

公告

# Linux RCU 机制详解

## 阅读目录

- 1、简介：
- 2、应用场景：
- 3、相应资料：
- 4、实现过程：
  - 4.1 宽限期：
  - 4.2 订阅——发布机制：
  - 4.3 数据读取的完整性：
- 5、小结：

昵称：yooooooo  
园龄：4年  
粉丝：72  
关注：2  
+加关注

<	2019年10月						>
日	一	二	三	四	五	六	
29	30	1	2	3	4	5	
6	7	8	9	10	11	12	
13	14	15	16	17	18	19	
20	21	22	23	24	25	26	
27	28	29	30	31	1	2	
3	4	5	6	7	8	9	

搜索

找找看

谷歌搜索

常用链接

我的随笔  
我的评论  
我的参与  
最新评论  
我的标签

积分与排名

积分 - 182028  
排名 - 2391

随笔分类

- Android(8)
- Android recovery模式(4)
- ARM(11)
- Audio(12)

## 正文

回到顶部

### 1、简介：

RCU ( Read-Copy Update ) 是数据同步的一种方式，在当前的Linux内核中发挥着重要的作用。

RCU主要针对的数据对象是链表，目的是提高遍历读取数据的效率，为了达到目的使用RCU机制读取数据的时候不对链表进行耗时的加锁操作。这样在同一时间可以有多个线程同时读取该链表，并且允许一个线程对链表进行修改（修改的时候，需要加锁）。

回到顶部

### 2、应用场景：

RCU适用于需要频繁的读取数据，而相应修改数据并不多的情景，例如在文件系统中，经常需要查找定位目录，而对目录的修改相对来说并不多，这就是RCU发挥作用的最佳场景。

回到顶部

### 3、相应资料：

Linux内核源码当中,关于RCU的文档比较齐全，你可以在 /Documentation/RCU/ 目录下找到这些文件。

Paul E. McKenney 是内核中RCU源码的主要实现者，他也写了很多RCU方面的文章。他把这些文章和一些关于RCU的论文的链接整理到了一起。相应链接如下：

http://www2.rdrop.com/users/paulmck/RCU/

00

回到顶部

- Battery Driver(16)
- Bootloader(12)
- C Programming(13)
- Cache(9)
- Camera(6)
- CPU(9)
- DevOps(5)
- Display Driver(16)
- Driver(21)
- File System(17)
- Git(2)
- Hardware(3)
- Hisilicon(3)
- IPC(1)
- Kernel(33)
- Linux(6)
- Linux 内核完全剖析-基于0.12内核(3)
- Makefile(6)
- Memory Management(55)
- MMC driver(10)
- Modem(1)
- NFC Driver(2)
- OOL(8)
- Process Management(36)
- Protocol(9)
- Python(1)
- Real-time system(19)
- Script(2)
- Security(5)
- Sensor(7)
- TCP/IP 协议栈(10)
- USB(3)
- 程序员的自我修养(10)
- 读书笔记
- 设计模式(2)
- 视频编码(2)
- 数据结构与算法分析：C语言描述(11)
- 网络流媒体(8)

## 随笔档案

- 2019年10月(10)
- 2019年9月(21)
- 2019年8月(11)
- 2019年7月(15)
- 2019年6月(17)
- 2019年5月(15)
- 2019年4月(16)
- 2019年3月(15)
- 2019年2月(14)
- 2019年1月(12)
- 2018年12月(23)
- 2018年11月(27)
- 2018年10月(9)

## 4. 实现过程：

在RCU的实现过程中，我们主要解决以下问题：

- 1, 在读取过程中，另外一个线程删除了一个节点。删除线程可以把这个节点从链表中移除，但它不能直接销毁这个节点，必须等到所有的读取线程读取完成以后，才进行销毁操作。RCU中把这个过程称为宽限期（Grace period）。
- 2, 在读取过程中，另外一个线程插入了一个新节点，而读线程读到了这个节点，那么需要保证读到的这个节点是完整的。这里涉及到了发布-订阅机制（Publish-Subscribe Mechanism）。
- 3, 保证读取链表的完整性。新增或者删除一个节点，不至于导致遍历一个链表从中间断开。但是RCU并不保证一定能读到新增的节点或者不读要被删除的节点。

## 4.1 宽限期：

通过例子，方便理解这个内容。以下例子修改于Paul的文章。

```
1 struct foo{
2     int a;
3     char b;
4     long c;
5 };
6
7 DEFINE_SPINLOCK(foo_mutex);
8
9 void foo_read(void)
10 {
11     foo *fp = gbl_foo;
12     if( fp != NULL )
13     {
14         dosomething(fp->a, fp->b, fp->c);
15     }
16 }
17
18 void foo_update(foo * new_fp)
19 {
20     spin_lock(&foo_mutex);
21     foo *old_fp = gbl_foo;
22     gbl_foo = new_fp;
23     spin_unlock(&foo_mutex);
24 }
```

