RCU in the Linux Kernel: One Decade Later

by: Paul E. Mckenney, Silas Boyd-Wickizer, Jonathan Walpole

Slides by David Kennedy (and sources)
During this same time period, the usage of reader-writer locks in Linux has drastically decreased. Although the two approaches have extremely different semantics, this shows that RCU is a viable replacement for most rw lock use cases.
RCU Uses in Linux

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Uses</th>
<th>LoC</th>
<th>Uses / KLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>virt</td>
<td>72</td>
<td>6,749</td>
<td>10.67</td>
</tr>
<tr>
<td>net</td>
<td>3251</td>
<td>740,382</td>
<td>4.39</td>
</tr>
<tr>
<td>ipc</td>
<td>35</td>
<td>8,306</td>
<td>4.21</td>
</tr>
<tr>
<td>security</td>
<td>251</td>
<td>68,494</td>
<td>3.66</td>
</tr>
<tr>
<td>kernel</td>
<td>628</td>
<td>198,304</td>
<td>3.17</td>
</tr>
<tr>
<td>mm</td>
<td>196</td>
<td>88,904</td>
<td>2.20</td>
</tr>
<tr>
<td>block</td>
<td>58</td>
<td>27,975</td>
<td>2.07</td>
</tr>
<tr>
<td>lib</td>
<td>70</td>
<td>52,235</td>
<td>1.34</td>
</tr>
<tr>
<td>fs</td>
<td>666</td>
<td>1,057,713</td>
<td>0.63</td>
</tr>
<tr>
<td>init</td>
<td>2</td>
<td>3,382</td>
<td>0.59</td>
</tr>
<tr>
<td>include</td>
<td>279</td>
<td>552,507</td>
<td>0.50</td>
</tr>
<tr>
<td>crypto</td>
<td>12</td>
<td>64,537</td>
<td>0.19</td>
</tr>
<tr>
<td>drivers</td>
<td>1061</td>
<td>8,530,160</td>
<td>0.12</td>
</tr>
<tr>
<td>arch</td>
<td>183</td>
<td>2,459,105</td>
<td>0.07</td>
</tr>
<tr>
<td>Total</td>
<td>6764</td>
<td>13,858,753</td>
<td>0.49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of Usage</th>
<th>API Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCU critical sections</td>
<td>3035</td>
</tr>
<tr>
<td>RCU dereference</td>
<td>972</td>
</tr>
<tr>
<td>RCU synchronization</td>
<td>696</td>
</tr>
<tr>
<td>RCU list traversal</td>
<td>574</td>
</tr>
<tr>
<td>RCU list update</td>
<td>524</td>
</tr>
<tr>
<td>RCU assign</td>
<td>358</td>
</tr>
<tr>
<td>Annotation of RCU-protected pointers</td>
<td>304</td>
</tr>
<tr>
<td>Initialization and cleanup</td>
<td>273</td>
</tr>
<tr>
<td>RCU lockdep assertion</td>
<td>28</td>
</tr>
<tr>
<td>Total</td>
<td>6764</td>
</tr>
</tbody>
</table>

Figure 10: Linux 3.7 RCU usage by subsystem.

Figure 11: Linux 3.7 RCU usage by RCU API function.
Simplified RCU Implementation

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

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

void synchronize_rcu(void)
{
    for_each_cpu(int cpu)
        run_on(cpu);
}
Waiting for Context Switches

In practice, synchronize_rcu does not run on every cpu. Instead, it waits for a context switch on each cpu.

Must update state to indicate a context switch occurs, which must be done relatively infrequently (once per clock tick)

Around 1,000 calls to synchronize_rcu or call_rcu can be done in each batch

Writers wait for longer than strictly necessary, but this results in very low-cost read-side critical sections
RCU Design Patterns

• There are many conventions that should be used when doing RCU, depending on the type of semantics you need.

• Whether you perform multiple updates and then call synchronize_rcu once or call synchronize_rcu after each update makes a huge difference in what your program means.

