering semantics is work in progress within the C and C++ standards committees.

3.3 RCU Grace Periods Are Strongly Ordered

If any portion of any RCU read-side critical section causally precedes a given RCU grace period, then the entirety of that RCU read-side critical section causally precedes any code that causally follows that same RCU grace period. Similarly, if any portion of any RCU read-side critical section causally follows a given RCU grace period, then the entirety of that RCU read-side critical section causally follows any code that causally precedes that same RCU grace period. If no part of a given RCU read-side critical section casally follows or causally precedes a given grace period, then there is no guarantee of ordering against any other RCU read-side critical section with respect to that grace period.[5] Roughly speaking, causal ordering is defined to be the union of modification order, reads-from, from-reads, and program order.

Figure 6 shows an example of this ordering. If `r1==0`, we know that line 6 of `t0()`'s RCU read-side critical section causally precedes `t1()`'s grace period. Therefore, we know that both lines 6 and 7 causally precede line 15, implying that `r2==0`. On the other hand, if `r2==1`, we know that line 7 of `t0()`'s RCU read-side critical section causally follows `t1()`'s grace period. Therefore, we know that both lines 6 and 7 causally follow line 13, implying that `r1==1`. The outcome `r1==0&&r2==1` is therefore excluded.

[5] In this case, the grace period completely overlaps the given RCU read-side critical section.

```
 1 int x, y;
 2
 3 void t0(void)
 4 {
 5   rcu_read_lock();
 6   r1 = READ_ONCE(x);
 7   r2 = READ_ONCE(y);
 8   rcu_read_unlock();
 9 }
10
11 void t1(void)
12 {
13   WRITE_ONCE(x, 1);
14   synchronize_rcu();
15   WRITE_ONCE(y, 1);
16 }
```

Figure 6: RCU Grace Periods Provide Strong Ordering

In contrast, if all of Figure 6's loads and stores were `memory_order_seq_cst`, and if the `synchronize_rcu()` on line 14 were omitted, all four possible outcomes would be allowed.

The ordering provided by the combination of RCU read-side critical sections and RCU grace periods is thus extremely strong, and this ordering has proven to be extremely useful in practice.[6]

4 RCU Bag

Figure 7 shows a possibly buggy implementation of a bag (which permits duplicates) using an RCU-protected linked list. Such an implementation might be useful when:

1. The bag is to be queried far more frequently than it is to be modified.
2. The integers in the bag might be quite large, ruling out use of small arrays of counters.
3. The number of elements in the bag will be quite small, so that the overhead of traversing the linked list is negligible.

Of course, very similar code implements mappings, bags, and ordered sets, but let's keep things simple.

```
struct elem {
  struct list_head *list;
  int key;
};
LIST_HEAD(mybag);
DEFINE_SPINLOCK(mybaglock);

int member(int key)
{
  struct elem *ep;

  rcu_read_lock();
  list_for_each_entry_rcu(ep, &mybag, list)
    if (ep->key == key) {
      rcu_read_unlock();
      return 1;
    }
  rcu_read_unlock();
  return 0;
}

int add(int key)
{
  struct elem *ep;

  ep = kmalloc(sizeof(*ep), GFP_KERNEL);
  if (!ep)
    return 0;
  ep->key = key;
  spin_lock(&mybaglock);
  list_add_rcu(ep, &mybag);
  spin_unlock(&mybaglock);
  return 1;
}

int remove(int key)
{
  struct elem *ep;

  spin_lock(&mybaglock);
  list_for_each_entry_rcu(ep, &mybag, list)
    if (ep->key == key) {
      list_del_rcu(ep);
      spin_unlock(&mybaglock);
      synchronize_rcu();
      kfree(ep);
      return 1;
    }
  spin_unlock(&mybaglock);
  return 0;
}
```

Figure 7: Bag Implemented Using RCU-Protected Linked List

[6] Kudos to Jade Alglave for calling attention to the strong ordering characteristics of RCU grace periods.

4.1 Bag Implementation

Lines 1-4 show the data structure representing an element in this bag, which contains a pair of pointers for insertion into a circular doubly linked list and an integer representing the value of the element. Line 5 represents the head of the list, which is also a pair of pointers. Line 6 defines the spinlock that guards modifcation of this list, initially in unlocked state.

