+---------------------------------------------------+  
|                 写一个块设备驱动                  |  
+---------------------------------------------------+  
| 作者：赵磊                                        |  
| 网名：OstrichFly、飞翔的鸵鸟                      |  
| email: zhaoleidd@hotmail.com                      |  
+---------------------------------------------------+  
| 文章版权归原作者所有。                            |  
| 大家可以自由转载这篇文章，但原版权信息必须保留。  |  
| 如需用于商业用途，请务必与原作者联系，若因未取得  |  
| 授权而收起的版权争议，由侵权者自行负责。          |  
+---------------------------------------------------+

# 第1章 ramdisk起步

同样是读书，读小说可以行云流水，读完后心情舒畅，意犹未尽；读电脑书却举步艰难，读完后目光呆滞，也是意犹未尽，只不过未尽的是痛苦的回忆。  
研究证明，痛苦的记忆比快乐的更难忘记，因此电脑书中的内容比小说记得持久。  
而这套教程的目的是要打破这种状况，以至于读者在忘记小说内容忘记本文。  
  
在这套教程中，我们通过写一个建立在内存中的块设备驱动，来学习linux内核和相关设备驱动知识。  
选择写块设备驱动的原因是：  
1：容易上手  
2：可以牵连出更多的内核知识  
3：像本文这样的块设备驱动教程不多，所以需要一个  
  
好吧，扯淡到此结束，我们开始写了。  
  
本章的目的用尽可能最简单的方法写出一个能用的块设备驱动。  
所谓的能用，是指我们可以对这个驱动生成的块设备进行mkfs，mount和读写文件。  
为了尽可能简单，这个驱动的规模不是1000行，也不是500行，而是100行以内。  
  
这里插一句，我们不打算在这里介绍如何写模块，理由是介绍的文章已经满天飞舞了。  
如果你能看得懂、并且成功地编译、运行了这段代码，我们认为你已经达到了本教程的入学资格，  
当然，如果你不幸的卡在这段代码中，那么请等到搞定它以后再往下看：  
mod.c:  
#include <linux/module.h>  
  
static int \_\_init init\_base(void)  
{  
        printk("----Hello. World----\n");  
        return 0;  
}  
  
static void \_\_exit exit\_base(void)  
{  
        printk("----Bye----\n");  
}  
  
module\_init(init\_base);  
module\_exit(exit\_base);  
  
MODULE\_LICENSE ("GPL");  
MODULE\_AUTHOR("Zhao Lei");  
MODULE\_DESCRIPTION("For test");  
  
Makefile:  
obj-m := mod.o  
KDIR := /lib/modules/$(shell uname -r)/build  
PWD := $(shell pwd)  
  
default:  
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules  
clean:  
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean  
        rm -rf Module.markers modules.order Module.symvers  
  
  
好了，这里我们假定你已经搞定上面的最简单的模块了，懂得什么是看模块，以及简单模块的编写、编译、加载和卸载。  
还有就是，什么是块设备，什么是块设备驱动，这个也请自行google吧，因为我们已经迫不及待要写完程序下课。  
  
为了建立一个可用的块设备，我们需要做......1件事情：  
1：用add\_disk()函数向系统中添加这个块设备  
   添加一个全局的  
   static struct gendisk \*simp\_blkdev\_disk;  
   然后申明模块的入口和出口：  
   module\_init(simp\_blkdev\_init);  
   module\_exit(simp\_blkdev\_exit);  
   然后在入口处添加这个设备、出口处释放这个设备：  
   static int \_\_init simp\_blkdev\_init(void)  
   {  
           add\_disk(simp\_blkdev\_disk);  
        return 0;  
   }  
   static void \_\_exit simp\_blkdev\_exit(void)  
   {  
           del\_gendisk(simp\_blkdev\_disk);  
   }  
  
当然，在添加设备之前我们需要申请这个设备的资源，这用到了alloc\_disk()函数，因此模块入口函数simp\_blkdev\_init(void)应该是：  
   static int \_\_init simp\_blkdev\_init(void)  
   {  
        simp\_blkdev\_disk = alloc\_disk(1);  
        if (!simp\_blkdev\_disk) {  
                ret = -ENOMEM;  
                goto err\_alloc\_disk;  
        }  
  
           add\_disk(simp\_blkdev\_disk);  
  
        return 0;  
  
   err\_alloc\_disk:  
        return ret;  
   }  
还有别忘了在卸载模块的代码中也加一个行清理函数：  
  put\_disk(simp\_blkdev\_disk);  
  
还有就是，设备有关的属性也是需要设置的，因此在alloc\_disk()和add\_disk()之间我们需要：  
        strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);  
        simp\_blkdev\_disk->major = ?1;  
        simp\_blkdev\_disk->first\_minor = 0;  
        simp\_blkdev\_disk->fops = ?2;  
        simp\_blkdev\_disk->queue = ?3;  
        set\_capacity(simp\_blkdev\_disk, ?4);  
  
SIMP\_BLKDEV\_DISKNAME其实是这个块设备的名称，为了绅士一些，我们把它定义成宏了：  
#define SIMP\_BLKDEV\_DISKNAME        "simp\_blkdev"  
  
