Deterministic Synchronization in Multicore Systems: 
*The Role of RCU*

Paul E. McKenney
IBM Distinguished Engineer & CTO Linux
Linux Technology Center
Overview

- A Brief Overview of RCU
- When to Use RCU (and not)
- User-Level RCU
A Brief Overview of RCU
Brief Overview of RCU: Use Case

- Combines waiting for readers and multiple versions:
  - Writer removes element B from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free B (kfree())
- Readers access the list with little or no synchronization
Brief Overview of RCU: Code for Use Case

```c
struct foo_head {
    struct list_head list;
    spinlock_t mutex;
};

struct foo {
    struct list_head list;
    int key;
};

int search(struct foo_head *fhp, int k) {
    struct foo *p;
    struct list_head *head = &fhp->list;
    rcu_read_lock();
    list_for_each_entry_rcu(p, head, list) {
        if (p->key == k) {
            rcu_read_unlock();
            return 1;
        }
    }
    rcu_read_unlock();
    return 0;
}

int delete(struct foo_head *fhp, int k) {
    struct foo *p;
    struct list_head *head = &fhp->list;
    spin_lock(&fhp->mutex);
    list_for_each_entry(p, head, list) {
        if (p->key == k) {
            list_del_rcu(p);
            spin_unlock(&fhp->mutex);
            synchronize_rcu();
            kfree(p);
            return 1;
        }
    }
    spin_unlock(&fhp->mutex);
    return 0;
}
```

foo_head → foo (A) → foo (B) → foo (C)
Brief Overview of RCU: Code for rwlock

```c
struct foo_head {
    struct list_head list;
    rwlock_t mutex;
};

struct foo {
    struct list_head list;
    int key;
};

int search(struct foo_head *fhp, int k) {
    struct foo *p;
    struct list_head *head = &fhp->list;
    read_lock(&fhp->mutex);
    list_for_each_entry(p, head, list) {
        if (p->key == k) {
            read_lock(&fhp->mutex);
            return 1;
        }
    }
    write_unlock(&fhp->mutex);
    return 0;
}

int delete(struct foo_head *fhp, int k) {
    struct foo *p;
    struct list_head *head = &fhp->list;
    write_lock(&fhp->mutex);
    list_for_each_entry(p, head, list) {
        if (p->key == k) {
            list_del(p);
            write_unlock(&fhp->mutex);
            /* synchronize_rcu(); */
            kfree(p);
            return 1;
        }
    }
    write_unlock(&fhp->mutex);
    return 0;
}
```

```c
struct foo_head {
    struct list_head list;
    rwlock_t mutex;
};

struct foo {
    struct list_head list;
    int key;
};
```
What Does synchronize_rcu() Do???
Brief Overview of RCU: Waiting for Readers

```
rcu_read_lock()

Reader
Reader
Reader
Reader
Reader

Reader

RCU readers concurrent with updates

Reader

synchronize_rcu()

Change
Change Visible to All Readers

Grace Period

Grace period extends as needed.

A grace period is not permitted to end until all pre-existing readers have completed.
```
Brief Overview of RCU: Sort of Reference Count

struct foo *gp;

Publish

p = malloc(sizeof(p));
initialize(p);
rcu_assign_pointer(gp, p);

Subscribe

rcu_read_lock();
q = rcu_dereference(gp);
/* *q guaranteed to exist */
do_something_with(q);
rcu_read_unlock();
/* *q might be freed */

Retract

p = gp;
gp = NULL;
synchronize_rcu();
free(p);

Release "reference"

Wait for pre-existing "references" to be released
Brief Overview of RCU: Performance

![Graph showing overhead in nanoseconds vs. number of CPUs for CONFIG_PREEMPT kernel build]

Overhead (nanoseconds)

Number of CPUs

CONFIG_PREEMPT kernel build
When to Use RCU (and not)
When to Use RCU (and not)
When to Use RCU (and not)
When to Use RCU (and not)

