

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: 80620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

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源码分析之访问控制

分类: LDD3源码分析

2012-03-29 16:29

1153人阅读

评论(0)

收藏

举报

struct

access

list

file

module

测试

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/ access.c examples/scull/main.c

一、访问控制设备的注册

本文分析LDD3第6章介绍的设备文件访问控制的实现，涉及的代码主要在access.c文件中，但是作为分析的起点，我们还是要看一下main.c文件中的scull_init_module函数，在该函数中，有如下语句：

[cpp]

01. 657 dev = MKDEV(scull_major, scull_minor + scull_nr_devs);

02. 658 dev += scull_p_init(dev);

03. 659 dev += scull_access_init(dev);

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动态分配值，次设备号分别是0，1，2，3。

658行，调用了pipe.c文件中的scull_p_init函数，我们在前面的《LDD3源码分析之阻塞型I/O》一文中对这个函数进行了分析，它创建了scullpipe0 - scullpipe3四个设备，对应的主设备号是系统动态分配值，次设备号为4，5，6，7。而scull_p_init函数的返回值是4，所以658行把dev的值再加4，然后传递给659行的scull_access_init函数。

659行，调用scull_access_init函数，这个函数就是我们今天要分析的起点，在access.c文件中定义，下面看这个函数的代码：

[cpp]

01. 366int scull_access_init(dev_t firstdev)

02. 367{

03. 368 int result, i;

04. 369

05. 370 /* Get our number space */

06. 371 result = register_chrdev_region (firstdev, SCULL_N_ADEVS, "sculla");

07. 372 if (result < 0) {

08. 373 printk(KERN_WARNING "sculla: device number registration failed\n");

09. 374 return 0;

10. 375 }

<http://blog.csdn.net/liuhaoyutz/article/details/7407700>

1/12

最新评论

- LDD3源码分析之内存映射
wzw88486969:
@jhlhlong: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ADI (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)

```
11. 376     scull_a_firstdev = firstdev;
12. 377
13. 378     /* Set up each device. */
14. 379     for (i = 0; i < SCULL_N_ADEVS; i++)
15. 380         scull_access_setup (firstdev + i, scull_access_devs + i);
16. 381     return SCULL_N_ADEVS;
17. 382}
```

371行注册访问控制相关设备的设备号，起始设备号是由参数传递进来的，注册的设备编号的个数是SCULL_N_ADEVS，它的值是4。

376行，保存第一个设备的设备编号。

379 - 380行，调用scull_access_setup函数，循环初始化4个访问控制相关设备。注意传递给scull_access_setup函数的第二个参数是scull_access_devs + i，先看一下scull_access_devs的定义：

```
[cpp]
01. 327static struct scull_adev_info {
02. 328     char *name;
03. 329     struct scull_dev *scullddev;
04. 330     struct file_operations *fops;
05. 331} scull_access_devs[] = {
06. 332     { "scullsingle", &scull_s_device, &scull_sngl_fops },
07. 333     { "sculluid", &scull_u_device, &scull_user_fops },
08. 334     { "scullwuid", &scull_w_device, &scull_wusr_fops },
09. 335     { "sullpriv", &scull_c_device, &scull_priv_fops }
10. 336};
```

可见，scull_access_devs是一个scull_adev_info结构体数组，该结构体代表一个访问控制设备，scull_adev_info有3个成员，第一个代表设备名，第二个是第3章中介绍的scull设备，第三个是对于这个访问控制设备的操作函数集。

scull_access_devs数组定义了4个访问控制设备，这4个设备使用不同的访问控制策略。第一个设备叫scullsingle，对应的“bare scull device”是scull_s_device,定义在49行，对应的操作函数集是scull_sngl_fops，定义在78行：

```
[cpp]
01. 49static struct scull_dev scull_s_device;
```

```
[cpp]
01. 78struct file_operations scull_sngl_fops = {
02. 79     .owner =      THIS_MODULE,
03. 80     .llseek =     scull_llseek,
04. 81     .read =       scull_read,
05. 82     .write =      scull_write,
06. 83     .ioctl =      scull_ioctl,
07. 84     .open =       scull_s_open,
08. 85     .release =    scull_s_release,
09. 86};
```

其它三个设备分别是sculluid、scullwuid、sullpriv，它们对应的“bare scull device”和操作函数集也都是 在access.c中定义，这里不一一列出了，后面分析相应设备时再详细介绍。由上面的内容可以看出，访问控制设备的实现是建立在“bare scull device”的基础上的，很多代码都是与“bare scull device”复用的。