这里又引出了4个问号。(天哪，是不是有种受骗的感觉，像是陪老婆去做头发)  
第1个问号：  
  每个设备需要对应的主、从驱动号。  
  我们的设备当然也需要，但很明显我不是脑科医生，因此跟写linux的那帮疯子不熟，得不到预先为我保留的设备号。  
  还有一种方法是使用动态分配的设备号，但在这一章中我们希望尽可能做得简单，因此也不采用这种方法。  
  那么我们采用的是：抢别人的设备号。  
  我们手头没有AK47，因此不敢干的太轰轰烈烈，而偷偷摸摸的事情倒是可以考虑的。  
  柿子要捡软的捏，而我们试图找出一个不怎么用得上的设备，然后抢他的ID。  
  打开linux/include/linux/major.h，把所有的设备一个个看下来，我们觉得最胜任被抢设备号的家伙非COMPAQ\_SMART2\_XXX莫属。  
  第一因为它不强势，基本不会被用到，因此也不会造成冲突；第二因为它有钱，从COMPAQ\_SMART2\_MAJOR到COMPAQ\_SMART2\_MAJOR7有那8个之多的设备号可以被抢，不过瘾的话还有它妹妹：COMPAQ\_CISS\_MAJOR~COMPAQ\_CISS\_MAJOR7。  
  为了让抢劫显得绅士一些，我们在外面又定义一个宏：  
  #define SIMP\_BLKDEV\_DEVICEMAJOR        COMPAQ\_SMART2\_MAJOR  
  然后在?1的位置填上SIMP\_BLKDEV\_DEVICEMAJOR。  
第2个问号：  
  gendisk结构需要设置fops指针，虽然我们用不到，但该设还是要设的。  
  好吧，就设个空得给它：  
  在全局部分添加：  
  struct block\_device\_operations simp\_blkdev\_fops = {  
          .owner                = THIS\_MODULE,  
  };  
  然后把?2的位置填上&simp\_blkdev\_fops。  
第3个问号：  
  这个比较麻烦一些。  
  首先介绍请求队列的概念。对大多数块设备来说，系统会把对块设备的访问需求用bio和bio\_vec表示，然后提交给通用块层。  
  通用块层为了减少块设备在寻道时损失的时间，使用I/O调度器对这些访问需求进行排序，以尽可能提高块设备效率。  
  关于I/O调度器在本章中不打算进行深入的讲解，但我们必须知道的是：  
  1：I/O调度器把排序后的访问需求通过request\_queue结构传递给块设备驱动程序处理  
  2：我们的驱动程序需要设置一个request\_queue结构  
  申请request\_queue结构的函数是blk\_init\_queue()，而调用blk\_init\_queue()函数时需要传入一个函数的地址，这个函数担负着处理对块设备数据的请求。  
  因此我们需要做的就是：  
  1：实现一个static void simp\_blkdev\_do\_request(struct request\_queue \*q)函数。  
  2：加入一个全局变量，指向块设备需要的请求队列：  
     static struct request\_queue \*simp\_blkdev\_queue;  
  3：在加载模块时用simp\_blkdev\_do\_request()函数的地址作参数调用blk\_init\_queue()初始化一个请求队列：  
     simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL);  
     if (!simp\_blkdev\_queue) {  
             ret = -ENOMEM;  
             goto err\_init\_queue;  
     }  
  4：卸载模块时把simp\_blkdev\_queue还回去：  
     blk\_cleanup\_queue(simp\_blkdev\_queue);  
  5：在?3的位置填上simp\_blkdev\_queue。  
第4个问号：  
  这个还好，比前面的简单多了，这里需要设置块设备的大小。  
  块设备的大小使用扇区作为单位设置，而扇区的大小默认是512字节。  
  当然，在把字节为单位的大小转换为以扇区为单位时，我们需要除以512，或者右移9位可能更快一些。  
  同样，我们试图把这一步也做得绅士一些，因此使用宏定义了块设备的大小，目前我们定为16M：  
  #define SIMP\_BLKDEV\_BYTES        (16\*1024\*1024)  
  然后在?4的位置填上SIMP\_BLKDEV\_BYTES>>9。  
  
看到这里，是不是有种身陷茫茫大海的无助感？并且一波未平，一波又起，在搞定这4个问号的同时，居然又引入了simp\_blkdev\_do\_request函数！  
当然，如果在身陷茫茫波涛中时你认为到处都是海，因此绝望，那么恭喜你可以不必挨到65岁再退休；  
反之，如果你认为到处都是没有三聚氰胺鲜鱼，并且随便哪个方向都是岸时，那么也恭喜你，你可以活着回来继续享受身为纳税人的荣誉。  
  
为了理清思路，我们把目前为止涉及到的代码整理出来：  
#define SIMP\_BLKDEV\_DEVICEMAJOR        COMPAQ\_SMART2\_MAJOR  
#define SIMP\_BLKDEV\_DISKNAME        "simp\_blkdev"  
#define SIMP\_BLKDEV\_BYTES        (16\*1024\*1024)  
  
static struct request\_queue \*simp\_blkdev\_queue;  
static struct gendisk \*simp\_blkdev\_disk;  
  
static void simp\_blkdev\_do\_request(struct request\_queue \*q);  
  
struct block\_device\_operations simp\_blkdev\_fops = {  
        .owner                = THIS\_MODULE,  
};  
  
static int \_\_init simp\_blkdev\_init(void)  
{  
        int ret;  
  
        simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL);  
        if (!simp\_blkdev\_queue) {  
                ret = -ENOMEM;  
                goto err\_init\_queue;  
        }  
  
        simp\_blkdev\_disk = alloc\_disk(1);  
        if (!simp\_blkdev\_disk) {  
                ret = -ENOMEM;  
                goto err\_alloc\_disk;  
        }  
  
        strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);  
        simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;  
        simp\_blkdev\_disk->first\_minor = 0;  
        simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;  
        simp\_blkdev\_disk->queue = simp\_blkdev\_queue;  
        set\_capacity(simp\_blkdev\_disk, SIMP\_BLKDEV\_BYTES>>9);  
        add\_disk(simp\_blkdev\_disk);  
  
        return 0;  
  
err\_alloc\_disk:  
        blk\_cleanup\_queue(simp\_blkdev\_queue);  
err\_init\_queue:  
        return ret;  
}  
  
static void \_\_exit simp\_blkdev\_exit(void)  
{  
        del\_gendisk(simp\_blkdev\_disk);  
        put\_disk(simp\_blkdev\_disk);  
        blk\_cleanup\_queue(simp\_blkdev\_queue);  
}  
  
module\_init(simp\_blkdev\_init);  
module\_exit(simp\_blkdev\_exit);  
  
剩下部分的不多了，真的不多了。请相信我，因为我不在质监局上班。  
我写的文章诚实可靠，并且不拿你纳税的钱。  
  
我们还有一个最重要的函数需要实现，就是负责处理块设备请求的simp\_blkdev\_do\_request()。  
首先我们看看究竟把块设备的数据以什么方式放在内存中。  
毕竟这是在第1章，因此我们将使用最simple的方式实现，也就是，数组。  
我们在全局代码中定义：  
unsigned char simp\_blkdev\_data[SIMP\_BLKDEV\_BYTES];  
对驱动程序来说，这个数组看起来大了一些，如果不幸被懂行的人看到，将100%遭到最无情、最严重的鄙视。  
而我们却从极少数公仆那里学到了最有效的应对之策，那就是：无视他，然后把他定为成“不明真相的群众”。  
  
然后我们着手实现simp\_blkdev\_do\_request。  
这里介绍elv\_next\_request()函数，原型是：  
struct request \*elv\_next\_request(struct request\_queue \*q);  
用来从一个请求队列中拿出一条请求(其实严格来说，拿出的可能是请求中的一段)。  
随后的处理请求本质上是根据rq\_data\_dir(req)返回的该请求的方向(读/写)，把块设备中的数据装入req->buffer、或是把req->buffer中的数据写入块设备。  
刚才已经提及了与request结构相关的rq\_data\_dir()宏和.buffer成员，其他几个相关的结构成员和函数是：  
request.sector：请求的开始磁道  
request.current\_nr\_sectors：请求磁道数  
end\_request()：结束一个请求，第2个参数表示请求处理结果，成功时设定为1，失败时设置为0或者错误号。  
因此我们的simp\_blkdev\_do\_request()函数为：