• Following design patterns enforces restrictions on possible orderings amongst threads and help to ensure that RCU is used in safe ways (ex. not allowed to block in a read-side critical section).
Wait for completion

- Readers call `rcu_read_lock()` and `rcu_read_unlock()` and nothing else
- Writers synchronize their operations with one another by acquiring a mutex to write
- After performing one update, call `synchronize_rcu()` to wait for all pre-existing readers to finish
- Subsequent RCU read-side critical sections must see whatever change was made
- Would this be true if we used `call_rcu`?
void register_nmi_handler(handler_t cb)
{
    spin_lock(&nmi_list_lock);
    rcu_list_add(&nmi_list, cb);
    spin_unlock(&nmi_list_lock);
}

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

rcu_list_t nmi_list;
spinlock_t nmi_list_lock;

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

• Allows for dynamic replacement of NMI handlers

• Very lightweight read side primitives and no read-side blocking so deterministic completion time of read side critical sections

• Unlike reader-writer locking, many readers can access the list of NMI handlers without causing cache invalidations on other CPUs
• Why don’t we wait for completion after calling register_nmi_handler?

• If we were modifying an existing NMI handler, would we have to wait for completion?
NMI Handlers

• Non-Maskable interrupts
• usually indicates non-recoverable hardware errors (chipset errors, memory corruption)
• Response time is critical
• So this seems like a situation where reads would be because the conditions NMI handlers rarely and updates to the list of NMI handlers would also be rare.
• So why use RCU, other than for deterministic completion times?
NMI as a Debugging Tool

• When used for debugging or profiling with tools like Perf and OProfile, very many invocations of NMIs would be triggered

• Also, used in this way it is clearer why dynamic replacement would be a benefit
Using RCU to replace Reference Counting

- RCU read-side critical sections can be thought of as incrementing a reference count when entering, and decrementing when leaving, but without the cost of atomic operations on reference counts.
- Also avoids the problem of bouncing the cache line containing the reference count among many CPUs.
- RCU as reference counter very scalable to very large number of processes.
Keeping a reference to IP options while copying them to a packet

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

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

    queue_packet(packet);
}

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

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

    /* Handle other opt values */
}
```
Note that the old version of ip options is first copied to a local pointer, then the global version is atomically swapped to point to null.

The old version can be freed through the reference in the local pointer, either synchronously or asynchronously.
Reference Counting Vs. RCU

Rcu is very scalable!
Some limitations

• You cannot sleep in an read-side critical section (unless using SRCU)
• Will not work when a reference must be passed among separate tasks:
  ex: a reference is acquired when an I/O operation begins and the reference is released in an I/O completion handler
Publish Subscribe

• Initialize a new item to add to a data structure locally,

• Once initialized, atomically swing some global pointer to the newly created object

• \texttt{rcu\_assign\_pointer} and \texttt{rcu\_dereference} assure that the compiler does not reorder the following orderings of events
  
  initialize data $\rightarrow$ publish pointer
  dereference pointer $\rightarrow$ read data

  As we discussed last class, \texttt{rcu\_assign\_pointer} also issues a memory barrier as does \texttt{rcu\_dereference} on the DEC\-Alpha
Publish-Subscribe for Dynamic System Calls

```c
syscall_t *table;
spinlock_t table_lock;

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

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

    return r;
}
```

```c
void retract_table()
{
    syscall_t *local_table;
    spin_lock(&table_lock);
    local_table = table;
    rcu_assign_pointer(&table, NULL);
    spin_unlock(&table_lock);

