

Linux/Android开发记录

学习、记录、分享Linux/Android开发技术

目录视图

摘要视图

RSS 订阅

个人资料



liuhaoyutz

访问: 80608次

积分: 1673分

排名: 第7877名

原创: 83篇

转载: 0篇

译文: 0篇

评论: 59条

博客声明

本博客文章均为原创，欢迎转载交流。转载请注明出处，禁止用于商业目的。

博客专栏



Android应用开发学习笔记本

文章: 30篇

阅读: 17067



LDD3源码分析

文章: 17篇

阅读: 29965

文章分类

LDD3源码分析 (18)

ADC驱动 (1)

触摸屏驱动 (1)

LCD驱动 (1)

Linux设备模型 (8)

USB驱动 (0)

Android架构分析 (12)

Cocos2d-x (1)

C陷阱与缺陷 (3)

Android应用开发 (30)

Linux设备驱动程序架构分析 (8)

有奖征资源，博文分享有内涵

5月推荐博文汇总

大数据读书汇--获奖名单公布

2014 CSDN博文大赛

LDD3源码分析之阻塞型I/O

分类: LDD3源码分析

2012-03-26 15:00

1501人阅读

评论(3)

收藏

举报

struct

semaphore

asynchronous

buffer

signal

up

作者: 刘昊昱

博客: <http://blog.csdn.net/liuhaoyutz>

编译环境: Ubuntu 10.10

内核版本: 2.6.32-38-generic-pae

LDD3源码路径: examples/scull/pipe.c examples/scull/main.c

本文分析LDD3第六章介绍的scullpipe设备是如何实现阻塞I/O的。另外，我发现scullpipe的实现代码有一个问题，在文章的最后，对这个问题进行了说明，并给出了修正代码。

一、scullpipe设备实现阻塞I/O分析

scuopipe设备被实现为具有类似管道特性，用以演示如何实现阻塞型和非阻塞型I/O。实际的设备驱动中，这通常通过硬件中断的方式实现：当等待事件发生时，硬件发出一个中断，然后在中断处理程序中，驱动程序会唤醒等待进程。作为一个演示程序，scullpipe没有利用中断，而是选择使用另一个进程来产生数据并唤醒读取进程，类似的，利用读取进程来唤醒等待缓冲区可用的写入进程。

scullpipe的主体实现在examples/scull/pipe.c中，但是也利用了examples/scull/main.c中的一些代码。我们分析scullpipe的起点在main.c中，首先看main.c中的如下代码：

```
[cpp]
648      /* Initialize each device. */
649      for (i = 0; i < scull_nr_devs; i++) {
650          scull_devices[i].quantum = scull_quantum;
651          scull_devices[i].qset = scull_qset;
652          init_MUTEX(&scull_devices[i].sem);
653          scull_setup_cdev(&scull_devices[i], i);
654      }
655
656      /* At this point call the init function for any friend device */
657      dev = MKDEV(scull_major, scull_minor + scull_nr_devs);
658      dev += scull_p_init(dev);
659      dev += scull_access_init(dev);
```

649 - 654行，我们在分析第三章中的scull设备时已经分析过了，用for循环初始化scull0 - scull3，但当时657 - 659行我们跳过没有分析。

现在来看657行，scull_major和scull_minor的默认值都是0，scull_nr_devs的默认值是4。dev变量在618行定义，它是dev_t类型，注意dev_t类型用来保存设备编号，包括主设备号和次设备号。所以，657行，我们通过MKDEV宏得到一个主设备号为0，次设备号为4的设备编号保存在dev中。这里之所以让

<http://blog.csdn.net/liuhaoyutz/article/details/7395057>

1/10

最新评论

- LDD3源码分析之内存映射
wzw88486969:
@jhlhlonng:unsigned long offset
= vma->vm_pgoff <v...
Linux设备驱动程序架构分析之I2
teamos:看了你的i2c的几篇文
章，真是受益匪浅，虽然让自己
写还是ie不出来。非常感谢
LDD3源码分析之块设备驱动程序
elecfan2011: 感谢楼主的精彩讲
解，受益匪浅啊！
LDD3源码分析之slab高速缓存
donghuwuwei: 省去了不少修改
的时间，真是太好了
LDD3源码分析之时间与延迟操作
donghuwuwei: jit.c代码需要加上
一个头文件。
LDD3源码分析之slab高速缓存
捧灰: 今天学到这里了，可是为什
么我没有修改源码一遍就通过了
额。。。内核版本是2.6.18-
53.el5-x...
LDD3源码分析之字符设备驱动程
捧灰: 参照楼主的博客在自学~谢
谢楼主！
LDD3源码分析之调试技术
fantasyhujian: 分析的很清楚，
赞一个！
LDD3源码分析之字符设备驱动程
fantasyhujian: 有时间再好好读
读，真的分析的不错！
LDD3源码分析之hello.c与Makef
fantasyhujian: 写的很详细，对
初学者很有帮助！！

