第 14 讲: Concurrency in OS Kernel

第三节：Scalable Concurrency – RCU

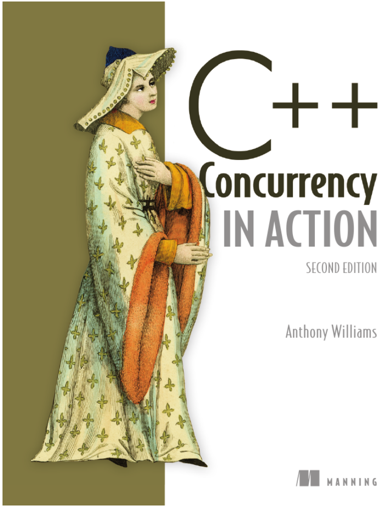
陈渝

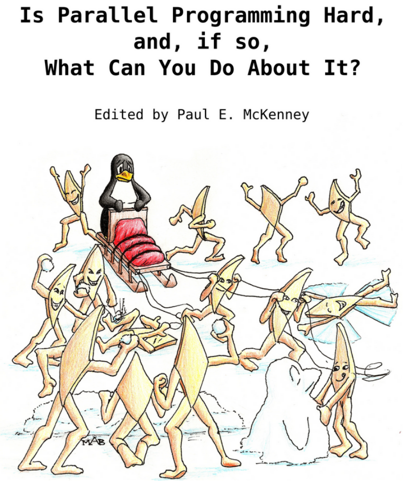
清华大学计算机系

yuchen@tsinghua.edu.cn

2020 年 5 月 18 日

Resource





Reference:
”Is Parallel Programming Hard, And, If So, What Can You Do About It?”,Paul McKenney;
”C++Concurrency in Action”, ANTHONY WILLIAMS;
”CS510 - Advanced Topics in Concurrency”, Jonathan Walpole; Adam Belay from MIT PDOS

Motivation

- Modern CPUs are predominantly multicore
- Applications rely heavily on kernel for networking, filesystem, etc.
- If kernel can’ t scale across many cores, applications that rely on it won’ t scale either
- Have to be able to execute system calls in parallel

Problem is sharing

- OS maintains many data structures
- They depend on locks to maintain invariants
- Applications may contend on locks, limiting scalability

Read-heavy data structures

Kernels often have data that is read much more often than it is modified

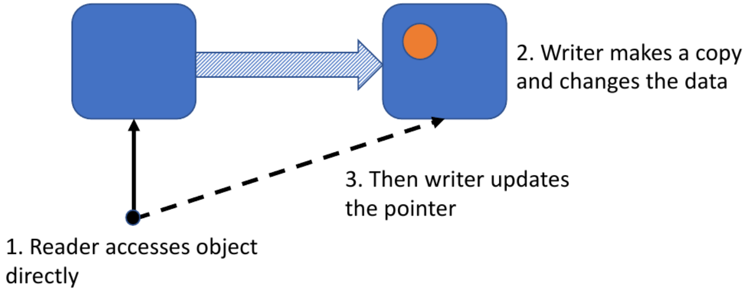
- Network tables: routing, ARP
- File descriptor arrays
- system call state

Goals

- Concurrent reads even during updates
- Low space overhead
- Low execution overhead

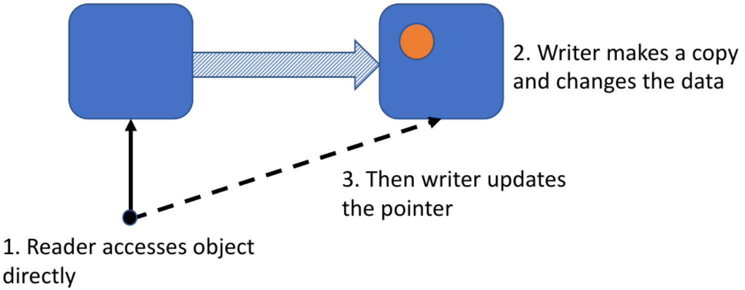
Idea: Read-copy update (RCU)

- Readers just access objects directly (no locks)
- Writers make a copy of object, change it, then update the pointer to the new copy



Three fundamental mechanisms

- Publish-Subscribe Mechanism (for insertion)
- Wait For Pre-Existing RCU Readers to Complete (for deletion)
- Maintain Multiple Versions of Recently Updated Objects (for readers)



RCU – Publish-Subscribe Mechanism

One key attribute of RCU is the ability to safely scan data, even though that data is being modified concurrently.

```
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;
```

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.

```
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;
rcu_assign_pointer(gp, p);
```

Three fundamental mechanisms

- Publish-Subscribe Mechanism (for insertion)
- Wait For Pre-Existing RCU Readers to Complete (for deletion)
- Maintain Multiple Versions of Recently Updated Objects (for readers)

RCU – Publish-Subscribe Mechanism

However, it is not sufficient to only enforce ordering at the updater, as the reader must enforce proper ordering as well.