    synchronize_rcu();
    kfree(local_table);
}
```
Some other cases...

• How would you implement modifying the extended system call table?

• How about modifying one system call entry in the extended system call table?
Reverse Page Map

There is a page_t for each physical page in memory.
Each page_t contains a pointer to a list of mappings to this page.

page_t pg

anon_vma_t a → anon_vma_t b → anon_vma_t c

pg is a memory location shared by multiple processes.
Reverse Page Map Data Race

Process 1: reads anon_vma b

Process 2: unmaps physical page pg, which deallocates all anon_vma_t’s mapped to pg

P1:

anon_vma_t vp;
//protects access to b
rcu_read_lock();          //this removes pg and all mappings to pg
vp = read_mapping(pg, b);
rcu_read_unlock();

P2:

unmap_page(pg);
Unmapping a Physical Page

unmap_page(page_t * arg) {
    for each (anon_vma_t a <- arg.vmalist)
        call_rcu(free, a)
    free(arg);
}

Problem: In Linux, the function used to free anon_vma_ts can acquire a mutex and block. If this mutex is held when the asynchronous call to free is executed, this would stall the thread executing all rcu callbacks. This could lead to a memory leak.
Possible Solutions

Associate a lock with each page_t: but this requires a lock for each physical page

Don’t acquire mutex in the function that deallocates anon_vma_t:
  but in Linux this function can acquire a mutex, and it would be a ton of work to change this in the source
Further, if the deallocation callback DOES block, you might stall the thread that executes all deallocation callbacks, which would lead to a memory leak
Solution (pt 1)

Do not use call_rcu to wait for each process reading an anon_vma_t to complete its read-side-critical section before freeing an anon_vma_t.

Instead of returning each anon_vma_t to the linux page allocator, return it to a freelist representing pages of memory allocated by the linux slab allocator, each page divided up into only instances of anon_vma_ts

When all the anon_vma_t’s on a page allocated by the slab allocator become free, the slab allocator will return the page to the linux page allocator, which could reallocate the page as a slab of a different type
Solution (pt 2)
Still, each read of an anon_vma_t must be protected by a read-side critical section

However, do not wait for all read-side critical sections to complete before returning memory to the freelist of pages for vma_ts.

Only wait for read-side critical sections to complete when a page of type safe memory contains no currently allocated objects and that page should be returned to the page allocator.
• This method can be enabled by setting a flag in the slab called SLAB_DESTROY_BY_RCU

• Since SLAB_DESTROY_BY_RCU does not wait for all reads to finish before returning an anon_vma_t to the freelist, rcu readers must check bits in the anon_vma_t data structure to check if the object has been returned to the freelist
Problems with Reader Writer Locks

• Costs more to acquire read-lock than to perform anything but a very long read-side critical section, resulting in no read concurrency in many applications

• In the best case, if the lock is in your cache, an atomic update to the read-count will take 50 times longer than a normal instruction

• In the worst case, if you were not the last processor to read (or write) and the lock is not in your cache, it will take 500 times longer than a normal instruction to update the read-count
• With the reader-writer lock, a writer must wait for all readers that successfully entered the critical section to complete the critical section before a writer can begin.

• Writers can be starved if there are a lot of readers.

• Conversely, if a writer is performing an update and there are lots of writers spinning to acquire the lock and only one reader spinning to acquire the lock, it is likely that the reader will be starved.
• Using “synchronize_rcu” will block, but only after the update has already occurred. If conventions are followed correctly (as below) RCU will never block before the update occurs (unlike rwlocks)

```
struct foo {
    int a;
};
struct foo *gp = NULL;
p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
rcu_assign_pointer(gp, p);
synchronize_rcu();
```

• The result is that updates become visible more quickly in most scenarios using RCU
Because all RCU readers overlap the updater, all readers could see updated value.

However, cannot as easily reason about when update takes effect and which readers will see the update.
• This increased performance comes at the cost of changing the semantics of reader-writer locking in ways that may be inappropriate for some applications

• With reader-writer locks, all readers that begin after a writer completes will see the writer’s update atomically, no matter how complex

• All readers will see the same object for the duration of their execution.
Example of RCU for RW locking

```c
pid_table_entry_t pid_table[];

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

void pid_remove(int pid)
{
    process_t **p;
    spin_lock(&pid_table[pid_hash(pid)].lock);
    p = &pid_table[pid_hash(pid)].process;
    rcu_assign_pointer(p, NULL);
    spin_unlock(&pid_table[pid_hash(pid)].lock);

