What is RCU, Fundamentally?

By: Paul E. McKenney
Jonathan Walpole

Presenter: Dany Madden
Agenda

- Review: What is the problem?
- Authors Background
- What is RCU?
- RCU Publish & Subscribe
- Wait For Pre-Existing RCU Readers to Complete
- RCU Deletion & Replacement
- Conclusion
Review

- **Spinlock:**
  - Solved critical section. No concurrency.
  - Freeing old object is trivial.

- **Non-blocking:**
  - Only one thread will succeed.
    - CAS caused ABA problem. LL/SC fixed it.
    - Freeing old object can be done with hazard pointers.

- **Reader-writer locks**
  - Atomic operation to acquire the read lock prevents concurrent reads.
  - One writer, with no reader presence. Write is expensive!

- **Compiler and hardware optimization**
The Problem

• How to increase concurrency and safely and efficiently reclaim unused memory?!
A Possible Solution

• Read Copy Update
  – Readers can continue reading while an update is in progress.
    • More concurrency than the reader-writer lock
  – Freeing unused memory is straightforward in non-CONFIG_PREEMPT kernels.
    • Is it less overhead than using hazard pointers?
Authors Background

- **Jonathan Walpole**
  - Professor at PSU
  - Research Interests: OS, Parallel and Distributed Systems
  - Paul Mckenney's PhD Thesis Advisor

- **Paul Mckenney**
  - One of the RCU inventors, RCU Maintainer for the Linux Kernel
  - Distinguished Engineer at IBM, Linux Technology Center
  - Worked on shared-memory and parallel computing for over 20 years, real-time linux, networking research, sys admin, and university business application.
What is RCU

• Publishing of new data
• Subscribing to the current version of data
• Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  – Each reader continues traversing its copy of the data while a new copy might be being created concurrently by each updater.
    • Hence the name Read-Copy Update, or RCU
  – Once all pre-existing RCU readers are done with them, old versions of the data may be discarded, (free().)
RCU Publish - Subscribe

• Code re-ordering Background

• Original code:
  ```c
  p = malloc (sizeof (*p));
  p->a = 1;
  p->b = 2;
  p->c = 3;
  gp = p;
  ```

• code with a mischievous compiler and cpu:
  ```c
  p = malloc (sizeof (*p));
  gp = p;
  p->a = 1;
  p->b = 2;
  p->c = 3;
  ```

What happen when gp = p is executed before the fields assignments?
** Publish mechanism:** When a memory location is updated it forces the CPU and the compiler to execute pointer assignment and object initialization in the right order using rcu_assign_pointers().

```
1 p->a = 1;
2 p->b = 2;
3 p->c = 3;
4 gp = p;
```

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

• How does rcu_assign_pointer() ensure the execution order?
• Forcing order on the writer isn't enough. Readers must do the same

• Consider this:

  • Original code
    
    ```c
    p = gp;
    if (p != NULL)
      do-something-with (p->a, p->b, p->c);
    ```

  • Code with a mischievous compiler and CPU:
    
    ```c
    retry:
    p = guess (gp);
    if (p != NULL)
      do-something-with (p->a, p->b, p->c);
    if (p != gp)
      goto retry;
    ```

RCU Publish - Subscribe

- **Subscribe** mechanism: Reader uses `rcu_dereference()` to read a value of a specified pointer, ensuring that it see any initialization that occurred before the corresponding `rcu_assign_pointer()`. How exactly?

- The `rcu_dereference()` uses memory barrier (on DEC Alpha) and compiler directives to tell the cpu and compiler to fetch values in the right order.

- `rcu_dereference()` must be enclosed in `rcu_read_lock()` and `rcu_read_unlock()` to mark the reader-side critical section. More on this later...

```c
1 p = gp;
2 if (p != NULL) {
3   do_something_with(p->a, p->b, p->c);
4 }
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();
```
The list* and hlist* are higher constructs, build from rcu_assign_pointer() and rcu_deference() primitives.

When is it safe to do *replace_rcu() or *del_rcu()?

Reclaiming memory is necessary to avoid memory exhaustion (because RCU maintains multiple copies of the shared object.)
Wait for Pre-Existing RCU Readers to Complete

- **RCU is a way to wait for things to finish without explicitly tracking them.**
  - Why would it want to wait for readers to complete?
  - How does it wait without tracking them?

