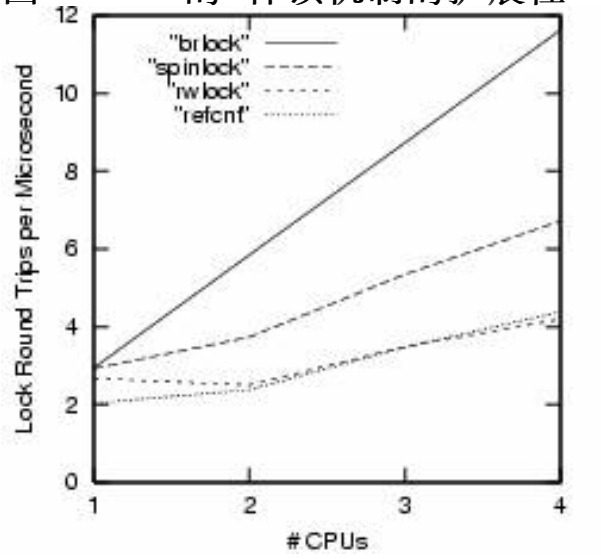
原子加 1	58.2
比较交换(cmpxchg)原子加 1	107.3
访问主存	162.4
CPU 本地 lock	163.7
缓存传输	170.4-360.9

表1是在700MHz的奔腾III机器上的基本操作的开销，在该机器上一个时钟周期能够执行两条
700MHz的奔腾III机器慢75纳秒(ns)，尽管CPU速度快两倍多。

这种锁机制的另一个问题在于其可扩展性，在多处理器系统上，可扩展性非常重要，否则根

图 1 Linux的4种锁机制的扩展性



注：refcnt表示自旋锁与引用记数一起使用。

读写锁rwlock在两个CPU的情况下性能反倒比一个CPU的差，在四个CPU的情况下，refcnt的
性能的39%,自旋缩spinlock的性能明显好于refcnt和rwlock，但它也只达到了理性性能的57%，
Ingo Molnar实现的一个高性能的rwlock，它适用于读特多而写特少的情况，读者获得brlock的
锁，用户无法随便定义并使用这种锁，它也需要为每个CPU定义一个锁状态数组，因此这种
方使用到。

正是在这种背景下，一个高性能的锁机制RCU呼之欲出，它克服了以上锁的缺点，具有很好
写少的情况，如网络路由表的查询更新、设备状态表的维护、数据结构的延迟释放以及多径

RCU并不是新的锁机制，它只是对Linux内核而言是新的。早在二十世纪八十年代就有了这种

统中使用了这种机制，但这种早期的实现并不太好，在二十世纪九十年代出现了一个比较高
包含在2.6内核中。

二、RCU的原理

RCU(Read-Copy Update)，顾名思义就是读-拷贝修改，它是基于其原理命名的。对于被RCU保
者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制
据。这个时机就是所有引用该数据的CPU都退出对共享数据的操作。

因此RCU实际上是一种改进的rwlock，读者几乎没有什么同步开销，它不需要锁，不使用原子
Barrier），因此不会导致锁竞争，内存延迟以及流水线停滞。不需要锁也使得使用更容易，

按照第二节所讲原理，对于读者，RCU 仅需要抢占失效，因此获得读锁和释放读锁分别定义

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

它们有一个变种：

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

这个变种只在修改是通过 call_rcu_bh 进行的情况下使用，因为 call_rcu_bh将把 softirq 的执行
call_rcu_bh 进行的，在进程上下文的读端临界区必须使用这一变种。

每一个 CPU 维护两个数据结构rcu_data,rcu_bh_data，它们用于保存回调函数，函数call_rcu和
rcu_data，而后者则把回调函数注册到rcu_bh_data，在每一个数据结构上，回调函数被组成一

当在CPU上发生进程切换时，函数rcu_qsctr_inc将被调用以标记该CPU已经经历了一个quiesce

时钟中断触发垃圾收集器运行，它会检查：

1. 否在该CPU上有需要处理的回调函数并且已经经过一个grace period；
2. 否没有需要处理的回调函数但有注册的回调函数；
3. 否该CPU已经完成回调函数的处理；
4. 否该CPU正在等待一个quiescent state的到来；

如果以上四个条件只要有一个满足，它就调用函数rcu_check_callbacks。

函数rcu_check_callbacks首先检查该CPU是否经历了一个quiescent state，如果：

1． 当前进程运行在用户态；
或

2． 当前进程为idle且当前不处在运行softirq状态，也不处在运行IRQ处理函数的状态；

那么，该CPU已经经历了一个quiescent state，因此通过调用函数rcu_qsctr_inc标记该CPU的数
CPU已经经历一个quiescent state。

