

# Read-Copy-Update (RCU)

Josh Triplett

May 22, 2006

## Topics

- The RCU API
- How it works
- How to use it
- What happens if you don't use it correctly
- Example uses

## Recurring Example - Writer

```
1 void write_thing()
2 {
3     struct thing *t, *old;
4     t = kmalloc(sizeof(*t), GFP_KERNEL);
5     spin_lock(&thing_lock);
6     t->contents = some_value;
7     old = global_thing;
8     global_thing = t;
9     spin_unlock(&thing_lock);
10    kfree(old);
11 }
```

## Recurring Example - Reader

```
1 void read_thing()
2 {
3     spin_lock(&thing_lock);
4     printk(KERN_INFO "thing: %d\n",
5           global_thing->contents);
6     spin_unlock(&thing_lock);
7 }
```

## The RCU API

- rcu\_read\_lock/rcu\_read\_unlock
- synchronize\_rcu
- call\_rcu
- rcu\_barrier
- \_bh variants
- rcu\_assign\_pointer
- rcu\_dereference

### rcu\_read\_lock/rcu\_read\_unlock - Description

- Delimit an RCU read-side critical section
- Allows writers to detect concurrent readers
- Prevents “quiescent state”
- Reclamation deferred until current readers complete
- May run concurrently with other readers **and with writers**
- No corresponding writer lock: use other synchronization

### rcu\_read\_lock/rcu\_read\_unlock - Usage

```
1 void read_thing()
2 {
3     rcu_read_lock();
4     printk(KERN_INFO "thing: %d\n",
5             global_thing->contents);
6     rcu_read_unlock();
7 }
```

### rcu\_read\_lock/rcu\_read\_unlock - Implementation

```
1 #define rcu_read_lock()    preempt_disable()
2 #define rcu_read_unlock() preempt_enable()
```

- No overhead without CONFIG\_PREEMPT
- Low overhead with CONFIG\_PREEMPT
- Quiescent state: context switch
- Readers may not block

### **synchronize\_rcu - Description**

- Guarantees that all current readers have finished
- Block until quiescent state on all CPUs
- Use after removing item for future readers
- Use before freeing item concurrent readers could still access

### **synchronize\_rcu - Usage**

```
1 void write_thing()
2 {
3     struct thing *t, *old;
4     t = kmalloc(sizeof(*t), GFP_KERNEL);
5     spin_lock(&thing_lock);
6     t->contents = some_value;
7     old = global_thing;
8     global_thing = t;
9     spin_unlock(&thing_lock);
10    synchronize_rcu();
11    kfree(old);
12 }
```

### **synchronize\_rcu - Toy implementation**

```
1 void synchronize_rcu()
2 {
3     int cpu;
4     for_each_cpu(cpu)
5         run_on_only(cpu);
6     run_on_all_cpus();
7 }
```

- Real, non-toy operating systems used this algorithm

### **call\_rcu - Description**

- Invoke callback when current readers have finished
- Remove item from view of future readers first
- Reclaim item in callback
- Does not block

#### call\_rcu - Usage (Data structure)

```
1 struct thing {
2 int contents;
3 struct rcu_head rcu;
4 };
```

#### call\_rcu - Usage (Writer)

```
1 void write_thing()
2 {
3     struct thing *t, *old;
4     t = kmalloc(sizeof(*t), GFP_KERNEL);
5     spin_lock(&thing_lock);
6     t->contents = some_value;
7     old = global_thing;
8     global_thing = t;
9     spin_unlock(&thing_lock);
10    call_rcu(old->rcu, reclaim_thing);
11 }
```

#### call\_rcu - Usage (Callback)

```
1 void reclaim_thing(struct rcu_head *r)
2 {
3     struct thing *t;
4     t = container_of(r, struct thing, rcu);
5     kfree(t);
6 }
```

- `container_of` gives structure pointer from member pointer

#### call\_rcu - Implementation

- `struct rcu_head` contains list pointer
- `call_rcu` queues `rcu_head` in per-CPU “next” list
- “next” list moves to “current” list in quiescent state at start of grace period
- “current” list moves to “done” list in quiescent state at end of grace period
- Callbacks on “done” list get called and discarded

### **synchronize\_rcu - Real implementation**

```
1 void synchronize_rcu() {
2     struct rcu_synchronize rcu;
3     init_completion(&rcu.completion);
4     call_rcu(&rcu.head, wakeme_after_rcu);
5     wait_for_completion(&rcu.completion);
6 }
7 static void wakeme_after_rcu(
8     struct rcu_head *head) {
9     struct rcu_synchronize *rcu;
10    rcu = container_of(head,
11        struct rcu_synchronize, head);
12    complete(&rcu->completion);
13 }
```