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

the DEC Alpha CPU 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!

RCU – Publish-Subscribe Mechanism

The rcu_dereference() primitive uses whatever memory-barrier instructions and compiler directives are required for this purpose:

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

RCU – Publish-Subscribe Mechanism

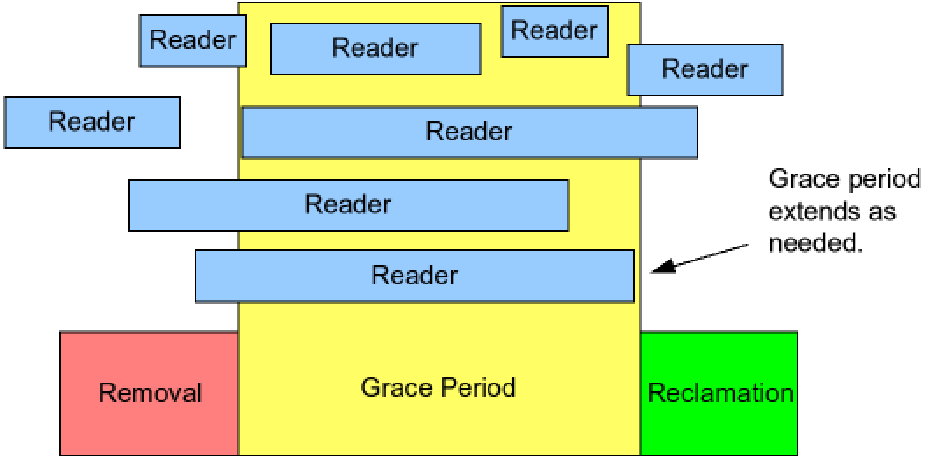
Category	Publish	Retract	Subscribe
Pointers	`rcu_assign_pointer()`	`rcu_assign_pointer(..., NULL)`	`rcu_dereference()`

Lists	`list_add_rcu()`
	`list_add_tail_rcu()`
	`list_replace_rcu()`
		`list_del_rcu()`	`list_for_each_entry_rcu()`

Hlists	`hlist_add_after_rcu()`
	`hlist_add_before_rcu()`
	`hlist_add_head_rcu()`
	`hlist_replace_rcu()`
		`hlist_del_rcu()`	`hlist_for_each_entry_rcu()`

RCU – Wait For Pre-Existing RCU Readers to Complete

In its most basic form, RCU is a way of waiting for things to finish.



RCU to wait for readers:

- Make a change, for example, replace an element in a linked list.
- 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.
- Clean up, for example, free the element that was replaced above.