下面看scull_access_setup函数的定义：

```
[cpp]
01. 339/*
02. 340 * Set up a single device.
03. 341 */
04. 342static void scull_access_setup (dev_t devno, struct scull_adev_info *devinfo)
05. 343{
06. 344     struct scull_dev *dev = devinfo->scullddev;
07. 345     int err;
08. 346
09. 347     /* Initialize the device structure */
10. 348     dev->quantum = scull_quantum;
```

[2014年01月 \(1\)](#)[2013年12月 \(6\)](#)[展开](#)

文章搜索

推荐文章

```
11. 349 dev->qset = scull_qset;
12. 350 init_MUTEX(&dev->sem);
13. 351
14. 352 /* Do the cdev stuff. */
15. 353 cdev_init(&dev->cdev, devinfo->fops);
16. 354 kobject_set_name(&dev->cdev.kobj, devinfo->name);
17. 355 dev->cdev.owner = THIS_MODULE;
18. 356 err = cdev_add (&dev->cdev, devno, 1);
19. 357 /* Fail gracefully if need be */
20. 358 if (err) {
21. 359     printk(KERN_NOTICE "Error %d adding %s\n", err, devinfo->name);
22. 360     kobject_put(&dev->cdev.kobj);
23. 361 } else
24. 362     printk(KERN_NOTICE "%s registered at %x\n", devinfo->name, devno);
25. 363}
```

348 - 353行，和第三章中初始化scull设备一样，分别初始化了量子数，量子集数，信号量和cdev成员。353行还将字符设备关联了相应的文件操作函数集。

354行，注册了sys系统中的名字。

356行，将字符设备注册到系统中，完成注册。

这样，就完成了对字符设备的初始化和注册，现在我们有4个采用不同访问控制策略的设备，分别是scullsingle、sculluid、scullwuid和scullpriv。

为了对这个4设备的访问控制策略进行测试，我编写了一个简单的测试程序access_control.c，其代码如下：

```
[cpp]
01. #include <stdio.h>
02. #include <unistd.h>
03. #include <fcntl.h>
04. #include <string.h>
05. #include <sys/types.h>
06. #include <sys/stat.h>
07.
08. #define BUF_SIZE 50
09.
10. int main(int argc, char *argv[])
11. {
12.     int fd;
13.     int num, n;
14.     char buf[BUF_SIZE];
15.
16.     fd = open(argv[1], O_RDWR);
17.     if(fd < 0)
18.     {
19.         printf("open scull error!\n");
20.         return -1;
21.     }
22.
23.     n = 0;
24.     while(n < 10)
25.     {
26.         lseek(fd, 0, SEEK_SET);
27.         memset(buf, 0, BUF_SIZE);
28.         num = read(fd, buf, BUF_SIZE);
29.         if( num > 0)
30.         {
31.             buf[num] = 0;
32.             printf("%s\n", buf);
33.         }
34.         sleep(2);
35.         n++;
36.     }
37.
38.     return 0;
39. }
```

后面将使用这个测试程序对不同的设备进行测试。

二、独享设备

这种访问控制一次只允许一个进程访问设备，最好避免使用这种技术，因为它限制了用户的灵活性。scullsingle设备实现了独享设备的策略，其主要代码如下：

```
[cpp]
01. 49static struct scull_dev scull_s_device;
02. 50static atomic_t scull_s_available = ATOMIC_INIT(1);
03. 51
04. 52static int scull_s_open(struct inode *inode, struct file *filp)
05. 53{
06. 54     struct scull_dev *dev = &scull_s_device; /* device information */
07. 55
08. 56     if (!atomic_dec_and_test(&scull_s_available)) {
09. 57         atomic_inc(&scull_s_available);
10. 58         return -EBUSY; /* already open */
11. 59     }
12. 60
13. 61     /* then, everything else is copied from the bare scull device */
14. 62     if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
15. 63         scull_trim(dev);
16. 64     filp->private_data = dev;
17. 65     return 0; /* success */
18. 66}
19. 67
20. 68static int scull_s_release(struct inode *inode, struct file *filp)
21. 69{
22. 70     atomic_inc(&scull_s_available); /* release the device */
23. 71     return 0;
24. 72}
25. 73
26. 74
27. 75/*
28. 76 * The other operations for the single-open device come from the bare device
29. 77 */
30. 78struct file_operations scull_sngl_fops = {
31. 79     .owner = THIS_MODULE,
32. 80     .llseek = scull_llseek,
33. 81     .read = scull_read,
34. 82     .write = scull_write,
35. 83     .ioctl = scull_ioctl,
36. 84     .open = scull_s_open,
37. 85     .release = scull_s_release,
38. 86};
```

