lab3_rcu 实验说明

本实验是一个 rcu 的测试实验。
编写一个简单的内核模块，创建一个读者内核线程和一个写者内核线程来模拟同步访问共享变量的情景。

基本实验步骤

1. 进入 rlk_lab/rlk_basic/chapter_9/lab3 目录。

```
# export ARCH=arm
# export CROSS_COMPILE=arm-linux-gnueabi-
# make BASEINCLUDE=/home/figo/work/runninglinuxkernel/runninglinuxkernel_4.0
```

这里 BASEINCLUDE 指定你当前 runninglinuxkernel_4.0 的目录路径。

然后把 rcu.ko 拷贝到 runninglinuxkernel_4.0/kmodules 目录下面。

运行如下脚本启动 Qemu。

```
#cd runninglinuxkernel_4.0
# sh run.sh arm32    #启动虚拟机
```

在 Qemu 虚拟机:

```
#cd /mnt
# insmod rcu.ko
```

```
/mnt # insmod rcu.ko
figo: my module init
/mnt # myrcu_reader_thread1: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread1: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread1: read a=0
myrcu_reader_thread1: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread2: read a=0
myrcu_reader_thread1: read a=0
myrcu_writer_thread: write to new 5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_del: a=0
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_reader_thread2: read a=5
myrcu_reader_thread2: read a=5
myrcu_reader_thread1: read a=5
myrcu_writer_thread: write to new 6
myrcu_reader_thread2: read a=6
myrcu_reader_thread1: read a=6
myrcu_del: a=5
```

可以看到：

该例子的目的是通过 RCU 机制保护 my_test_init()分配的共享数据结构 g_ptr，另外创建了一个读者线程和一个写者线程来模拟同步场景。

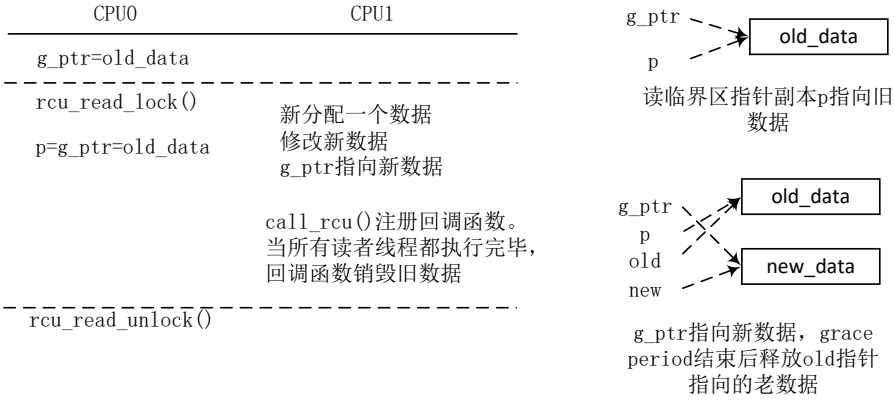
对于读者线程 myrcu_reader_thread：

- 通过 rcu_read_lock()和 rcu_read_unlock()来构建一个读者临界区。
- 调用 rcu_dereference()获取被保护数据 g_ptr 指针的一个副本，即指针 p，这时 p 和 g_ptr 都指向旧的被保护数据。
- 读者线程每隔一段时间读取一次被保护数据。

对于写者线程 myrcu_writer_thread：

- 分配一个新的保护数据 new_ptr，并修改相应数据。
- rcu_assign_pointer()让 g_ptr 指向新数据。
- call_rcu()注册一个回调函数，确保所有对旧数据的引用都执行完成之后，才调用回调函数来删除旧数据 old_data。
- 写者线程每隔一段时间修改被保护数据。

上述过程如图所示。



RCU时序图

在所有的读访问完成之后，内核可以释放旧数据，对于何时释放旧数据，内核提供了两个 API 函数：synchronize_rcu()和 call_rcu()。

进阶思考

笨叔在实验里，设置一个进阶思考的问题。
代码里有一个 test_issue.c 文件，最大的不同就是使用 malloc 来分配 user buffer，而不是通过 mmap 来分配的匿名页面。

```
34
35          read_buffer = malloc(len);
36          if (!read_buffer)
37                  goto open_fail;
38
39          write_buffer = malloc(len);
40          if (!write_buffer)
41                  goto buffer_fail;
42
```

运行 test_issue 程序，得到如下结果：

```
/mnt # ./test_issue
demodrv_open: major=10, minor=58
driver max buffer size=4096
demodrv_read_write: len=4096, npage=1
pin 1 pages from user done
demodrv_read_write: write user buffer 4096 bytes done
demodrv_write: write nbytes=4096 done at pos=0
demodrv_read_write: len=4096, npage=1
pin 1 pages from user done
demodrv_read_write: read user buffer 4096 bytes done
demodrv_read: read nbytes=4096 done at pos=0
buffer compare fail
```

程序的最后结果是“buffer compare fail”，说明程序运行有问题，也就是 read buffer 和 write buffer 数据不对，究竟怎么回事呢？

遇到这种问题，我们最简单最粗暴也是最有效的调试办法就是把 buffer 打印出来看看。在内核里，可以使用 print_hex_dump_bytes()函数。比如在 demodrv_read_write()函数的第 61,67,72 行添加打印语句。

```
59          for (i = 0; i < npages; i++) {
60                  kmap_addr = kmap(pages[i]);
61                  //print_hex_dump_bytes("kmap:", DUMP_PREFIX_OFFSET, kmap_addr, PAGE_SIZE);
62                  size = min_t(size_t, PAGE_SIZE, len);
63                  switch(rw) {
64                  case MYDEMO_READ:
65                          memcpy(kmap_addr, dev_buf + PAGE_SIZE *i,
66                                          size);
67                          //print_hex_dump_bytes("read:", DUMP_PREFIX_OFFSET, kmap_addr, size);
68                          break;
69                  case MYDEMO_WRITE:
70                          memcpy(dev_buf + PAGE_SIZE*i, kmap_addr,
71                                          size);
72                          //print_hex_dump_bytes("write:", DUMP_PREFIX_OFFSET, dev_buf + PAGE_SIZE*i, size);
73                          break;
74                  default:
75                          break;
76                  }
```

打印结果如下：

```
pin 1 pages from user done
kmap:00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:000000c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
kmap:000000d0: 00 00 00 00 09 10 00 00 55 55 55 55 55 55 55 55  ........UUUUUUUU
kmap:000000e0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:000000f0: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:00000100: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:00000110: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:00000120: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:00000130: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
kmap:00000140: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55  UUUUUUUUUUUUUUUU
```

发现 pin 的 page，开头的数据都是 0，而不是 0x55，这是为什么呢？理论上这个 page 应该全部都是 0x55 才对。

这是为什么呢？

如果大家对这个问题感兴趣，可以关注笨叔的第一季旗舰篇视频，笨叔会在视频中和大家详细解答。

微店：

微店 奔跑吧Linux内核

扫码识别

淘宝店：https://shop115683645.taobao.com/

微信公众号：