RCU – Wait For Pre-Existing RCU Readers to Complete

```
1 struct foo {
2   struct list_head list;
3   int a, b, c;
6 };
7 LIST_HEAD(head); ...
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; q->b = 2;  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.

RCU – Wait For Pre-Existing RCU Readers to Complete

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

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

RCU – Wait For Pre-Existing RCU Readers to Complete

RCU readers:

- 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!
- RCU Classic read-side critical sections delimited by rcu_read_lock() and rcu_read_unlock() are not permitted to block or sleep.

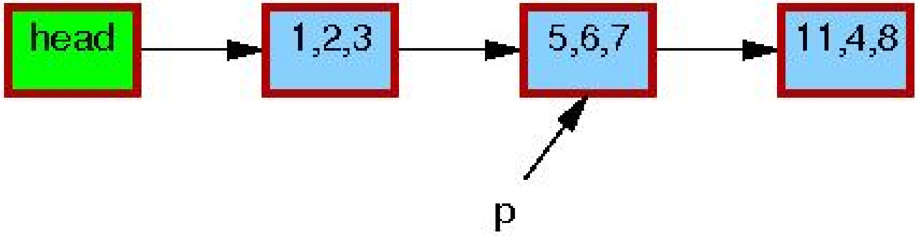
RCU – Maintain Multiple Versions of Recently Updated Objects

Example 1: Maintaining Multiple Versions During Deletion

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

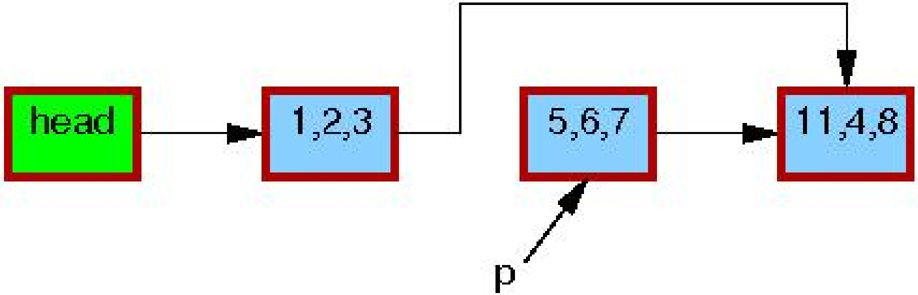
Example 1: Maintaining Multiple Versions During Deletion

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



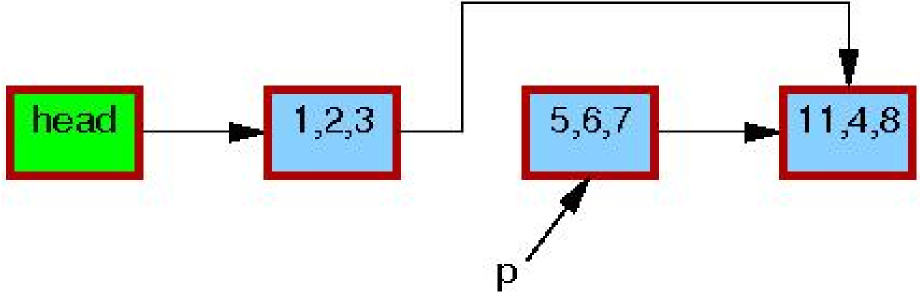
Example 1: Maintaining Multiple Versions During Deletion

After the list_del_rcu() on line 3 has completed, the 5,6,7 element has been removed from the list, as shown below.



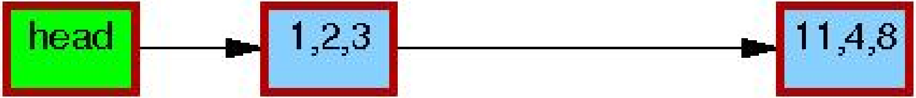
Example 1: Maintaining Multiple Versions During Deletion

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.



Example 1: Maintaining Multiple Versions During Deletion

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

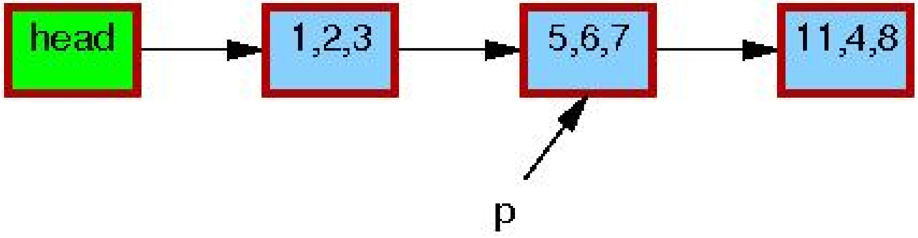


Example 2: Maintaining Multiple Versions During Replacement