Lines 9-20 show the membership-query function. Line 12 enters an RCU read-side critical section, and lines 15 and 18 exit it. Line 13 iterates over the list headed by the `mybag` global variable, and for each element executes the code on lines 14-17. If line 14 determines that the current element matches the desired key, line 15 exits the RCU read-side critical section and line 16 returns `1` to indicate that the specified value is a member of the bag. Otherwise, control eventually reaches line 18, which exits the RCU read-side critical section, allowing line 19 to return `0` to indicate that the specified value is not a member of the bag.

Lines 22-34 show the member-addition function. Line 26 allocates a new `elem` structure, and lines 27 and 28 handle the out-of-memory case. Line 29 initializes the element's key. Line 30 acquires the lock, line 31 does the pointer manipulations required to add the new member to the bag along with any needed memory-ordering constraints, and line 32 releases the lock. Finally, line 33 returns `1` to indicate success.

Lines 36-51 show the member-removal function. This is similar to `member()`, but substitutes locking on lines 40, 44, and 49 for the RCU operations on lines 12, 15, and 18. Because we hold the lock, there can of course be no concurrent updates to the linked list. Once an element is found, line 43 does the linked-list manipulations required to remove the element, line 44 releases the lock, and line 45 waits for an RCU grace period to elapse. Once control reaches line 46, there can no longer be any readers holding a reference to the newly deleted element, so line 46 can safely free it.

This represents a straightforward concurrent implementation of a bag, with excellent read-side performance and scalability.

```
 1 void t0(void)
 2 {
 3   add(0);
 4 }
 5
 6 void t1(void)
 7 {
 8   add(1);
 9 }
10
11 void t2(void)
12 {
13   r1 = member(0);
14   r2 = member(1);
15 }
16
17 void t3(void)
18 {
19   r1 = member(1);
20   r2 = member(0);
21 }
```

Figure 8: RCU Bags Strongly Ordered For Addition

Candidate bug injections include omitting the `synchronize_rcu()`, prematurely terminating the loops in `member()` and `remove()`, randomly refusing to add the element, randomly perturbing the key, and so on.

4.2 Bag Memory-Ordering Properties

The insertions and removals will be fully ordered, courtesy of `mybaglock`. Interestingly enough, if there are additions but no removals, membership queries will be strongly ordered, as illustrated by Figure 8. The key point is that the list pointers are subject to coherence properties similar to those of C11 `memory_order_relaxed` accesses. Therefore, if element `0` was added to the list before element `1`, if `t3()` sees element `1` on line 19, it must also see element `0` on line 20. The opposite insertion order imposes the corresponding constraint of `t2()`. Therefore, the IRIW cycle is forbidden.

In short, sequential consistency (or something very similar to it) has been restored at the linked-list level for additions and membership queries, despite RCU's extremely weak read-side ordering properties.

This raises the question of whether removals and membership operations are similarly ordered. This question is left as an exercise for the reader. In my uneducated opinion, removals and membership operations are in fact strongly ordered. If you disagree, provide a trace demonstrating weak ordering, for exmaple, by replacing the `add()` calls in Figure 8 with `remove()`.

4.3 RCU Bag Timing

In the absence of updates, readers incur no cache misses, and on average incur a number of loads that increases linearly with the number of elemnets in the bag. This data structure is therefore quite suitable for situations where the bag has few elements and updates are rare. In contrast, frequent updates will of course result in excessive contention on `mybaglock`, and will further inflict cache misses on all readers.

5 RCU Large Bag

If a bag has a large number of members, the linked-list implementation discussed in Section 4 will of course suffer from poor performance due to a linear search through a linked list. One time-honored solution to this problem is to use a hash table, as shown in Figure 9. The code is quite similar to that in Figure 7, with the differences being the use of a hash function `hash()` to select one of `NBUCKETS` linked lists instead of using a single linked list. Note also that global locking has been replaced by per-bucket locking.

Of course, the need to select a hash-table size at compile time can be quite inconvenient, however, implementation of RCU-protected resizable hash tables is reasonably straightforward [28, 27, 20]. Two of these variants may be found in the Linux kernel. An overview of one of the simpler implementations is also available [13, Chapter 10].