49行定义了scullsingle设备对应的“bare scull device” scull_s_device。

50行定义了一个原子变量(atomic_t)scull_s_available，其初始值为1，表明设备可用。如果其值为0，表明设备不可用。

56 - 59行，对原子变量scull_s_available执行atomic_dec_and_test操作，该函数将原子变量减1并测试其值是否为0，如果为0，返回TRUE，说明没有进程在使用设备，可以独享使用了。如果测试返回FALSE，说明有进程正在使用设备，将原子变量加1后，返回-EBUSY退出。

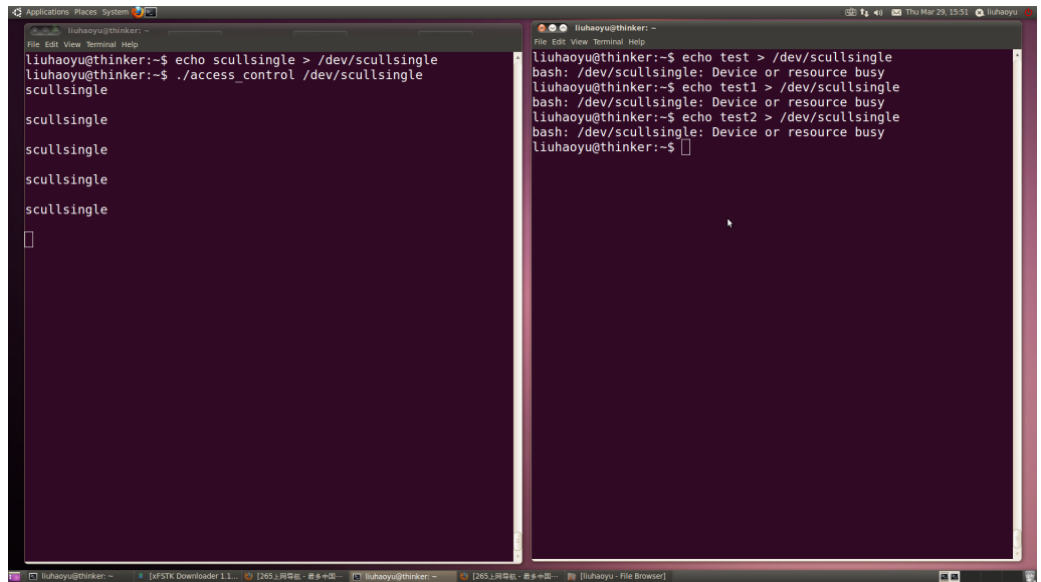
68 - 72行，定义了scull_s_release函数，该函数在进程关闭设备文件时调用，其作用是将原子变量scull_s_available的值加1，表示释放设备。

78 - 86行，定义了scullsingle设备的操作函数集，可以看到，除了open和release函数外，其他函数都是复用的scull设备的操作函数。

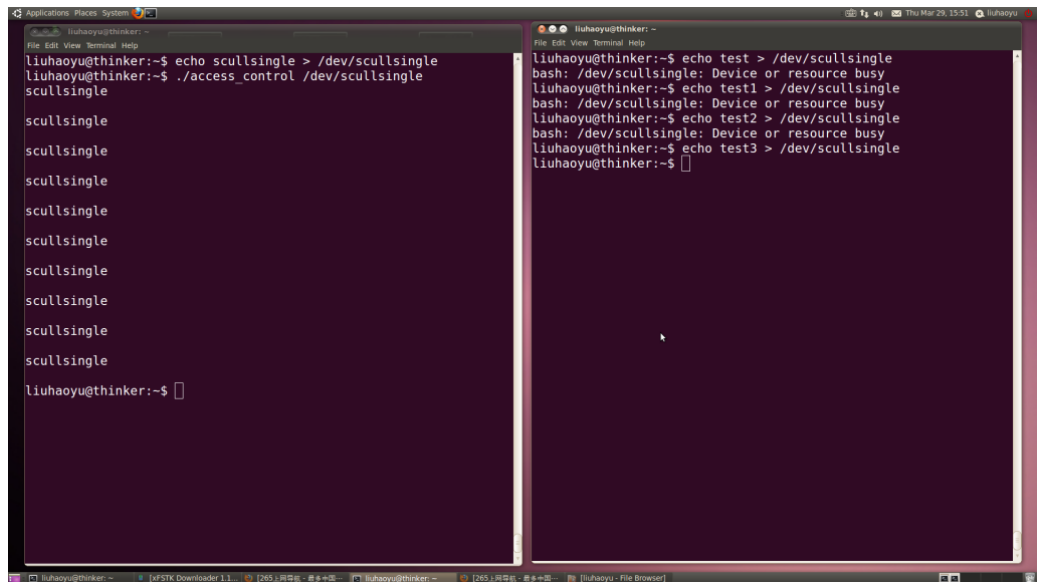
这样，通过加入一个原子变量，并在open函数中对其值进行判断，就能达到独享设备的目的了。