阅读排行

- LDD3源码分析之字符设: (3143)
LDD3源码分析之hello.c: (2701)
S3C2410驱动分析之LCI (2527)
Linux设备模型分析之kse (2435)
LDD3源码分析之内存映! (2336)
LDD3源码分析之与硬件! (2333)
Android架构分析之Andrc (2093)
LDD3源码分析之时间与; (1987)
LDD3源码分析之poll分析 (1972)
S3C2410驱动分析之AD (1948)

评论排行

- LDD3源码分析之字符设: (12)
S3C2410驱动分析之触摸 (7)
LDD3源码分析之内存映! (5)
LDD3源码分析之hello.c: (4)
Linux设备模型分析之kot (4)
LDD3源码分析之slab高; (4)
S3C2410驱动分析之LCI (3)
LDD3源码分析之阻塞型I (3)
LDD3源码分析之时间与; (3)
LDD3源码分析之poll分析 (2)

文章存档

- 2014年06月 (1)
2014年05月 (4)
2014年04月 (1)

次设备号为4，是因为前面已经注册了scull0 - scull3，它们的主设备号均为系统分配值，次设备号分别是0，1，2，3。虽然我们还没有看后面的代码，但我们由此可以推测出作者的意图，要创建scullpipe0 - scullpipe3，其主设备号均为系统分配值，次设备号分别为4，5，6，7。

658行是我们分析的重点。而659行以后才会涉及，在本文中不分析659行。注意，658行，把657行生成的设备编号dev做为参数传递给scull_p_init函数，现在看这个函数的实现，在examples/scull/pipe.c文件中：

[cpp]

01. 343/*
02. 344 * Initialize the pipe devs; return how many we did.
03. 345 */
04. 346int scull_p_init(dev_t firstdev)
05. 347{
06. 348 int i, result;
07. 349
08. 350 result = register_chrdev_region(firstdev, scull_p_nr_devs, "scullp");
09. 351 if (result < 0) {
10. 352 printk(KERN_NOTICE "Unable to get scullp region, error %d\n", result);
11. 353 return 0;
12. 354 }
13. 355 scull_p_devno = firstdev;
14. 356 scull_p_devices = kmalloc(scull_p_nr_devs * sizeof(struct scull_pipe), GFP_KERNEL);
15. 357 if (scull_p_devices == NULL) {
16. 358 unregister_chrdev_region(firstdev, scull_p_nr_devs);
17. 359 return 0;
18. 360 }
19. 361 memset(scull_p_devices, 0, scull_p_nr_devs * sizeof(struct scull_pipe));
20. 362 for (i = 0; i < scull_p_nr_devs; i++) {
21. 363 init_waitqueue_head(&(scull_p_devices[i].inq));
22. 364 init_waitqueue_head(&(scull_p_devices[i].outq));
23. 365 init_MUTEX(&scull_p_devices[i].sem);
24. 366 scull_p_setup_cdev(scull_p_devices + i, i);
25. 367 }
26. 368#ifdef SCULL_DEBUG
27. 369 create_proc_read_entry("scullpipe", 0, NULL, scull_read_p_mem, NULL);
28. 370#endif
29. 371 return scull_p_nr_devs;
30. 372}

350行，调用register_chrdev_region函数申请从firstdev开始的scull_p_nr_devs个设备编号，firstdev是scull_p_init函数的形参，所以firstdev是前面main.c中658行传的设备编号。而scull_p_nr_devs的默认值为4。

355行，scull_p_devno = firstdev; scull_p_devno用于记录第一个scullpipe设备的设备编号。

356行，为scullpipe0 - scullpipe3的设备结构体分配内存空间。scullpipe设备由scull_pipe结构体表示，其定义如下：

[html]