- rcu\_synchronize contains rcu\_head and completion
- wait\_for\_completion blocks until complete called

### **rcu\_barrier**

- Blocks until all RCU callbacks on all CPUs have completed
- Usage example: module unloading
- Implementation: CPU count and wait\_for\_completion

### **\_bh variants**

- Used for “bottom half” handlers
- Need shorter grace periods
- Quiescent state: no bottom half running
- Read-side critical sections:

```
1 #define rcu_read_lock_bh()    local_bh_disable()
2 #define rcu_read_unlock_bh() local_bh_enable()
```

- call\_rcu\_bh: different queues

### **rcu\_assign\_pointer - Description**

- Assign to an RCU-protected pointer
- Use after initializing item
- Makes item visible to readers
- Includes appropriate memory barrier

### Without `rcu_assign_pointer`

- Writes could get reordered
- Reader could see:

```
1  global_thing = t;
2  t->contents = some_value;
```
- Reader can read `global_thing->contents` in between
- Reader gets random uninitialized contents

### `rcu_assign_pointer` - Usage

```
1 void write_thing()
2 {
3     struct thing *t, *old;
4     t = kmalloc(sizeof(*t), GFP_KERNEL);
5     spin_lock(&thing_lock);
6     t->contents = some_value;
7     old = global_thing;
8     rcu_assign_pointer(global_thing, t);
9     spin_unlock(&thing_lock);
10    synchronize_rcu();
11    kfree(old);
12 }
```

### `rcu_assign_pointer` - Implementation

```
1 #define rcu_assign_pointer(p, v) \
2     ({ \
3         smp_wmb(); \
4         (p) = (v); \
5     })
```

`smp_wmb()` provides a write memory barrier in SMP kernels.

### `rcu_dereference` - Description

- Get a copy of an RCU-protected pointer to dereference
- Use inside `rcu_read_lock()/rcu_read_unlock()`
- Includes appropriate memory barrier
- Prevents read reordering

### Without `rcu_dereference`

- Reads could get reordered
- Write memory barrier forces write of contents, then pointer
- Reader can read new pointer, dereference, and find old contents
- Only an issue on Alpha CPUs

### `rcu_dereference` - Usage

```
1 void read_thing()
2 {
3     rcu_read_lock();
4     printk(KERN_INFO "thing: %d\n",
5           rcu_dereference(global_thing)->contents);
6     rcu_read_unlock();
7 }
```

### `rcu_dereference` - Alternate Usage

```
1 void read_thing()
2 {
3     struct thing *local_thing;
4     rcu_read_lock();
5     local_thing = rcu_dereference(global_thing);
6     printk(KERN_INFO "thing: %d\n",
7           local_thing->contents);
8     rcu_read_unlock();
9 }
```

- Useful if using `local_thing` repeatedly
- Cannot use `local_thing` after `rcu_read_unlock()`

### `rcu_dereference` - Implementation

```
1 #define rcu_dereference(p) \
2     ({ \
3         typeof(p) _____p1 = p; \
4         smp_read_barrier_depends(); \
5         (_____p1); \
6     })
```

- Uses GCC extension “statements as expressions”
- Saves copy of pointer, calls `smp_read_barrier_depends()`, returns copy

- Allows use of `rcu_dereference()` in expressions
- `smp_read_barrier_depends()` no-op except on SMP Alpha

#### Final version of writer

```

1 void write_thing()
2 {
3     struct thing *t, *old;
4     t = kmalloc(sizeof(*t), GFP_KERNEL);
5     spin_lock(&thing_lock);
6     t->contents = some_value;
7     old = global_thing;
8     rcu_assign_pointer(global_thing, t);
9     spin_unlock(&thing_lock);
10    synchronize_rcu();
11    kfree(old);
12 }

```

#### Final version of reader

```

1 void read_thing()
2 {
3     rcu_read_lock();
4     printk(KERN_INFO "thing: %d\n",
5           rcu_dereference(global_thing)->contents);
6     rcu_read_unlock();
7 }

```

#### RCU API summary

- `rcu_read_lock/rcu_read_unlock`
- `synchronize_rcu`
- `call_rcu`
- `rcu_barrier`
- `_bh` variants
- `rcu_assign_pointer`
- `rcu_dereference`