注意：通常应该把scull_s_available变量放在设备结构中(这里是scull_dev结构)，因为从概念上讲它本身属于设备。但是scullsingle设备的实现是把scull_s_available定义为一个全局变量，这样做是为了与scull复用代码。

使用测试程序access_control.c测试scullsingle设备的过程如下图所示：



因为access_control进程会占用scullsingle设备20秒，从上图可以看出，在access_control执行的这20秒内，另外一个进程即echo试图操作(这里是写)scullsingle设备会返回设备正忙的错误信息，这说明，scullsingle设备同时只能被一个进程访问。如下图所示，当access_control退出后，echo进程就可以操作scullsingle设备了：



三、限制每次只由一个用户访问

这种访问策略允许一个用户的多个进程同时访问设备，但是不允许多个用户同时访问设备。与独享设备的策略相比，这种方法更加灵活。此时需要增加两个数据项，一个打开计数器和一个设备属主UID。同样，这两个数据项最好保存在设备结构体内部，但是为了与scull复用代码，在实现时我们把这两个变量定义为全局变量。

使用这种策略实现的设备叫sculluid，其主要代码如下：

```
[cpp]
01. 95static struct scull_dev scull_u_device;
02. 96static int scull_u_count; /* initialized to 0 by default */
03. 97static uid_t scull_u_owner; /* initialized to 0 by default */
04. 98static spinlock_t scull_u_lock = SPIN_LOCK_UNLOCKED;
05. 99
06. 100static int scull_u_open(struct inode *inode, struct file *filp)
07. 101{
08. 102     struct scull_dev *dev = &scull_u_device; /* device information */
09. 103
10. 104     spin_lock(&scull_u_lock);
11. 105     if (scull_u_count &&
```

```

12. 106         (scull_u_owner != current->uid) && /* allow user */
13. 107         (scull_u_owner != current->euid) && /* allow whoever did su */
14. 108         !capable(CAP_DAC_OVERRIDE)) { /* still allow root */
15. 109         spin_unlock(&scull_u_lock);
16. 110         return -EBUSY; /* -EPERM would confuse the user */
17. 111     }
18. 112
19. 113     if (scull_u_count == 0)
20. 114         scull_u_owner = current->uid; /* grab it */
21. 115
22. 116     scull_u_count++;
23. 117     spin_unlock(&scull_u_lock);
24. 118
25. 119 /* then, everything else is copied from the bare scull device */
26. 120
27. 121     if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
28. 122         scull_trim(dev);
29. 123     filp->private_data = dev;
30. 124     return 0; /* success */
31. 125}
32. 126
33. 127static int scull_u_release(struct inode *inode, struct file *filp)
34. 128{
35. 129     spin_lock(&scull_u_lock);
36. 130     scull_u_count--; /* nothing else */
37. 131     spin_unlock(&scull_u_lock);
38. 132     return 0;
39. 133}
40. 134
41. 135
42. 136
43. 137/*
44. 138 * The other operations for the device come from the bare device
45. 139 */
46. 140struct file_operations scull_user_fops = {
47. 141     .owner = THIS_MODULE,
48. 142     .llseek = scull_llseek,
49. 143     .read = scull_read,
50. 144     .write = scull_write,
51. 145     .ioctl = scull_ioctl,
52. 146     .open = scull_u_open,
53. 147     .release = scull_u_release,
54. 148};

```

95行定义了设备结构体scull_u_device。

96行定义了访问计数器变量scull_u_count，该变量用来保存正在访问设备的进程数。

97行定义了uid_t变量scull_u_owner，用来保存正在访问设备的用户UID。

105 - 111行，如果不是当前进程不是第一个访问设备的进程，并且当前进程的uid或euid不等于scull_u_owner变量的值，并且不是root权限用户，则返回-EBUSY退出。表明有另外一个用户正在访问设备。