// note: end\_request改为\_\_blk\_end\_request\_all  
static void simp\_blkdev\_do\_request(struct request\_queue \*q)  
{  
        struct request \*req;  
        while ((req = elv\_next\_request(q)) != NULL) {  
                if ((req->sector + req->current\_nr\_sectors) << 9  
                        > SIMP\_BLKDEV\_BYTES) {  
                        printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME  
                                ": bad request: block=%llu, count=%u\n",  
                                (unsigned long long)req->sector,  
                                req->current\_nr\_sectors);  
                        end\_request(req, 0);  
                        continue;  
                }  
  
                switch (rq\_data\_dir(req)) {  
                case READ:  
                        memcpy(req->buffer,  
                                simp\_blkdev\_data + (req->sector << 9),  
                                req->current\_nr\_sectors << 9);  
                        end\_request(req, 1);  
                        break;  
                case WRITE:  
                        memcpy(simp\_blkdev\_data + (req->sector << 9),  
                                req->buffer, req->current\_nr\_sectors << 9);  
                        end\_request(req, 1);  
                        break;  
                default:  
                        /\* No default because rq\_data\_dir(req) is 1 bit \*/  
                        break;  
                }  
        }  
}  
函数使用elv\_next\_request()遍历struct request\_queue \*q中使用struct request \*req表示的每一段，首先判断这个请求是否超过了我们的块设备的最大容量，  
然后根据请求的方向rq\_data\_dir(req)进行相应的请求处理。由于我们使用的是指简单的数组，因此请求处理仅仅是2条memcpy。  
memcpy中也牵涉到了扇区号到线性地址的转换操作，我想对坚持到这里的读者来说，这个操作应该不需要进一步解释了。  
  
编码到此结束，然后我们试试这个程序：  
首先编译：  
# make  
make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step1 modules  
make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'  
  CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step1/simp\_blkdev.o  
  Building modules, stage 2.  
  MODPOST  
  CC      /root/test/simp\_blkdev/simp\_blkdev\_step1/simp\_blkdev.mod.o  
  LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step1/simp\_blkdev.ko  
make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686'  
#  
加载模块  
# insmod simp\_blkdev.ko  
#  
用lsmod看看。  
这里我们注意到，该模块的Used by为0，因为它既没有被其他模块使用，也没有被mount。  
# lsmod  
Module                  Size  Used by  
simp\_blkdev         16784008  0  
...  
#  
如果当前系统支持udev，在调用add\_disk()函数时即插即用机制会自动为我们在/dev/目录下建立设备文件。  
设备文件的名称为我们在gendisk.disk\_name中设置的simp\_blkdev，主、从设备号也是我们在程序中设定的72和0。  
如果当前系统不支持udev，那么很不幸，你需要自己用mknod /dev/simp\_blkdev  b 72 0来创建设备文件了。  
# ls -l /dev/simp\_blkdev  
brw-r----- 1 root disk 72, 0 11-10 18:13 /dev/simp\_blkdev  
#  
在块设备中创建文件系统，这里我们创建常用的ext3。  
当然，作为通用的块设备，创建其他类型的文件系统也没问题。  
# mkfs.ext3 /dev/simp\_blkdev  
mke2fs 1.39 (29-May-2006)  
Filesystem label=  
OS type: Linux  
Block size=1024 (log=0)  
Fragment size=1024 (log=0)  
4096 inodes, 16384 blocks  
819 blocks (5.00%) reserved for the super user  
First data block=1  
Maximum filesystem blocks=16777216  
2 block groups  
8192 blocks per group, 8192 fragments per group  
2048 inodes per group  
Superblock backups stored on blocks:  
        8193  
  
Writing inode tables: done  
Creating journal (1024 blocks): done  
Writing superblocks and filesystem accounting information: done  
  
This filesystem will be automatically checked every 38 mounts or  
180 days, whichever comes first.  Use tune2fs -c or -i to override.  
#  
如果这是第一次使用，建议创建一个目录用来mount这个设备中的文件系统。  
当然，这不是必需的。如果你对mount之类的用法很熟，你完全能够自己决定在这里干什么，甚至把这个设备mount成root。  
# mkdir -p /mnt/temp1  
#  
把建立好文件系统的块设备mount到刚才建立的目录中  
# mount /dev/simp\_blkdev /mnt/temp1  
#  
看看现在的mount表  
# mount  
...  
/dev/simp\_blkdev on /mnt/temp1 type ext3 (rw)  
#  
看看现在的模块引用计数，从刚才的0变成1了，  
原因是我们mount了。  
# lsmod  
Module                  Size  Used by  
simp\_blkdev         16784008  1  
...  
#  
看看文件系统的内容，有个mkfs时自动建立的lost+found目录。  
# ls /mnt/temp1  
lost+found  
#  
随便拷点东西进去  
# cp /etc/init.d/\* /mnt/temp1  
#  
再看看  
# ls /mnt/temp1  
acpid           conman              functions  irqbalance    mdmpd           NetworkManagerDispatcher  rdisc            sendmail        winbind  
anacron         cpuspeed            gpm        kdump         messagebus      nfs                       readahead\_early  setroubleshoot  wpa\_supplicant  
apmd            crond               haldaemon  killall       microcode\_ctl   nfslock                   readahead\_later  single          xfs  
atd             cups                halt       krb524        multipathd      nscd                      restorecond      smartd          xinetd  
auditd          cups-config-daemon  hidd       kudzu         netconsole      ntpd                      rhnsd            smb             ypbind  
autofs          dhcdbd              ip6tables  lost+found    netfs           pand                      rpcgssd          sshd            yum-updatesd  
avahi-daemon    dund                ipmi       lvm2-monitor  netplugd        pcscd                     rpcidmapd        syslog  
avahi-dnsconfd  firstboot           iptables   mcstrans      network         portmap                   rpcsvcgssd       vmware  
bluetooth       frecord             irda       mdmonitor     NetworkManager  psacct                    saslauthd        vncserver  
#  
现在这个块设备的使用情况是  
# df  
文件系统               1K-块        已用     可用 已用% 挂载点  
...  
/dev/simp\_blkdev         15863      1440     13604  10% /mnt/temp1  
#  
再全删了玩玩  
# rm -rf /mnt/temp1/\*  
#  
看看删完了没有  
# ls /mnt/temp1  
#  
好了，大概玩够了，我们把文件系统umount掉  
# umount /mnt/temp1  
#  
模块的引用计数应该还原成0了吧  
# lsmod  
Module                  Size  Used by  
simp\_blkdev         16784008  0  
...  
#  
最后一步，移除模块  
# rmmod simp\_blkdev  
#  
  
这是这部教程的第1章，不好意思的是，内容比预期还是难了一些。  
当初还有一种考虑是在本章中仅仅实现一个写了就丢的块设备驱动，也就是说，对这个块设备的操作只能到mkfs这一部，而不能继续mount，因为刚才写的数据全被扔了。  
或者更简单些，仅仅写一个hello world的模块。  
但最后还是写成了现在这样没，因为我觉得拿出一个真正可用的块设备驱动程序对读者来说更有成就感。  
  