01. 33struct scull_pipe {
02. 34 wait_queue_head_t inq, outq; /* read and write queues */
03. 35 char *buffer, *end; /* begin of buf, end of buf */
04. 36 int buffersize; /* used in pointer arithmetic */
05. 37 char *rp, *wp; /* where to read, where to write */
06. 38 int nreaders, nwriters; /* number of openings for r/w */
07. 39 struct fasync_struct *asynch_queue; /* asynchronous readers */
08. 40 struct semaphore sem; /* mutual exclusion semaphore */
09. 41 struct cdev cdev; /* Char device structure */
10. 42};

361行，将scull_pipe结构体数组的内容清0。

362 - 367行，初始化scullpipe设备对应的scull_pipe结构体。每次for循环初始化一个scullpipe设备。

363 - 364行，初始化两个等待队列头scull_p_devices[i].inq和scull_p_devices[i].outq，这两个等待队列头分别代表休眠在scullpipe设备上的等待读取和等待写入的进程。

365行初始化信号量scull_p_devices[i].sem。

366行，调用scull_p_setup_cdev函数完成进一步的初始化。下面看这个函数的实现：

[cpp]

2014年01月 (1)

2013年12月 (6)

展开

文章搜索

推荐文章

```
01. 326/*
02. 327 * Set up a cdev entry.
03. 328 */
04. 329static void scull_p_setup_cdev(struct scull_pipe *dev, int index)
05. 330{
06. 331     int err, devno = scull_p_devno + index;
07. 332
08. 333     cdev_init(&dev->cdev, &scull_pipe_fops);
09. 334     dev->cdev.owner = THIS_MODULE;
10. 335     err = cdev_add (&dev->cdev, devno, 1);
11. 336     /* Fail gracefully if need be */
12. 337     if (err)
13. 338         printk(KERN_NOTICE "Error %d adding scullpipe%d", err, index);
14. 339}
```

333行, 调用cdev_init函数初始化dev->cdev, 指定设备操作函数集是scull_pipe_fops。

335行, 调用cdev_add函数将dev->cdev注册到系统中, devno是对应的设备编号。

至此, scullpipe设备就初始化完毕并且注册到系统中。

再回到scull_p_init函数, 368 - 370行, 注册了scullpipe的/proc接口。因为本文主要分析阻塞型I/O, 所以不讨论这个接口。但需要注意的是, scullpipe的/proc接口实现与scull的/proc接口实现不一样, 主要是start参数相关的问题, 有兴趣可以仔细研究一下。

371行, 返回注册的scullpipe设备个数。

下面我们来看scullpipe的read函数, 即scull_p_read函数, 其代码如下所示:

```
[cpp]
01. 118static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t count,
02. 119                             loff_t *f_pos)
03. 120{
04. 121     struct scull_pipe *dev = filp->private_data;
05. 122
06. 123     if (down_interruptible(&dev->sem))
07. 124         return -ERESTARTSYS;
08. 125
09. 126     while (dev->rp == dev->wp) { /* nothing to read */
10. 127         up(&dev->sem); /* release the lock */
11. 128         if (filp->f_flags & O_NONBLOCK)
12. 129             return -EAGAIN;
13. 130         PDEBUG("\'%s\' reading: going to sleep\n", current->comm);
14. 131         if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
15. 132             return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
16. 133         /* otherwise loop, but first reacquire the lock */
17. 134         if (down_interruptible(&dev->sem))
18. 135             return -ERESTARTSYS;
19. 136     }
20. 137     /* ok, data is there, return something */
21. 138     if (dev->wp > dev->rp)
22. 139         count = min(count, (size_t)(dev->wp - dev->rp));
23. 140     else /* the write pointer has wrapped, return data up to dev->end */
24. 141         count = min(count, (size_t)(dev->end - dev->rp));
25. 142     if (copy_to_user(buf, dev->rp, count)) {
26. 143         up (&dev->sem);
27. 144         return -EFAULT;
28. 145     }
29. 146     dev->rp += count;
30. 147     if (dev->rp == dev->end)
31. 148         dev->rp = dev->buffer; /* wrapped */
32. 149     up (&dev->sem);
33. 150
34. 151     /* finally, awake any writers and return */
35. 152     wake_up_interruptible(&dev->outq);
36. 153     PDEBUG("\'%s\' did read %li bytes\n", current->comm, (long)count);
37. 154     return count;
38. 155}
```