The memory-ordering litmus test shown in Figure 8 also applies to the hash-table implementation. However, the coherence property no longer holds because there is now more than one list header. The hash-table implementation can therefore produce the counter-intuitive outcome `r1==1&&r2==0&&r3==1&&r4==0`, so that the strong ordering restored at the linked-list level is lost at the hash-table level.

```
struct elem {
  struct list_head *list;
  int key;
};
struct bucket {
  struct list_head *head;
  spinlock_t lock
};
struct bucket mybag[NBUCKETS];

int member(int key}
{
  struct elem *ep;
  int idx = hash(key);

  rcu_read_lock();
  list_for_each_entry_rcu(ep, &mybag[idx].head, list)
    if (ep->key == key) {
      rcu_read_unlock();
      return 1;
    }
  rcu_read_unlock();
  return 0;
}

int add(int key)
{
  struct elem *ep;
  int idx = hash(key);

  ep = kmalloc(sizeof(*ep), GFP_KERNEL);
  if (!ep)
    return 0;
  ep->key = key;
  spin_lock(&mybag[idx].lock);
  list_add_rcu(ep, &mybag[idx].head);
  spin_unlock(&mybag[idx].lock);
  return 1;
}

int remove(int key)
{
  struct elem *ep;
  int idx = hash(key);

  spin_lock(&mybag[idx].lock);
  list_for_each_entry_rcu(ep, &mybag[idx].head, list)
    if (ep->key == key) {
      list_del_rcu(ep);
      spin_unlock(&mybag[idx].lock);
      synchronize_rcu();
      kfree(ep);
      return 1;
    }
  spin_unlock(&mybag[idx].lock);
  return 0;
}
```

Figure 9: Bag Implemented Using RCU-Protected Hash Table

That said, most Linux-kernel use cases of bags have very weak ordering requirements, which in turn means that the typical Linux kernel hacker will likely consider any discussion of ordering properties to be a pointless distraction.

The bug-injection candidates are similar to those called out in Section 4.1, with the addition of bugs involving inconsistent hash functions for different operations.

The timing of each bucket of this RCU hash-table bag is similar to that of the linked-list version, although of course the hash-table bag can tolerate higher update rates, at least assuming that the updates spread nicely across the hash table. Depending on the size of the hash table, the level of memory pressure, and the frequency of updates, it might or might not be wise to cache-align the elements of the `mybag` array.

6 Kernel-Hacker View of RCU

RCU is rarely used by itself, but instead in conjunction with other synchronization techniques, with a variety of combinations used to achieve a wide range of synchronization goals [11, 18, 19, 13]. For example, RCU can be used in conjunction with locking to implement an atomic move of elements from one binary search tree to another, which can be generalized to other types of linked structures [14, 12, 16]. This in turn means that it is not sufficient to merely understand RCU. It is instead necessary to understand RCU usage patterns that involve combinations of other synchronization primitives. For but one example, the Linux kernel's mechanism for traversing filesystem directory hierarchies uses a combination of locking, reference counting, sequence locking, and RCU. Enumerating the full space of such combinations is beyond the scope of this paper, although some information may be found elsewhere [13, Chapters 9 and 13]. The remainder of this section focuses on RCU timing (Section 6.1), RCU semantics (Section 6.2), and RCU applicability (Section 6.3.

6.1 RCU Timing

RCU's read-side primitives are exceedingly fast and scalable. In the limiting case of a server-class Linux kernel build, `rcu_read_lock()` and `rcu_read_unlock()` are zero cost, as can be seen in Figure 3. Other implementations require non-atomic updates to per-thread variables, often with no ordering constraints. However, real-time Linux-kernel implementations that do priority boosting on RCU readers will require invoking the scheduler in order to do deboosting at `rcu_read_unlock()` time. However, if a given RCU reader is running at the highest possible priority level, it will not be preempted, and will therefore not be priority boosted. Therefore, RCU readers running at the highest priority level enjoy full read-side performance.

The `rcu_dereference()` primitive applies volatile semantics, and thus might result in a slight decrease in performance due to suppression of certain types of compiler optimizations. In addition, on DEC Alpha, `rcu_dereference()` incurs the overhead of a full memory barrier. However, on current commodity hardware, `rcu_dereference()` has overhead roughly that of an unordered load instruction. The overheads of `rcu_read_lock()`, `rcu_read_unlock()`, and `rcu_dereference()` have been empirically shown to be constant with the number of CPUs up to 1024 CPUs on a Power server.