无论如何，本章是一个开始，而你，已经跨入了学习块设备驱动教室的大门，或者通俗来说，上了贼船。  
而在后续的章节中，我们将陆续完善对这个程序，通过追加或者强化这个程序，来学习与块设备有关、或与块设备无关但与linux有关的方方面面。  
总之，我希望通过这部教程，起码让读者学到有用的知识，或者更进一步，引导读者对linux的兴趣，甚至领悟学习一切科学所需要的钻研精神。  
  
作为第一章的结尾，引用我在另一篇文章中的序言：  
谨以此文向读者示范什么叫做严谨的研究。  
呼唤踏实的治学态度，反对浮躁的论坛风气。  
--OstrichFly

# 第2章 设置调度器

|  |
| --- |
| 上一章不但实现了一个最简单的块设备驱动程序，而且可能也成功地吓退了不少准备继续看下去的读者。 因为第一章看起来好像太难了。 不过读者也不要过于埋怨作者，因为大多数情况下第一次都不是什么好的体验......  对于坚持到这里的读者，这一章中，我们准备了一些简单的内容来犒劳大家。  关于块设备与I/O调度器的关系，我们在上一章中已经有所提及。 I/O调度器可以通过合并请求、重排块设备操作顺序等方式提高块设备访问的顺序。 就好像吃街边的大排档，如果点一个冷门的品种，可能会等更长的时间，而如果点的恰好与旁边桌子上刚点的相同，那么会很快上来，因为厨师八成索性一起炒了。 然而I/O调度器和块设备的情况却有一些微妙的区别，大概可以类比成人家点了个西红柿鸡蛋汤你接着就点了个西红柿炒蛋。 聪明的厨师一定会先做你的菜，因为随后可以直接往锅里加水煮汤，可怜比你先来的人喝的却是你的刷锅水。 两个菜一锅煮表现在块设备上可以类比成先后访问块设备的同一个位置，这倒是与I/O调度器无关，有空学习linux缓存策略时可以想想这种情况。  一个女孩子换了好多件衣服问我漂不漂亮，而我的评价只要一眼就能拿出来。 对方总觉得衣服要牌子好、面料好、搭配合理、要符合个人的气质、要有文化，而我的标准却简单的多：越薄越好。 所谓臭气相投，我写的块设备驱动程序对I/O调度器的要求大概也是如此。 究其原因倒不是因为块设备驱动程序好色，而是这个所谓块设备中的数据都是在内存中的。 这也意味着我们的“块设备”读写迅速、并且不存在磁盘之类设备通常面临的寻道时间。 因此对这个“块设备”而言，一个复杂的I/O调度器不但发挥不了丝毫作用，反而其本身将白白耗掉不少内存和CPU。 同样的情况还出现在固态硬盘、U盘、记忆棒之类驱动中。将来固态硬盘流行之时，大概就是I/O调度器消亡之日了。  这里我们试图给我们的块设备驱动选择一个最简单的I/O调度器。 目前linux中包含anticipatory、cfq、deadline和noop这4个I/O调度器。 2.6.18之前的linux默认使用anticipatory，而之后的默认使用cfq。 关于这4个调度器的原理和特性我们不打算在这里介绍，原因是相关的介绍满网都是。 但我们还是不能避免在这里提及一下noop调度器，因为我们马上要用到它。 noop顾名思义，是一个基本上不干事的调度器。它基本不对请求进行什么附加的处理，仅仅假惺惺地告诉通用块设备层：我处理完了。 但与吃空饷的公仆不同，noop的存在还是有不少进步意义的。至少我们现在就需要一个不要没事添乱的I/O调度器。  选择一个指定的I/O调度器需要这个函数： int elevator\_init(struct request\_queue \*q, char \*name); q是请求队列的指针，name是需要设定的I/O调度器的名称。 如果name为NULL，那么内核会首先尝试选择启动参数"elevator="中指定的调度器， 不成功的话就去选择编译内核时指定的默认调度器，如果运气太背还是不成功，就去选择"noop"调度器。 不要问我怎么知道的，一切皆在RTFSC(Read the F\*\*ing Source Code --Linus Torvalds)。  对于我们的代码，就是在simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL)后面加上： elevator\_init(simp\_blkdev\_queue, "noop");  但问题是在blk\_init\_queue()函数中系统已经帮我们申请一个了，因此这里我们需要费点周折，把老的那个送回去。 所以我们的代码应该是： simp\_blkdev\_init()函数开头处： elevator\_t \*old\_e; blk\_init\_queue()函数之后： old\_e = simp\_blkdev\_queue->elevator; if (IS\_ERR\_VALUE(elevator\_init(simp\_blkdev\_queue, "noop")))         printk(KERN\_WARNING "Switch elevator failed, using default\n"); else         elevator\_exit(old\_e);  为方便阅读并提高本文在google磁盘中的占用率，我们给出修改后的整个simp\_blkdev\_init()函数： static int \_\_init simp\_blkdev\_init(void) {         int ret;         elevator\_t \*old\_e;          simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL);         if (!simp\_blkdev\_queue) {                 ret = -ENOMEM;                 goto err\_init\_queue;         }  /\* 新版本代码中, 如果request\_queue->elevator不是NULL,就不会替换 \*/         old\_e = simp\_blkdev\_queue->elevator;         if (IS\_ERR\_VALUE(elevator\_init(simp\_blkdev\_queue, "noop")))                 printk(KERN\_WARNING "Switch elevator failed, using default\n");         else                 elevator\_exit(old\_e);          simp\_blkdev\_disk = alloc\_disk(1);         if (!simp\_blkdev\_disk) {                 ret = -ENOMEM;                 goto err\_alloc\_disk;         }          strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);         simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;         simp\_blkdev\_disk->first\_minor = 0;         simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;         simp\_blkdev\_disk->queue = simp\_blkdev\_queue;         set\_capacity(simp\_blkdev\_disk, SIMP\_BLKDEV\_BYTES>>9);         add\_disk(simp\_blkdev\_disk);          return 0;  err\_alloc\_disk:         blk\_cleanup\_queue(simp\_blkdev\_queue); err\_init\_queue:         return ret; }  本章的改动很小，我们现在测试一下这段代码： 首先我们像原先那样编译模块并加载： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step2 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step2/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step2/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step2/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # insmod simp\_blkdev.ko # 然后看一看咱们的这个块设备现在使用的I/O调度器： # cat /sys/block/simp\_blkdev/queue/scheduler [noop] anticipatory deadline cfq # 看样子是成功了。  哦，上一章中忘了看老程序的调度器信息了，这里补上老程序的情况： # cat /sys/block/simp\_blkdev/queue/scheduler noop anticipatory deadline [cfq] #  OK，我们完成简单的一章，并且用事实说明了作者并没有在开头撒谎。 当然，作者也会力图让接下来的章节同样比小说易读。 |