如上的程序，是针对全局变量gbl\_foo的操作。假设以下场景。有两个线程同时运行foo\_read和foo\_update的时候，当foo\_read执行完赋值操作后，线程发生切换；此时另一个线程开始执行foo\_update并执行完成。当foo\_read运行的进程切换回来后，运行dosomething的时候，fp已经被删除，这将对系统造成危害。为了防止此类事件的发生，RCU里增加了一个新的概念叫宽限期（Grace period）。如下图所示：



- 2018年9月(22)
- 2018年8月(9)
- 2018年7月(10)
- 2018年6月(21)
- 2018年5月(11)
- 2018年4月(14)
- 2018年3月(15)
- 2018年2月(11)
- 2018年1月(15)
- 2017年12月(10)
- 2017年11月(12)
- 2017年10月(4)
- 2017年9月(10)
- 2017年8月(8)
- 2017年7月(6)
- 2017年6月(5)
- 2017年5月(5)
- 2017年4月(9)
- 2017年3月(3)
- 2017年2月(1)
- 2016年12月(1)
- 2016年11月(1)
- 2016年10月(1)
- 2016年9月(2)
- 2016年7月(2)
- 2016年6月(1)

#### 最新评论

##### 1. Re:Android音频系统

@ 我呆不了一个月了这些都是韦东山的视频上的图片，我暂时发不了呢...  
--yooooooo

##### 2. Re:Android音频系统

楼主您好，请问文中的图片有较为清楚版本吗？  
--我呆不了一个月了

##### 3. Re:6. [mmc subsystem] mmc core (第六章)——mmc core主模块

看了一系列的文章，只能感叹大佬大佬

--不知也

##### 4. Re:生成前N个自然数随机置换的3个程序

算法3写错了洗牌算法每个元素和未确定元素交换一次之后就确定 你的做法应该从高循环到低 for (int i = n-1; i > 0; i--) { swap(a[i], a[ RandInt(0,...

--求道于盲

##### 5. Re:MMU工作原理

@ 1577670619@1577670619引用在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写。而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存...

--yooooooo

#### 阅读排行榜

1. Git的gc功能(7155)
2. C语言函数不定参数实现方式(4990)
3. main函数是主线程吗(4096)
4. 信号量、互斥锁、自旋锁、原子操作(3939)
5. 高通GPIO驱动 ( DTS 方式 ) (3929)

图中每行代表一个线程，最下面的一行是删除线程，当它执行完删除操作后，线程进入了宽限期。宽限期（grace period）的意思是，在一个删除动作发生后，它必须等待所有在宽限期开始前已经开始的读线程结束，才可以进行销毁操作。这是因为这些线程有可能读到了要删除的元素。图中的宽限期必须等待1和2结束；而读线程5在宽限期开始前已经结束，不需要考虑；而3,4,6也不需要考虑，因为在宽限期结束后开始后的线程不可能读到已删除的元素。为此RCU机制提供了相应的API来实现这个功能。

```
1 void foo_read(void)
2 {
3     rcu_read_lock();
4     foo *fp = gbl_foo;
5     if( fp != NULL )
6         dosomething(fp->a, fp->b, fp->c);
7     rcu_read_unlock();
8 }
9
10 void foo_update(foo *new_fp)
11 {
12     spin_lock(&foo_mutex);
13     foo *old_fp = gbl_foo;
14     gbl_foo = new_fp;
15     spin_unlock(&foo_mutex);
16     synchronize_rcu();
17     kfree(old_fp);
18 }
```

其中foo\_read中增加了rcu\_read\_lock和rcu\_read\_unlock，这两个函数用来标记一个RCU读过程的开始和结束。其实作用就是帮助检测宽限期是否结束。foo\_update增加了一个函数synchronize\_rcu()，调用该函数意味着一个宽限期的开始，而直到宽限期结束，该函数才会返回。我们再对比着图看一看，线程1和2，在synchronize\_rcu之前可能得到了旧的gbl\_foo，也就是foo\_update中的old\_fp，如果不等它们运行结束，就调用kfree(old\_fp)，极有可能造成系统崩溃。而3,4,6在synchronize\_rcu之后运行，此时它们已经不可能得到old\_fp，此次的kfree将不对它们产生影响。