```
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);
```

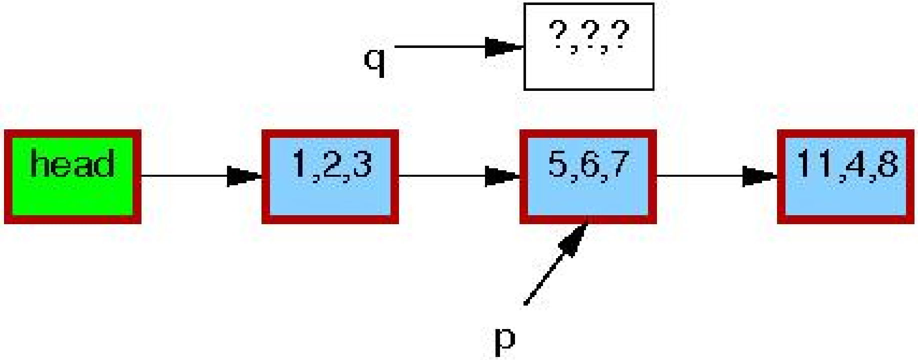
Example 2: Maintaining Multiple Versions During Replacement

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



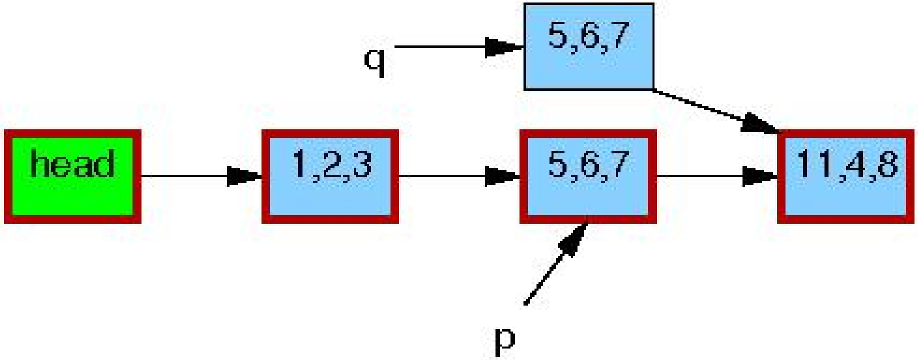
Example 2: Maintaining Multiple Versions During Replacement

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



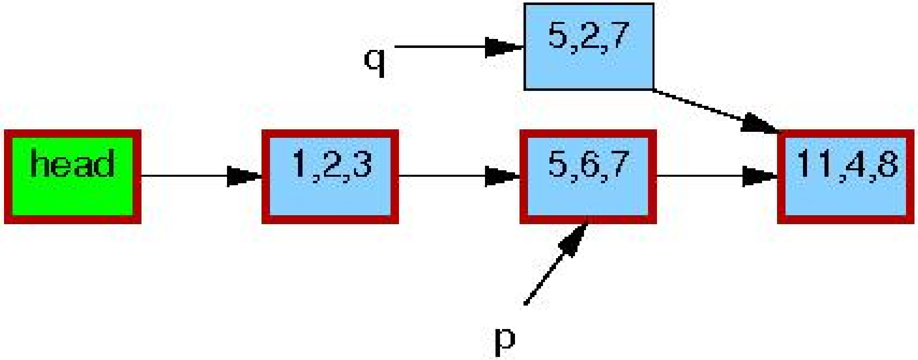
Example 2: Maintaining Multiple Versions During Replacement

Line 2 copies the old element to the new one:



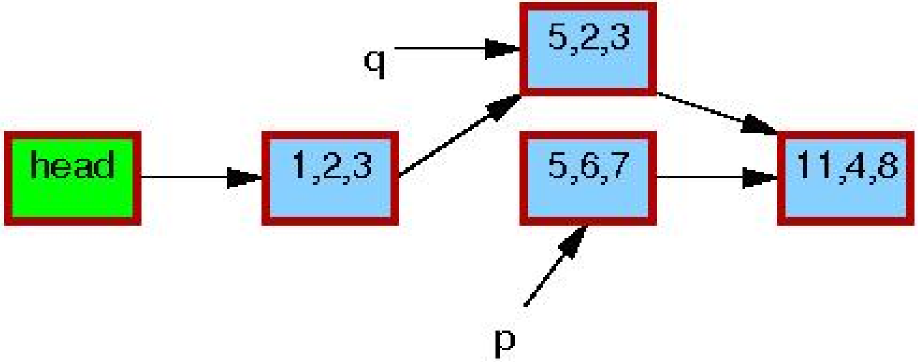
Example 2: Maintaining Multiple Versions During Replacement

Line 3 updates q->b to the value ”2”:



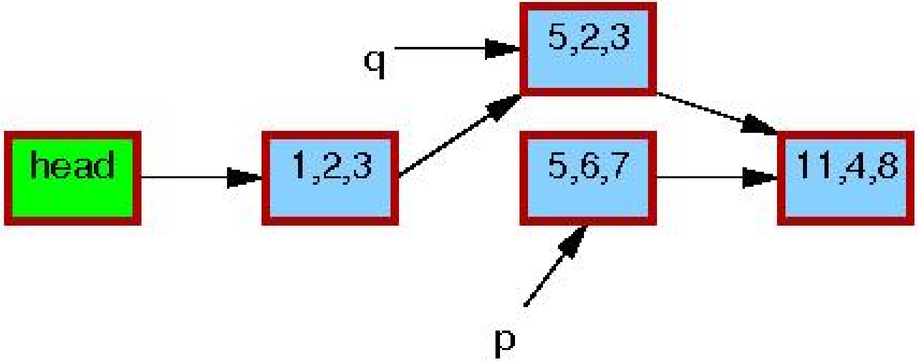
Example 2: Maintaining Multiple Versions During Replacement

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.



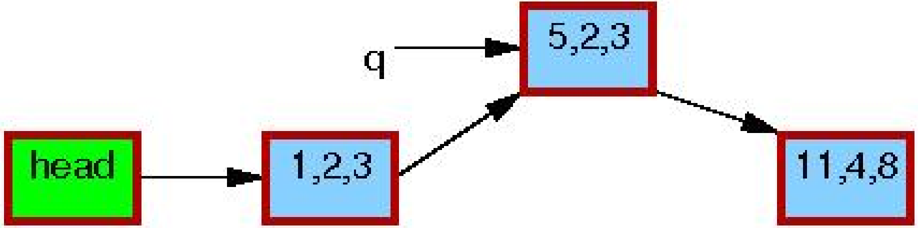
Example 2: Maintaining Multiple Versions During Replacement

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.



Example 2: Maintaining Multiple Versions During Replacement

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



Conclusion

- RCU enables zero-cost read-only access at the expense of slightly more expensive updates
- Very useful for read-mostly data (extremely common in kernels)