- Use **RCU read-side critical section**
  - Start with `rcu_read_lock()`, end with `rcu_read_unlock()`.
  - Critical section can be nested.
    - Must not block or sleep. How do we ensure this?
    - “SRCU” permits general sleeping. Outside the scope of this presentation.
1) Make a change, ie: replace an an element in a linked list
2) Wait for all pre-existing RCU readers critical sections to completely finish with `synchronize_rcu()`.
3) Clean up, ie: free the element that was replaced above.
Wait for Pre-Existing RCU Readers to Complete

1 struct foo {
2     struct list_head list;
3     int a;
4     int b;
5     int c;
6 };
7 LIST_HEAD(head);
8 /* . . . */
9
10 p = search(head, key);
11 if (p == NULL) {
12     /* Take appropriate action, unlock, and return. */
13 }
14 }
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);

- Must be synchronized with another update thread.
  - Where would you put a lock?
- Or ... have this be the only thread that can update.
- While allowing concurrent reads, line 16 copies and line 17-19 do an update.
- synchronize_rcu() waits for pre-existing RCU readers to complete. How?
Wait for Pre-Existing RCU Readers to Complete

• RCU Classic read-side critical sections are not permitted to be blocked or sleep.
  - When a CPU execute a context switch, a prior RCU read-side critical section has completed.
  - When each CPU does a context switch, all prior RCU read-side critical sections are guaranteed to have completed. synchronize_rcu() can safely return.

• Context switch works for non-CONFIG_PREEMPT

• CONFIG_PREEMPT and -rt kernels use a different approach, which is outside the scope of this presentation.
Maintain Multiple Versions of Recently Updated Objects

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

Delete
Maintain Multiple Versions of Recently Updated Objects

1 \( p = \text{search}(\text{head}, \text{key}); \)
2 if \( p \neq \text{NULL} \) {
3   \text{list_del_rcu}(&p->\text{list});
4   \text{synchronize_rcu}();
5   \text{kfree}(p);
6 }
Maintain Multiple Versions of Recently Updated Objects

1 p = search(head, key);
2 if (p != NULL) {
3   list_del_rcu(&p->list);
4   synchronize_rcu();
5   kfree(p);
6 }
Maintain Multiple Versions of Recently Updated Objects

1 \( p = \text{search}(\text{head}, \text{key}); \)
2 if (\( p \neq \text{NULL} \)) {
3   \text{list\_del\_rcu}(&p->\text{list});
4   \text{synchronize\_rcu}();
5   \text{kfree}(p);
6 }

Delete
Maintain Multiple Versions of Recently Updated Objects

1. \( q = kmalloc(\text{sizeof}(p), \text{GFP\_KERNEL}); \)
2. \( *q = *p; \)
3. \( q->b = 2; \)
4. \( q->c = 3; \)
5. \( \text{list Replace}_\text{rcu}(&p->list, &q->list); \)
6. \( \text{synchronize}_\text{rcu}(); \)
7. \( \text{kfree}(p); \)

Replace
Maintain Multiple Versions of Recently Updated Objects

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);

Replace
Maintain Multiple Versions of Recently Updated Objects

1 \( q = \text{kmalloc}(	ext{sizeof}(*p), \text{GFP\_KERNEL}); \)
2 \( *q = *p; \)
3 \( q->b = 2; \)
4 \( q->c = 3; \)
5 \( \text{list\_replace\_rcu}(&p->\text{list}, &q->\text{list}); \)
6 \( \text{synchronize\_rcu}(); \)
7 \( \text{kfree}(p); \)

Replace
Maintain Multiple Versions of Recently Updated Objects

1 \( q = \text{kmalloc} (\text{sizeof}(*p), \text{GFP} \_\text{KERNEL}); \)
2 \( \ast q = \ast p; \)
3 \( q->b = 2; \)
4 \( q->c = 3; \)
5 \text{list\_replace\_rcu} (&p->\text{list}, &q->\text{list});
6 \text{synchronize\_rcu}();
7 \text{kfree} (p);

Replace
Maintain Multiple Versions of Recently Updated Objects

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);
Maintain Multiple Versions of Recently Updated Objects

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);
Maintain Multiple Versions of Recently Updated Objects

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);
Conclusion

● 3 different ways to use RCU
  – A publish-subscribe mechanism for adding new data.
  – A way to wait for pre-existing RCU readers to finish.
  – A way to maintain multiple versions of recently updated object without delaying concurrent readers.

● RCU is a step closer towards solving concurrency
  – Readings have no overhead and occur concurrently with an update. (update has to be synchronized!)
  – Memory can be reclaimed when reads are finished.

● RCU is very scalable and heavily used in the Linux Kernel. Next paper!
Graphical Summary

References

- http://lwn.net/Articles/262464
- Daniel Mansour (CS510 2013)
- Jonathan Walpole (CS510 2011)