一:前言

RCU机制出现的比较早,只是在linux kernel中一直到2.5版本的时候才被采用.关于RCU机制,这里就不做过多的介绍了,网上有很多有关RCU介绍和使用的文档.请自行查阅.本文主要是从linux kernel源代码的角度.来分析RCU的实现.

在讨论RCU的实现之前.有必要重申以下几点:

1:RCU使用在读者多而写者少的情况.RCU和读写锁相似.但RCU的读者占锁没有任何的系统开销.写者与写写者之间必须要保持同步,且写者必须要等它之前的读者全部都退出之后才能释放之前的资源.

2:RCU保护的是指针.这一点尤其重要.因为指针赋值是一条单指令.也就是说是一个原子操作.因它更改指针指向没必要考虑它的同步.只需要考虑cache的影响.

3:读者是可以嵌套的.也就是说rcu_read_lock()可以嵌套调用.

4:读者在持有rcu_read_lock()的时候,不能发生进程上下文切换.否则,因为写者需要要等待读者完成,写者进程也会一直被阻塞.

以下的代码是基于linux kernel 2.6.26

二:使用RCU的实例

Linux kernel中自己附带有详细的文档来介绍RCU,这些文档位于linux-2.6.26.3/Documentation/RCU. 这些文档值得多花点时间去仔细研读一下.

下面以whatisRCU.txt中的例子作为今天分析的起点:

```
struct foo {
	int a;
	char b;
	long c;
};
DEFINE_SPINLOCK(foo_mutex);

struct foo *gbl_foo;
void foo_update_a(int new_a)
{
	struct foo *new_fp;
	struct foo *old_fp;

	new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
	spin_lock(&foo_mutex);
	old_fp = gbl_foo;
	*new_fp = *old_fp;
	new_fp->a = new_a;
	rcu_assign_pointer(gbl_foo, new_fp);
	spin_unlock(&foo_mutex);
	synchronize_rcu();
	kfree(old_fp);
}

int foo_get_a(void)
```

```
{
        int retval;

        rcu_read_lock();
        retval = rcu_dereference(gbl_foo)->a;
        rcu_read_unlock();
        return retval;
}
```

如上代码所示,RCU被用来保护全局指针struct foo *gbl_foo. foo_get_a()用来从RCU保护的结构中取得gbl_foo的值.而foo_update_a()用来更新被RCU保护的gbl_foo的值.

另外,我们思考一下,为什么要在foo_update_a()中使用自旋锁foo_mutex呢?

假设中间没有使用自旋锁.那foo_update_a()的代码如下:

```
void foo_update_a(int new_a)
{
        struct foo *new_fp;
        struct foo *old_fp;

        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);

        old_fp = gbl_foo;
        1:------------------------
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);

        synchronize_rcu();
        kfree(old_fp);
}
```

假设A进程在上图----

标识处被B进程抢点.B进程也执行了goo_ipdate_a().等B执行完后，再切换回A进程.此时,A进程所持的old_fd实际上已经被B进程给释放掉了.此后A进程对old_fd的操作都是非法的.

另外,我们在上面也看到了几个有关RCU的核心API.它们为别是:

rcu_read_lock()

rcu_read_unlock()

synchronize_rcu()

rcu_assign_pointer()

rcu_dereference()

其中,rcu_read_lock()和rcu_read_unlock()用来保持一个读者的RCU临界区.在该临界区内不允许发生上下文切换.

rcu_dereference():读者调用它来获得一个被RCU保护的指针.

Rcu_assign_pointer():写者使用该函数来为被RCU保护的指针分配一个新的值.这样是为了安

全从写者到读者更改其值.这个函数会返回一个新值

三:RCU API实现分析

Rcu_read_lock()和rcu_read_unlock()的实现如下:

```
#define rcu_read_lock() __rcu_read_lock()
#define rcu_read_unlock() __rcu_read_unlock()

#define __rcu_read_lock() \
   do { \
      preempt_disable(); \
      __acquire(RCU); \
      rcu_read_acquire(); \
   } while (0)
#define __rcu_read_unlock() \
   do { \
      rcu_read_release(); \
      __release(RCU); \
      preempt_enable(); \
   } while (0)
```