宽限期是RCU实现中最复杂的部分,原因是在提高读数据性能的同时，删除数据的性能也不能太差。

#### 4.2 订阅——发布机制

当前使用的编译器大多会对代码做一定程度的优化，CPU也会对执行指令做一些优化调整,目的是提高代码的执行效率，但这样的优化，有时候会带来不期望的结果。如例：

```
1 void foo_update(foo *new_fp)
2 {
3     spin_lock(&foo_mutex);
4     foo *old_fp = gbl_foo;
5
6     new_fp->a = 1;
7     new_fp->b = 'b';
8     new_fp->c = 100;
9
10    gbl_foo = new_fp;
11    spin_unlock(&foo_mutex);
12    synchronize_rcu();
13    kfree(old_fp);
14 }
```

这段代码中，我们期望的是6，7，8行的代码在第10行代码之前执行。但优化后的代码并不对执行顺序做出保证。在这种情形下，一个读线程很可能读到了 new\_fp，但new\_fp的成员赋值还没执行完成。当读线程执行

0

0

## 评论排行榜

1. 什么是重定位？为什么需要重定位？【转】(4)
2. I2C通讯协议(3)
3. C语言函数不定参数实现方式(2)
4. Linux内核书籍(2)
5. 高通调试 SPI 屏的 开机一段时间黑屏(2)

## 推荐排行榜

1. 静态库和动态库的区别(3)
2. 信号量、互斥锁、自旋锁、原子操作(3)
3. Linux内存描述之概述--Linux内存管理(一)(3)
4. Memory Map(2)
5. 2. Linux-3.14.12内存管理笔记【系统启动阶段的memblock算法(2)】(2)

dosomething(fp->a, fp->b, fp->c) 的时候，就有不确定的参数传入到dosomething，极有可能造成不可预期的结果，甚至程序崩溃。可以通过优化屏障来解决该问题，RCU机制对优化屏障做了包装，提供了专用的API来解决该问题。这时候，第十行不再是直接的指针赋值，而应该改为：

```
rcu_assign_pointer(gbl_foo,new_fp);
```

rcu\_assign\_pointer的实现比较简单，如下：

```
1 #define rcu_assign_pointer(p, v) \
2     __rcu_assign_pointer((p), (v), __rcu)
```

```
1 #define RCU_INIT_POINTER(p, v) \
2     p = (typeof(*v)) __force __rcu *) (v)
```

在DEC Alpha CPU机器上还有一种更强悍的优化，如下所示：

```
1 void foo_read(void)
2 {
3     rcu_read_lock();
4     foo *fp = gbl_foo;
5     if ( fp != NULL )
6         dosomething(fp->a, fp->b, fp->c);
7     rcu_read_unlock();
8 }
```

第六行的 fp->a,fp->b,fp->c会在第3行还没执行的时候就预先判断运行，当他和foo\_update同时运行的时候，可能导致传入dosomething的一部分属于旧的gbl\_foo，而另外的属于新的。这样导致运行结果的错误。为了避免该类问题，RCU还是提供了宏来解决该问题：

```
1 #define rcu_dereference_check(p, c) \
2     __rcu_dereference_check((p), rcu_read_lock_held() || (c), __rcu)
3
4 #define __rcu_dereference_check(p, c, space) \
5     ({ \
6         typeof(*p) *_____p1 = (typeof(*p))__force ACCESS_ONCE(p); \
7         rcu_lockdep_assert(c, "suspicious rcu_dereference_check()" \
8             " usage"); \
9         rcu_dereference_sparse(p, space); \
10        smp_read_barrier_depends(); \
11        ((typeof(*p)) __force __kernel *) (_____p1)); \
12    })
13
14 static inline int rcu_read_lock_held(void)
15 {
16     if (!debug_lockdep_rcu_enabled())
17         return 1;
18     if (rcu_is_cpu_idle())
19         return 0;
20     if (!rcu_lockdep_current_cpu_online())
21         return 0;
22     return lock_is_held(&rcu_lock_map);
23 }
```

这段代码中加入了调试信息，去除调试信息，可以是以下的形式（其实这也是旧版本中的代码）：

```
1 #define rcu_dereference_check(p) ({ \
2     typeof(p) _____p1 = p; \
3     smp_read_barrier_depends(); \
```

Fork me on GitHub

```

4         (____p1); \
5     })

