CS510 Advanced Topics in Concurrency

Jonathan Walpole

Portland State University

Read-Copy Update (RCU)

The Problem

How can we know when its 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

```
    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

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

rcu_assign_pointer()

- is opaque to the compiler
- contains the appropriate memory barrier to prevent hardware reordering
- and *then* does the assignment

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

```
p = gp;
if (p != NULL) {
    do_something_with(p->a, p->b, p->c);
}
```

Problems:

- What if p is concurrently freed?
- Some architectures (DEC Alpha) can reorder lines 1-3

Solution Using RCU

```
rcu_read_lock();
p = rcu_dereference(gp);
if (p != NULL) {
	do_something_with(p->a, p->b, p->c);
}
rcu_read_unlock();
```

What Do These Statements Do?

rcu_read_lock and rcu_read_unlock delimit a kind of critical section

- it excludes the rcu garbage collector
- objects can not be freed
- does not exclude concurrent writers!

rcu_dereference()

- reads the pointer, then executes whatever memory barrier is necessary to prevent reordering

Portland State UNIVERSITY

Summary of 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

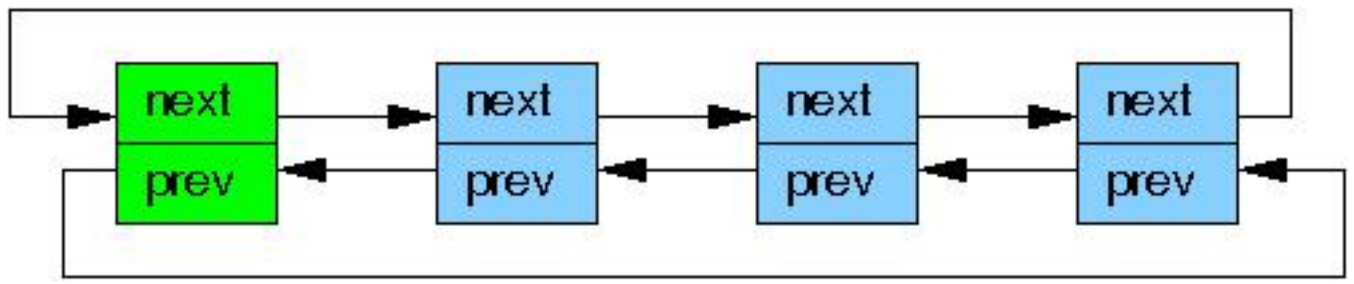
- prevent memory reclamation
- but do not prevent concurrent writing!

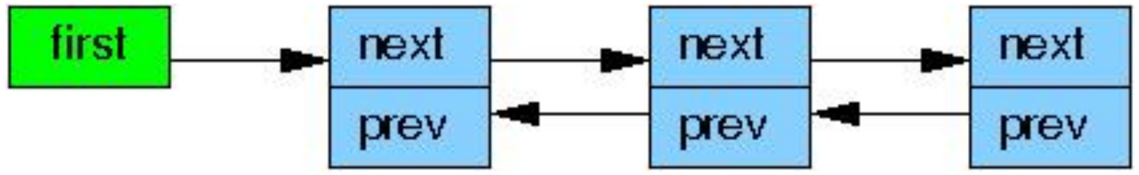
RCU-Based ADTs

Linux defines various ADTs including lists, hash tables, RB trees, 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


next
prev
next
prev
next
prev
next
prev


first
next
prev
next
prev
next
prev

RCU Primitives

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

RCU Publish in List Operations

```
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 Reading in List Operations

```
rcu_read_lock();
list_for_each_entry_rcu(p, head, list) {
        do_something_with(p->a, p->b, p->c);
 }
 rcu_read_unlock();
```

Portland State UNIVERSITY

Deletion (and freeing)

