What is RCU, Fundamentally

By: Paul E. McKenney
Jonathan Walpole

Presenter: Jim Santmyer
Agenda

- Definition RCU
- RCU Publish Subscribe
- Memory Reclamation
- Example walk thru, Deletion and Replacement
- Conclusion
RCU Definition

- Synchronization mechanism
- Concurrency between Multiple readers, single Writer
  - Locking, mutual exclusion, no real concurrency
  - Non-Blocking
    - concurrency with work thrown out
  - Reader/Writer locks
    - Multiple Readers,
    - Exclusive Writer
      - Writer starvation – reader to writer deadlock
      - One thread crash, whole system effected
RCU Definition (Continued)

- RCU ensures reads are coherent
  - Maintains multiple versions of objects
  - Versions not freed until read-side critical sections done
    - Will define read-side critical section later
- Scalable mechanism for publishing and reading of global data
  - Reads extremely fast, lot of them
  - Some cases where read side RCU primitives have zero over head.
    - More about this later
RCU Definition (Continued)

- RCU three fundamental mechanisms
  - Publish – Subscribe (for deletion)
    - More than Writer/Reader
      - i.e. concurrent Update/Readers
  - Wait for Pre-Existing RCU Readers to complete
    - Ensure safe memory reclamation
  - Maintain Multiple Versions of recently updated objects
    - Memory coherency
RCU – Publish, Subscribe

- Concurrent threads can be viewed as communicating with each other via global objects.
- This communication involves a lot more than simple pointer assignment and dereferencing.
- Need to deal with Hardware and Compiler optimizations that reorder code.
- Memory reordering can effect the order data is written and the order data is read.
- The communication process is RCU's publish and subscribe mechanism.
RCU – Publish, Subscribe Example

Writer Code example:

```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;
```

What happens if the compiler reorders the code and the pointer `gp` is assigned the address of `p` before each of the integers in `p` are assigned their values?
RCU – Publish, Subscribe - Example

- Memory Barriers, Compiler directives
  - Difficult to use, hardware specific
  - Non-portable
  - Repetitive
- RCU solution, Publish procedure
  - Ensures correct ordering of operations
- `rcu_assign_pointer(pointer)`
  - Wraps pointer assignment, includes memory barriers and compiler directives
RCU – Publish, Subscribe - Example

Solution Code example:

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

- rcu_assign_pointer() guarantees published pointer is correct.
RCU – Publish, Subscribe - Example

• Readers have their own issues:

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

• Value-speculation
  – Compiler optimization
  – Guess value of p, fetch p->a, p->b, p->c, before assignment of gp address
RCU – Publish, Subscribe - Example

- `rcu_dereference()` primitive
  - Wraps memory barriers and compiler directives
  - Portable
  - Uses published value of gp to assign value of p

```
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_read_lock` & `rcu_read_unlock` covered later
## RCU Publish, Subscribe Primitives

<table>
<thead>
<tr>
<th>Category</th>
<th>Publish</th>
<th>Retract</th>
<th>Subscribe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers</td>
<td>rcu_assign_pointer()</td>
<td>rcu_assign_pointer(..., NULL)</td>
<td>rcu_dereference()</td>
</tr>
<tr>
<td>Lists</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>Hlists</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>
Issue – Memory Reclamation

- Two operations on global object
  - Insertion of new object, free old object
    - Copy global object,
    - Modify copy of object,
    - Atomic operation to publish copy of object
    - Free old object
  - Removing object, freeing object
    - Remove object from global structure
    - Free Object
Issue – Memory Reclamation

- Each of these two operations use a two step process to perform memory reclamation
  - Retire object – remove object from global structure
    - CAS
    - LL/SC
  - Free object at a later time
    - Need to determine when it is safe to free object
      - Reference counter
      - Hazard Pointers
    - Deferred destruction mechanism
RCU – Memory Reclamation

- Reader Side Critical Section
  - Primitives used to delimit critical section
    - `rcu_read_lock()`: may generate no code
    - `rcu_read_unlock()`: may generate no code
  - Can be nested
  - Can delimit any code
  - Must not block or sleep (SRCU)
  - Used to “wait for something to finish”
  - Not a critical section as we have previously discussed.
**RCU – Memory Reclamation**

- **Example Reader Side Code:**

  ```
  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_read_lock` & `rcu_read_unlock` bounds critical section
- On preemptive kernel, may disable preemption
- Note: no synchronization, unlimited concurrency
RCU – Memory Reclamation

Writer Side RCU Primitives:

- **list_replace_rcu()**
  - Performs the replacement of local copy with global
  - Contains all code necessary to perform atomic replace
  - More like this

- **synchronize_rcu()**
  - Synchronous wait for readers to complete

- **call_rcu()**
  - Asynchronous wait for readers to complete
Given update code as shown:

```
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(); // block until read side completes
kfree(p);
```

Question: What needs to occur before the memory can be reclaimed via the kfree(p) call?

Answer: All readers started before or during update must complete before memory can be reclaimed.
RCU Memory Reclamation

• Synchronize with Readers before memory reclamation:
  
  1 synchronize_rcu(cpu)
  2 {
  3     for_each_online_cpu(cpu)
  4     run_on(cpu)
  5  }

• As noted in previous slide the Reader disabled preemption (i.e. context switch) during the Read-Side critical section. If the run_on(cpu) returns kernel preemption must be enabled, therefore Reader done.
RCU Memory Reclamation

- Only works on non-preempt kernels
- Real Time kernels with preemption must use another method such as reference counters.
RCU – Ex: Deletion and Replacement

- RCU allows multiple views of the global object
  - Readers may “see” different versions
- Multiple views occur during node deletion and replacement
Maintain Multiple Versions of Objects
Deletion

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

Deletion

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

Deletion

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

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

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 Objects Replacement

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 Objects

Replacement

```
1 q = kmalloc(sizeof(*p), GFP_KERNEL);
2 *q = *p;  //copy of 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 Objects
Replacement

```c
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 Objects
Replacement

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);

NOTE: Two versions of list. Pre-existing Readers see node(5,6,7), new Readers see node(5,2,3)
Maintain Multiple Versions of Objects
Replacement

1 \( q = \text{kmalloc} \) \( \text{sizeof} \) \( (*p) \), GFP_KERNEL); 
2 \(*q = *p; \)
3 \( q->b = 2; \)
4 \( q->c = 3; \)
5 \( \text{list_replace_rcu} \) (&p->list, &q->list); 
6 \( \text{synchronize_rcu}(); \)
7 \( \text{kfree}(p); \)

NOTE: No more Readers reference node(5,6,7) after synchronize_rcu returns
Maintain Multiple Versions of Objects
Replacement

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

- RCU supports:
  - True concurrency between multiple Readers and single Updater
  - Readers maintain coherent view of memory
  - Automated Memory reclamation
- Highly scalable