126行, 当读指针dev->rp与写指针 dev->wp相等时, 说明缓冲区中没有数据可读。这种情况下, 要根据用户指定的标志位决定是进入休眠等待还是直接返回。

127行, 进入休眠之前, 要将已经获得的锁释放掉。

128 - 129行，如果用户空间的read函数指定了O_NONBLOCK标志，则不阻塞进程，直接退出。

131 - 132行，调用wait_event_interruptible函数休眠在dev->inq等待队列上，注意被唤醒的条件是(dev->rp != dev->wp)，即缓冲区中有数据可读。由这一句也可以看出，用户空间的read进程对应的等待队列是dev->inq。另外，wait_event_interruptible的返回值有两种，返回0表示缓冲区中有数据可读，被唤醒。返回非0值表示休眠被某个信号中断，这时，132行直接返回-ERESTARTSYS。

134 - 135行，休眠被唤醒后，首先重新获得互斥锁，然后返回到while循环开始处判断是否有数据可读。

138 - 141行，确认有数据可读后，退出while循环，根据读写指针的位置位及count值，决定要读多少数据。

142行，读数据到用户空间。

146 - 148行，更新读指针的位置。

149行，释放互斥锁。

152行，读取结束后，就有缓冲区空间可以进行写操作了。所以唤醒所有休眠在dev->outq上的写进程，唤醒函数是wake_up_interruptible。

154行，返回读取的字节数。

注意，scull_p_read在142行调用copy_to_user将数据拷贝到用户空间过程中，可能会休眠。此时，虽然进程拥有互斥锁，但这种情况下拥有互斥锁休眠是可以接受的，因为我们知道内核会把数据复制到用户空间并唤醒我们，同时不会试图锁上同一个信号量。

下面我们来看scullpipe的write函数，即scull_p_write函数，其代码如下所示：

```
[cpp]
01. 188static ssize_t scull_p_write(struct file *filp, const char __user *buf, size_t count,
02. 189                             loff_t *f_pos)
03. 190{
04. 191     struct scull_pipe *dev = filp->private_data;
05. 192     int result;
06. 193
07. 194     if (down_interruptible(&dev->sem))
08. 195         return -ERESTARTSYS;
09. 196
10. 197     /* Make sure there's space to write */
11. 198     result = scull_getwritespace(dev, filp);
12. 199     if (result)
13. 200         return result; /* scull_getwritespace called up(&dev->sem) */
14. 201
15. 202     /* ok, space is there, accept something */
16. 203     count = min(count, (size_t)spacefree(dev));
17. 204     if (dev->wp >= dev->rp)
18. 205         count = min(count, (size_t)(dev->end - dev->wp)); /* to end-of-buf */
19. 206     else /* the write pointer has wrapped, fill up to rp-1 */
20. 207         count = min(count, (size_t)(dev->rp - dev->wp - 1));
21. 208     PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp, buf);
22. 209     if (copy_from_user(dev->wp, buf, count)) {
23. 210         up(&dev->sem);
24. 211         return -EFAULT;
25. 212     }
26. 213     dev->wp += count;
27. 214     if (dev->wp == dev->end)
28. 215         dev->wp = dev->buffer; /* wrapped */
29. 216     up(&dev->sem);
30. 217
31. 218     /* finally, awake any reader */
32. 219     wake_up_interruptible(&dev->inq); /* blocked in read() and select() */
33. 220
34. 221     /* and signal asynchronous readers, explained late in chapter 5 */
35. 222     if (dev->async_queue)
36. 223         kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
37. 224     PDEBUG("%s" did write %li bytes\n", current->comm, (long)count);
38. 225     return count;
39. 226}
```

与scull_p_read采用简单休眠不同，scull_p_write采用了手工休眠的方式。

194行, 获得互斥锁。

198行, 调用scull_getwritespace函数。下面看这个函数的定义:

```
[cpp]
01. 157/* Wait for space for writing; caller must hold device semaphore. On
02. 158 * error the semaphore will be released before returning. */
03. 159static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
04. 160{
05. 161     while (spacefree(dev) == 0) { /* full */
06. 162         DEFINE_WAIT(wait);
07. 163
08. 164         up(&dev->sem);
09. 165         if (filp->f_flags & O_NONBLOCK)
10. 166             return -EAGAIN;
11. 167         PDEBUG("\'%s\'" writing: going to sleep\n",current->comm);
12. 168         prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
13. 169         if (spacefree(dev) == 0)
14. 170             schedule();
15. 171         finish_wait(&dev->outq, &wait);
16. 172         if (signal_pending(current))
17. 173             return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
18. 174         if (down_interruptible(&dev->sem))
19. 175             return -ERESTARTSYS;
20. 176     }
21. 177     return 0;
22. 178}
```