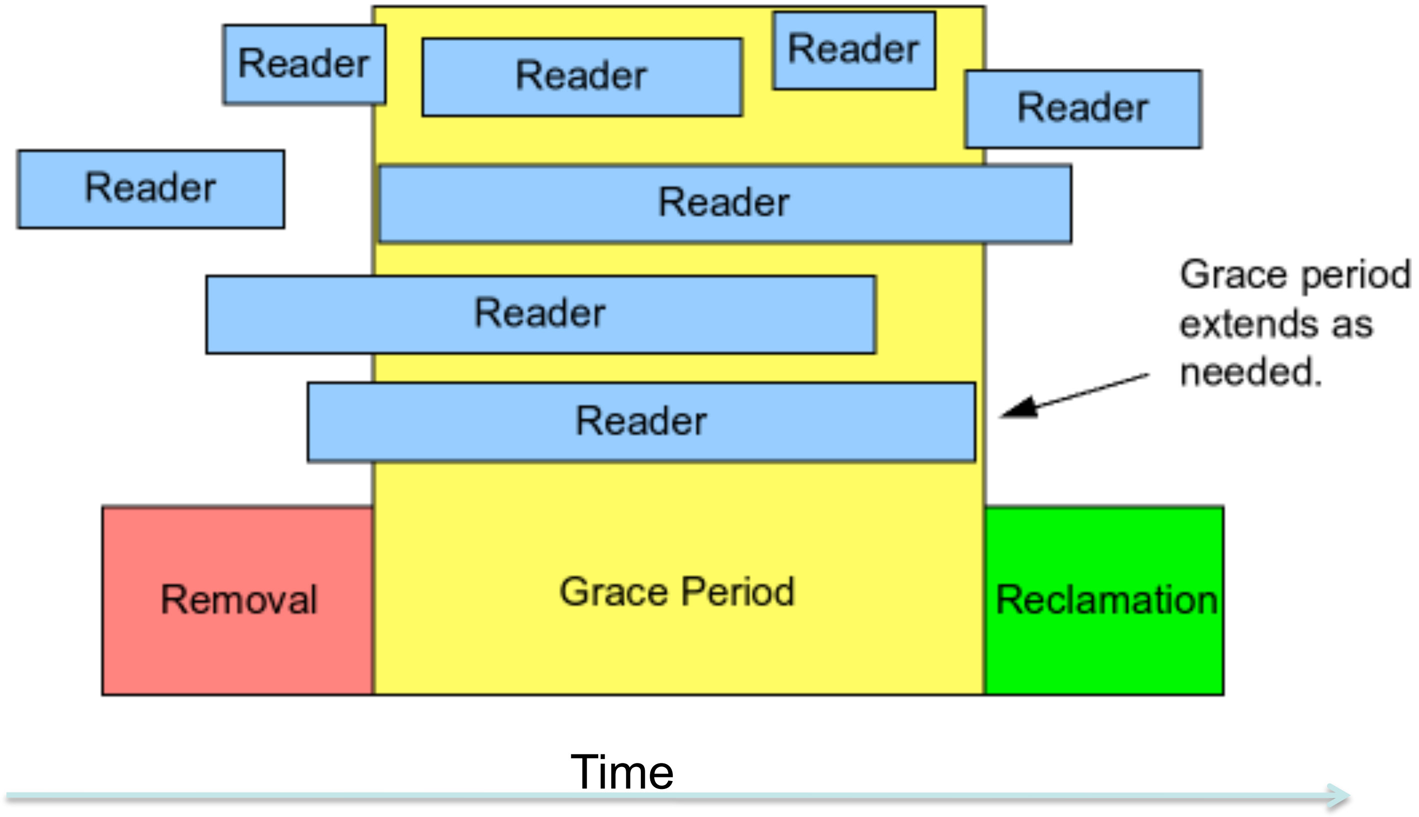
```
p = search(head, key);
if (p != NULL) {
        list_del_rcu(&p->list);
        synchronize_rcu();
        kfree(p);
}
```

Deferring Reclamation

How long do we have to wait before its safe to reclaim (free) an object?

- 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



RCU Primitives for Deferring

Does a writer need to wait while reclamation is deferred?

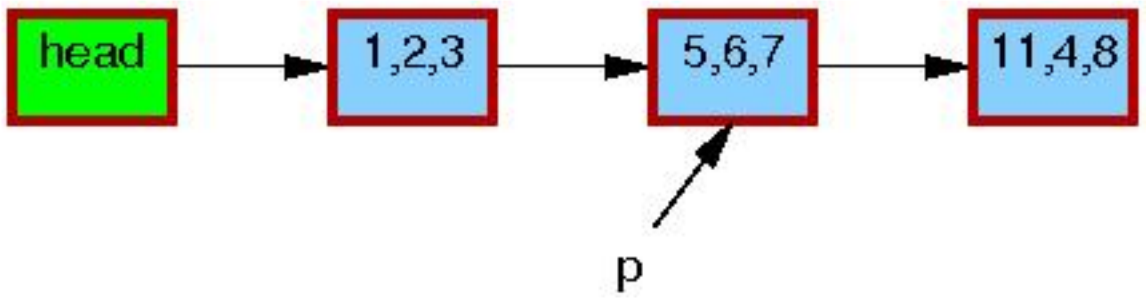
RCU provides synchronous and asynchronous primitives for deferring an action (typically memory reclamation)

