

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: 80603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源码分析之调试技术

分类: LDD3源码分析

2012-03-22 16:06

1592人阅读

评论(1)

收藏

举报

struct

file

makefile

function

null

debugging

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/main.c

本文分析LDD3第四章相关代码，主要是使用/proc文件系统(包括seq\_file接口)进行调试操作，即在/proc目录下生成/proc/scullmem和/proc/scullseq文件，用户可以通过这些文件获得scull设备相关信息。对应的源码文件主要是examples/scull/main.c。

一、使用proc文件系统

需要注意的一点是，如果要测试本章相关代码，即建立/proc/scullmem和/proc/scullseq文件并进行操作，需要打开DEBUG选项，否则编译出的模块是不包含调试功能的。打开DEBUG选项的方法是修改Makefile的第2行，打开DEBUG=y选项。

scull模块是在模块初始化函数scull\_init\_module中调用scull\_create\_proc函数创建/proc/scullmem和/proc/scullseq两个文件的，列出代码如下：

[cpp]

01. 661#ifdef SCULL\_DEBUG /\* only when debugging \*/

02. 662 scull\_create\_proc();

03. 663#endif

由代码可见，只有定义了SCULL\_DEBUG变量，才会调用scull\_create\_proc函数。那么SCULL\_DEBUG是在哪里定义的呢？答案是在Makefile文件中：

[cpp]

01. 2DEBUG = y

02. 3

03. 4

04. 5# Add your debugging flag (or not) to CFLAGS

05. 6ifeq (\$(DEBUG),y)

06. 7 DEBFLAGS = -O -g -DSCULL\_DEBUG # "-O" is needed to expand inlines

07. 8else

08. 9 DEBFLAGS = -O2

09. 10endif

在Makefile的第2行我们打开了DEBUG=y(默认情况下这行是被屏蔽的)。所以第6行判断成立，就会执行

<http://blog.csdn.net/liuhaoyutz/article/details/7383563>

1/7

最新评论

- LDD3源码分析之内存映射  
wzw88486969:  
@fjlhlonng: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  
electfan2011: 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序  
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序  
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! (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! (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! (3)
- LDD3源码分析之时间与! (3)
- LDD3源码分析之poll分! (2)

文章存档

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

第7行，其中-DSCULL\_DEBUG在编译时会传递给gcc，参考gcc的-D选项，这等价于在头文件中定义了#define SCULL\_DEBUG，所以main.c的661行#ifndef SCULL\_DEBUG成立，进而调用662行scull\_create\_proc创建proc文件。

```
[cpp]
01. 209static void scull_create_proc(void)
02. 210{
03. 211     struct proc_dir_entry *entry;
04. 212     create_proc_read_entry("scullmem", 0 /* default mode */,
05. 213         NULL /* parent dir */, scull_read_procmem,
06. 214         NULL /* client data */);
07. 215     entry = create_proc_entry("scullseq", 0, NULL);
08. 216     if (entry)
09. 217         entry->proc_fops = &scull_proc_ops;
10. 218}
```

212行，调用create\_proc\_read\_entry函数创建/proc/scullmem，该函数函数原型如下：

```
struct proc_dir_entry *create_proc_read_entry(const char *name,
                                             mode_t mode,
                                             struct proc_dir_entry *base,
                                             read_proc_t *read_proc,
                                             void *data);
```

name是要创建的文件名称；  
mode是该文件的保护掩码(传入0表示使用系统默认值)；

base指定该文件所在的目录(如果base为NULL，则该文件将创建在/proc根目录下)；

read\_proc是实现该文件的读操作的函数；

data是传递给read\_proc的参数。

这里我们重点看read\_proc函数。为了创建一个只读的/proc文件，驱动程序必须实现一个函数，用于在读取文件时生成数据，这个函数称为read\_proc。当某个进程读取这个/proc文件时(使用read系统调用)，就会调用相应驱动程序的read\_proc函数。

read\_proc函数的原型如下：

```
int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data)
```

这个函数的参数比较难理解，这里我偷下懒，直接把文档Linux Kernel Procfs Guide上对这个函数的解释复制过来，大家自己看。另外，LDD3上说的比较难以理解，特别是中文版，翻译上有许多不对的地方，建议大家看英文版的描述。

The read function is a call back function that allows userland processes to read data from the kernel.