其中__acquire(),rcu_read_read_acquire(),rcu_read_release(),rcu_read_release()都是一些选择编译函数,可以忽略不可看.因此可以得知.rcu_read_lock(),rcu_read_unlock()只是禁止和启用抢占.因为在读者临界区,不允许发生上下文切换.

rcu_dereference()和rcu_assign_pointer()的实现如下:

```
#define rcu_dereference(p)     ({ \
            typeof(p) _________p1 = ACCESS_ONCE(p); \
            smp_read_barrier_depends(); \
            (_________p1); \
            })
#define rcu_assign_pointer(p, v) \
   ({ \
      if (!__builtin_constant_p(v) || \
         ((v) != NULL)) \
         smp_wmb(); \
      (p) = (v); \
   })
```

它们的实现也很简单.因为它们本身都是原子操作.因为只是为了cache一致性,插上了内存屏障.可以让其它的读者/写者可以看到保护指针的最新值.

synchronize_rcu()在RCU中是一个最核心的函数,它用来等待之前的读者全部退出.我们后面的大部份分析也是围绕着它而进行.实现如下:

```
void synchronize_rcu(void)
{
   struct rcu_synchronize rcu;
```

```
    init_completion(&rcu.completion);
    /* Will wake me after RCU finished */
    call_rcu(&rcu.head, wakeme_after_rcu);

    /* Wait for it */
    wait_for_completion(&rcu.completion);
}
```

我们可以看到,它初始化了一个本地变量,它的类型为struct rcu_synchronize.调用call_rcu()之后.一直等待条件变量rcu.competion的满足.

在这里看到了RCU的另一个核心API,它就是call_run().它的定义如下:

```
void call_rcu(struct rcu_head *head,
              void (*func)(struct rcu_head *rcu))
```

它用来等待之前的读者操作完成之后,就会调用函数func.
我们也可以看到,在synchronize_rcu()中,读者操作完了要调用的函数就是wakeme_after_rcu().另外,call_rcu()用在不可睡眠的条件中,如果中断环境,禁止抢占环境等.而synchronize_rcu()用在可睡眠的环境下.先跟踪看一下wakeme_after_rcu():

```
static void wakeme_after_rcu(struct rcu_head  *head)
{
    struct rcu_synchronize *rcu;

    rcu = container_of(head, struct rcu_synchronize, head);
    complete(&rcu->completion);
}
```

我们可以看到,该函数将条件变量置真,然后唤醒了在条件变量上等待的进程.

看下call_rcu():

```
void call_rcu(struct rcu_head *head,
              void (*func)(struct rcu_head *rcu))
{
    unsigned long flags;
    struct rcu_data *rdp;

    head->func = func;
    head->next = NULL;
    local_irq_save(flags);
    rdp = &__get_cpu_var(rcu_data);
    *rdp->nxttail = head;
    rdp->nxttail = &head->next;
    if (unlikely(++rdp->qlen > qhimark)) {
        rdp->blimit = INT_MAX;
        force_quiescent_state(rdp, &rcu_ctrlblk);
```

```
    }
    local_irq_restore(flags);
}
```

该函数也很简单,就是将head加在了per_cpu变量rcu_data的tail链表上.

Rcu_data定义如下:

```
DEFINE_PER_CPU(struct rcu_data, rcu_data) = { 0L };
```

由此,我们可以得知,每一个CPU都有一个rcu_data.每个调用call_rcu()/synchronize_rcu()进程所代表的head都会挂到rcu_data的tail链表上.

那究竟怎么去判断当前的写者已经操作完了呢?我们在之前看到,不是读者在调用rcu_read_lock()的时候要禁止抢占么?因此,我们只需要判断如有的CPU都进过了一次上下文切换,就说明所有读者已经退出了.

引用>((

http://www.ibm.com/developerworks/cn/linux/l-rcu/

)中有关这个过程的描述:

“等待适当时机的这一时期称为grace period，而CPU发生了上下文切换称为经历一个quiescent state，grace period就是所有CPU都经历一次quiescent state所需要的等待的时间。垃圾收集器就是在grace period之后调用写者注册的回调函数来完成真正的数据修改或数据释放操作的”

要彻底弄清楚这个问题,我们得从RCU的初始化说起.