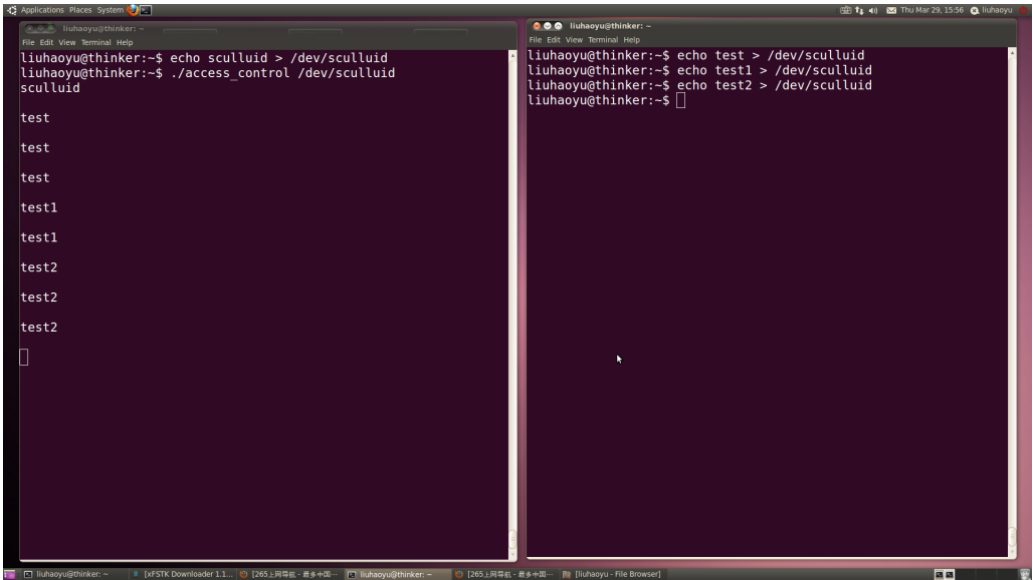
113 - 114行，如果是第一个访问设备的进程，则将进程的UID保存在scull_u_owner。

116行，将访问计数器值加1。

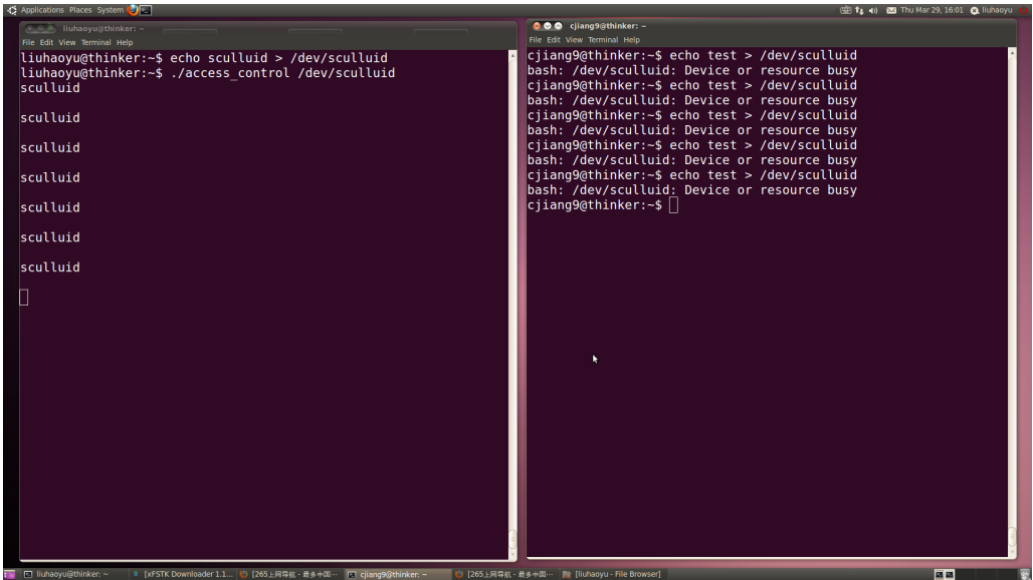
127 - 133行，scull_u_release函数在关闭设备文件时调用，其作用是将访问计数器值减1。