- synchronize_rcu
- call_rcu

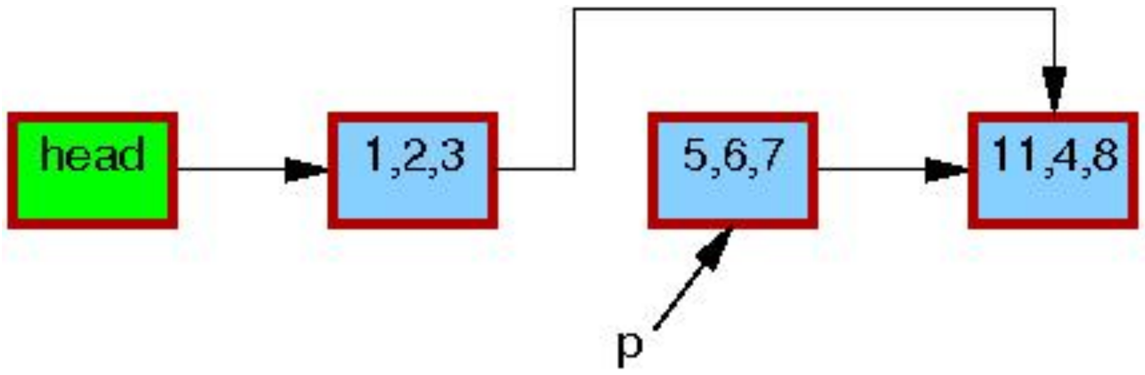
Example – synchronize_rcu

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

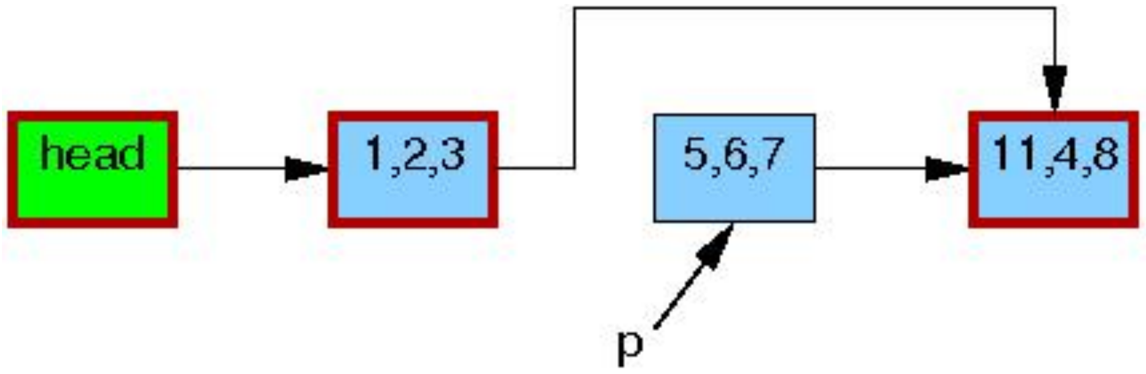
Initial State of List



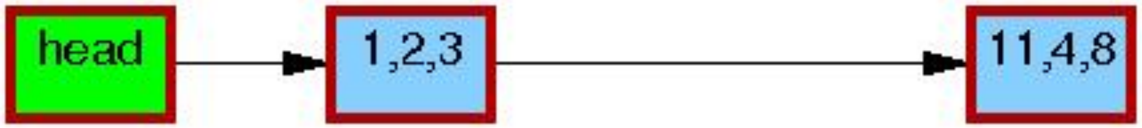
After list_del_rcu()



After synchronize_rcu



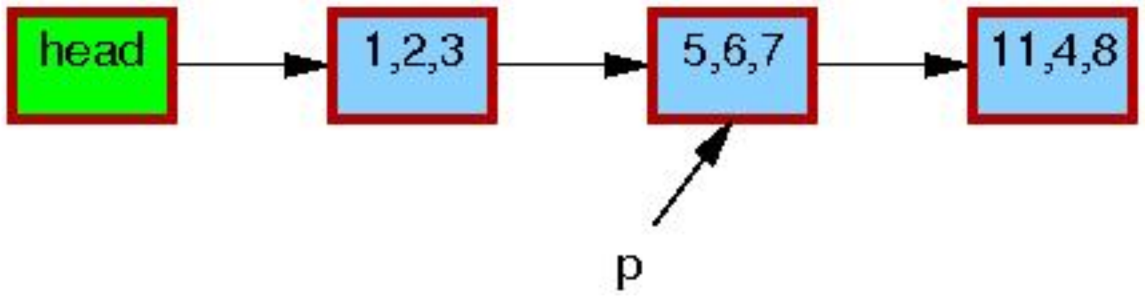
After Free



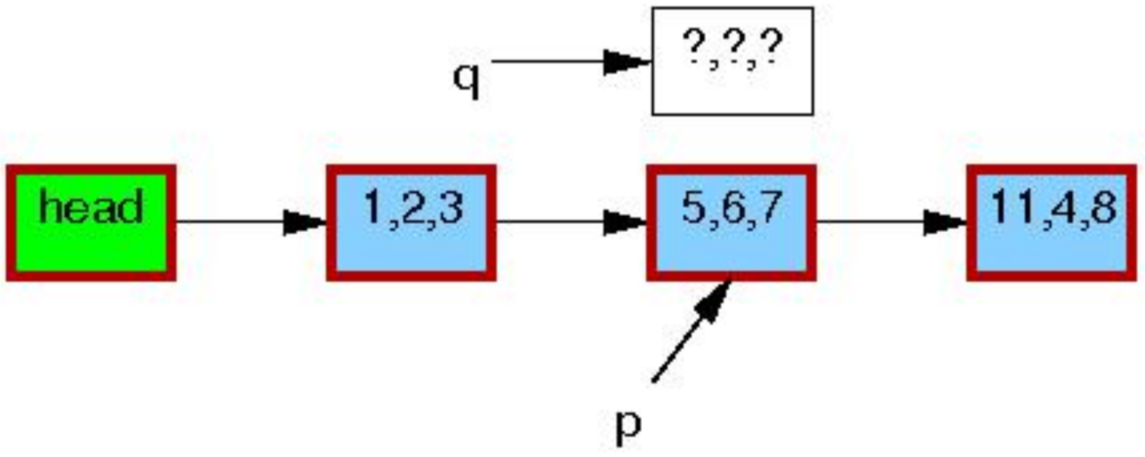
Replacing a List Element

```
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);
```

