lab2_mutex 实验说明

在第 5 章的虚拟 FIFO 设备中，我们并没有考虑多个进程同时访问设备驱动的情况，请使用互斥锁对虚拟 FIFO 设备驱动程序进行并发保护

基本实验步骤

1. 进入 rlk_lab/rlk_basic/chapter_9/lab2 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。

编译 test 测试 app。

```
# arm-linux-gnueabi-gcc test.c -o test
```

然后把 mydemo_fasync.ko 和 test 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 mydemo_fasync.ko
```

```
/mnt # insmod mydemo_fasync.ko
my_class mydemo:252:0: create device: 252:0
mydemo_fifo=ee098f5c
my_class mydemo:252:1: create device: 252:1
mydemo_fifo=ee0981dc
my_class mydemo:252:2: create device: 252:2
mydemo_fifo=ee09829c
my_class mydemo:252:3: create device: 252:3
mydemo_fifo=ee09835c
my_class mydemo:252:4: create device: 252:4
mydemo_fifo=ee09841c
my_class mydemo:252:5: create device: 252:5
mydemo_fifo=ee09889c
my_class mydemo:252:6: create device: 252:6
mydemo_fifo=ee098e9c
my_class mydemo:252:7: create device: 252:7
mydemo_fifo=ee09859c
succeeded register char device: mydemo_dev
```

你会看到创建了 8 个设备。你可以到/sys/class/my_class/目录下面看到这些设备。

```
/mnt # cd /sys/class/my_class/
/sys/class/my_class # ls
mydemo:252:0  mydemo:252:2  mydemo:252:4  mydemo:252:6
mydemo:252:1  mydemo:252:3  mydemo:252:5  mydemo:252:7
/sys/class/my_class #
```

我们可以看到创建了主设备号为 252 的设备。我们再来看一下/dev/目录。

```
/sys/class/my_class # ls -l /dev
total 0
crw-rw----    1 0        0          14,   4 Feb  1 09:46 audio
crw-rw----    1 0        0           5,   1 Feb  1 09:46 console
crw-rw----    1 0        0          10,  63 Feb  1 09:46 cpu_dma_latency
crw-rw----    1 0        0          14,   3 Feb  1 09:46 dsp
crw-rw----    1 0        0          29,   0 Feb  1 09:46 fb0
crw-rw----    1 0        0          29,   1 Feb  1 09:46 fb1
crw-rw----    1 0        0           1,   7 Feb  1 09:46 full
crw-rw----    1 0        0          10, 183 Feb  1 09:46 hwrng
drwxr-xr-x    2 0        0              120 Feb  1 09:46 input
crw-rw----    1 0        0           1,   2 Feb  1 09:46 kmem
crw-rw----    1 0        0           1,  11 Feb  1 09:46 kmsg
crw-rw----    1 0        0           1,   1 Feb  1 09:46 mem
crw-rw----    1 0        0          10,  60 Feb  1 09:46 memory_bandwidth
crw-rw----    1 0        0          14,   0 Feb  1 09:46 mixer
crw-rw----    1 0        0          90,   0 Feb  1 09:46 mtd0
```

发现并没有主设备为 252 的设备。

所以我们需要手工创建一个设备用来 test app。

```
#mknod /dev/mydemo0 c 252 1
```

接下来跑我们的 test 程序：

```
# ./test &     #这里让 test 程序在后台跑
```

```
/mnt # ./test &
/mnt # my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
my_class mydemo:252:1: demodrv_fasync send SIGIO
```

然后使用 echo 命令来往/dev/mydemo0 这个设备写入字符串。

```
/mnt # echo "i am study linux now" > /dev/mydemo0
my_class mydemo:252:1: demodrv_open: major=252, minor=1, device=mydemo_dev1
demodrv_write kill fasync
my_class mydemo:252:1: demodrv_write:mydemo_dev1 pid=700, actual_write =21, ppos=0, ret=0
FIFO is not empty
my_class mydemo:252:1: demodrv_read:mydemo_dev1, pid=772, actual_readed=21, pos=0
i am study linux now
```

可以看到从 demodrv_read()函数把刚才写入的字符串已经读到用户空间了。

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                  }
```

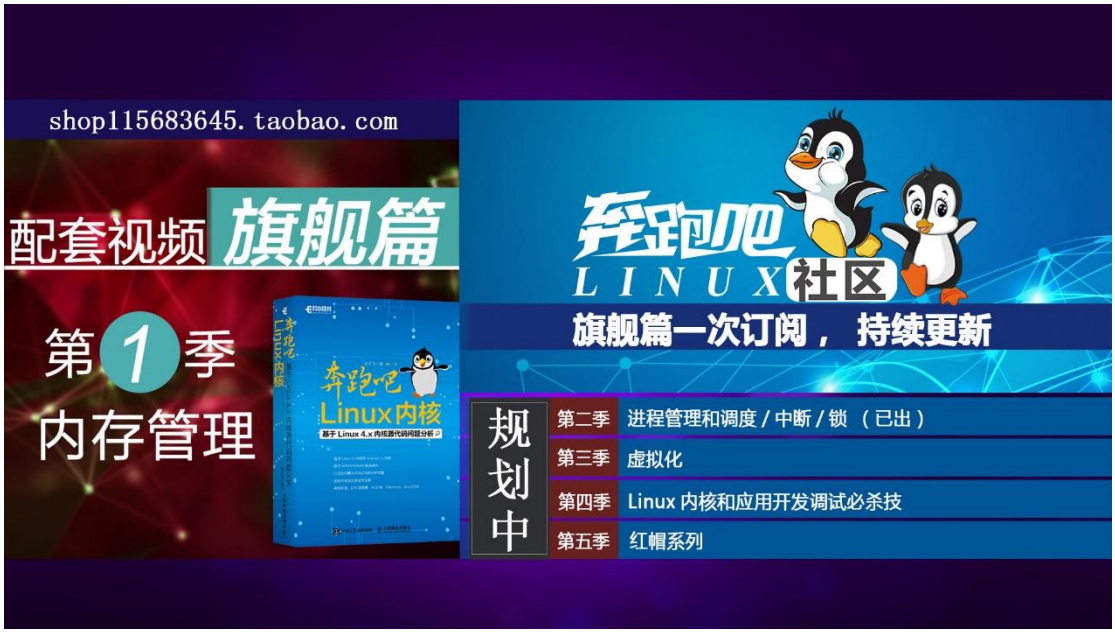
打印结果如下：

```
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。


shop115683645.taobao.com
配套视频 旗舰篇
第1季
内存管理
奔跑吧 Linux内核
基于 Linux 4.x 内核源代码问题分析
奔跑吧 LINUX社区
旗舰篇一次订阅， 持续更新
规划中
第二季 进程管理和调度 / 中断 / 锁 （已出）
第三季 虚拟化
第四季 Linux 内核和应用开发调试必杀技
第五季 红帽系列


shop115683645.taobao.com
Linux视频课程
奔跑吧 LINUX社区
微信公众号： 奔跑吧 linux 社区
1. 一键订阅， 持续更新
2. 最有深度和广度的 Linux 视频
3. 手把手解读 Linux 内核代码
4. 紧跟 Linux 开源社区技术热点
5. 笨叔叔的 VIP 私密群答疑
6. 图书 + 视频， 全新学习模式

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：