四:从RCU的初始化说起

RCU的初始化位于start_kernel()àrcu_init().代码如下:

```
void __init rcu_init(void)
{
    __rcu_init();
}

void __init __rcu_init(void)
{
    rcu_cpu_notify(&rcu_nb, CPU_UP_PREPARE,
            (void *)(long)smp_processor_id());
    /* Register notifier for non-boot CPUs */
    register_cpu_notifier(&rcu_nb);
}
```

Reqister_cpu_notifier()是关于通知链表的操作,可以忽略不看.

跟进rcu_cpu_notify():

```
static int __cpuinit rcu_cpu_notify(struct notifier_block *self,
                unsigned long action, void *hcpu)
{
    long cpu = (long)hcpu;

    switch (action) {
    case CPU_UP_PREPARE:
```

```
    case CPU_UP_PREPARE_FROZEN:
        rcu_online_cpu(cpu);
        break;
    case CPU_DEAD:
    case CPU_DEAD_FROZEN:
        rcu_offline_cpu(cpu);
        break;
    default:
        break;
    }
    return NOTIFY_OK;
}
```

注意到,在__rcu_init()中是以CPU_UP_PREPARE为参数调用此函数,对应流程转入rcu_online_cpu中:

```
static void __cpuinit rcu_online_cpu(int cpu)
{
    struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
    struct rcu_data *bh_rdp = &per_cpu(rcu_bh_data, cpu);

    rcu_init_percpu_data(cpu, &rcu_ctrlblk, rdp);
    rcu_init_percpu_data(cpu, &rcu_bh_ctrlblk, bh_rdp);
    open_softirq(RCU_SOFTIRQ, rcu_process_callbacks, NULL);
}
```

我们从这里又看到了另一个per_cpu变量,rcu_bh_data.有关bh的部份之后再来分析.在这里略过这些部份.

Rcu_init_percpu_data()如下:

```
static void rcu_init_percpu_data(int cpu, struct rcu_ctrlblk *rcp,
                    struct rcu_data *rdp)
{
    memset(rdp, 0, sizeof(*rdp));
    rdp->curtail = &rdp->curlist;
    rdp->nxttail = &rdp->nxtlist;
    rdp->donetail = &rdp->donelist;
    rdp->quiescbatch = rcp->completed;
    rdp->qs_pending = 0;
    rdp->cpu = cpu;
    rdp->blimit = blimit;
}
```

调用这个函数的第二个参数是一个全局变量rcu_ctlblk.定义如下:

```
static struct rcu_ctrlblk rcu_ctrlblk = {
    .cur = -300,
    .completed = -300,
    .lock = __SPIN_LOCK_UNLOCKED(&rcu_ctrlblk.lock),
    .cpumask = CPU_MASK_NONE,
```

```
};
static struct rcu_ctrlblk rcu_bh_ctrlblk = {
   .cur = -300,
   .completed = -300,
   .lock = __SPIN_LOCK_UNLOCKED(&rcu_bh_ctrlblk.lock),
   .cpumask = CPU_MASK_NONE,
};
```

在rcu_init_percpu_data中,初始化了三个链表,分别是taillist,curlist和donelist.另外, 将rdp->quiescbatch 赋值为 rcp->completed.这个是一个很重要的操作.

Rdp-> quiescbatch表示rcu_data已经完成的grace period序号(在代码中也被称为了batch),rcp->completed表示全部变量rcu_ctrlblk计数已经完成的grace period序号.将rdp->quiescbatch = rcp->completed;,表示不需要等待grace period.

回到rcu_online_cpu()中:

```
open_softirq(RCU_SOFTIRQ, rcu_process_callbacks, NULL);
```

初始化了RCU_SOFTIRQ类型的软中断.但这个软中断什么时候被打开,还需要之后来分析.

之后,每个CPU的初始化都会经过start_kernel()->rcu_init().相应的,也为每个CPU初始化了RCU的相关结构.

五:等待RCU读者操作完成

之前,我们看完了RCU的初始化,现在可以来看一下RCU如何来判断当前的RCU读者已经退出了.

