CS510 Concurrent Systems

Jonathan Walpole
What is RCU, Fundamentally?

Paul McKenney and
Jonathan Walpole
The Problem

How can we know when it's safe to reclaim memory without paying too high a cost?
- especially in the read path
Possible Approaches

Reference counts

- increment count when a reference is taken, decrement count when reference is dropped
- requires multiple atomic read-modify-write instructions and memory barriers in each read section
- very slow and non-scalable for readers!
Possible Approaches

Hazard Pointers

- readers update their own hazard pointer list when taking and dropping references
- don’t need atomic read-modify-write instructions, just regular stores which are atomic if word sized and word aligned
- but we need at least two memory barriers, and potentially two per hazard pointer update
- memory barriers are expensive!
But How Could We Do Better?

Batching and amortizing costs

- we need to know that a reader is done, but we don’t need to know immediately!
  - delaying the communication allows multiple read sections to share the costs of a barrier
  - you pay by using more memory than necessary
- we need to know that no thread is using this item, but it may be cheaper to track read completion on a group or system wide basis, rather than per item
  - can be cheap to determine that no threads are reading
  - the cost is extra memory usage
RCU

RCU (read-copy update) is a collection of primitives for safely deferring memory reclamation in an efficient and scalable way
- but they can be used for much more than that!
Why Call It RCU?

The “copy” part of RCU comes from the use of multiple versions of an object
- writers perform updates by creating a new copy
- readers read from the old copy
- multiple versions (copies) enable readers and writers of the same data item to run concurrently
Immutability?

This sounds like an immutable data model
- But if we never update something in place how do new readers find the new version?

Pointers are mutable (i.e., updated in place)
- written using word size, word aligned stores, which are atomic on all current architectures
- but is atomicity enough?

RCU has a publish primitive for updating pointers
- it includes the necessary order-constraining instructions (memory barriers and compiler directives)
Write Side With a Reordering Problem

```c
struct foo {
    int a;
    int b;
    int c;
};
struct foo *gp = NULL;

/* . . . */
p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
gp = p;
```
Solution Using RCU

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

Does preventing reordering on the write side guarantee that the uninitialized data will not be visible to readers?

- read-side reordering must be prevented too!
Read Side With Several Problems

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

Solution Using RCU

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();
RCU Primitives

rcu_assign_pointer
  - prevents write side reordering that could break publishing

rcu_dereference
  - prevents read side reordering that could break publishing

rcu_read_lock and rcu_read_unlock
  - prevent memory reclamation
  - but do not prevent concurrent writing!
RCU-Based ADTs

Linux defines various ADTs including lists, RB trees and radix trees

- Supported operations include iterators that simplify programming
- Each ADT has a wide selection of operations to allow for different optimization cases (i.e., they are not that abstract)
- Each has support for RCU-based synchronization
Linux Lists
RCU Publish in List Operations

```c
struct foo {
    struct list_head list;
    int a;
    int b;
    int c;
};
LIST_HEAD(head);
/* . . . */
/* . . . */
p = kmalloc(sizeof(*p), GFP_KERNEL);
p->a = 1;
p->b = 2;
p->c = 3;
list_add_rcu(&p->list, &head);
```
RCU Subscribing in List Operations

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

<table>
<thead>
<tr>
<th></th>
<th>Publish</th>
<th>Retract</th>
<th>Subscribe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pointers</strong></td>
<td>rcu_assign_pointer()</td>
<td>rcu_assign_pointer(..., NULL)</td>
<td>rcu_dereference()</td>
</tr>
<tr>
<td><strong>Lists</strong></td>
<td>list_add_rcu()</td>
<td>list_del_rcu()</td>
<td>list_for_each_entry_rcu()</td>
</tr>
<tr>
<td></td>
<td>list_add_tail_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list_replace_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hlists</strong></td>
<td>hlist_add_after_rcu()</td>
<td>hlist_del_rcu()</td>
<td>hlist_for_each_entry_rcu()</td>
</tr>
<tr>
<td></td>
<td>hlist_add_before_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlist_add_head_rcu()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>hlist_replace_rcu()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deferring Reclamation

How long do we have to wait before it's safe to reclaim an old version?