否则，如果当前不处在运行softirq状态，那么，只标记该CPU的数据结构rcu_bh_data的标记
意，该标记只对rcu_bh_data有效。

然后，函数rcu_check_callbacks将调用tasklet_schedule，它将调度为该CPU设置的tasklet rcu_tas

在时钟中断返回后，rcu_tasklet将在softirq上下文被运行。

rcu_tasklet将运行函数rcu_process_callbacks，函数rcu_process_callbacks可能做以下事情：

1． 开始一个新的grace period；这通过调用函数rcu_start_batch实现。

2． 运行需要处理的回调函数；这通过调用函数rcu_do_batch实现。

3． 检查该CPU是否经历一个quiescent state；这通过函数rcu_check_quiescent_state实现

如果还没有开始grace period，就调用rcu_start_batch开始新的grace period。调用函数rcu_check_
且是最后一个经历quiescent state的CPU，那么就结束grace period，并开始新的grace period。如

synchronize_kernel()

其他非RCU的内核代码使用该函数来等待所有CPU处在可抢占状态，目前功能等同于synchro

synchronize_sched()

该函数用于等待所有CPU都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，
保证所有CPU都处理完正在运行的读端临界区。 注：在2.6.12内核中，synchronize_kernel和sy
实际是完全等同的，但是将来将可能有大的变化，因此务必根据需求选择恰当的函数。

```
void fastcall call_rcu(struct rcu_head *head,
                       void (*func)(struct rcu_head *rcu))
struct rcu_head {
      struct rcu_head *next;
      void (*func)(struct rcu_head *head);
};
```

函数 call_rcu 也由 RCU 写端调用，它不会使写者阻塞，因而可以在中断上下文或 softirq 使用
在进程上下文使用。该函数将把函数 func 挂接到 RCU回调函数链上，然后立即返回。一旦所
除的将绝不在被应用的数据。参数 head 用于记录回调函数 func，一般该结构会作为被 RCU
操作。需要指出的是，函数 synchronize_rcu 的实现实际上使用函数call_rcu。

```
void fastcall call_rcu_bh(struct rcu_head *head,
                       void (*func)(struct rcu_head *rcu))
```

函数call_ruc_bh功能几乎与call_rcu完全相同，唯一差别就是它把softirq的完成也当作经历一个
端必须使用rcu_read_lock_bh。

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

该宏用于在RCU读端临界区获得一个RCU保护的指针，该指针可以在以后安全地引用，内存

除了这些API，RCU还增加了链表操作的RCU版本，因为对于RCU，对共享数据的操作必须保
的。

static inline void list_add_rcu(struct list_head *new, struct list_head *head) 该函数把链表项new插入
插入的链表项之前，新链表项的链接指针的修改对所有读者是可见的。

```
static inline void list_add_tail_rcu(struct list_head *new,
                                struct list_head *head)
```

该函数类似于list_add_rcu，它将把新的链表项new添加到被RCU保护的链表的末尾。

该宏用于在退出点之后继续遍历由RCU保护的链表head。

```
static inline void hlist_del_rcu(struct hlist_node *n)
```

它从由RCU保护的哈希链表中移走链表项n，并设置n的ppre指针为LIST_POISON2，但并没有
链表。

```
static inline void hlist_add_head_rcu(struct hlist_node *n,
                                      struct hlist_head *h)
```

该函数用于把链表项n插入到被RCU保护的哈希链表的开头，但同时允许读者对该哈希链表的
者可见。

```
hlist_for_each_rcu(pos, head)
```

该宏用于遍历由RCU保护的哈希链表head，只要在读端临界区使用该函数，它就可以安全地

```
hlist_for_each_entry_rcu(tpos, pos, head, member)
```

类似于hlist_for_each_rcu，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表

五、RCU 典型应用

在 linux 2.6 内核中，RCU 被内核使用的越来越广泛。下面是在最新的 2.6.12内核中搜索得到的

表 1 rcu_read_lock 的使用情况统计

net/ipv6/icmp.c	1
net/ipv6/af_inet6.c	1
net/ipv6/ndisc.c	1
net/ipv6/ip6_input.c	1
net/sctp/protocol.c	1
net/802/psnap.c	1
net/decnet/dn_neigh.c	1
net/decnet/dn_route.c	1
net/8021q/vlan_dev.c	1
net/econet/af_econet.c	1
security/selinux/avc.c	3
security/selinux/netif.c	1