# 第3章 直接处理请求

|  |
| --- |
| 上一章中我们讨论了mm的衣服问题，并成功地为她换上了一件轻如鸿毛、关键是薄如蝉翼的新衣服。 而这一章中，我们打算稍稍再前进一步，也就是：给她脱光。 目的是更加符合我们的审美观、并且能够更加深入地了解该mm(喜欢制服皮草的读者除外)。 付出的代价是这一章的内容要稍稍复杂一些。  虽然noop调度器确实已经很简单了，简单到比我们的驱动程序还简单，在2.6.27中的120行代码量已经充分说明了这个问题。 但显而易见的是，不管它多简单，只要它存在，我们就把它看成累赘。 这里我们不打算再次去反复磨嘴皮子论证不使用I/O调度器能给我们的驱动程序带来什么样的好处、面临的困难、以及如何与国际接轨的诸多事宜， 毕竟现在不是在讨论汽油降价，而我们也不是中石油。我们更关心的是实实在在地做一些对驱动程序有益的事情。  不过I/O调度器这层遮体衣服倒也不是这么容易脱掉的，因为实际上我们还使用了它捆绑的另一个功能，就是请求队列。 因此我们在前两章中的程序才如此简单。 从细节上来说，请求队列request\_queue中有个make\_request\_fn成员变量，我们看它的定义： struct request\_queue {         ...         make\_request\_fn         \*make\_request\_fn;         ... } 它实际上是： typedef int (make\_request\_fn) (struct request\_queue \*q, struct bio \*bio); 也就是一个函数的指针。  > NOTE: [make\_request\_fn默认设置为blk\_queue\_bio@block/blk-core.c,](mailto:make_request_fn默认设置为blk_queue_bio@block/blk-core.c,) 在blk\_init\_queue/blk\_init\_queue\_node/blk\_init\_allocated\_queue/blk\_queue\_make\_request中设置  如果上面这段话让读者感到莫名其妙，那么请搬个板凳坐下，Let's Begin the Story。  对通用块层的访问，比如请求读某个块设备上的一段数据，通常是准备一个bio，然后调用generic\_make\_request()函数来实现的。  > NOTE: generic\_make\_request@block/blk-core.c 调用者是幸运的，因为他往往不需要去关心generic\_make\_request()函数如何做的，只需要知道这个神奇的函数会为他搞定所有的问题就OK了。 而我们却没有这么幸运，因为对一个块设备驱动的设计者来说，如果不知道generic\_make\_request()函数的内部情况，很可能会让驱动的使用者得不到安全感。  了解generic\_make\_request()内部的有效方法还是RTFSC，但这里会给出一些提示。 我们可以在generic\_make\_request()中找到\_\_generic\_make\_request(bio)这么一句，然后在\_\_generic\_make\_request()函数中找到ret = q->make\_request\_fn(q, bio)这么一行。 偷懒省略掉解开谜题的所有关键步骤后，这里可以得出一个作者相信但读者不一定相信的正确结论： generic\_make\_request()最终是通过调用request\_queue.make\_request\_fn函数完成bio所描述的请求处理的。  Story到此结束，现在我们可以解释刚才为什么列出那段莫名其妙的数据结构的意图了。 对于块设备驱动来说，正是request\_queue.make\_request\_fn函数负责处理这个块设备上的所有请求。 也就是说，只要我们实现了request\_queue.make\_request\_fn，那么块设备驱动的Primary Mission就接近完成了。 在本章中，我们要做的就是： 1：让request\_queue.make\_request\_fn指向我们设计的make\_request函数 2：把我们设计的make\_request函数写出来  如果读者现在已经意气风发地拿起键盘跃跃欲试了，作者一定会假装谦虚地问读者一个问题： 你的钻研精神遇到城管了？ 如果这句话问得读者莫名其妙的话，作者将补充另一个问题： 前两章中明显没有实现make\_request函数，那时的驱动程序倒是如何工作的？ 然后就是清清嗓子自问自答。  前两章确实没有用到make\_request函数，但当我们使用blk\_init\_queue()获得request\_queue时， 万能的系统知道我们搞IT的都低收入，因此救济了我们一个，这就是大名鼎鼎的\_\_make\_request()函数。 request\_queue.make\_request\_fn指向了\_\_make\_request()函数，因此对块设备的所有请求被导向了\_\_make\_request()函数中。  > NOTE: \_\_make\_request已经改成了blk\_queue\_make\_request  \_\_make\_request()函数不是吃素的，马上喊上了他的兄弟，也就是I/O调度器来帮忙，结果就是bio请求被I/O调度器处理了。 同时，\_\_make\_request()自身也没闲着，它把bio这条咸鱼嗅了嗅，舔了舔，然后放到嘴里嚼了嚼，把鱼刺鱼鳞剔掉， 然后情意绵绵地通过do\_request函数(也就是blk\_init\_queue的第一个参数)喂到驱动程序作者的口中。 这就解释了前两章中我们如何通过simp\_blkdev\_do\_request()函数处理块设备请求的。  我们理解\_\_make\_request()函数本意不错，它把bio这条咸鱼嚼成request\_queue喂给do\_request函数，能让我们的到如下好处： 1：request.buffer不在高端内存    这意味着我们不需要考虑映射高端内存到虚存的情况 2：request.buffer的内存是连续的    因此我们不需要考虑request.buffer对应的内存地址是否分成几段的问题 这些好处看起来都很自然，正如某些行政不作为的“有关部门”认为老百姓纳税养他们也自然，但不久我们就会看到不很自然的情况。  如果读者是mm，或许会认为一个摔锅把咸鱼嚼好了含情脉脉地喂过来是一件很浪漫的事情(也希望这位读者与作者联系)，但对于大多数男性IT工作者来说，除非取向问题，否则...... 因此现在我们宁可把\_\_make\_request()函数一脚踢飞，然后自己去嚼bio这条咸鱼。 当然，踢飞\_\_make\_request()函数也意味着摆脱了I/O调度器的处理。  踢飞\_\_make\_request()很容易，使用blk\_alloc\_queue()函数代替blk\_init\_queue()函数来获取request\_queue就行了。 也就是说，我们把原先的 simp\_blkdev\_queue = blk\_init\_queue(simp\_blkdev\_do\_request, NULL); 改成了 simp\_blkdev\_queue = blk\_alloc\_queue(GFP\_KERNEL); 这样。  至于嚼人家口水渣的simp\_blkdev\_do\_request()函数，我们也一并扔掉： 把simp\_blkdev\_do\_request()函数从头到尾删掉。  同时，由于现在要脱光，所以上一章中我们费好大劲换上的那件薄内衣也不需要了， 也就是把上一章中增加的elevator\_init()这部分的函数也删了，也就是删掉如下部分： old\_e = simp\_blkdev\_queue->elevator; if (IS\_ERR\_VALUE(elevator\_init(simp\_blkdev\_queue, "noop")))         printk(KERN\_WARNING "Switch elevator failed, using default\n"); else         elevator\_exit(old\_e);  到这里我们已经成功地让\_\_make\_request()升空了，但要自己嚼bio，还需要添加一些东西： 首先给request\_queue指定我们自己的bio处理函数，这是通过blk\_queue\_make\_request()函数实现的，把这面这行加在blk\_alloc\_queue()之后： blk\_queue\_make\_request(simp\_blkdev\_queue, simp\_blkdev\_make\_request); 然后实现我们自己的simp\_blkdev\_make\_request()函数，然后编译。  如果按照上述的描述修改出的代码让读者感到信心不足，我们在此列出修改过的simp\_blkdev\_init()函数： static int \_\_init simp\_blkdev\_init(void) {         int ret;          simp\_blkdev\_queue = blk\_alloc\_queue(GFP\_KERNEL);         if (!simp\_blkdev\_queue) {                 ret = -ENOMEM;                 goto err\_alloc\_queue;         }         blk\_queue\_make\_request(simp\_blkdev\_queue, simp\_blkdev\_make\_request);          simp\_blkdev\_disk = alloc\_disk(1);         if (!simp\_blkdev\_disk) {                 ret = -ENOMEM;                 goto err\_alloc\_disk;         }          strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);         simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;         simp\_blkdev\_disk->first\_minor = 0;         simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;         simp\_blkdev\_disk->queue = simp\_blkdev\_queue;         set\_capacity(simp\_blkdev\_disk, SIMP\_BLKDEV\_BYTES>>9);         add\_disk(simp\_blkdev\_disk);          return 0;  err\_alloc\_disk:         blk\_cleanup\_queue(simp\_blkdev\_queue); err\_alloc\_queue:         return ret; } 这里还把err\_init\_queue也改成了err\_alloc\_queue，希望读者不要打算就这一点进行提问。  正如本章开头所述，这一章的内容可能要复杂一些，而现在看来似乎已经做到了。 而现在的进度大概是......一半! 不过值得安慰的是，余下的内容只有我们的simp\_blkdev\_make\_request()函数了。  首先给出函数原型： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio); 该函数用来处理一个bio请求。 函数接受struct request\_queue \*q和struct bio \*bio作为参数，与请求有关的信息在bio参数中，而struct request\_queue \*q并没有经过\_\_make\_request()的处理，这也意味着我们不能用前几章那种方式使用q。 因此这里我们关注的是：bio。  关于bio和bio\_vec的格式我们仍然不打算在这里做过多的解释，理由同样是因为我们要避免与google出的一大堆文章撞衫。 这里我们只说一句话： bio对应块设备上一段连续空间的请求，bio中包含的多个bio\_vec用来指出这个请求对应的每段内存。 因此simp\_blkdev\_make\_request()本质上是在一个循环中搞定bio中的每个bio\_vec。  