在每一次进程切换的时候,都会调用rcu_qsctr_inc().如下代码片段如示:

```
asmlinkage void __sched schedule(void)
{
   · · · · · ·
   · · · · · ·
   rcu_qsctr_inc(cpu);
   ......
}
```

Rcu_qsctr_inc()代码如下:

```
static inline void rcu_qsctr_inc(int cpu)
{
   struct rcu_data *rdp = &per_cpu(rcu_data, cpu);
   rdp->passed_quiesc = 1;
}
```

该函数将对应CPU上的rcu_data的passed_quiesc成员设为了1.

或许你已经发现了,这个过程就标识该CPU经过了一次quiescent state.没错:-)

另外,在时钟中断中,会进行以下操作:

```
void update_process_times(int user_tick)
{
   · · · · · ·
   · · · · · ·
```

```
    if (rcu_pending(cpu))
        rcu_check_callbacks(cpu, user_tick);
    . . . . . .
    . . . . . .
}
```

在每一次时钟中断,都会检查是否有需要更新的RCU需要处理,如果有,就会为其调用rcu_check_callbacks().

Rcu_pending()的代码如下:

```
int rcu_pending(int cpu)
{
    return __rcu_pending(&rcu_ctrlblk, &per_cpu(rcu_data, cpu)) ||
        __rcu_pending(&rcu_bh_ctrlblk, &per_cpu(rcu_bh_data, cpu));
}
```

同上面一样,忽略bh的部份.

```
static int __rcu_pending(struct rcu_ctrlblk *rcp, struct rcu_data *rdp)
{
    /* This cpu has pending rcu entries and the grace period
     * for them has completed.
     */
    if (rdp->curlist && !rcu_batch_before(rcp->completed, rdp->batch))
        return 1;

    /* This cpu has no pending entries, but there are new entries */
    if (!rdp->curlist && rdp->nxtlist)
        return 1;

    /* This cpu has finished callbacks to invoke */
    if (rdp->donelist)
        return 1;

    /* The rcu core waits for a quiescent state from the cpu */
    if (rdp->quiescbatch != rcp->cur || rdp->qs_pending)
        return 1;

    /* nothing to do */
    return 0;
}
```

上面有四种情况会返回1,分别对应:

1:该CPU上有等待处理的回调函数,且已经经过了一个batch(grace period).rdp->datch表示rdp在等待的batch序号

2:上一个等待已经处理完了,又有了新注册的回调函数.

3:等待已经完成,但尚未调用该次等待的回调函数.

4:在等待quiescent state.
关于rcp和rdp结构中成员的含义,我们等用到的时候再来分析.

如果rcu_pending返回1,就会进入到rcu_check_callbacks().代码如下:

```
void rcu_check_callbacks(int cpu, int user)
{
   if (user ||
      (idle_cpu(cpu) && !in_softirq() &&
            hardirq_count()
      rcu_qsctr_inc(cpu);
      rcu_bh_qsctr_inc(cpu);
   } else if (!in_softirq())
      rcu_bh_qsctr_inc(cpu);
   raise_rcu_softirq();
}
```

如果已经CPU中运行的进程是用户空间进程或者是CPU空闲且不处于中断环境,那么,它也已经进过了一次切换.注意,RCU只能在内核空间使用.
最后调用raise_rcu_softirq()打开了软中断处理.相应的,也就调用RCU的软中断处理函数.结合上面分析的初始化流程,软中断的处理函数为rcu_process_callbacks().
代码如下:

```
static void rcu_process_callbacks(struct softirq_action *unused)
{
   __rcu_process_callbacks(&rcu_ctrlblk, &__get_cpu_var(rcu_data));
   __rcu_process_callbacks(&rcu_bh_ctrlblk, &__get_cpu_var(rcu_bh_data));
}
```

在阅读__rcu_process_callbacks()之前,先来了解一下rdp中几个链表的含义:
每次新注册的回调函数,都会链入到rdp->taillist.
当前等待grace period完成的函数都会链入到rdp->curlist上.
到等待的grace period已经到来,就会将curlist上的链表移到donelist上.
当一个grace period过了之后,就会将taillist上的数据移到rdp->curlist上.之后加册的回调函数又会将其加到rdp->taillist上.