The `rcu_assign_pointer()` primitive applies volatile release semantics, which can inflict measurable overhead on RCU updaters. However, this overhead is negligible compared to the latency and overhead of `synchronize_rcu()`. In the Linux kernel, `synchronize_rcu()` has a latency of at least several milliseconds, and sometimes triggers warnings that are emitted if an RCU grace period lasts longer than 21 seconds. The CPU overhead of a given grace period is substantial, consuming microseconds on small systems and potentially even hundreds of microseconds on thousand-CPU systems. However, the Linux kernel uses batching, so that a given grace period might be shared by more than a thousand RCU updaters, reducing the per-update CPU overhead to negligible levels.

6.2 RCU Semantics

The two fundamental guarantees of RCU are the publish-subscribe guarantee linking `rcu_assign_pointer()` with `rcu_dereference()`, and the grace-period guarantee linking `synchronize_rcu()` with `rcu_read_lock()` and `rcu_read_unlock()` [5]. The grace-period guarantee can be stated in two ways: (1) The `synchronize_rcu()` primitive waits for all pre-existing RCU readers, and (2) If a task has executed a given `rcu_read_lock()` and accesses a given RCU-protected resource, some property P of that resource will be preserved until that task reaches the corresponding `rcu_read_unlock()`.

Here are some properties that RCU is used to preserve:

1. Existence.
2. Identity.
3. Type safety.
4. System state.

Each property is discussed in one of the following sections.

6.2.1 RCU Existence Guarantees

Existence is the property preserved in the RCU-protected bags discussed in Sections 4 and 5. Updates preserve this property by interposing an RCU grace period between the time that the resource is rendered inaccessible to readers and the time that the resource is reclaimed (in this case, freed).

Reliance on existence guarantees is illustrated by lines 17-19 of `add()` in Figure 9. Any element traversed by `list_for_each_entry_rcu()` must remain in existence (that is, cannot be freed) until after control reaches one of the `rcu_read_unlock()` calls on either line 19 or 22. This existence guarantee ensures that the `->key` access on line 18 remains meaningful, even in the face of concurrent removals.

If a guarantee is to be relied on, something somewhere must provide that guarantee, and in this case the provider is lines 49-52 of `remove()` in Figure 9. Line 49 renders the element inaccessible to readers, but the element is not passed to `kfree()` (line 52) until after an RCU grace period has elapsed (line 51).

```
 1 int go_undercover(int key)
 2 {
 3   struct elem *ep;
 4   int idx = hash(key);
 5
 6   spin_lock(&mybag[idx].lock);
 7   list_for_each_entry_rcu(ep, &mybag[idx].head, list)
 8     if (ep->key == key) {
 9       list_del_rcu(ep);
10       spin_unlock(&mybag[idx].lock);
11       synchronize_rcu();
12       ep->key = -ep->key;
13       idx = hash(ep->key);
14       spin_lock(&mybag[idx].lock);
15       list_add_rcu(ep, &mybag[idx].head);
16       spin_unlock(&mybag[idx].lock);
17       return 1;
18     }
19   spin_unlock(&mybag[idx].lock);
20   return 0;
21 }
```

Figure 10: RCU-Protected Hash Table Bag Identity Change

6.2.2 RCU Identity Guarantees

In some cases, a resource is not destroyed, but rather its identity is changed. For example, consider the function `go_undercover()` shown in Figure 10. Rather than freeing the element, as is done by `remove()`, lines 12-16 negate the element's key and then add it back into the bag. This code is therefore not providing an existence guarantee, but rather an identity guarantee.

Readers can rely on this guarantee in exactly the same way that they rely on existence guarantees.

6.2.3 RCU Type-Safety Guarantees

There are several situations in the Linux kernel where lightweight readers are needed, but where the overhead and latency of RCU grace periods cannot be tolerated. The `SLAB_DESTROY_BY_RCU` slab-allocator flag is used for this purpose, which allows an immediate free operation, but which guarantees that the type of the object will not change until the reader exits its RCU read-side critical section. However, the identity of the object can change at any time, so read-

```
 1 #define STATE_NORMAL        0
 2 #define STATE_WANT_SERVICE  1
 3 #define STATE_SERVICING     2
 4
 5 int state;
 6
 7 void do_something_service(void)
 8 {
 9   int state_snap;
10
11   rcu_read_lock();
12   state_snap = READ_ONCE(state);
13   if (state_snap == STATE_NORMAL)
14     do_something();
15   else
16     do_something_carefully();
17   rcu_read_unlock();
18 }
19
20 void start_service(void)
21 {
22   WRITE_ONCE(state, STATE_WANT_SERVICE);
23   synchronize_rcu();
24   WRITE_ONCE(state, STATE_SERVICING);
25 }
```

Figure 11: RCU-Mediated System-State Change

ers must carry out identity checks, usually after acquiring a lock or reference count associated with the object. These identity checks often rely on subtle global invariants.