表 2 rcu_read_unlock 的使用情况统计

net/ipv6/af_inet6.c	2
net/ipv6/ndisc.c	2
net/ipv6/ip6_input.c	2
net/sctp/protocol.c	2
net/802/psnap.c	1
net/decnet/dn_neigh.c	3
net/decnet/dn_route.c	2
net/8021q/vlan_dev.c	5
net/econet/af_econet.c	1
security/selinux/avc.c	3
security/selinux/netif.c	2

表 3 rcu_read_lock_bh 的使用情况统计

文件名	使用次数
net/ipv4/route.c	4
net/decnet/dn_route.c	4

表 4 rcu_read_unlock_bh 的使用情况统计

文件名	使用次数
net/ipv4/route.c	8
net/decnet/dn_route.c	7

表 5 call_rcu 的使用情况统计

文件名	使用次数
arch/ppc64/mm/tlb.c	1
fs/dcache.c	1
ipc/util.c	2
kernel/auditsc.c	1
mm/slab.c	1
net/core/neighbour.c	1
net/ipv4/devinet.c	2
net/ipv4/multipath_wrandom.c	2
net/bridge/br_fdb.c	1
net/bridge/br_if.c	1
net/sched/sch_generic.c	1
net/8021q/vlan.c	1
security/selinux/avc.c	2
security/selinux/netif.c	1

表 6 call_rcu_bh 的使用情况统计

文件名	使用次数
net/ipv4/route.c	2
net/decnet/dn_route.c	2

表 7 list API 的使用情况统计

arch/x86_64/kernel/mce.c	2
drivers/s390/net/qeth_main.c	1
ipc/util.c	1
net/core/netfilter.c	1
net/core/dev.c	2
net/ipv4/ip_input.c	1
net/ipv4/arp.c	1
net/ipv4/icmp.c	1
net/ipv4/route.c	10
net/ipv4/multipath_drr.c	2
net/ipv4/multipath_random.c	2
net/ipv4/multipath_rr.c	2
net/ipv4/multipath_wrandom.c	2
net/atm/clip.c	1
net/ipv6/icmp.c	1
net/ipv6/ip6_input.c	1
net/decnet/dn_neigh.c	1
net/decnet/dn_route.c	7

从以上统计结果可以看出，RCU已经在网络驱动层、网络核心层、IPC、dcache、内存设备层
的使用统计汇总（表 10），不难看出，RCU已经是一个非常重要的内核锁机制。

表 10 所有RCU API使用情况总汇

函数名	使用次数
rcu_read_lock	96
rcu_read_unlock	126
rcu_read_lock_bh	8
rcu_read_unlock_bh	15
call_rcu	18
call_rcu_bh	4
list API	69
synchronize_rcu	8
rcu_dereference	39
合计	383

因此，如何正确使用 RCU 对于内核开发者而言非常重要。

下面部分将就 RCU 的几种典型应用情况详细讲解。

1．只有增加和删除的链表操作

在这种应用情况下，绝大部分是对链表的遍历，即读操作，而很少出现的写操作只有增加或
常容易，从rwlock转换成RCU非常自然。路由表的维护就是这种情况的典型应用，对路由表的
加或删除，因此使用RCU替换原来的rwlock顺理成章。系统调用审计也是这样的情况。

这是一段使用rwlock的系统调用审计部分的读端代码：

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
        enum audit_state   state;

        read_lock(&auditsc_lock);
```

```
{
        struct audit_entry  *e;
        write_lock(&auditsc_lock);
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del(&e->list);
                        write_unlock(&auditsc_lock);
                        return 0;
                }
        }
        write_unlock(&auditsc_lock);
        return -EFAULT;         /* No matching rule */
}
static inline int audit_add_rule(struct audit_entry *entry,
                        struct list_head *list)
{
        write_lock(&auditsc_lock);
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add(&entry->list, list);
        } else {
                list_add_tail(&entry->list, list);
        }
        write_unlock(&auditsc_lock);
        return 0;
}
```

使用RCU后写端代码变成为：