__rcu_process_callbacks()代码分段分析如下:

```
static void __rcu_process_callbacks(struct rcu_ctrlblk *rcp,
            struct rcu_data *rdp)
{
   if (rdp->curlist && !rcu_batch_before(rcp->completed, rdp->batch)) {
      *rdp->donetail = rdp->curlist;
      rdp->donetail = rdp->curtail;
      rdp->curlist = NULL;
      rdp->curtail = &rdp->curlist;
   }
```

如果有需要处理的回调函数,且已经经过了一次grace period.就将curlist上的数据移到donetlist上.
其中,crp->completed表示已经完成的grace period.rdp->batch表示该CPU正在等待的grace period序号.

```
if (rdp->nxtlist && !rdp->curlist) {
    local_irq_disable();
    rdp->curlist = rdp->nxtlist;
    rdp->curtail = rdp->nxttail;
    rdp->nxtlist = NULL;
    rdp->nxttail = &rdp->nxtlist;
    local_irq_enable();

    /*
     * start the next batch of callbacks
     */

    /* determine batch number */
    rdp->batch = rcp->cur + 1;
    /* see the comment and corresponding wmb() in
     * the rcu_start_batch()
     */
    smp_rmb();

    if (!rcp->next_pending) {
        /* and start it/schedule start if it's a new batch */
        spin_lock(&rcp->lock);
        rcp->next_pending = 1;
        rcu_start_batch(rcp);
        spin_unlock(&rcp->lock);
    }
}
```

如果上一个等待的回调函数处理完了,而且又有了新注册的回调函数.就将taillist上的数据移动到curlist上.并开启新的grace period等待.
注意里面几个变量的赋值:
rdp->batch = rcp->cur + 1表示该CPU等待的grace period置为当前已发生grace period序号的下一个.
每次启动一个新的grace period等待之后,就会将rcp->next_pending.在启动的过程中,也就是rcu_start_batch()的过程中,会将rcp->next_pending置为1.设置这个变量主要是防止多个写者竞争的情况

```
//更新相关信息
rcu_check_quiescent_state(rcp, rdp);
//处理等待完成的回调函数
```

```
    if (rdp->donelist)
        rcu_do_batch(rdp);
}
```

接着,更新相关的信息,例如,判断当前CPU是否进行了quiescent state.或者grace period是否已经完成.
最后再处理挂在rdp->donelist上的链表.

这里面有几个子函数值得好好分析,分别分析如下:
第一个要分析的是rcu_start_batch():

```
static void rcu_start_batch(struct rcu_ctrlblk *rcp)
{
    if (rcp->next_pending &&
            rcp->completed == rcp->cur) {
        rcp->next_pending = 0;
        smp_wmb();
        rcp->cur++;
        smp_mb();
        cpus_andnot(rcp->cpumask, cpu_online_map, nohz_cpu_mask);

        rcp->signaled = 0;
    }
}
```

这个函数的代码虽然很简单,但隐藏了很多玄机.
每次启动一个新的grace period等待的时候就将rcp->cur加1,将rcp->cpumask中,将存在的CPU的位置1.
其中,if判断必须要满足二个条件:
第一:rcp->next_pending必须为1.我们把这个函数放到__rcu_process_callbacks()这个大环境中看一下:

```
static void __rcu_process_callbacks(struct rcu_ctrlblk *rcp,
                struct rcu_data *rdp)
{
    . . . . . .
    . . . . . .
    if (rdp->nxtlist && !rdp->curlist) {
        . . . . . .
        if (!rcp->next_pending) {
            /* and start it/schedule start if it's a new batch */
            spin_lock(&rcp->lock);
            rcp->next_pending = 1;
            rcu_start_batch(rcp);
            spin_unlock(&rcp->lock);
        }
    }
}
```

首先，rcp->next_pending为0才会调用rcu_start_batch()启动一个新的进程．然后，将rcp->next_pending置为1,再调用rcu_start_batch().在这里要注意中间的自旋锁.然后在rcu_start_batch()中,再次判断rcp->next_pending为1后,再进行后续操作,并将rcp->next_pending置为0.
为什么这里需要这样的判断呢? 如果其它CPU正在开启一个新的grace period等待,那就用不着再次开启一个新的等待了,直接返回即可.

第二: rcu_start_batch()中if要满足的第二个条件为rcp->completed == rcp->cur.也就是说前面的grace period全部都完成了.每次开启新等待的时候都会将rcp->cur加1.每一个等待完成之后,都会将rc-> completed等于rcp->cur.