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

process_t *pid_lookup(int pid)
{
    process_t *p
    rcu_read_lock();
    p = pid_table[pid_hash(pid)].process;
    if (p)
        atomic_inc(&p->ref);
    rcu_read_unlock();
    return p;
}
```
PID Hash Table discussion

• This uses a hybrid of reference counts as well as RCU, possibly for historical reasons

• We need RCU in addition to reference counts because a reference is acquired before increment its reference count

• Otherwise, there would be a race condition between a thread trying to delete the process and one looking it up and about to increment its reference count
Semantic Differences Example

The following 4 events happen concurrently

1. Writer 1 adds element A to one bucket in the PID hash table
2. Writer 2 adds element B to a different bucket in the PID hash table
3. Reader 1 looks for A then B, finds A but not B
4. Reader 2 looks for B then A, finds B but not A

No global ordering. Different reads see writes in different orders.

This is not possible with RW locks because reads and writes do not occur concurrently. Readers will either happen before both updates, between the two updates, or after both updates.
A Read Concurrent with Two Writes

R could see...
1. W1 and W2
2. Just W1
3. Just W2
4. Neither W1 nor W2
In the previous example, what could we do on the write-side to make sure that concurrent readers will see at most one update?
Semantic Differences From RW Locking

- Reader-Writer Locks forbid concurrency between readers and writers
- RCU allows concurrency between readers and writers
- This is sometimes problematic, but can be solved in 3 ways
  - Impose a Level of Indirection to Make Writes Atomic
  - Mark Obsolete Objects
  - Retry Readers
Modify Writes to Make Them Atomic

• If your code requires that all updates become visible to readers atomically, insert a level of indirection and use the publish-subscribe pattern to update a pointer.

• This works if the update you need is for one element to be added, one element to be deleted, or one element to be modified. As the next example shows, if you need to perform a more complex update, RCU may or may not work.
Atomic Move

- It is easy to make an insertion or a deletion in a linked list atomic.
- But can we move an element in a linked list atomically?
Move operation

• In this example, we must swap at least 2 pointers

• Depending on timing, we could see D once, twice, or not at all

• We could insert another level of indirection and copy the entire list, make the move on the copy and then swap the head pointer from the old list to the new list, but this would be very expensive

• If we had a 2CAS and could swap both pointers atomically, would this be a solution?
Move Operation

• If we had a 2CAS and could swap both pointers atomically, would this be a solution?
• No!
• When operations on a data structure become this complex, we may need to enforce mutual exclusion between readers and writers by using a RW lock
• Sometimes however, this move might work
• In a hash table, if we are performing a lookup, it would not be okay to not see D at all but it would be okay to see it either once or twice
Mark Obsolete Objects

• Sometimes readers should not be able to see old versions of objects
• To guarantee old objects are not accessed, when the object is being updated, the writer will mark a flag in the old version and readers will check this flag
• Depending on the circumstances, you will either want to abort the operation and return something indicating failure or retry the operation
Deleting a Semaphore

• In the System V semaphore implementation, when a reader looks up a semaphore in a hash table via a semaphore ID, if that semaphore has been deleted by another thread, the function should return a value indicating failure if the semaphore is marked for deletion.

• If instead, when a semaphore is deleted, the read-side function attempted to retry the operation, this would lead to an infinite loop in the read side, and so the read side critical section would never terminate, and because of this the write-side would wait indefinitely to delete the semaphore.

• Thus, when deleting, mark and abort.

• When updating, mark and retry.
Using Seqlocks to Prevent Reader/Writer Concurrency

• RCU can be used in combination with other approaches to achieve many different forms of read-write semantics

• A seqlock can be used to implement a version of RCU where a reader will retry if executing concurrently with a writer

• This is useful in cases where a data item is modified in-place
Linux Seqlocks

Can acquire in read mode or write mode.
Seqlock writer increments counter when it starts a write, increments again when it finishes a write. Value of seqlock is even if a writer is not currently in a critical section, odd otherwise.

Seqlock reader will spin waiting to enter the critical section if seqlock value is odd, begin otherwise. At the end of the read-side critical section a reader will again read the seqlock value. If the value when entering the critical section is not the same as the value upon leaving, retry critical section.

Performance: Read-side critical sections don’t block but could need to retry many times an a situation with frequent writes.
seqlock_t lock;
unsigned int seq;

while (1){
    seq = readseqbegin(&thelock);
    if (seq % 2 != 0) continue; // if seq is odd, retry
    rcu_read_lock();
    read_data();
    rcu_read_unlock();
    if (read_seqretry(&lock, seq) == 0) break; // 0 if no retry needed
}
References

Jonathan Walpole’s slides:


“A Comparison of Relativistic and Reader-Writer Locking Approaches to Shared Data Access” by Walpole, Howard, Triplett:

“The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux” by Guniguntala, McKenney, Triplett, Walpole
http://web.cecs.pdx.edu/~walpole/class/cs510/papers/IBMSysJ.pdf