1. **通过打印调试**

**PrintK**

这里有 2 个 printk 命令的例子, 一个调试消息, 一个紧急消息:

printk(KERN\_DEBUG "Here I am: %s:%i\n", \_\_FILE\_\_, \_\_LINE\_\_);

printk(KERN\_CRIT "I'm trashed; giving up on %p\n", ptr);

有 8 种可能的记录字串, 在头文件 <linux/kernel.h> 里定义; 我们按照严重

性递减的顺序列出它们:

KERN\_EMERG

用于紧急消息, 常常是那些崩溃前的消息.

KERN\_ALERT

需要立刻动作的情形.

KERN\_CRIT 严重情况, 常常与严重的硬件或者软件失效有关.

KERN\_ERR

用来报告错误情况; 设备驱动常常使用 KERN\_ERR 来报告硬件故障.

KERN\_WARNING

有问题的情况的警告, 这些情况自己不会引起系统的严重问题.

KERN\_NOTICE

正常情况, 但是仍然值得注意. 在这个级别一些安全相关的情况会报告.

KERN\_INFO

信息型消息. 在这个级别, 很多驱动在启动时打印它们发现的硬件的信

息.

KERN\_DEBUG

用作调试消息.

注：在发布驱动程序时我们应该移除这些调试printK

1. **使用/proc文件系统**

/proc 文件系统是一个特殊的软件创建的文件系统, 内核用来输出消息到外界. /proc 下的每个文件都绑到一个内核函数上, 当文件被读的时候即时产生文件内容.

我们不鼓励在 /proc 下添加文件. /proc 文件 系统在内核开发者看作是有点无法控制的混乱, 建议新代码中使信息可获取的方法是利用 sysfs.

所有使用 /proc 的模块应当包含 <linux/proc\_fs.h>

**实现read\_proc函数**

实现/proc需要实现如下函数

int (\*read\_proc)(char \*page, char \*\*start, off\_t offset, int count, int \*eof, void \*data);

page 指针是你写你的数据的缓存区，缓冲区大小为一页（4k）;

start 是这个参数传入为空，如果在函数中设置了\*start的值，那么下一次调用offset为设置的值， 此参数一般用于需要多页信息输出

offset 和 count 对于 read 方法有同样的含义.

eof 参数指向一个整数, 必须由驱动设置来指示它不再有数据返回,

data 是驱动特定的数据指针, 你可以用做内部用途.

示例：

int scull\_read\_procmem(char \*buf, char \*\*start, off\_t offset, int count, int \*eof, void \*data)

{

    int i, j, len = 0;

    int limit = count - 80;     /\* Don't print more than this \*/

    for (i = 0; i < scull\_nr\_devs && len <= limit; i++)

    {

        struct scull\_dev \*d = &scull\_devices[i];

        struct scull\_qset \*qs = d->data;

        if (down\_interruptible(&d->sem))

            return -ERESTARTSYS;

        // 向缓冲区写入数据

        // 写入设备信息

        len += sprintf(buf + len, "\nDevice %i: qset %i, q %i, sz %li\n", i, d->qset, d->quantum, d->size);

        for (; qs && len <= limit; qs = qs->next)

        {

            // 写入scull\_qset的信息

            len += sprintf(buf + len, " item at %p, qset at %p\n", qs, qs->data);

        }

        up(&scull\_devices[i].sem);

    }

    \*eof = 1;       // 指示数据以完全返回

    return len;     // 返回写入的数量

}

**创建/proc文件**

使用一个如下函数创建/proc文件:

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 是要创建的文件名子

mod 是文件的保护掩码(缺省系统范围时可以 作为 0 传递)

base 指出要创建的文件的目录( 如果 base 是 NULL, 文件在 /proc 根下创建 )

read\_proc 是实现文件的 read\_proc 函数

data 被内核忽略( 但是传递给 read\_proc).

示例：

create\_proc\_read\_entry("scullmem", 0 /\* default mode \*/,

                       NULL /\* parent dir \*/,

                       scull\_read\_procmem,

                       NULL /\* client data \*/);

这里, 我们创建了一个名为 scullmem 的文件, 直接在 /proc 下, 带有缺省的,

全局可读的保护.

**移除/proc文件**

应当在模块卸载后应当移除/proc文件

remove\_proc\_entry("scullmem", NULL /\* parent dir \*/);

1. **seq\_file 接口**

在 /proc 下的大文件的实现有点麻烦. /proc的替代方法 seq\_file 接口.

为使用 seq\_file, 你必须创建 4 个 iterator 方法, 称为 start, next, stop, 和 show. 头文件<linux/seq\_file.h>

**Start方法**

start 方法是首先调用. 这个函数的原型是:

void \*start(struct seq\_file \*sfile, loff\_t \*pos);

sfile 参数可以几乎是一直被忽略.

pos 是一个整型位置值,

返回值 NULL或者迭代器iterator

示例：

static void \*scull\_seq\_start(struct seq\_file \*s, loff\_t \*pos)

{

    if (\*pos >= scull\_nr\_devs)

        return NULL; /\* No more to read \*/

    return scull\_devices + \*pos;

}

**Next方法**

next 函数应当动 iterator 到下一个位置, 如果序列里什么都没有剩下就返 回 NULL.