140 - 148行，定义了sculluid的设备操作函数集，可以看到，除了open和release函数，其它函数都是与scull复用的。

使用acell_control测试sculluid设备的过程如下图所示：



由上图操作过程可以看出，同一用户在两个终端下可以同时sculluid设备进行操作。不同用户对sculluid同时进行操作是否可以呢？如下图所示：



由上图可以看出，不同用户，不能同时对sculluid进行操作。

另外，一个用普通用户，一个用root用户，能不能同时操作sculluid呢？大家可以自己试验一下。

四、阻塞型open

上面两种访问控制方法当设备不能访问时，都是返回-EBUSY退出，但是有些情况下，可能需要让进程阻塞等待，这时就需要实现阻塞型open。

scullwuid设备实现了阻塞型open，其主要代码如下：

```
[cpp]
01. 156static struct scull_dev scull_w_device;
02. 157static int scull_w_count; /* initialized to 0 by default */
03. 158static uid_t scull_w_owner; /* initialized to 0 by default */
04. 159static DECLARE_WAIT_QUEUE_HEAD(scull_w_wait);
05. 160static spinlock_t scull_w_lock = SPIN_LOCK_UNLOCKED;
06. 161
07. 162static inline int scull_w_available(void)
08. 163{
09. 164     return scull_w_count == 0 ||
10. 165         scull_w_owner == current->uid ||
11. 166         scull_w_owner == current->euid ||
12. 167         capable(CAP_DAC_OVERRIDE);
13. 168}
```



```
14. 169
15. 170
16. 171static int scull_w_open(struct inode *inode, struct file *filp)
17. 172{
18. 173     struct scull_dev *dev = &scull_w_device; /* device information */
19. 174
20. 175     spin_lock(&scull_w_lock);
21. 176     while (! scull_w_available()) {
22. 177         spin_unlock(&scull_w_lock);
23. 178         if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
24. 179         if (wait_event_interruptible (scull_w_wait, scull_w_available()))
25. 180             return -ERESTARTSYS; /* tell the fs layer to handle it */
26. 181         spin_lock(&scull_w_lock);
27. 182     }
28. 183     if (scull_w_count == 0)
29. 184         scull_w_owner = current->uid; /* grab it */
30. 185     scull_w_count++;
31. 186     spin_unlock(&scull_w_lock);
32. 187
33. 188     /* then, everything else is copied from the bare scull device */
34. 189     if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
35. 190         scull_trim(dev);
36. 191     filp->private_data = dev;
37. 192     return 0; /* success */
38. 193}
39. 194
40. 195static int scull_w_release(struct inode *inode, struct file *filp)
41. 196{
42. 197     int temp;
43. 198
44. 199     spin_lock(&scull_w_lock);
45. 200     scull_w_count--;
46. 201     temp = scull_w_count;
47. 202     spin_unlock(&scull_w_lock);
48. 203
49. 204     if (temp == 0)
50. 205         wake_up_interruptible_sync(&scull_w_wait); /* awake other uid's */
51. 206     return 0;
52. 207}
53. 208
54. 209
55. 210/*
56. 211 * The other operations for the device come from the bare device
57. 212 */
58. 213struct file_operations scull_wusr_fops = {
59. 214     .owner =      THIS_MODULE,
60. 215     .llseek =     scull_llseek,
61. 216     .read =       scull_read,
62. 217     .write =      scull_write,
63. 218     .ioctl =      scull_ioctl,
64. 219     .open =       scull_w_open,
65. 220     .release =    scull_w_release,
66. 221};
```