- until all readers have finished reading?
  - no, that's too strong!
  - if new readers picked up a newer version we don't need to wait for them to finish
  - we just need to wait for readers who might be reading the version we want to reclaim
Deferring for a Grace Period

[Diagrame showing the process of deferring for a grace period with stages labeled Reader, Removal, Grace Period, and Reclamation. The grace period extends as needed.]
RCU Primitives for Deferring

Does a writer need to wait while reclamation is deferred?

RCU provides synchronous and asynchronous primitives for deferring an action (typically memory reclamation)

- synchronize_rcu
- call_rcu
Example – synchronize_rcu

```c
struct foo {
    struct list_head list;
    int a;
    int b;
    int c;
};
LIST_HEAD(head);

/* . . . */
p = search(head, key);
if (p == NULL) {
    /* Take appropriate action, unlock, and return. */
}
q = kmalloc(sizeof(*p), GFP_KERNEL);
*q = *p;
q->b = 2;
q->c = 3;
list_replace_rcu(&p->list, &q->list);
synchronize_rcu();
kfree(p);
```
RCU-based List Deletion

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

Initial State of List
After list\_del\_rcu()
After `synchronize_rcu`
After Free
Replacing a List Element

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);`
Initial State
Allocation of New Element
Partial Initialization of New Element
Partial Initialization
Initialization Complete
After Publishing
After synchronize_rcu
After Free
How Does RCU Know When Its Safe?

Could use reference counts or hazard pointers, but that’s too expensive in the read path.

RCU batches and amortizes costs:
- different implementations of RCU make different choices, even within the Linux kernel.
- we’ll look at some examples.
Non-Preemptable Kernel Example

Basic rule:
- Readers must not give up the CPU (yield or sleep) during a read side critical section
RCU read side primitives need do nothing!
- so long as barriers enforced by act of yielding or sleeping!

Synchronize RCU can be as simple as:

1. for_each_online_cpu(cpu)
2. run_on(cpu);

Why?
Preemptable Kernels?

RCU read side primitives must disable and re-enable preemption
- do such actions have to include a barrier?

What if we’re running untrusted, i.e., at user level?
- we’re preemptable and we can’t disable it!
- could we apply same techniques in a thread library?
User Level RCU Implementations

Approach 1:
- reader threads signal quiescent states explicitly (i.e., periodically call into the RCU)
- RCU keeps track
- a quiescent state occurs between read sections
- frequency of calling quiescent states determines trade-off between read side overhead and memory overhead
  - need not be once per read section!
User Level RCU Implementations

Approach 2
- readers have a flag to indicate active reading
- RCU maintains a global counter of grace periods
- readers copy counter value at start of read section
- writers advance value of global counter
- synchronize_rcu updates the counter and waits until all reader threads are either not in a read section or have advanced beyond the old value
- like batched/amortized hazard pointers!
Problems with Approach 2

It is susceptible to counter overflow using 32 bit counters (its ok with 64 bit counters on current architectures)

overflow problem is fixed by an approach that uses “phases” instead of counting
  - if a reader is observed to go through two phase transitions it can not possibly be reading the data to be deleted
Memory Barrier Optimization

Both approaches 1 and 2 (and the fix) require memory barriers in the read-side primitives
- these are expensive!

They can be removed/amortized so long as the writer knows each reading thread will have executed a memory barrier before collection occurs
- this can be forced, by the writer, by sending signals to all the active reading threads and waiting for an acknowledgement
- signal handling includes a memory barrier
Summary

We have mechanisms for concurrent reading and writing
They let us safely reclaim memory
There are many different ways to implement them, making different trade-offs
- which is most appropriate for your situation?