- **Read-Mostly, Stale & Inconsistent Data OK** (RCU Works Great!!!)
- **Read-Mostly, Need Consistent Data** (RCU Works OK, Especially for Real-Time)
- **Read-Write, Need Consistent Data** (RCU Might Be OK... Perhaps for Real-Time Readers)
- **Update-Mostly, Need Consistent Data** (RCU is Really Unlikely to be the Right Tool For The Job) (Instead, use locking, atomic operations, message passing, etc.)
Non-Preemptible RCU: Classic RCU !PREEMPT

```c
1 static inline void rcu_read_lock(void)
2 {
3 }
4
5 static inline void rcu_read_unlock(void)
6 {
7 }
```
Preemptible RCU: Classic RCU PREEMPT

1 static inline void rcu_read_lock(void)
2 {
3    preempt_disable();
4 }
5
6 static inline void rcu_read_unlock(void)
7 {
8    preempt_enable();
9 }
Preemptible RCU: 2005 Preemptible RCU

```c
1 void rcu_read_lock(void) {
2     int f;
3     unsigned long oldirq;
4     struct task_struct *t = current;
5     raw_local_irq_save(oldirq);
6     if (t->rcu_read_lock_nesting++ == 0) {
7         f = rcu_ctrlblk.completed & 1;
8         atomic_inc(t->rcu_flipctr1);
9         smp_read_barrier_depends();
10        t->rcu_flipctr1 =
11           &(__get_cpu_var(rcu_flipctr)[f]);
12        atomic_inc(t->rcu_flipctr1);
13        smp_mb__after_atomic_inc();
14        if (f != (rcu_ctrlblk.completed & 1)) {
15           t->rcu_flipctr2 =
16             &(__get_cpu_var(rcu_flipctr)[!f]);
17             atomic_inc(t->rcu_flipctr2);
18             smp_mb__after_atomic_inc();
19        }
20    }
21    raw_local_irq_restore(oldirq);
22 }
```

```c
1 void rcu_read_unlock(void) {
2     unsigned long oldirq;
3     struct task_struct *t = current;
4     raw_local_irq_save(oldirq);
5     if (--t->rcu_read_lock_nesting == 0) {
6         smp_mb__before_atomic_dec();
7         atomic_dec(t->rcu_flipctr1);
8         t->rcu_flipctr1 = NULL;
9         if (t->rcu_flipctr2 != NULL) {
10            atomic_dec(t->rcu_flipctr2);
11            t->rcu_flipctr2 = NULL;
12        }
13    }
14    raw_local_irq_restore(oldirq);
15 }
```
Preemptible RCU: 2007 Preemptible RCU

```c
void __rcu_read_lock(void) {
    int idx;
    struct task_struct *t = current;
    int nesting;

    nesting = __read_lock_nesting(t);

    if (nesting != 0) {
        t->rcu_read_lock_nesting = nesting + 1;
        unsigned long flags;
        local_irq_save(flags);
        idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1;
        ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])++;
        ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1;
        ACCESS_ONCE(t->rcu_flipctr_idx) = idx;
        local_irq_restore(flags);
    } else {
        unsigned long flags;
        local_irq_save(flags);
        idx = ACCESS_ONCE(t->rcu_flipctr_idx);
        ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
        ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
        local_irq_restore(flags);
    }
}
```

No longer the mainline preemptible RCU implementation
Preemptible RCU: 2009 Preemptible RCU

```
1 void __rcu_read_lock(void)
2 {
3   ACCESS_ONCE(current->rcu_read_lock_nesting)++;
4   barrier();
5 }

7 void __rcu_read_unlock(void)
8 {
9   struct task_struct *t = current;
10  barrier();
11  if (--ACCESS_ONCE(t->rcu_read_lock_nesting) == 0 &&
12         unlikely(ACCESS_ONCE(t->rcu_read_unlock_special)))
13    rcu_read_unlock_special(t);
14 }
```