第二个要分析的函数是rcu_check_quiescent_state().代码如下:

```
static void rcu_check_quiescent_state(struct rcu_ctrlblk *rcp,
                struct rcu_data *rdp)
{
   if (rdp->quiescbatch != rcp->cur) {
      /* start new grace period: */
      rdp->qs_pending = 1;
      rdp->passed_quiesc = 0;
      rdp->quiescbatch = rcp->cur;
      return;
   }

   /* Grace period already completed for this cpu?
    * qs_pending is checked instead of the actual bitmap to avoid
    * cacheline trashing.
    */
   if (!rdp->qs_pending)
      return;

   /*
    * Was there a quiescent state since the beginning of the grace
    * period? If no, then exit and wait for the next call.
    */
   if (!rdp->passed_quiesc)
      return;
   rdp->qs_pending = 0;

   spin_lock(&rcp->lock);
   /*
    * rdp->quiescbatch/rcp->cur and the cpu bitmap can come out of sync
    * during cpu startup. Ignore the quiescent state.
    */
   if (likely(rdp->quiescbatch == rcp->cur))
      cpu_quiet(rdp->cpu, rcp);
```

```
    spin_unlock(&rcp->lock);
}
```
首先,如果rdp->quiescbatch != rcp->cur.则说明又开启了一个新的等待,因此需要重新处理这个等待,首先将rdp->quiescbatch更新为rcp->cur.然后,使rdp->qs_pending为1.表示有等待需要处理. passed_quiesc也被清成了0.然后,再判断rdp->passed_quiesc是否为真,记得我们在之前分析过,在每次进程切换或者进程切换的时候,都会调用rcu_qsctr_inc().该函数会将rdp->passed_quiesc置为1.
因此,在这里判断这个值是为了检测该CPU上是否发生了上下文切换.
之后,就是一段被rcp->lock保护的一段区域.如果还是等待没有发生改变,就会调用cpu_quiet(rdp->cpu, rcp)将该CPU位清零.如果是一个新的等待了,就用不着清了,因为需要重新判断该CPU上是否发生了上下文切换.

cpu_quiet()函数代码如下:
```
static void cpu_quiet(int cpu, struct rcu_ctrlblk *rcp)
{
    cpu_clear(cpu, rcp->cpumask);
    if (cpus_empty(rcp->cpumask)) {
        /* batch completed ! */
        rcp->completed = rcp->cur;
        rcu_start_batch(rcp);
    }
}
```
它清除当前CPU对应的位,如果CPMMASK为空,对应所有的CPU都发生了进程切换,就会将rcp->completed = rcp->cur.并且根据需要是否开始一个grace period等待.

最后一个要分析的函数是rcu_do_batch().它进行的是清尾的工作.如果等待完成了,那就必须要处理donelist链表上挂载的数据了.代码如下:
```
static void rcu_do_batch(struct rcu_data *rdp)
{
    struct rcu_head *next, *list;
    int count = 0;

    list = rdp->donelist;
    while (list) {
        next = list->next;
        prefetch(next);
        list->func(list);
        list = next;
        if (++count >= rdp->blimit)
            break;
    }
```

```
    rdp->donelist = list;

    local_irq_disable();
    rdp->qlen -= count;
    local_irq_enable();
    if (rdp->blimit == INT_MAX && rdp->qlen
        rdp->blimit = blimit;

    if (!rdp->donelist)
        rdp->donetail = &rdp->donelist;
    else
        raise_rcu_softirq();
}
```

它遍历处理挂在链表上的回调函数.在这里,注意每次调用的回调函数有最大值限制.这样做主要是防止一次调用过多的回调函数而产生不必要系统负载.如果donelist中还有没处理完的数据,打开RCU软中断,在下次软中断到来的时候接着处理.

五:几种RCU情况分析

1:如果CPU

1上有进程调用rcu_read_lock进入临界区,之后退出来,发生了进程切换,新进程又通过rcu_read_lock进入临界区.由于RCU软中断中只判断一次上下文切换,因此,在调用回调函数的时候,仍然有进程处于RCU的读临界区,这样会不会有问题呢?