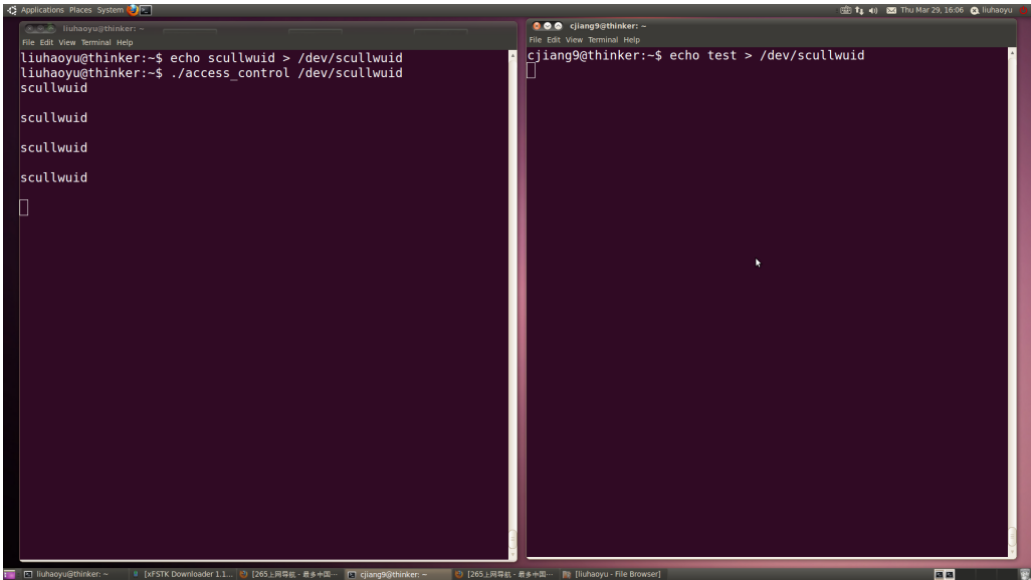
159行定义了一个等待队列scull_w_wait。

176 - 182行，判断能否访问设备的方法与sculluid相同，但是，如果不能访问设备，阻塞在scull_w_wait上等待而不是返回-EBUSY退出。

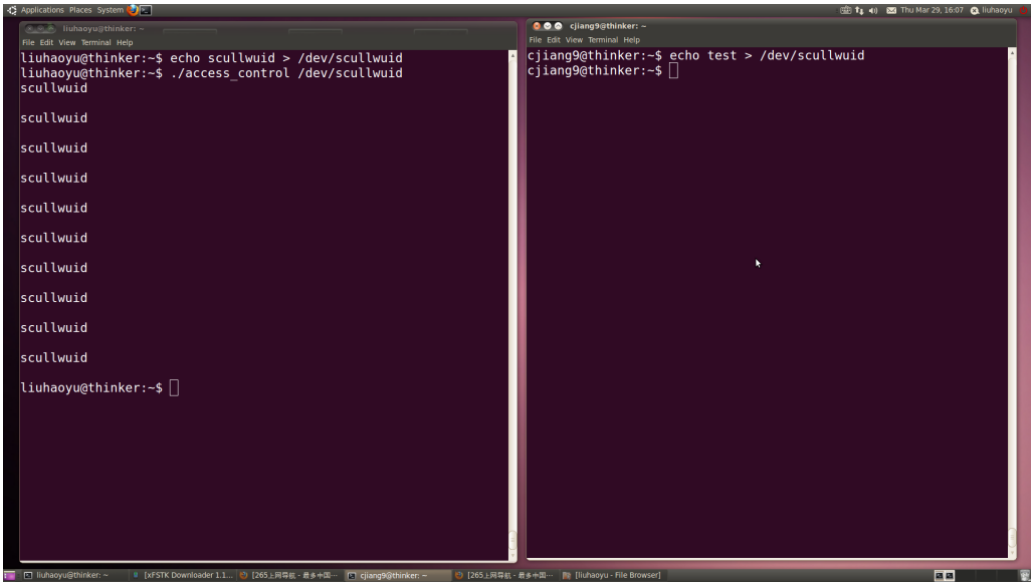
195 - 207行，scull_w_release函数在关闭设备文件时调用，它将使用计数器值减1，如果使用计数为0，则唤醒在等待队列scull_w_wait中阻塞等待的进程。

213 - 221行，定义scullwuid的文件操作函数集，除了open和release函数外，其它的函数都是与scull复用代码。

使用access_control测试scullwuid设备的过程如下图所示。注意，测试必须使用两个不同的普通用户进行。



由上图可以看出，当一个用户在操作scullwuid时，另一个用户的如果要打开scullwuid设备会被阻塞住。而当前一个用户操作完成了，被阻塞用户会解除阻塞，继续执行，如下图所示：



五、打开时clone设备

另一个实现访问控制的方法是，在进程打开设备时clone一个设备给进程使用。使用这种控制策略实现的设备是scullpriv，它使用当前进程控制终端的设备号作为访问虚拟设备的键值，也可以使用任意整数做键值，但是不同的键值将导致不同的访问策略。例如，使用uid作为键值，则会给每个用户clone一个设备。使用pid作为键值，则会给每个进程clone一个设备。所以，对于scullpriv，不同终端上的进程会有不同的clone设备。