这个神奇的循环是这样的： dsk\_mem = simp\_blkdev\_data + (bio->bi\_sector << 9);  bio\_for\_each\_segment(bvec, bio, i) {         void \*iovec\_mem;          switch (bio\_rw(bio)) {         case READ:         case READA:                 iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                 memcpy(iovec\_mem, dsk\_mem, bvec->bv\_len);                 kunmap(bvec->bv\_page);                 break;         case WRITE:                 iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                 memcpy(dsk\_mem, iovec\_mem, bvec->bv\_len);                 kunmap(bvec->bv\_page);                 break;         default:                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": unknown value of bio\_rw: %lu\n",                         bio\_rw(bio)); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                 bio\_endio(bio, 0, -EIO); #else                 bio\_endio(bio, -EIO); #endif                 return 0;         }         dsk\_mem += bvec->bv\_len; } bio请求的块设备起始扇区和扇区数存储在bio.bi\_sector和bio.bi\_size中， 我们首先通过bio.bi\_sector获得这个bio请求在我们的块设备内存中的起始部分位置，存入dsk\_mem。 然后遍历bio中的每个bio\_vec，这里我们使用了系统提供的bio\_for\_each\_segment宏。  循环中的代码看上去有些眼熟，无非是根据请求的类型作相应的处理。READA意味着预读，精心设计的预读请求可以提高I/O效率， 这有点像内存中的prefetch()，我们同样不在这里做更详细的介绍，因为这本身就能写一整篇文章，对于我们的基于内存的块设备驱动， 只要按照READ请求同样处理就OK了。  在很眼熟的memcpy前后，我们发现了kmap和kunmap这两个新面孔。 这也证明了咸鱼要比烂肉难啃的道理。 bio\_vec中的内存地址是使用page \*描述的，这也意味着内存页面有可能处于高端内存中而无法直接访问。 这种情况下，常规的处理方法是用kmap映射到非线性映射区域进行访问，当然，访问完后要记得把映射的区域还回去，不要仗着你内存大就不还，实际上在i386结构中，你内存越大可用的非线性映射区域越紧张。 关于高端内存的细节也请自行google，反正在我的印象中intel总是有事没事就弄些硬件限制给程序员找麻烦以帮助程序员的就业。 所幸的是逐渐流行的64位机的限制应该不那么容易突破了，至少我这么认为。  switch中的default用来处理其它情况，而我们的处理却很简单，抛出一条错误信息，然后调用bio\_endio()告诉上层这个bio错了。 不过这个万恶的bio\_endio()函数在2.6.24中改了，如果我们的驱动程序是内核的一部分，那么我们只要同步更新调用bio\_endio()的语句就行了， 但现在的情况显然不是，而我们又希望这个驱动程序能够同时适应2.6.24之前和之后的内核，因此这里使用条件编译来比较内核版本。 同时，由于使用到了LINUX\_VERSION\_CODE和KERNEL\_VERSION宏，因此还需要增加#include <linux/version.h>。  循环的最后把这一轮循环中完成处理的字节数加到dsk\_mem中，这样dsk\_mem指向在下一个bio\_vec对应的块设备中的数据。  读者或许开始耐不住性子想这一章怎么还不结束了，是的，马上就结束，不过我们还要在循环的前后加上一丁点： 1：循环之前的变量声明：    struct bio\_vec \*bvec;    int i;    void \*dsk\_mem; 2：循环之前检测访问请求是否超越了块设备限制：    if ((bio->bi\_sector << 9) + bio->bi\_size > SIMP\_BLKDEV\_BYTES) {            printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                    ": bad request: block=%llu, count=%u\n",                    (unsigned long long)bio->bi\_sector, bio->bi\_size);    #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)            bio\_endio(bio, 0, -EIO);    #else            bio\_endio(bio, -EIO);    #endif            return 0;    } 3：循环之后结束这个bio，并返回成功：    #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)    bio\_endio(bio, bio->bi\_size, 0);    #else    bio\_endio(bio, 0);    #endif    return 0;    bio\_endio用于返回这个对bio请求的处理结果，在2.6.24之后的内核中，第一个参数是被处理的bio指针，第二个参数成功时为0，失败时为-ERRNO。    在2.6.24之前的内核中，中间还多了个unsigned int bytes\_done，用于返回搞定了的字节数。  现在可以长长地舒一口气了，我们完工了。 还是附上simp\_blkdev\_make\_request()的完成代码： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         struct bio\_vec \*bvec;         int i;         void \*dsk\_mem;          if ((bio->bi\_sector << 9) + bio->bi\_size > SIMP\_BLKDEV\_BYTES) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                 bio\_endio(bio, 0, -EIO); #else                 bio\_endio(bio, -EIO); #endif                 return 0;         }          dsk\_mem = simp\_blkdev\_data + (bio->bi\_sector << 9);          bio\_for\_each\_segment(bvec, bio, i) {                 void \*iovec\_mem;                  switch (bio\_rw(bio)) {                 case READ:                 case READA:                         iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                         memcpy(iovec\_mem, dsk\_mem, bvec->bv\_len);                         kunmap(bvec->bv\_page);                         break;                 case WRITE:                         iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                         memcpy(dsk\_mem, iovec\_mem, bvec->bv\_len);                         kunmap(bvec->bv\_page);                         break;                 default:                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": unknown value of bio\_rw: %lu\n",                                 bio\_rw(bio)); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                         bio\_endio(bio, 0, -EIO); #else                         bio\_endio(bio, -EIO); #endif                         return 0;                 }                 dsk\_mem += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0; }  读者可以直接用本章的simp\_blkdev\_make\_request()函数替换掉上一章的simp\_blkdev\_do\_request()函数， 然后用本章的simp\_blkdev\_init()函数替换掉上一章的同名函数，再在文件头部增加#include <linux/version.h>， 就得到了本章的最终代码。  在结束本章之前，我们还是试验一下： 首先还是编译和加载： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step3 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step3/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step3/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step3/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # insmod simp\_blkdev.ko # 然后使用上一章中的方法看看sysfs中的这个设备的信息： # ls /sys/block/simp\_blkdev dev  holders  range  removable  size  slaves  stat  subsystem  uevent # 我们发现我们的驱动程序在sysfs目录中的queue子目录不见了。 这并不奇怪，否则就要抓狂了。  本章中我们实现自己的make\_request函数来处理bio，以此摆脱了I/O调度器和通用的\_\_make\_request()对bio的处理。 由于我们的块设备中的数据都是存在于内存中，不牵涉到DMA操作、并且不需要寻道，因此这应该是最适合这种形态的块设备的处理方式。 在linux中类似的驱动程序大多使用了本章中的处理方式，但对大多数基于物理磁盘的块设备驱动来说，使用适合的I/O调度器更能提高性能。 同时，\_\_make\_request()中包含的回弹机制对需要进行DMA操作的块设备驱动来说，也能提供不错帮助。  虽然说量变产生质变，通常质变比量变要复杂得多。 同理，相比前一章，把mm衣服脱光也比让她换一件薄一些的衣服要困难得多。 不过无论如何，我们总算连哄带骗地让mm脱下来了，而付出了满头大汗的代价： 本章内容的复杂度相比前一章大大加深了。  如果本章的内容不幸使读者感觉头部体积有所增加的话，作为弥补，我们将宣布一个好消息： 因为根据惯例，随后的1、2章将会出现一些轻松的内容让读者得到充分休息。 |