```
static inline int audit_del_rule(struct audit_rule *rule,
                        struct list_head *list)
{
        struct audit_entry  *e;
        /* Do not use the _rcu iterator here, since this is the only
         * deletion routine. */
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        list_del_rcu(&e->list);
                        call_rcu(&e->rcu, audit_free_rule, e);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
static inline int audit_add_rule(struct audit_entry *entry,
                        struct list_head *list)
{
        if (entry->rule.flags & AUDIT_PREPEND) {
                entry->rule.flags &= ~AUDIT_PREPEND;
                list_add_rcu(&entry->list, list);
        } else {
                list_add_tail_rcu(&entry->list, list);
        }
        return 0;
}
```

对于链表删除操作，list_del替换为list_del_rcu和call_rcu，这是因为被删除的链表项可能还在被

```
static inline int audit_upd_rule(struct audit_rule *rule,
                        struct list_head *list,
                        __u32 newaction,
                        __u32 newfield_count)
{
        struct audit_entry  *e;
        struct audit_newentry *ne;
        list_for_each_entry(e, list, list) {
                if (!audit_compare_rule(rule, &e->rule)) {
                        ne = kmalloc(sizeof(*entry), GFP_ATOMIC);
                        if (ne == NULL)
                                return -ENOMEM;
                        audit_copy_rule(&ne->rule, &e->rule);
                        ne->rule.action = newaction;
                        ne->rule.file_count = newfield_count;
                        list_replace_rcu(e, ne);
                        call_rcu(&e->rcu, audit_free_rule, e);
                        return 0;
                }
        }
        return -EFAULT;         /* No matching rule */
}
```

3．修改操作立即可见

前面两种情况，读者能够容忍修改可以在一段时间后看到，也就说读者在修改后某一时间段
到旧的数据，这种情况下，需要使用一些新措施，如System V IPC，它在每一个链表条目中地
置为真，否则设置为假，当代码在遍历链表时，核对每一个条目的deleted字段，如果为真，

还是以系统调用审计代码为例，如果它不能容忍旧数据，那么，读端代码应该修改为：

```
static enum audit_state audit_filter_task(struct task_struct *tsk)
{
        struct audit_entry *e;
        enum audit_state   state;
        rcu_read_lock();
        list_for_each_entry_rcu(e, &audit_tsklist, list) {
                if (audit_filter_rules(tsk, &e->rule, NULL, &state)) {
                        spin_lock(&e->lock);
                        if (e->deleted) {
                                spin_unlock(&e->lock);
                                rcu_read_unlock();
                                return AUDIT_BUILD_CONTEXT;
                        }
                        rcu_read_unlock();
                        return state;
                }
        }
        rcu_read_unlock();
        return AUDIT_BUILD_CONTEXT;
}
```

注意，对于这种情况，每一个链表条目都需要一个spinlock保护，因为删除操作将修改条目的
条目的锁，因为只有这样，才能看到新的修改的数据，否则，仍然可能看到就的数据。

参考资料

[1] Linux RCU实现者之一Paul E. McKenney的RCU资源链接，http://www.rdrop.com/users/paul

[2] Paul E. McKenney的博士论文，"Exploiting Deferred Destruction: An Analysis of Read-Copy Up Kernels"，http://www.rdrop.com/users/paulmck/rclock/RCUdissertation.2004.07.14e1.pdf。

[3] Paul E. McKenney's paper in Ottawa Linux Summit 2002, Read-Copy Update，http://www.rdrop

[4] Linux Journal在2003年10月对RCU的简介, Kernel Korner - Using RCU in the Linux 2.5 Kernel，

[5] Scaling dcache with RCU, http://linuxjournal.com/article/7124。

[6] Patch: Real-Time Preemption and RCU，http://lwn.net/Articles/128228/。

[7] Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel, http://www.rdrop

[8] Linux 2.6.12 kernel source。

[9] Linux kernel documentation, Documentation/RCU/*。

关于作者

杨燚，计算机科学硕士，毕业于中科院计算技术研究所，有4年的Linux内核编程经验，目前
过yang.yi@bmrtech.com或yyang@ch.mvista.com与作者联系。

5 星

提交

添加评论:

请 登录 或 注册 后发表评论。

注意：评论中不支持 HTML 语法

有新评论时提醒我剩余 1000 字符

发布

共有评论 (3)

你好，我想问下你问题，如果rcu保护的是list的话，那么对于对于list中单个成员用rcu保护的

由 **mode_yang** 于 2012年05月16日发布

报告滥用

很不错。

由 **è¥¿åéå³°** 于 2012年03月10日发布

报告滥用

很详细呀！很期待更多的文档，多谢了！

由 **ä¸ç¹ç¹** 于 2012年02月14日发布

报告滥用