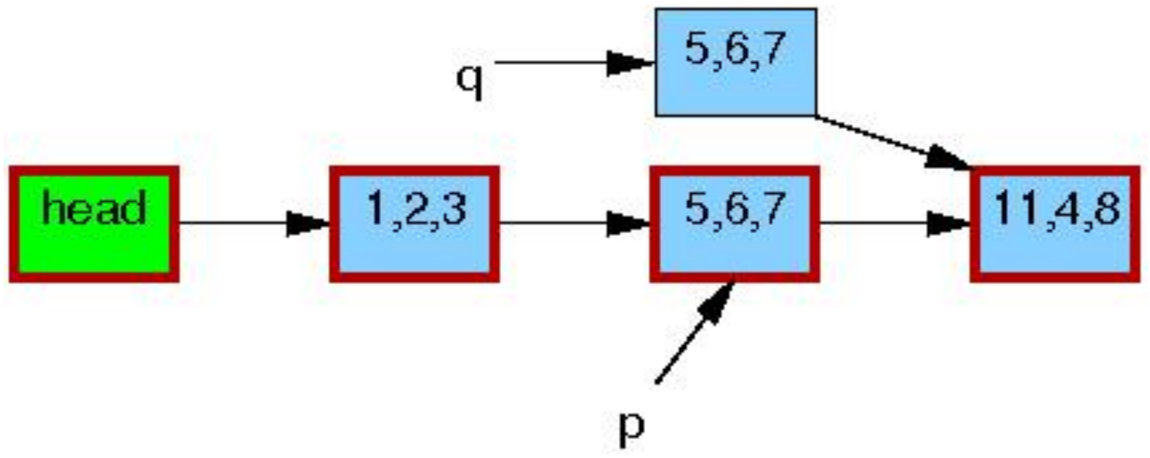
Initial State



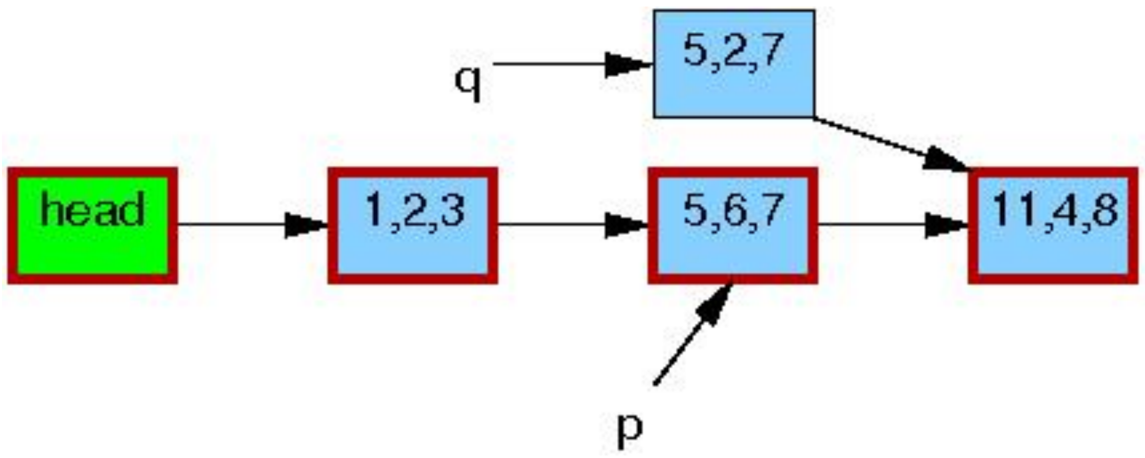
Allocation of New Element



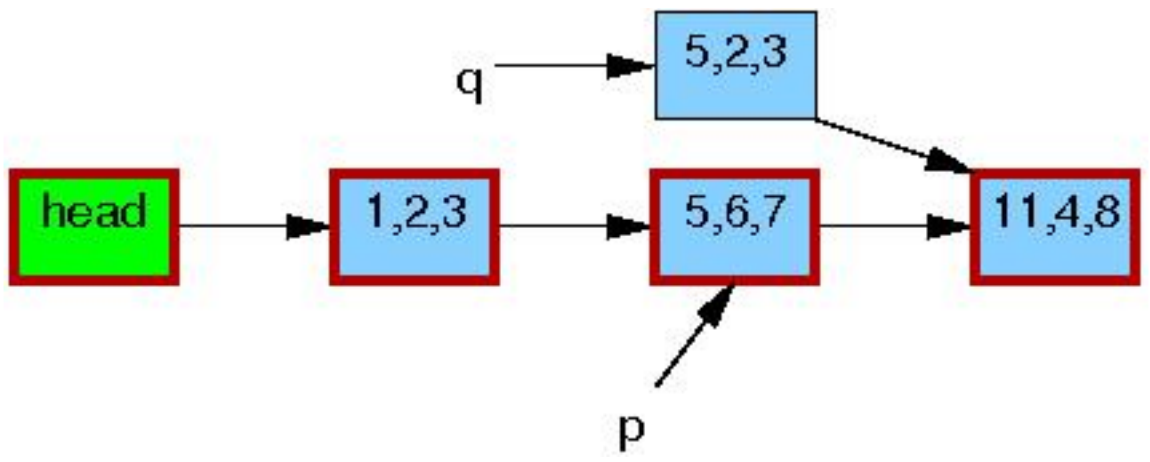
Partial Initialization of New Element



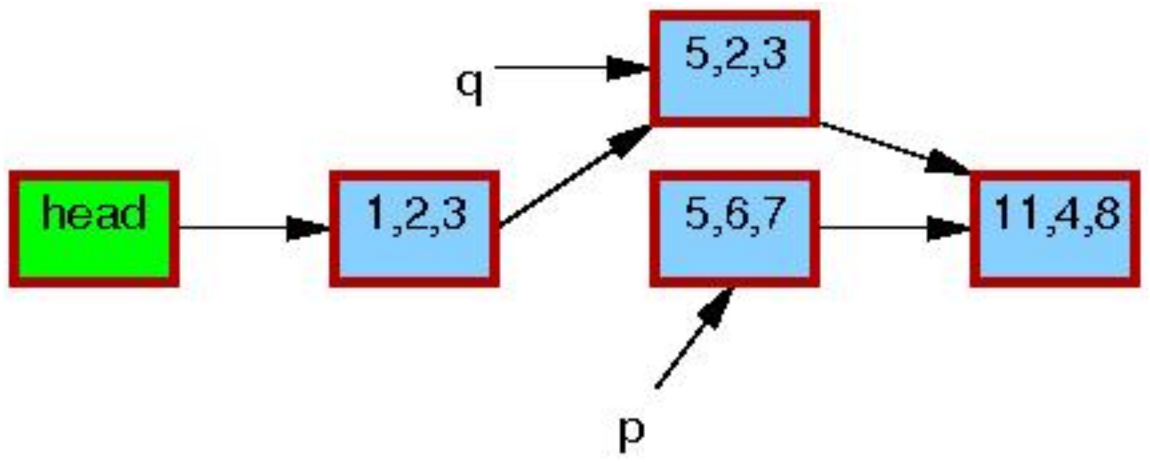
Partial Initialization



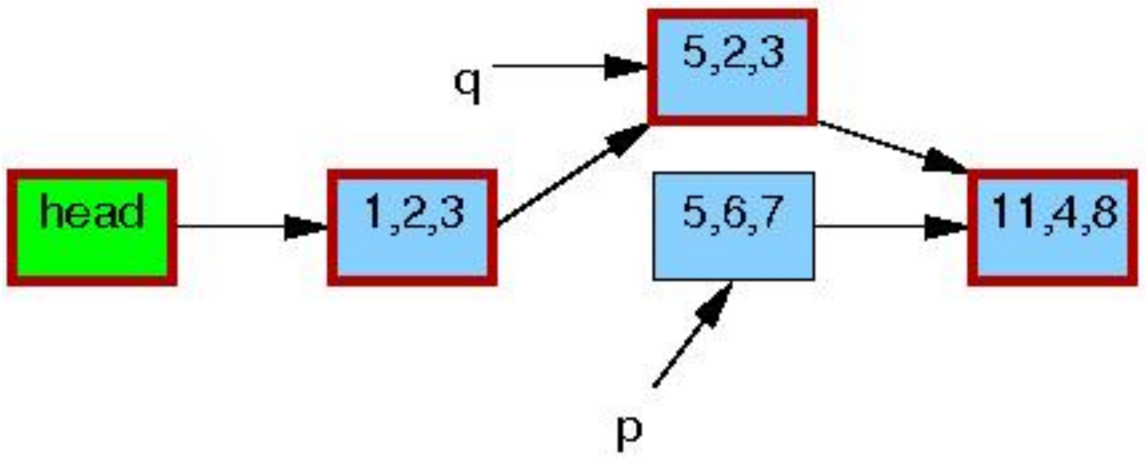
Initialization Complete