6.2.4 RCU System-State Guarantees

A trivial form of system-state guarantee is illustrated by the code in Figure 11. The system must periodically undergo servicing, during which time normal operations can still be carried out, but alternative implementations of those operations must be used. For example, during servicing, `do_something_carefully()` must be invoked instead of the usual `do_something()`.

This transition to servicing state is mediated by the variable `state`, which can take on the three values called out on lines 1-3, but which is initially `STATE_NORMAL`, courtesy of the C language's default initialization to zero.

The `do_something_service()` function on lines 7-18 enters an RCU read-side critical section, then on line 12 takes a snapshot of the global `state` variable. Line 13 checks for normal state, which results in a call to `do_something()`, otherwise, `do_something_carefully()` is called. This code clearly needs a system-state guarantee: If line 13 determines that the state is `STATE_NORMAL`, then the state cannot change to `STATE_SERVICING` until after `do_something()` returns.

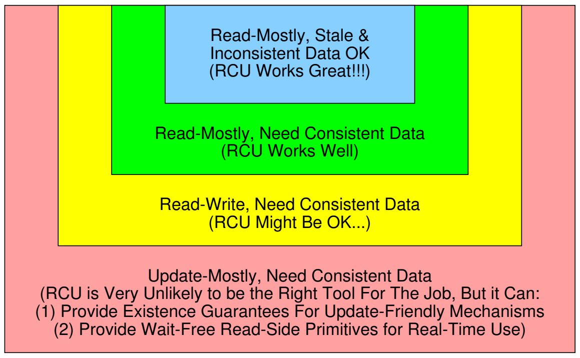
Read-Mostly, Stale &
Inconsistent Data OK
(RCU Works Great!!!)
Read-Mostly, Need Consistent Data
(RCU Works Well)
Read-Write, Need Consistent Data
(RCU Might Be OK...)
Update-Mostly, Need Consistent Data
(RCU is Very Unlikely to be the Right Tool For The Job, But it Can:
(1) Provide Existence Guarantees For Update-Friendly Mechanisms
(2) Provide Wait-Free Read-Side Primitives for Real-Time Use)

Figure 12: RCU Areas of Applicability

This system-state guarantee is provided by the `start_service()` function shown on lines 20-25. Line 22 sets `state` to `STATE_WANT_SERVICE`, and then line 23 waits for an RCU grace period to elapse. By the time control passes to line 24, all RCU readers that might have called `do_something()` have completed, so it it safe to set `state` to `STATE_SERVICING`.

This system-state guarantee was one of the first uses of RCU within Sequent's DYNIX/ptx clustering product [22, 17].

6.3 RCU Applicability

RCU can provide excellent performance and scalability, but it does so via specialization. As illustrated in the upper blue box in Figure 12, RCU is best suited to workloads that are read-mostly in cases where RCU's clients do not require consistent data. In such cases, use of RCU means simply adding RCU "annotations" similar to those shown in the RCU-protected bag implementations in Sections 4 and 5. The net result is excellent performance and ease of use.

RCU can also be used for read-mostly workloads where RCU's clients require consistent data, as indicated by the green box. One way to achieve this is to include a lock and a `->deleted` flag in each data element. When rendering the element inaccessible to readers, the updater acquires the per-element lock, renders the element inaccessible to readers, sets the flag, releases the lock, and then continues as before. Readers acquire the per-elmeent lock and check the flag. If the flag is set, they release the lock and continue as if they had not found the element. Otherwise, if the flag is clear, they release the lock and proceed normally. This technique was applied to the Linux kernel's System-V IPC subsystem [2].

This same technique works for read-write workloads where RCU's clients require consistent data, as indicated by the yellow box. The same per-elmeent lock and flag may be used, but the performance and perhaps also the scalability will likely be somewhat degraded.