# 第4章 磁盘分区

|  |
| --- |
| 上一章结束时说过，本章会准备一些不需要动脑子的内容，现在我们开始履行诺言。  看上去简单的事情实际上往往会被弄得很复杂，比如取消公仆们的招待费用问题； 看上去复杂的事情真正做起来也可能很简单，比如本章中要让我们的块设备支持分区操作。  谈到分区，不懂电脑的人想到了去找“专家”帮忙；电脑入门者想到了“高手”这个名词； 渐入佳境者想到了fdisk；资深级玩家想到了dm；红点玩家想到了隐藏的系统恢复区； 程序员想到了分区表；病毒制造者想到了把分区表清空......  作为块设备驱动程序的设计者，我们似乎需要想的比他们更多一些， 我们大概需要在驱动程序开始识别块设备时访问设备上的分区表，读出里面的数据进行分析，找出这个块设备中包含哪一类的分区(奇怪吧，但真相是分区表确实有很多种，只是我们经常遇到的大概只有ibm类型罢了)、 几个分区，每个分区在块设备上的区域等信息，再在驱动程序中对每个分区进行注册、创建其管理信息...... 读到这里，正在系鞋带准备溜之大吉的同学们请稍等片刻听我说完， 虽然实际上作者也鼓励同学们多作尝试，甚至是这种无谓的尝试，但本章中的做法却比上述的内容简单得多。 因为这一回linux居然帮了我们的忙，并且不是I/O调度器的那种倒忙。  打开linux代码，我们会在fs/partitions/目录中发现一些文件，这些友好的文件将会默默无闻地帮我们的大忙。  而我们需要做的居然如此简单，还记得alloc\_disk()函数吗？ 我们一直用1作参数来调用它的，但现在，我们换成64，这意味着设定块设备最大支持63个分区。 然后......不要问然后，因为已经做完了。 当然，如果要让代码看起来漂亮一些的话，我们可以考虑用一个宏来定义最大分区数。 也就是，在文件的头部增加： /\* usable partitions is SIMP\_BLKDEV\_MAXPARTITIONS - 1 \*/ #define SIMP\_BLKDEV\_MAXPARTITIONS      (64)  然后把 simp\_blkdev\_disk = alloc\_disk(1); 改成 simp\_blkdev\_disk = alloc\_disk(SIMP\_BLKDEV\_MAXPARTITIONS);  好了，真的改好了。 上一章那样改得太多看起来会让读者不爽，那么这里改得太少，是不是也同样不爽？ 大概有关部门深信老百姓接受不了有害物质含量过少的食品，因此制定了食品中三聚氰胺含量的标准。 于是，今后我们大概会制定出一系列标准，比如插入多深才能叫强奸什么的。  为了达到所谓的标准，我们破例补充介绍一下alloc\_disk()函数： 这个函数的原型为： struct gendisk \*alloc\_disk(int minors); 用于申请一个gendisk结构，并做好一些初始化工作。 minors用于指定这个设备使用的次设备号数量，因为第一个次设备号已经用于表示整个块设备了， 因此余下的minors-1个设备号用于表示块设备中的分区，这就限制了这个块设备中的最大可访问分区数。 我们注意“最大可访问分区数”这个词： “最大”虽然指的是上限，但并不意味这是唯一的上限。 极端情况下如果这个块设备只有2个磁道，那么无论minors多大，块设备本身充其量也只能建立2个分区。 这时再谈minors值能到达多少简直就是扯淡，就像腐败不根除，建多少经济适用房都是白搭一样。 “可访问”指的是通过驱动程序可以访问的分区数量，这是因为我们只有那么多次设备号。 但这个数字并不妨碍用户在块设备上面建多少个区。比如我们把minors设定为4，那么最大可访问的分区数量是3， 足够变态的用户完全可以在块设备上建立几十个分区，只不过结果是只能使用前3个分区而已。  现在我们可以试试这个程序了。 与以往相同的是，我们编译和加载这个模块： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step04 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step04/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step04/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step04/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # insmod simp\_blkdev.ko # 与以往不同的是，这一次加载完模块后，我们并不直接在块设备上创建文件系统，而是进行分区： # fdisk /dev/simp\_blkdev Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel Building a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.  Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)  Command (m for help):   关于fdisk我们不打算在这里介绍，因为我们试图让这篇文档看起来专家一些。 使用n命令创建第一个主分区： Command (m for help): n Command action    e   extended    p   primary partition (1-4) p Partition number (1-4): 1 First cylinder (1-2, default 1): 1 Last cylinder or +size or +sizeM or +sizeK (1-2, default 2): 1  Command (m for help): 如果细心一些的话，在这里可以看出一个小麻烦，就是：这块磁盘一共只有2个磁道。 因此，我们只好指定第一个分区仅占用1个磁道。毕竟，还要为第2个分区留一些空间。 然后建立第二个分区： Command (m for help): n Command action    e   extended    p   primary partition (1-4) p Partition number (1-4): 2 First cylinder (2-2, default 2): 2  Command (m for help): 这一步中由于只剩下1个磁道，fdisk便不再问我们Last cylinder，而是自作主张地把最后一个磁道分配给新的分区。 这时我们的分区情况是： Command (m for help): p  Disk /dev/simp\_blkdev: 16 MB, 16777216 bytes 255 heads, 63 sectors/track, 2 cylinders Units = cylinders of 16065 \* 512 = 8225280 bytes             Device Boot      Start         End      Blocks   Id  System /dev/simp\_blkdev1               1           1        8001   83  Linux /dev/simp\_blkdev2               2           2        8032+  83  Linux  Command (m for help): 写入分区，退出fdisk： Command (m for help): w The partition table has been altered!  