The read function should write its information into the *page*. For proper use, the function should start writing at an offset of *off* in *page* and write at most *count* bytes, but because most read functions are quite simple and only return a small amount of information, these two parameters are usually ignored (it breaks pagers like more and less, but cat still works).

If the *off* and *count* parameters are properly used, *eof* should be used to signal that the end of the file has been reached by writing 1 to the memory location *eof* points to.

The parameter *start* doesn't seem to be used anywhere in the kernel. The *data* parameter can be used to create a single call back function for several files.

The read\_func function must return the number of bytes written into the *page*.

2014年01月 (1)

2013年12月 (6)

展开

文章搜索

推荐文章

对应212 - 214行, scull模块在/proc下创建了一个称为scullmem的文件, 并默认具有全局可读的权限, 对应该文件的read\_proc函数是scull\_read\_procmem, 列出如下:

```
[cpp]
01. 90int scull_read_procmem(char *buf, char **start, off_t offset,
02. 91                      int count, int *eof, void *data)
03. 92{
04. 93     int i, j, len = 0;
05. 94     int limit = count - 80; /* Don't print more than this */
06. 95
07. 96     for (i = 0; i < scull_nr_devs && len <= limit; i++) {
08. 97         struct scull_dev *d = &scull_devices[i];
09. 98         struct scull_qset *qs = d->data;
10. 99         if (down_interruptible(&d->sem))
11. 100             return -ERESTARTSYS;
12. 101         len += sprintf(buf+len, "\nDevice %i: qset %i, q %i, sz %li\n",
13. 102                        i, d->qset, d->quantum, d->size);
14. 103         for (; qs && len <= limit; qs = qs->next) { /* scan the list */
15. 104             len += sprintf(buf + len, " item at %p, qset at %p\n",
16. 105                            qs, qs->data);
17. 106             if (qs->data && !qs->next) /* dump only the last item */
18. 107                 for (j = 0; j < d->qset; j++) {
19. 108                     if (qs->data[j])
20. 109                         len += sprintf(buf + len,
21. 110                                       "    %4i: %8p\n",
22. 111                                       j, qs->data[j]);
23. 112                 }
24. 113             }
25. 114             up(&scull_devices[i].sem);
26. 115         }
27. 116         *eof = 1;
28. 117         return len;
29. 118 }
```

94行, 定义limit变量, 这里limit的值表明最多向buf中写count - 80个字符。实际上按照read\_proc的定义, 最多向参数buf中写入参数count指定个数的字符, 这里只是进一步限制最多写入字数。系统传递的count值应该小于一页大小, 在我的机器上验证count的值为3072。