这个方法的原型是:

void \*next(struct seq\_file \*sfile, void \*v, loff\_t \*pos);

v 是从前一个对 start 或者 next 的调用返回的 iterator

pos 是文件的当前位置. next 应当递增有 pos 指向的值; 根据你的 iterator 是如何工 作的, 你可能(尽管可能不会)需要递增 pos 不止是 1.

示例：

static void \*scull\_seq\_next(struct seq\_file \*s, void \*v, loff\_t \*pos)

{

    (\*pos)++;

    if (\*pos >= scull\_nr\_devs)

        return NULL;

    return scull\_devices + \*pos;

}

**Stop方法**

当内核处理完 iterator, 它调用 stop 来清理:

void stop(struct seq\_file \*sfile, void \*v);

scull 实现没有清理工作要做, 所以它的 stop 方法是空的.

**Show方法**

在这些调用中, 内核调用 show 方法来真正输出有用的东西给用户空间. 这个方法的原型是:

int show(struct seq\_file \*sfile, void \*v);

这个方法应当创建序列中由 iterator v 指示的项的输出.

Show中使用printk输出，有一套特殊的用作 seq\_file 输出的函数:

int seq\_printf(struct seq\_file \*sfile, const char \*fmt, ...);

回到我们的例子; 在 scull 使用的 show 方法是:

static int scull\_seq\_show(struct seq\_file \*s, void \*v)

{

    struct scull\_dev \*dev = (struct scull\_dev \*)v;

    struct scull\_qset \*d;

    int i;

    if (down\_interruptible(&dev->sem))

        return -ERESTARTSYS;

    seq\_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n", (int)(dev - scull\_devices), dev->qset, dev->quantum, dev->size);

    for (d = dev->data; d; d = d->next)

    {

        seq\_printf(s, " item at %p, qset at %p\n", d, d->data);

    }

    up(&dev->sem);

    return 0;

}

**填充seq\_operations**

现在已有了一个完整的 iterator 操作的集合, scull 必须包装起它们, 并且 连接它们到 /proc 中的一个文件.

第一步是填充一个 seq\_operations 结构:

static struct seq\_operations scull\_seq\_ops = {

    .start = scull\_seq\_start,

    .next = scull\_seq\_next,

    .stop = scull\_seq\_stop,

    .show = scull\_seq\_show

};

**填充file\_operations**

File\_operation结构我们只要实现open方法即可

static int scull\_proc\_open(struct inode \*inode, struct file \*file)

{

    // 将文件与我们的seq\_operations相连

    return seq\_open(file, &scull\_seq\_ops);

}

file\_operations 结构如下:

static struct file\_operations scull\_proc\_ops = {

    .owner = THIS\_MODULE,

    .open = scull\_proc\_open,

    .read = seq\_read,

    .llseek = seq\_lseek,

    .release = seq\_release

};

这里我们指定我们自己的 open 方法, 和使用预装好的方法 seq\_read, seq\_lseek, 和 seq\_release

**创建/proc文件**

最后的步骤是创建 /proc 中的实际文件：

entry = create\_proc\_entry("scullseq", 0, NULL);

if (entry)

    entry->proc\_fops = &scull\_proc\_ops;

有了上面代码, scull 有一个新的 /proc 入口, 看来很象前面的一个.

我们建议使用 seq\_file , 来实现包含多个非常小数目的输出行数的文件.

1. **Oops消息**

大部分错误是由于指针错误所引起的，这些错误通常导致一个opps消息

处理器使用的任何地址几乎都是一个虚拟地址, 通过一个复杂的页表结构映射

为物理地址(例外是内存管理子系统自己使用的物理地址). 当解引用一个无效

的指针, 分页机制无法映射指针到一个物理地址, 处理器发出一个页错误给操

作系统.

如下是一个错误消息

Unable to handle kernel NULL pointer dereference at virtual address

00000000

printing eip:

d083a064

Oops: 0002 [#1]

SMP

CPU: 0

EIP: 0060:[<d083a064>] Not tainted

EFLAGS: 00010246 (2.6.6)

EIP is at faulty\_write+0x4/0x10 [faulty]

eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000

esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74

ds: 007b es: 007b ss: 0068

Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)

Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460

cf8b2460 fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005

cf8b2480 00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005

00000005

Call Trace:

[<c0150558>] vfs\_write+0xb8/0x130

[<c0150682>] sys\_write+0x42/0x70

[<c0103f8f>] syscall\_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0

当你面对一个 oops, 第一件事是查看发生问题的位置, 如下消息：

EIP is at faulty\_write+0x4/0x10 [faulty]

我们看到, 我们在函数 faulty\_write, 位于 faulty 模块. 16 进制数指示指令指针是函数内 4 字节, 函数是 10 ( 16 进制 )字节长. 常常这就足够来知道问题是什么.