```

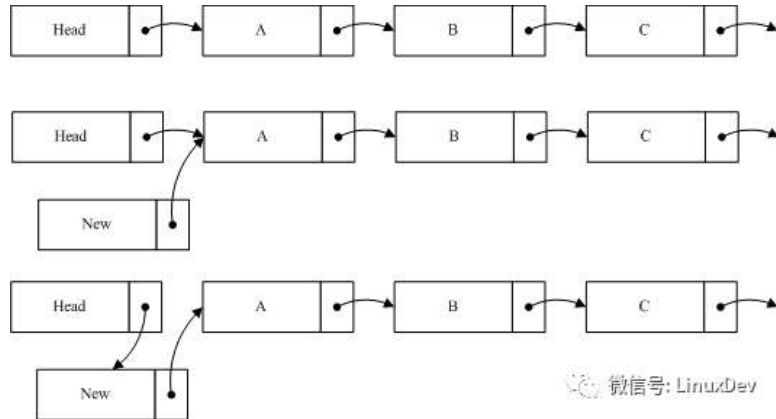
Fork me on GitHub

在赋值后加入优化屏障`smp_read_barrier_depends()`。

我们之前的第四行代码改为 `foo *fp = rcu_dereference(gbl_foo);`，就可以防止上述问题。

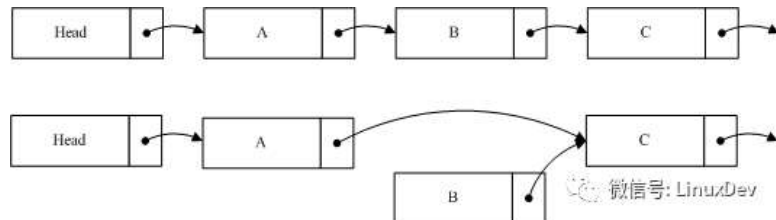
### 4.3 数据读取的完整性：

还是通过例子来说明这个问题：



如图我们在原list中加入一个节点new到A之前，所要做的第一步是将new的指针指向A节点，第二步才是将Head的指针指向new。这样做的目的是当插入操作完成第一步的时候，对于链表的读取并不产生影响，而执行完第二步的时候，读线程如果读到new节点，也可以继续遍历链表。如果把这个过程反过来，第一步head指向new，而这时一个线程读到new，由于new的指针指向的是Null，这样将导致读线程无法读取到A，B等后续节点。从以上过程中，可以看出RCU并不保证读线程读取到new节点。如果该节点对程序产生影响，那么就需要外部调用做相应的调整。如在文件系统中，通过RCU定位后，如果查找不到相应节点，就会进行其它形式的查找，相关内容等分析到文件系统的时候再进行叙述。

我们再看一下删除一个节点的例子：



如图我们希望删除B，这时候要做的就是将A的指针指向C，保持B的指针，然后删除程序将进入宽限期检测。由于B的内容并没有变更，读到B的线程仍然可以继续读取B的后续节点。B不能立即销毁，它必须等待宽限期结束后，才能进行相应销毁操作。由于A的节点已经指向了C，当宽限期开始之后所有的后续读操作通过A找到的是C，而B已经隐藏了，后续的读线程都不会读到它。这样就确保宽限期过后，删除B并不对系统造成影响。

[回到顶部](#)

### 5、小结：

RCU的原理并不复杂，应用也很简单。但代码的实现确并不是那么容易，难点都集中在了宽限期的检测上，后续分析源代码的时候，我们可以看到一些极富技巧的实现方式。

如果您觉得阅读本文对您有帮助，请点一下“推荐”按钮，您的“推荐”将是我最大的写作动力！

分类: [Kernel](#)

0

0

[好文要顶](#)
[关注我](#)
[收藏该文](#)




yooooooo

关注 - 2

粉丝 - 72

+加关注

« 上一篇： 如何使用C语言的面向对象  
» 下一篇： 什么是负载均衡？【转】

posted @ 2018-02-23 21:23 yooooooo 阅读(1433) 评论(0) 编辑 收藏



[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)， [访问](#) 网站首页。

- 【推荐】超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
- 【活动】京东云服务器\_云主机低于1折，低价高性能产品备战双11
- 【推荐】天翼云新用户专享，0元体验数十款云产品，立即开通
- 【活动】魔程社区技术沙龙—移动测试应用专场等你报名
- 【福利】学AI有奖：博客园&华为云 Modelarts 有奖训练营

相关博文：

- [Linux RCU机制详解](#)
- [RCU原理分析](#)
- [Linux RCU机制详解 \[ 转 \]](#)
- [Linux RCU 机制详解](#)
- [深入理解 Linux 的 RCU 机制](#)

最新 IT 新闻

- MIT的测试证明 达芬奇500多年前的桥梁设计仍具有可行性
- 首次公开！中国火星探测器"真容"曝光 预计明年发射
- 苹果收购Intel基带"未审先合并" 反垄断监管启动问询
- 研究称星际彗星2I/Borisov的“尾巴”将携带有毒气体
- 它是臭名昭著的兴奋剂，也是诺奖梦开始的地方
- » [更多新闻...](#)