96行, 循环处理scull0 - scull3, 每次循环处理1个设备。

101行, 将设备相关信息保存进buf中。包括设备编号, 量子集大小, 量子大小, 设备实际大小。

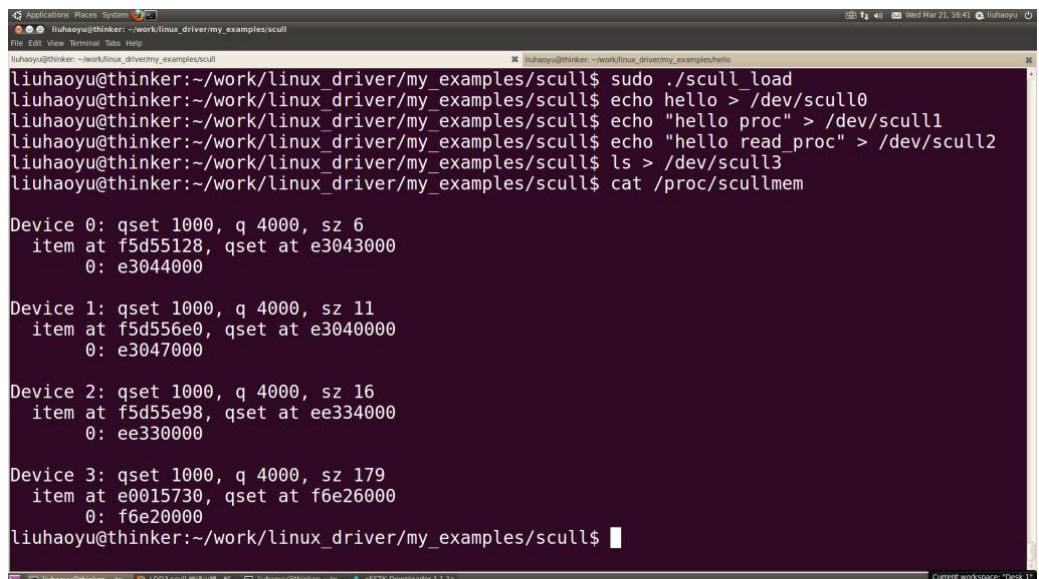
103行, 循环处理设备的scull\_qset, 每次循环处理一个scull\_qset。

104行, 将scull\_qset相关信息保存进buf中, 包括scull\_qset的地址和量子数组地址。

107行, 循环处理量子数组, 每次循环处理一个量子数组成员。

109行, 将量子数组成员的信息保存进buf中, 包括数组下标和相应数组成员地址。

下图是我对/proc/scullmem的测试过程:



```
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ sudo ./scull_load
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ echo hello > /dev/scull0
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ echo "hello proc" > /dev/scull1
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ echo "hello read_proc" > /dev/scull2
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ ls > /dev/scull3
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$ cat /proc/scullmem

Device 0: qset 1000, q 4000, sz 6
 item at f5d55128, qset at e3043000
 0: e3044000

Device 1: qset 1000, q 4000, sz 11
 item at f5d556e0, qset at e3040000
 0: e3047000

Device 2: qset 1000, q 4000, sz 16
 item at f5d55e98, qset at ee334000
 0: ee330000

Device 3: qset 1000, q 4000, sz 179
 item at e0015730, qset at f6e26000
 0: f6e20000
liuhaoyu@thinker:~/work/linux_driver/my_examples/scull$
```

## 二、使用seq\_file接口

针对/proc系统处理大文件比较困难的问题，内核提供了seq\_file接口。

seq\_file的实现基于/proc系统。要使用seq\_file，我们必须抽象出一个对象序列，然后可以依次遍历对象序列的每个成员。这个对象序列可以是链表，数组，哈希表等等。具体到scull模块，是把scull\_devices数组做为一个对象序列，每个对象就是一个scull\_dev结构。

seq\_file接口有两个重要数据结构：

```
struct seq_file {  
    char *buf;  
  
    size_t size;  
  
    size_t from;  
  
    size_t count;  
  
    loff_t index;  
  
    loff_t read_pos;  
  
    u64 version;  
  
    struct mutex lock;  
  
    const struct seq_operations *op;  
  
    void *private;  
};
```

seq\_file结构在seq\_open函数中创建，然后作为参数传递给每个seq\_file接口操作函数。

```
struct seq_operations {  
  
    void * (*start) (struct seq_file *m, loff_t *pos);  
  
    void (*stop) (struct seq_file *m, void *v);  
  
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);  
  
    int (*show) (struct seq_file *m, void *v);  
};
```