161行, 使用spacefree函数判断是否有空间可写, 该函数定义如下:

```
[cpp]
01. 180/* How much space is free? */
02. 181static int spacefree(struct scull_pipe *dev)
03. 182{
04. 183     if (dev->rp == dev->wp)
05. 184         return dev->buffersize - 1;
06. 185     return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
07. 186}
```

如果spacefree函数返回值为0, 说明现在scullpipe设备缓冲区已满, 没有空间可以写入。写进程需要根据调用者设置的标志位决定是否休眠等待。

162行, 使用DEFINE_WAIT函数创建并初始化一个等待队列入口。

164行, 在进入休眠前, 释放互斥锁。

165 - 166行, 如果调用者设置了O_NONBLOCK标志位, 则直接退出。

168行, 调用prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE)函数, 将等待队列入口wait插入到dev->outq等队列中, 并设置进程状态为TASK_INTERRUPTIBLE。

169 - 170行, 再次调用spacefree(dev)判断是否有空间可写, 如果还是确认没有空间, 则调用170行的schedule()进行进程调度。这里需要注意的是, 如果在169行的if判断和170行的调度之间, 有了可写入空间, 那么会发生什么情况? 这种情况下没有任何问题, 只要进程已经把自己放到了等待队列中并修改了进程状态, 就不会有任何问题, schedule会把进程状态重新设置为TASK_RUNNING, 并返回。

171行, 调用finish_wait执行清理操作。

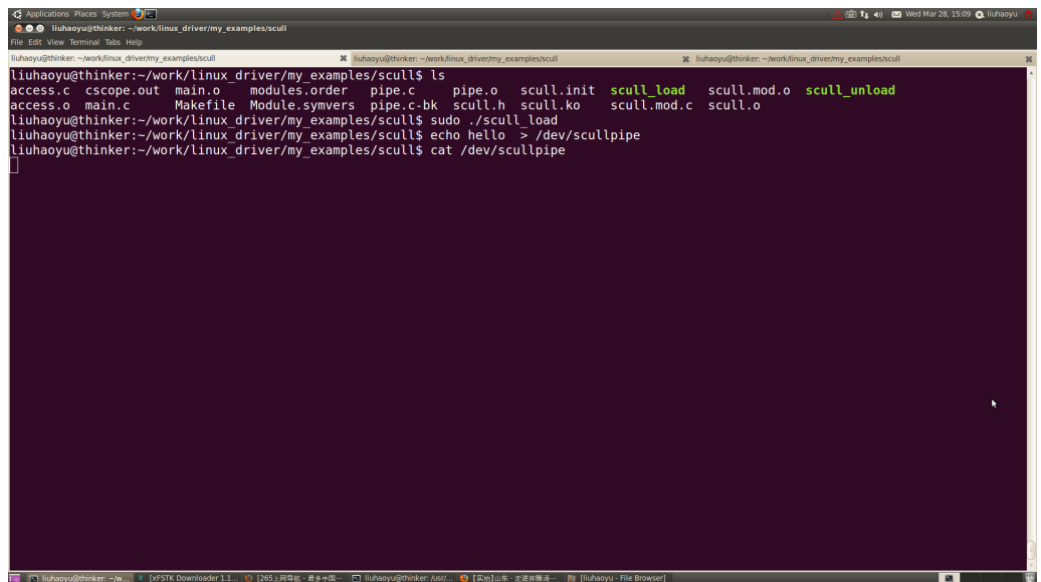
172 - 173行, 判断是否是因为收到信号而醒来的, 如果是返回-ERESTARTSYS。

174 - 176行, 重新申请互斥锁, 再次回到while开始处确认是否有空间写入。

177行, 如果有空间可写入, 返回0。

下面执行流程回到scull_p_write函数的198行, 如果有可写入的空间, scull_getwritespace返回值为0, 如果是因为收到信号而醒来的, scull_getwritespace返回非0值。