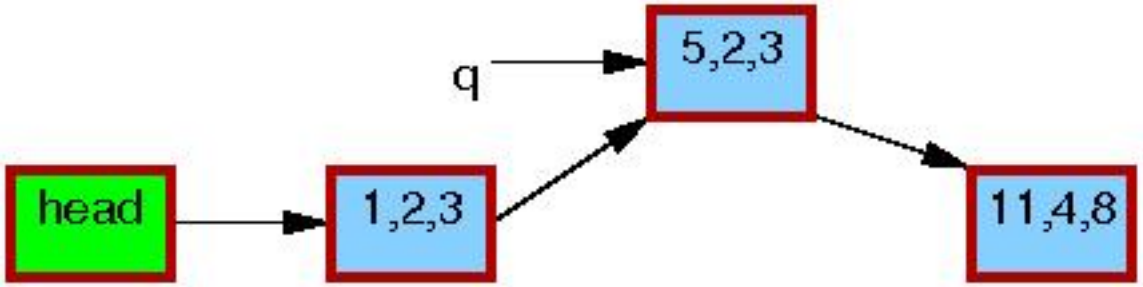
After Publishing



After synchronize_rcu



After Free



How Does RCU Know When Its Safe?

Could use reference counts or hazard pointers, but that's expensive, especially in the read path

RCU batches work 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:

```
for_each_online_cpu(cpu)
    run_on(cpu);
```

Why?