要使用seq\_file接口，必须实现四个操作函数，分别是start(), next(), show(), stop()。

start函数完成初始化工作，在遍历操作开始时调用，返回一个对象指针。

show函数对当前正在遍历的对象进行操作，利用seq\_printf, seq\_puts等函数，打印这个对象的信息。

next函数在遍历中寻找下一个对象并返回。

stop函数在遍历结束时调用，完成一些清理工作。

下面我们看scull模块中是怎样使用seq\_file接口的：

```
[cpp]  
01. 126/*  
02. 127 * Here are our sequence iteration methods. Our "position" is  
03. 128 * simply the device number.  
04. 129 */  
05. 130static void *scull_seq_start(struct seq_file *s, loff_t *pos)  
06. 131{  
07. 132     if (*pos >= scull_nr_devs)
```

```

08. 133         return NULL; /* No more to read */
09. 134     return scull_devices + *pos;
10. 135}
11. 136
12. 137static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
13. 138{
14. 139     (*pos)++;
15. 140     if (*pos >= scull_nr_devs)
16. 141         return NULL;
17. 142     return scull_devices + *pos;
18. 143}
19. 144
20. 145static void scull_seq_stop(struct seq_file *s, void *v)
21. 146{
22. 147     /* Actually, there's nothing to do here */
23. 148}
24. 149
25. 150static int scull_seq_show(struct seq_file *s, void *v)
26. 151{
27. 152     struct scull_dev *dev = (struct scull_dev *) v;
28. 153     struct scull_qset *d;
29. 154     int i;
30. 155
31. 156     if (down_interruptible(&dev->sem))
32. 157         return -ERESTARTSYS;
33. 158     seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
34. 159         (int) (dev - scull_devices), dev->qset,
35. 160         dev->quantum, dev->size);
36. 161     for (d = dev->data; d; d = d->next) { /* scan the list */
37. 162         seq_printf(s, "  item at %p, qset at %p\n", d, d->data);
38. 163         if (d->data && !d->next) /* dump only the last item */
39. 164             for (i = 0; i < dev->qset; i++) {
40. 165                 if (d->data[i])
41. 166                     seq_printf(s, "    %4i: %8p\n",
42. 167                         i, d->data[i]);
43. 168             }
44. 169     }
45. 170     up(&dev->sem);
46. 171     return 0;
47. 172}
48. 173
49. 174/*
50. 175 * Tie the sequence operators up.
51. 176 */
52. 177static struct seq_operations scull_seq_ops = {
53. 178     .start = scull_seq_start,
54. 179     .next = scull_seq_next,
55. 180     .stop = scull_seq_stop,
56. 181     .show = scull_seq_show
57. 182};
58. 183
59. 184/*
60. 185 * Now to implement the /proc file we need only make an open
61. 186 * method which sets up the sequence operators.
62. 187 */
63. 188static int scull_proc_open(struct inode *inode, struct file *file)
64. 189{
65. 190     return seq_open(file, &scull_seq_ops);
66. 191}
67. 192
68. 193/*
69. 194 * Create a set of file operations for our proc file.
70. 195 */
71. 196static struct file_operations scull_proc_ops = {
72. 197     .owner  = THIS_MODULE,
73. 198     .open   = scull_proc_open,
74. 199     .read   = seq_read,
75. 200     .llseek = seq_lseek,
76. 201     .release = seq_release
77. 202};
78. 203
79. 204
80. 205/*
81. 206 * Actually create (and remove) the /proc file(s).
82. 207 */
83. 208
84. 209static void scull_create_proc(void)
85. 210{
86. 211     struct proc_dir_entry *entry;

```

```
87. 212 create_proc_read_entry("scullmem", 0 /* default mode */,
88. 213 NULL /* parent dir */, scull_read_procmem,
89. 214 NULL /* client data */);
90. 215 entry = create_proc_entry("scullseq", 0, NULL);
91. 216 if (entry)
92. 217     entry->proc_fops = &scull_proc_ops;
93. 218}
```

126 - 148行，实现了scull\_seq\_start，scull\_seq\_next，scull\_seq\_stop三个函数，这三个函数比较简单，没有什么可说的。如果不明白，可以参考LDD3。