这样是不会有问题的.还是上面的例子:

```
        spin_lock(&foo_mutex);
        old_fp = gbl_foo;
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        synchronize_rcu();
        kfree(old_fp);
```

使用synchronize_rcu ()只是为了等待持有old_fd(也就是调用rcu_assign_pointer ()更新之前的gbl_foo)的进程退出.而不需要等待所有的读者全部退出.这是因为,在rcu_assign_pointer ()之后的读取取得的保护指针,已经是更新好的新值了.

2:上面分析的似乎是针对有挂载链表的CPU而言的,那对于只调用rcu_read_lock()的CPU,它们是怎么处理的呢?

首先,每次启动一次等待,肯定是会更新rcp->cur的.因此,在rcu_pending()的判断中,下面语句会被满足:

```
    if (rdp->quiescbatch != rcp->cur || rdp->qs_pending)
        return 1;
```

因此会进入到RCU的软中断.在软中断处理中:

rcu_process_callbacks() à __rcu_process_callbacks() àrcu_check_quiescent_state()

中,如果该CPU上有进程切换,就会各新rcp中的CPU 掩码数组.

3:如果一个CPU连续调用synchronize_rcu()或者call_rcu()它们会有什么影响呢?
如果当前有请求在等待,就会新请提交的回调函数挂到taillist上,一直到前一个等待完成,再将taillist的数据移到curlist,并开启一个新的等待,因此,也就是说,在前一个等待期间提交的请求,都会放到一起处理.也就是说,他们会共同等待所有CPU切换完成.
举例说明如下:
假设grace period时间是12ms.在12ms内,先后有A,B,C进程提交请求.
那系统在等待处理完后,交A,B,C移到curlist中,开始一个新的等待.

六:有关rcu_read_lock_bh()/rcu_read_unlock_bh()/call_rcu_bh().
在上面的代码分析的时候,经常看到带有bh的RCU代码.现在来看一下这些带bh的RCU是什么样的.

```
#define rcu_read_lock_bh() __rcu_read_lock_bh()
#define rcu_read_unlock_bh() __rcu_read_unlock_bh()

#define __rcu_read_lock_bh() \
   do { \
      local_bh_disable(); \
      __acquire(RCU_BH); \
      rcu_read_acquire(); \
   } while (0)
#define __rcu_read_unlock_bh() \
   do { \
      rcu_read_release(); \
      __release(RCU_BH); \
      local_bh_enable(); \
   } while (0)
```

根据上面的分析:bh RCU跟普通的RCU相比不同的是,普通RCU是禁止内核抢占,而bh RCU是禁止下半部.
其实,带bh的RCU一般在软中断使用,不过计算quiescent state并不是发生一次上下文切换.而是发生一次softirq.我们在后面的分析中可得到印证.

Call_rcu_bh()代码如下:

```
void call_rcu_bh(struct rcu_head *head,
            void (*func)(struct rcu_head *rcu))
{
   unsigned long flags;
   struct rcu_data *rdp;

   head->func = func;
   head->next = NULL;
   local_irq_save(flags);
   rdp = &__get_cpu_var(rcu_bh_data);
```

```
    *rdp->nxttail = head;
    rdp->nxttail = &head->next;

    if (unlikely(++rdp->qlen > qhimark)) {
        rdp->blimit = INT_MAX;
        force_quiescent_state(rdp, &rcu_bh_ctrlblk);
    }

    local_irq_restore(flags);
}
```

它跟call_rcu()不相同的是,rcu是取per_cpu变量rcu__data和全局变量rcu_ctrlblk.而bh RCU是取rcu_bh_data,rcu_bh_ctrlblk.他们的类型都是一样的,这样做只是为了区分BH和普通RCU的等待.

对于rcu_bh_qsctr_inc

```
static inline void rcu_bh_qsctr_inc(int cpu)
{
    struct rcu_data *rdp = &per_cpu(rcu_bh_data, cpu);
    rdp->passed_quiesc = 1;
}
```

它跟rcu_qsctr_inc()机同,也是更改对应成员.

所不同的是,调用rcu_bh_qsctr_inc()的地方发生了变化.

```
asmlinkage void __do_softirq(void)
{
    . . . . . .
        do {
        if (pending & 1) {
            h->action(h);
            rcu_bh_qsctr_inc(cpu);
        }
        h++;
        pending >>= 1;
    } while (pending);
    . . . . . .
}
```

也就是说，在发生软中断的时候，才会认为是经过了一次quiescent state.