202 - 207行, 计算可写入数据的大小。



上图中，先向/dev/scullpipe写入字符串“hello”，然后用cat命令读/dev/scullpipe，正常情况下应该能读出“hello”才对，但是从上图我们可以看出，cat被阻塞住了。原因是什么呢？我们先看scull_p_open函数的实现：

```
[cpp]
01. 61static int scull_p_open(struct inode *inode, struct file *filp)
02. 62{
03. 63     struct scull_pipe *dev;
04. 64
05. 65     dev = container_of(inode->i_cdev, struct scull_pipe, cdev);
06. 66     filp->private_data = dev;
07. 67
08. 68     if (down_interruptible(&dev->sem))
09. 69         return -ERESTARTSYS;
10. 70     if (!dev->buffer) {
11. 71         /* allocate the buffer */
12. 72         dev->buffer = kmalloc(scull_p_buffer, GFP_KERNEL);
13. 73         if (!dev->buffer) {
14. 74             up(&dev->sem);
15. 75             return -ENOMEM;
16. 76         }
17. 77     }
18. 78     dev->buffersize = scull_p_buffer;
19. 79     dev->end = dev->buffer + dev->buffersize;
20. 80     dev->rp = dev->wp = dev->buffer; /* rd and wr from the beginning */
21. 81
22. 82     /* use f_mode, not f_flags: it's cleaner (fs/open.c tells why) */
23. 83     if (filp->f_mode & FMODE_READ)
24. 84         dev->nreaders++;
25. 85     if (filp->f_mode & FMODE_WRITE)
26. 86         dev->nwriters++;
27. 87     up(&dev->sem);
28. 88
29. 89     return nonseekable_open(inode, filp);
30. 90 }
```

70 - 80行，如果dev->buffer为NULL，则给dev->buffer分配内存空间，然后设置缓冲区大小和结束位置，以及读写指针位置。

关键在第80行，这行将读写指针位置dev->rp和dev->wp设置为同时指向缓冲区的开始处，而在scullpipe设备中，dev->rp等于dev->wp表明缓冲区内没有数据。

所以，问题的原因就找到了，我们向scullpipe中写入了字符串，但是执行cat /dev/scullpipe时，cat命令会调用open打开/dev/scullpipe文件，这时，就会将dev->rp和dev->wp设置为指向缓冲区的开始处，然后，cat读/dev/scullpipe的时候，就会认为该文件中没有数据可读，就会阻塞住。直到下次有数据写入时，cat才能被唤醒并读出数据。

另外，这个scull_p_open函数的逻辑有问题。在open时给dev->buffer分配内存，就是说，每次打开scullpipe设备，都会为它分配内存，这本来就不合理，尽管70行加了一个判断，只有当dev->buffer

为空时，才分配内存。合理的位置应该是在初始化函数中完成这些操作。所以，我修改了scull_p_open函数和scull_p_setup_cdev函数，把对缓冲区的初始化工作放在了scull_p_setup_cdev函数中，该函数由初始化函数scull_p_init调用。列出这两个函数如下：

[cpp]

```
01. static int scull_p_open(struct inode *inode, struct file *filp)
02. {
03.     struct scull_pipe *dev;
04.
05.     dev = container_of(inode->i_cdev, struct scull_pipe, cdev);
06.     filp->private_data = dev;
07.
08.     if (down_interruptible(&dev->sem))
09.         return -ERESTARTSYS;
10.
11.     /* use f_mode, not f_flags: it's cleaner (fs/open.c tells why) */
12.     if (filp->f_mode & FMODE_READ)
13.         dev->nreaders++;
14.     if (filp->f_mode & FMODE_WRITE)
15.         dev->nwriters++;
16.     up(&dev->sem);
17.
18.     return nonseekable_open(inode, filp);
19. }
```

[cpp]

```
01. /*
02.  * Set up a cdev entry.
03.  */
04. static void scull_p_setup_cdev(struct scull_pipe *dev, int index)
05. {
06.     int err, devno;
07.
08.     /* allocate the buffer */
09.     dev->buffer = kmalloc(scull_p_buffer, GFP_KERNEL);
10.     if (!dev->buffer)
11.         return -ENOMEM;
12.
13.     dev->buffersize = scull_p_buffer;
14.     dev->end = dev->buffer + dev->buffersize;
15.     dev->rp = dev->wp = dev->buffer; /* rd and wr from the beginning */
16.
17.     devno = scull_p_devno + index;
18.     cdev_init(&dev->cdev, &scull_pipe_fops);
19.     dev->cdev.owner = THIS_MODULE;
20.     err = cdev_add (&dev->cdev, devno, 1);
21.     /* Fail gracefully if need be */
22.     if (err)
23.         printk(KERN_NOTICE "Error %d adding scullpipe%d", err, index);
24. }
```