150 - 172行，实现了scull\_seq\_show。这个函数与前面介绍的scull\_read\_procmem逻辑是一样的，主要区别是打印语句用seq\_printf，可对比参考理解。

177 - 182行，填充了一个seq\_operations结构体，这是seq\_file接口要求的。

接下来，我们要实现一个file\_operations结构，这个结构将实现在该/proc文件上进行读取和定位时所需要的所有操作。与第三章介绍的字符设备驱动程序不同，这里我们只要实现一个open方法，其他的方法可以直接使用seq\_file接口提供的函数。

188 - 191行，实现了open方法scull\_proc\_open。调用seq\_open(file, &scull\_seq\_ops)将file结构与seq\_operations结构体连接在一起。

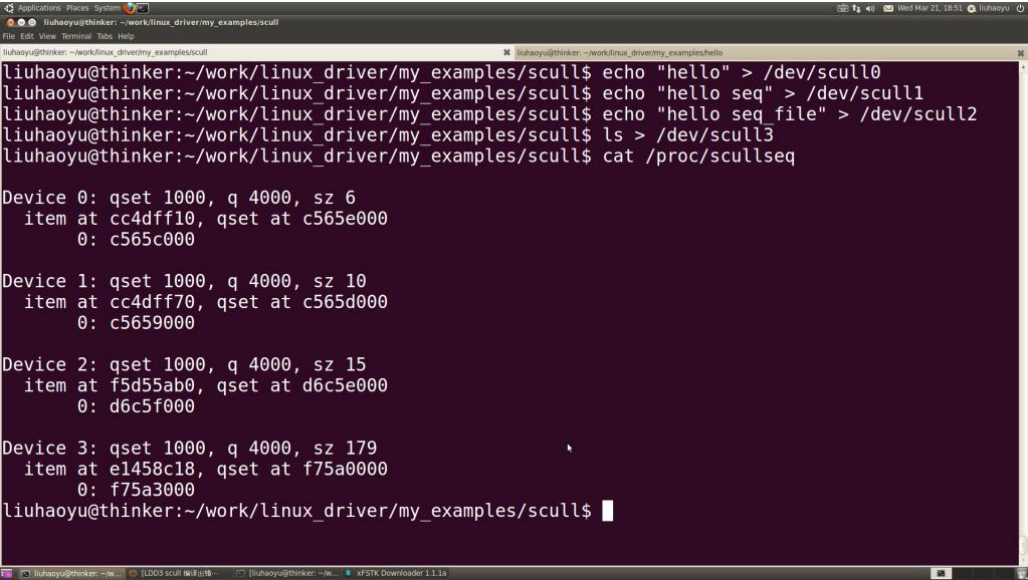
196 - 202行，定义了file\_operations结构体scull\_proc\_ops，其中，只有scull\_proc\_open是我们自己定义的，其他函数都是使用seq\_file接口提供的函数。

215行，调用create\_proc\_entry函数创建/proc/scullseq文件。

217行，接scull\_proc\_ops结构体与/proc/scullseq连接起来。

至此，使用seq\_file接口进行调试的过程我们就分析完了。

下图是在我的系统上使用seq\_file接口打印的信息：



更多 0

上一篇 LDD3源码分析之字符设备驱动程序

下一篇 LDD3源码分析之并发与竞态

顶 3 踩 0



猜你在找

wifi 架构	解读set_gpio_ctrl(GPIO_MODE_OUT   GPIO_H6)
如何在windows下面编译u-boot（原发于：2012-07-24	linux内核DMA内存分配
linux input输入子系统分析《二》：s3c2440的ADC简单	u-boot编译笔记
想成为Android高手必须学习的干货	Eclipse中选择git 的repository的某个项目，不要使用
Android WIFI 架构	STM32 对内部FLASH读写接口函数

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

1楼 [fantasyhujian](#) 2013-10-18 14:28发表



分析的很清楚，赞一个！

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