Accepted into mainline for 2.6.32 kernel
TREE_RCU Data Structures
TREE_RCU struct rcu_node

struct rcu_node

spinlock_t lock; /* CPUs waited for */
unsigned long qsmask; /* CPUs online */
unsigned long qsmaskinit; /* Bit in parent */
unsigned long grpmask; /* First CPU/group # */
int grplo; /* Last CPU/group # */
int grphi; /* Bit # in parent */
u8 grpnum; /* Root is level 0 */
u8 level /* Pointer to parent */
struct rcu_node parent;
TREE_PREEMPT_RCU struct rcu_node

struct rcu_node

spinlock_t lock;
long gpnum;
unsigned long qsmask;
unsigned long qsmaskinit;
unsigned long grpmask;
int grplo;
int grpHi;
u8 grpnum;
u8 level
struct rcu_node parent;
struct list_head blocked_tasks[2];
/* Grace period # */
/* CPUs waited for */
/* CPUs online */
/* Bit in parent */
/* First CPU/group # */
/* Last CPU/group # */
/* Bit # in parent */
/* Root is level 0 */
/* Pointer to parent */
/* In RCU R-S C-S */
TREE_PREEMPT_RCU Data Structures
Plug-In Architecture (kernel/rcutree_plugin.h)

static inline void rcu_bootup_announce(void);
long rcu_batches_completed(void);
static void rcu_preempt_note_context_switch(int cpu);
static int rcu_preempted_readers(struct rcu_node *rnp);
#ifdef CONFIG_RCU_CPU_STALL_DETECTOR
static void rcu_print_task_stall(struct rcu_node *rnp);
#endif /* #ifdef CONFIG_RCU_CPU_STALL_DETECTOR */
static void rcu_preempt_check_blocked_tasks(struct rcu_node *rnp);
#ifdef CONFIG_HOTPLUG_CPU
static void rcu_preempt_offline_tasks(struct rcu_state *rsp,
struct rcu_node *rnp,
struct rcu_data *rdp);
#endif /* #ifdef CONFIG_HOTPLUG_CPU */
static void rcu_preempt_offline_cpu(int cpu);
#endif /* #ifdef CONFIG_HOTPLUG_CPU */
static void rcu_preempt_check_callbacks(int cpu);
static void rcu_preempt_process_callbacks(void);
void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *rcu));
static int rcu_preempt_pending(int cpu);
static int rcu_preempt_needs_cpu(int cpu);
static void __cpuinit rcu_preempt_init_percpu_data(int cpu);
static void rcu_preempt_send_cbs_to_orphanage(void);
static void __init __rcu_init_preempt(void);
## Planned Configuration Selection

<table>
<thead>
<tr>
<th></th>
<th>SMP</th>
<th>!SMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREEMPT</td>
<td>TREE_PREEMPT_RCU</td>
<td>TINY_PREEMPT_RCU (TBD)</td>
</tr>
<tr>
<td>!PREEMPT</td>
<td>TREE_RCU</td>
<td>TINY_RCU</td>
</tr>
</tbody>
</table>
User-Level RCU
User-Level RCU

- Multiple experimental implementations:
  - Hart et al. IPDPS'06
  - git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git
- M. Desnoyers, 2009 Linux Plumbers Conference
  - Minimal read-side overhead
  - Library-ready implementation (read-side smp_mb())
  - Signal-based implementation
  - Likely to appear soon in a distro near you...
- Linux Audio project (P. Davis)
- And numerous others
Summary

- RCU is a specialized synchronization primitive that provides deterministic read-side overhead
- Kernel RCU implementation has grown increasingly simple and fast
  - "I would have provided a simpler implementation to start with, but I did not have time to do so."
- User-level RCU primitives used in production
  - A surprise to me...
  - And several production-quality implementations are under development
  - Likely to be useful in real-time applications
Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, Mathieu Desnoyers, and Jonathan Walpole
Questions?

- http://lwn.net/Articles/262464/ (What is RCU, Fundamentally?)
- http://lwn.net/Articles/263130/ (What is RCU's Usage?)
- http://lwn.net/Articles/264090/ (What is RCU's API?)
  - linux.conf.au paper comparing RCU vs. locking performance
  - RCU motivation, implementations, usage patterns, performance (micro+sys)
  - System-level performance for SELinux workload: >500x improvement
  - Comparison of RCU and NBS (later appeared in JPDC)
- http://doi.acm.org/10.1145/1400097.1400099
  - History of RCU in Linux (Linux changed RCU more than vice versa)
- git://lttng.org/userspace-rcu.git
  - Mathieu Desnoyers’s user-space RCU git repository