The Key Idea

Writer removes last global pointer to object

- it is now unreachable by new readers
- .. but it might still be in use by existing readers

Now writer waits until a context switch has been observed on all CPUs, before freeing

- if readers guarantee not to allow a context switch while in a read critical section
- then no reader that was active at the start of this wait will still be active at the end!

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 of all threads states
- 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 object or read section!
  - but they do need a memory barrier!

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
- 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!

Similarities to Hazard Pointers

It requires readers to write something locally

It requires communication between the readers and the thread waiting to free

But readers don’t have to write for every object they read

Memory barriers needed per read section, not per object

So, its cheaper for readers

But more memory is tied up, and for longer

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?

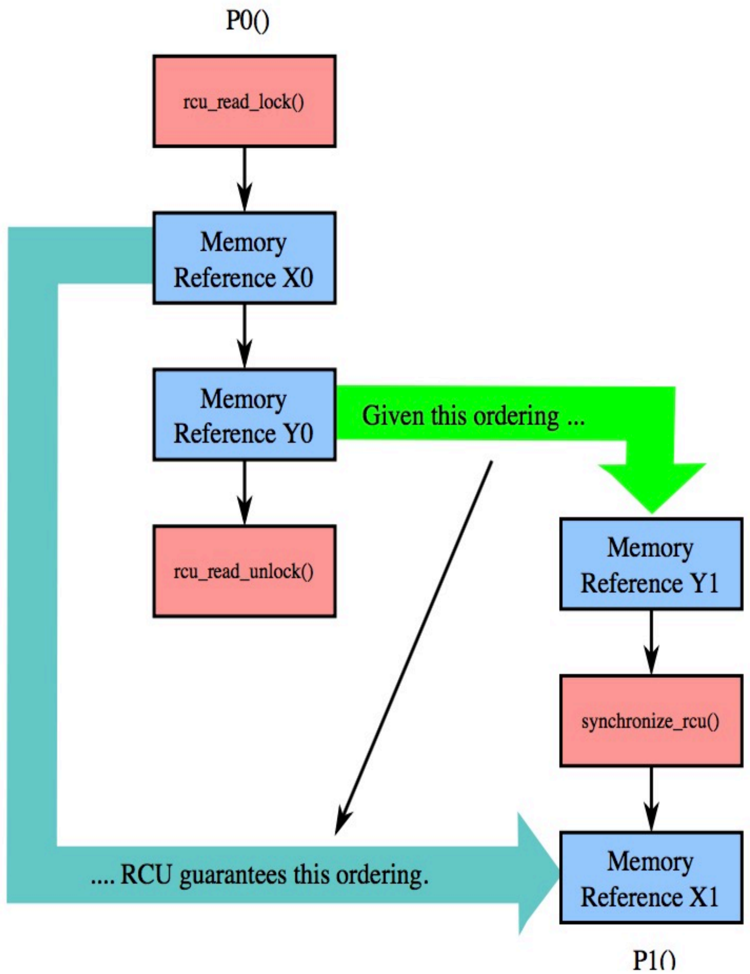
RCU Ordering Guarantees

RCU is useful for much more than deferred memory reclamation!

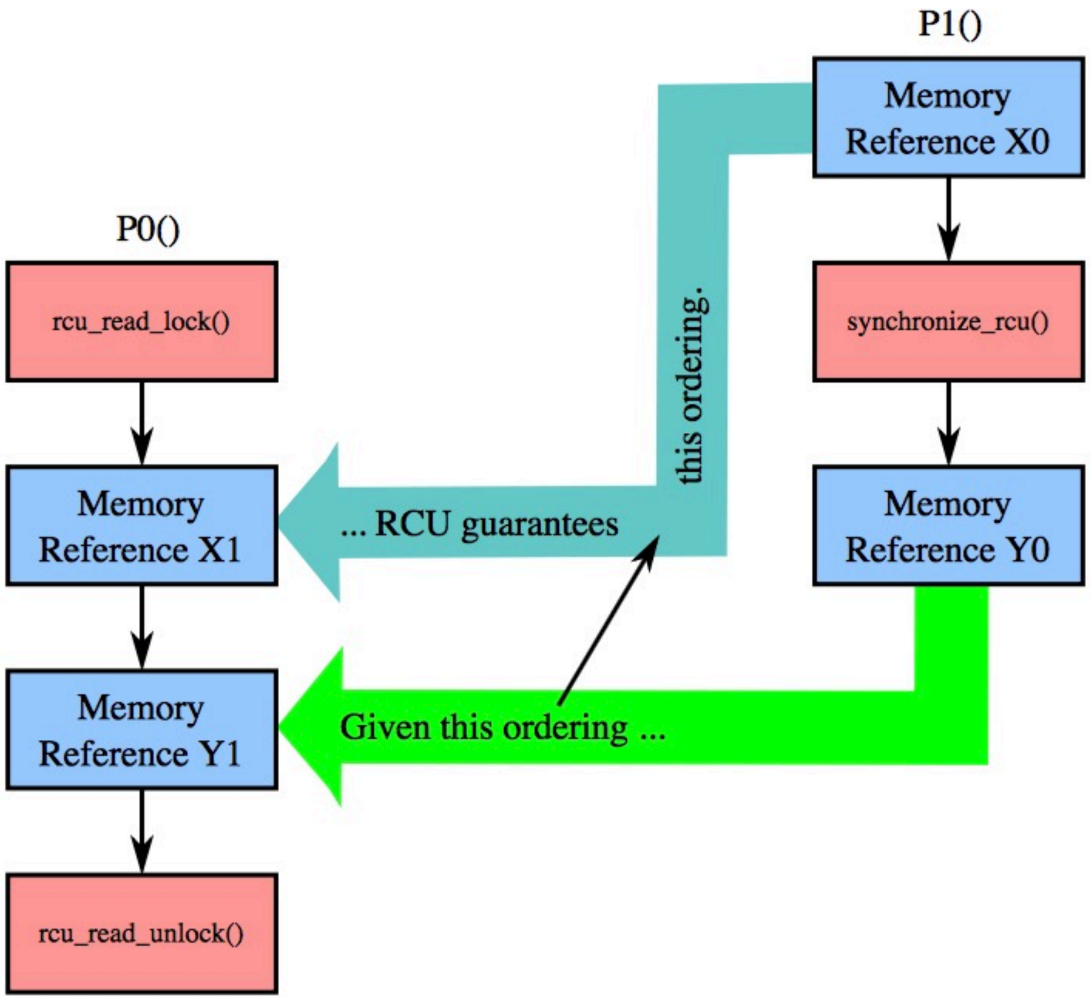
Synchronize RCU gives a useful ordering guarantee