Finally, as indicated by the red box, RCU is unlikely to be well-suited to write-mostly workloads where RCU's clients require consistent data. However, a couple of exceptions have been found thus far. In the first exception, RCU takes on the same role that garbage collectors do for certain types of non-blocking synchronization algorithms. In the second exception, a real-time workload has tight response-time constraints for a few infrequently executed code paths that include RCU readers, while the code paths containing RCU updates, though perhaps frequently executed, are not subject to response-time constraints.

7 Summary

This paper has illustrated a few simple examples of concurrent data structures, each of which has roughly similar counterparts in the Linux kernel. Each of these examples should be small enough to be compatible with formal-verification techniques, and is accompanied by timing information as well as an overview of the reasoning process that a kernel hacker might use when deciding whether or not a given example is applicable to the situation at hand.

Acknowledgments

I owe a debt of gratitude to Peter Sewell for his guidance and his careful review of an early draft of this material. We can all thank Cliff Jones for pointing out that what I was calling "unordered sets with duplicates" should instead be called "bags." We are all indebted to Derek Dreyer and Joe Tassarotti for asking questions that led to the writing of Section 6. I am also thankful to the organizers and attendees of the Schloss Dagstuhl workshop on Compositional Verification Methods for Next-Generation Concurrency for many interesting questions and fruitful discussions.

References

[1] 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. ???–???

[2] 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.

[3] Corbet, J. ACCESS_ONCE(). `http://lwn.net/Articles/508991/`, August 2012.

[4] Cowan, C., Autrey, T., Krasic, C., Pu, C., and Walpole, J. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems (ICCDS'96)* (Annapolis, MD, May 1996), p. 108.

[5] Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., and Walpole, J. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems 23* (2012), 375–382.

[6] 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.

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

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

[9] 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].

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

[11] McKenney, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[12] McKenney, P. E. C++ memory model meets high-update-rate data structures. `http://www2.rdrop.com/users/paulmck/RCU/C++Updates.2014.09.11a.pdf`, September 2014.

[13] McKenney, P. E. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (First Edition).* kernel.org, Corvallis, OR, USA, 2014.

[14] McKenney, P. E. N4037: Non-transactional implementation of atomic tree move. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf`, May 2014.

[15] McKenney, P. E. The RCU API, 2014 edition. `http://lwn.net/Articles/609904/`, September 2014.

[16] McKenney, P. E. High-performance and scalable updates: The issaquah challenge. `http://www2.rdrop.com/users/paulmck/scalability/paper/Updates.2015.01.16b.LCA.pdf`, January 2015.

[17] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001). Available: `http://www.linuxsymposium.org/2001/abstracts/readcopy.php` `http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf` [Viewed June 23, 2004].

[18] McKenney, P. E., Boyd-Wickizer, S., and Walpole, J. RCU usage in the linux kernel: One decade later. Technical report paulmck.2012.09.17, September 2012.

[19] McKenney, P. E., Boyd-Wickizer, S., and Walpole, J. RCU usage in the Linux kernel: One decade later. `http://rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf`, February 2013.

[20] McKenney, P. E., Desnoyers, M., and Jiangshan, L. The URCU hash table API. `https://lwn.net/Articles/573432/`, November 2013.

[21] McKenney, P. E., Riegel, T., Preshin, J., Boehm, H., Nelson, C., and Giroux, O. Towards implementation and use of `memory_order_consume`. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4321.pdf`, October 2014.

[22] 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.

[23] Pugh, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[24] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd Symposium on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, CA, October 1987), Association for Computing Machinery, pp. 31–39. Available: `http://www.cse.ucsc.edu/~randal/221/rashid-machvm.pdf` [Viewed February 17, 2005].

[25] Russell, R. Re: modular net drivers. Available: `http://oss.sgi.com/projects/netdev/archive/2000-06/msg00250.html` [Viewed April 10, 2006], June 2000.

[26] Tassarotti, J., Dreyer, D., and Vafeiadis, V. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 2015 Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation* (June 2015), ACM, pp. xxx–yyy.

[27] Triplett, J., McKenney, P. E., and Walpole, J. Scalable concurrent hash tables via relativistic programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[28] Xu, H. bridge: Add core IGMP snooping support. Available: `http://marc.info/?t=126719855400006&r=1&w=2` [Viewed March 20, 2011], February 2010.