下面是scullpriv的主要实现代码：

```
[cpp]
01. 229/* The clone-specific data structure includes a key field */
02. 230
03. 231struct scull_listitem {
04. 232     struct scull_dev device;
05. 233     dev_t key;
06. 234     struct list_head list;
07. 235
08. 236};
09. 237
10. 238/* The list of devices, and a lock to protect it */
11. 239static LIST_HEAD(scull_c_list);
12. 240static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;
13. 241
```

```

14. 242/* A placeholder scull_dev which really just holds the cdev stuff. */
15. 243static struct scull_dev scull_c_device;
16. 244
17. 245/* Look for a device or create one if missing */
18. 246static struct scull_dev *scull_c_lookfor_device(dev_t key)
19. 247{
20. 248     struct scull_listitem *lptr;
21. 249
22. 250     list_for_each_entry(lptr, &scull_c_list, list) {
23. 251         if (lptr->key == key)
24. 252             return &(lptr->device);
25. 253     }
26. 254
27. 255     /* not found */
28. 256     lptr = kmalloc(sizeof(struct scull_listitem), GFP_KERNEL);
29. 257     if (!lptr)
30. 258         return NULL;
31. 259
32. 260     /* initialize the device */
33. 261     memset(lptr, 0, sizeof(struct scull_listitem));
34. 262     lptr->key = key;
35. 263     scull_trim(&(lptr->device)); /* initialize it */
36. 264     init_MUTEX(&(lptr->device.sem));
37. 265
38. 266     /* place it in the list */
39. 267     list_add(&lptr->list, &scull_c_list);
40. 268
41. 269     return &(lptr->device);
42. 270}
43. 271
44. 272static int scull_c_open(struct inode *inode, struct file *filp)
45. 273{
46. 274     struct scull_dev *dev;
47. 275     dev_t key;
48. 276
49. 277     if (!current->signal->tty) {
50. 278         PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
51. 279         return -EINVAL;
52. 280     }
53. 281     key = tty_devnum(current->signal->tty);
54. 282
55. 283     /* look for a scullc device in the list */
56. 284     spin_lock(&scull_c_lock);
57. 285     dev = scull_c_lookfor_device(key);
58. 286     spin_unlock(&scull_c_lock);
59. 287
60. 288     if (!dev)
61. 289         return -ENOMEM;
62. 290
63. 291     /* then, everything else is copied from the bare scull device */
64. 292     if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
65. 293         scull_trim(dev);
66. 294     filp->private_data = dev;
67. 295     return 0; /* success */
68. 296}
69. 297
70. 298static int scull_c_release(struct inode *inode, struct file *filp)
71. 299{
72. 300     /*
73. 301      * Nothing to do, because the device is persistent.
74. 302      * A `real' cloned device should be freed on last close
75. 303      */
76. 304     return 0;
77. 305}
78. 306
79. 307
80. 308
81. 309/*
82. 310 * The other operations for the device come from the bare device
83. 311 */
84. 312struct file_operations scull_priv_fops = {
85. 313     .owner = THIS_MODULE,
86. 314     .llseek = scull_llseek,
87. 315     .read = scull_read,
88. 316     .write = scull_write,
89. 317     .ioctl = scull_ioctl,
90. 318     .open = scull_c_open,
91. 319     .release = scull_c_release,
92. 320};

```

我们从272scull_c_open函数开始分析。

277行，current->signal->tty代表当前进程的控制终端，如果当前进程没有控制终端，则退出。

281行，通过tty_devnum函数，取得当前进程控制终端的设备号，赋值给key。

285行，调用scull_c_lookfor_device(key)查找设备，如果没有，在scull_c_lookfor_device函数中会创建一个。注意，传递给scull_c_lookfor_device的参数是key。

下面看scull_c_lookfor_device函数的实现：

250 - 253行，遍历链表scull_c_list，如果有链表项的key值等于参数传递进来的key值，则说明已经为该控制终端clone过设备，则直接返回对应的设备结构。

256行，如果在scull_c_list链表中没有查找到对应key的节点，说明是第一次在该控制终端上打开设备，则为链表节点scull_listitem分配内存空间。

261 - 264行，初始化链表节点结构体。

267行，将链表节点加入到scull_c_list链表中。

269行，返回找到或新创建的scull_dev结构体。

298 - 305行，release函数不做任何事，因为scullpriv是永久存在的，如果是一个真正的clone设备，应该在最后一次关闭后释放空间。

312 - 320行，定义了设备文件操作函数集，除了open和release函数外，其它函数都是利用的scull的代码。

使用access_control测试scullpriv设备的过程如下图所示：

由上图可以看出，两个终端下的进程可以同时操作scullpriv，并且相互没有影响。因为两个终端操作的是两个不同的scullpriv的clone版本。

[更多](#) 0

[上一篇](#) LDD3源码分析之llseek分析

[下一篇](#) LDD3源码分析之时间与延迟操作

顶
1

踩
0

主题推荐

源码

全局变量

structure

quantum

sizeof

猜你在找

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

暂无评论

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

