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

```c
void write_thing()
{
    struct thing *t, *old;
    t = kmalloc(sizeof(*t), GFP_KERNEL);
    spin_lock(&thing_lock);
    t->contents = some_value;
    old = global_thing;
    global_thing = t;
    spin_unlock(&thing_lock);
    kfree(old);
}
```

Recurring Example - Reader

```c
void read_thing()
{
    spin_lock(&thing_lock);
    printk(KERN_INFO "thing: %d\n",
           global_thing->contents);
    spin_unlock(&thing_lock);
}
```
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**

```c
void read_thing()
{
    rcu_read_lock();
    printk(KERN_INFO "thing: \%d\n",
           global_thing->contents);
    rcu_read_unlock();
}
```

**rcu_read_lock/rcu_read_unlock - Implementation**

```c
#define rcu_read_lock() preempt_disable()
#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

```c
void write_thing()
{
    struct thing *t, *old;
    t = kmalloc(sizeof(*t), GFP_KERNEL);
    spin_lock(&thing_lock);
    t->contents = some_value;
    old = global_thing;
    global_thing = t;
    spin_unlock(&thing_lock);
    synchronize_rcu();
    kfree(old);
}
```

synchronize_rcu - Toy implementation

```c
void synchronize_rcu()
{
    int cpu;
    for_each_cpu(cpu)
        run_on_only(cpu);
    run_on_all_cpus();
}
```

- 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)

```c
struct thing {
    int contents;
    struct rcu_head rcu;
};
```

call_rcu - Usage (Writer)

```c
void write_thing()
{
    struct thing *t, *old;
    t = kmalloc(sizeof(*t), GFP_KERNEL);
    spin_lock(&thing_lock);
    t->contents = some_value;
    old = global_thing;
    global_thing = t;
    spin_unlock(&thing_lock);
    call_rcu(old->rcu, reclaim_thing);
}
```

call_rcu - Usage (Callback)

```c
void reclaim_thing(struct rcu_head *r)
{
    struct thing *t;
    t = container_of(r, struct thing, rcu);
    kfree(t);
}
```

- 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:

```c
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

```c
void write_thing()
{
struct thing *t, *old;
t = kmalloc(sizeof(*t), GFP_KERNEL);
spin_lock(&thing_lock);
t->contents = some_value;
old = global_thing;
rcu_assign_pointer(global_thing, t);
spin_unlock(&thing_lock);
synchronize_rcu();
kfree(old);
}
```

`rcu_assign_pointer` - Implementation

```c
#define rcu_assign_pointer(p, v) \
    ({ \
        smp_wmb(); \
        (p) = (v); \
    })
```

`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**

```c
void read_thing ()
{
    rcu_read_lock ();
    printk (KERN_INFO "thing: %d\n",
    rcu_dereference (global_thing) -> contents);
    rcu_read_unlock ();
}
```

**rcu_dereference - Alternate Usage**

```c
void read_thing ()
{
    struct thing * local_thing;
    rcu_read_lock ();
    local_thing = rcu_dereference (global_thing);
    printk (KERN_INFO "thing: %d\n",
    local_thing -> contents);
    rcu_read_unlock ();
}
```

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

**rcu_dereference - Implementation**

```c
#define rcu_dereference (p) \
    ({ \
        typeof (p) _________p1 = p; \ 
        smp_read_barrier_depends (); \ 
        (_________p1); \ 
    })
```

- 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**
```c
void write_thing()
{
    struct thing *t, *old;
    t = kmalloc(sizeof(*t), GFP_KERNEL);
    spin_lock(&thing_lock);
    t->contents = some_value;
    old = global_thing;
    rcu_assign_pointer(global_thing, t);
    spin_unlock(&thing_lock);
    synchronize_rcu();
    kfree(old);
}
```

**Final version of reader**
```c
void read_thing()
{
    rcu_read_lock();
    printk(KERN_INFO "thing: %d\n",
            rcu_dereference(global_thing)->contents);
    rcu_read_unlock();
}
```

**RCU API summary**
- `rcu_read_lock/rcu_read_unlock`
- `synchronize_rcu`
- `call_rcu`
- `rcu_barrier`
- `_bh variants`
- `rcu_assign_pointer`
- `rcu_dereference`