- Readers will not disagree about the order of writes that are separated by a synchronize_rcu wait (a grace period)
- Can use this to gain sequentially consistent ordering

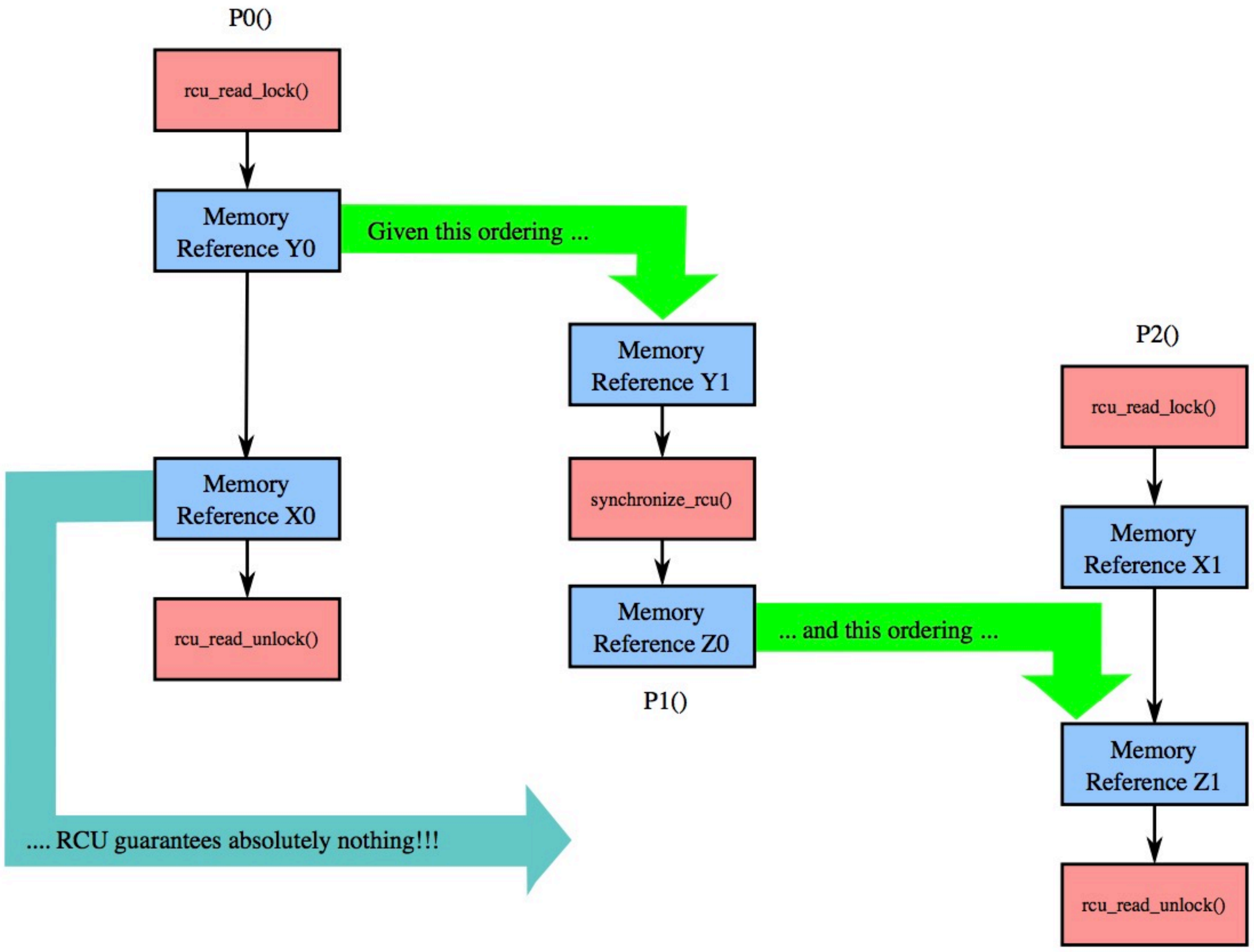
RCU Ordering Guarantees



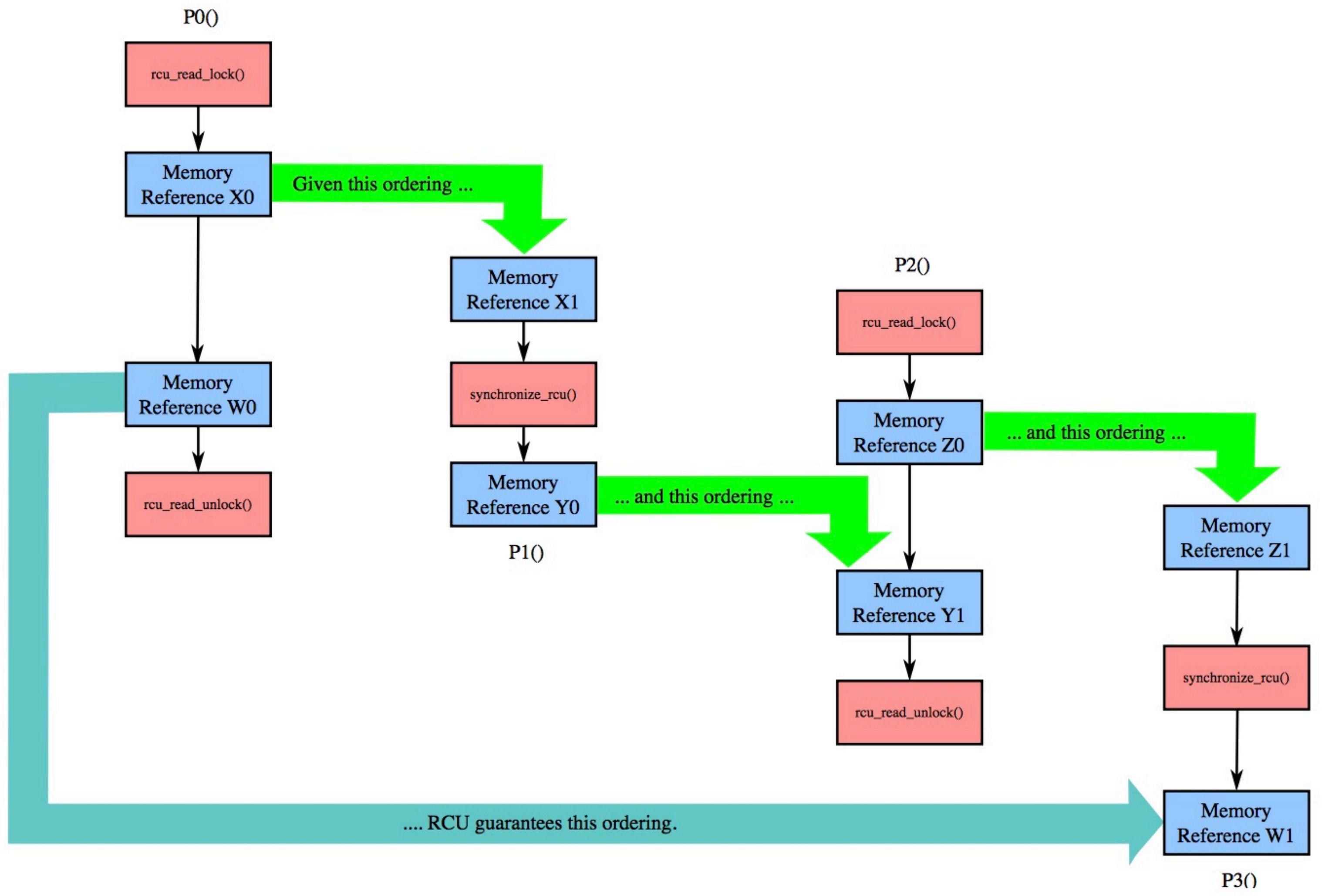
RCU Ordering Guarantees



RCU Ordering Guarantees



RCU Ordering Guarantees



Linux Kernel Memory Model

LKMM and tools (herd) understand RCU and can check your code

Automated checking is useful, especially for complex examples … even for experts

Basic Rules for RCU Usage

- Use rcu_read_lock() and rcu_read_unlock() to guard RCU read-side critical sections.
- Within an RCU read-side critical section, use rcu_dereference() to dereference RCU-protected pointers.
- Use some solid scheme (such as locks or semaphores) to keep concurrent updates from interfering with each other.

Basic Rules for RCU Usage

- Use rcu_assign_pointer() to update an RCU-protected pointer. Protects concurrent readers from the updater, *not* concurrent updates from each other!
- Use synchronize_rcu() *after* removing a data element from an RCU-protected data structure, but -before- reclaiming/freeing the data element, in order to wait for the completion of all RCU read-side critical sections that might be referencing that data item.