由于把分配缓冲区的工作从scull_p_open函数移到了模块初始化函数中，连锁反应，还有一个函数需要修改，那就是scull_p_release，修改后的代码如下：

[cpp]

```
01. static int scull_p_release(struct inode *inode, struct file *filp)
02. {
03.     struct scull_pipe *dev = filp->private_data;
04.
05.     /* remove this filp from the asynchronously notified filp's */
06.     scull_p_fasync(-1, filp, 0);
07.     down(&dev->sem);
08.     if (filp->f_mode & FMODE_READ)
09.         dev->nreaders--;
10.     if (filp->f_mode & FMODE_WRITE)
11.         dev->nwriters--;
12.     // if (dev->nreaders + dev->nwriters == 0) {
13.     //     kfree(dev->buffer);
14.     //     dev->buffer = NULL; /* the other fields are not checked on open */
15.     // }
```



```
16.         up(&dev->sem);
17.         return 0;
18.     }
```

因为原来在scull_p_open中分配缓冲区，所以在scull_p_release中释放缓冲区，现在scull_p_open中不再分配缓冲区了，所以scull_p_release也就不再释放缓冲区了。所以必须屏蔽掉scull_p_release函数中释放缓冲区的4条语句。

用上面的列出的scull_p_open、scull_p_setup_cdev、scull_p_release函数替换原来的函数，上面的问题就解决了，执行结果如下图所示：

更多 0

上一篇 LDD3源码分析之简单休眠
下一篇 LDD3源码分析之poll分析

顶 0 踩 0

主题推荐 源码 asynchronous something structure 内存

猜你在找


- | | |
|-------------------------------------|---|
| LDD3源码分析之slab高速缓存 | Touch Driver介绍 |
| Android 用Vibrator实现震动功能 | Ubuntu下JNI的实现与调用 |
| select() 和poll() 的区别是什么？ | intel dpdk api rte_eal_hugepage_init() 函数介绍 |
| linux input输入子系统分析《二》：s3c2440的ADC简单 | php请求超时过高导致系统负载高的优化方法？ |
| STM32 对内部FLASH读写接口函数 | linux下ALSA音频驱动软件开发 |

免费学习IT4个月,月薪12000

中国[官方授权]IT培训与就业示范基地,学成后名企直接招聘,月薪12000起!

查看评论

2楼 njxxdx 2013-04-03 20:42发表



问个问题 scull_getwritespace 里有个疑问
这个函数在没有使用信号量保存互斥的情况下 if (spacefree(dev) == 0)
访问了dev,会不会存在一种可能,另外一个线程执行scull_p_read 改变了值,而spacefree读出的是旧值

Re: haohuanfei 2013-05-19 17:34发表

回复njxxdx: line 194: if (down_interruptible(&dev->sem))



1楼 [ashergf](#) 2012-06-16 12:48发表



很好，谢谢分享！

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- 全部主题
- Java
- VPN
- Android
- iOS
- ERP
- IE10
- Eclipse
- CRM
- JavaScript
- Ubuntu
- NFC
- WAP
- jQuery
- 数据库
- BI
- HTML5
- Spring
- Apache
- Hadoop
- .NET
- API
- HTML
- SDK
- IIS
- Fedora
- XML
- LBS
- Unity
- Splashtop
- UML
- components
- Windows Mobile
- Rails
- QEMU
- KDE
- Cassandra
- CloudStack
- FTC
- coremail
- OPhone
- CouchBase
- 云计算
- iOS6
- Rackspace
- Web App
- SpringSide
- Maemo
- Compuware
- 大数据
- aptech
- Perl
- Tornado
- Ruby
- Hibernate
- ThinkPHP
- Spark
- HBase
- Pure
- Solr
- Angular
- Cloud Foundry
- Redis
- Scala
- Django
- Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2014, CSDN.NET, All Rights Reserved 