Calling ioctl() to re-read partition table. Syncing disks. #  然后我们在这两个分区中创建文件系统 # mkfs.ext3 /dev/simp\_blkdev1 mke2fs 1.39 (29-May-2006) Filesystem label= OS type: Linux Block size=1024 (log=0) Fragment size=1024 (log=0) 2000 inodes, 8000 blocks 400 blocks (5.00%) reserved for the super user First data block=1 Maximum filesystem blocks=8388608 1 block group 8192 blocks per group, 8192 fragments per group 2000 inodes per group  Writing inode tables: done Creating journal (1024 blocks): done Writing superblocks and filesystem accounting information: done  This filesystem will be automatically checked every 27 mounts or 180 days, whichever comes first.  Use tune2fs -c or -i to override. # mkfs.ext3 /dev/simp\_blkdev2 mke2fs 1.39 (29-May-2006) Filesystem label= OS type: Linux Block size=1024 (log=0) Fragment size=1024 (log=0) 2008 inodes, 8032 blocks 401 blocks (4.99%) reserved for the super user First data block=1 Maximum filesystem blocks=8388608 1 block group 8192 blocks per group, 8192 fragments per group 2008 inodes per group  Writing inode tables: done Creating journal (1024 blocks): done Writing superblocks and filesystem accounting information: done  This filesystem will be automatically checked every 23 mounts or 180 days, whichever comes first.  Use tune2fs -c or -i to override. # 然后mount设两个设备： # mount /dev/simp\_blkdev1 /mnt/temp1 # mount /dev/simp\_blkdev2 /mnt/temp2 # 看看结果： # mount /dev/hda1 on / type ext3 (rw) proc on /proc type proc (rw) sysfs on /sys type sysfs (rw) devpts on /dev/pts type devpts (rw,gid=5,mode=620) tmpfs on /dev/shm type tmpfs (rw) none on /proc/sys/fs/binfmt\_misc type binfmt\_misc (rw) /dev/simp\_blkdev1 on /mnt/temp1 type ext3 (rw) /dev/simp\_blkdev2 on /mnt/temp2 type ext3 (rw) # 然后读/写： # cp /etc/init.d/\* /mnt/temp1/ # cp /etc/passwd /mnt/temp2 # ls /mnt/temp1/ NetworkManager            avahi-dnsconfd      dund       ipmi        lost+found     netfs     portmap          rpcsvcgssd      vncserver NetworkManagerDispatcher  bluetooth           firstboot  iptables    lvm2-monitor   netplugd  psacct           saslauthd       winbind acpid                     capi                functions  irda        mcstrans       network   rdisc            sendmail        wpa\_supplicant anacron                   conman              gpm        irqbalance  mdmonitor      nfs       readahead\_early  setroubleshoot  xfs apmd                      cpuspeed            haldaemon  isdn        mdmpd          nfslock   readahead\_later  single          ypbind atd                       crond               halt       kdump       messagebus     nscd      restorecond      smartd          yum-updatesd auditd                    cups                hidd       killall     microcode\_ctl  ntpd      rhnsd            sshd autofs                    cups-config-daemon  hplip      krb524      multipathd     pand      rpcgssd          syslog avahi-daemon              dhcdbd              ip6tables  kudzu       netconsole     pcscd     rpcidmapd        vmware-tools # ls /mnt/temp2 lost+found  passwd # 收尾工作： # umount /dev/temp1 # umount /dev/temp2 # rmmod simp\_blkdev #  看起来本章应该结束了，但为了耽误大家更多的时间，我们来回忆一下刚才出现的小麻烦。 我们发现这块磁盘只有2个磁道，由于分区是以磁道为边界的，因此最大只能创建2个分区。 不过谢天谢地，好歹我们能够证明我们的程序是支持“多个”分区的......尽管只有2个。  那么为什么系统会认为我们的块设备只有2个磁道呢？其实这不怪系统，因为我们根本没有告诉系统我们的磁盘究竟有多少个磁道。 因此系统只好去猜、猜、猜，结果就猜成2个磁道了。 好吧，说的细节一些，传统的磁盘使用8个位表示盘面数、6个位表示每磁道扇区数、10个位表示磁道数，因此盘面、每磁道扇区、磁道的最大数值分别为255、63和1023。 这也是传说中启动操作系统时的1024柱面(磁道)和硬盘容量8G限制的根源。 现代磁盘采用线性寻址方式突破了这一限制，从本质上说，如果你的机器还没生锈，那么你的硬盘无论是内部结构还是访问方式都与常识中的盘面、每磁道扇区、磁道无关。 但为了与原先的理解兼容，对于现代磁盘，我们在访问时还是假设它具有传统的结构。目前比较通用的假设是：所有磁盘具有最大数目的(也就是恒定的)盘面和每磁道扇区数，而***磁盘大小与磁道数***成正比。 因此，对于一块80G的硬盘，根据假设，这块磁盘的盘面和每磁道扇区数肯定是255和63，磁道数为：80\*1024\*1024\*1024/512(字节每扇区)/255(盘面数)/63(每磁道扇区数)=10043(小数部分看作不完整的磁道被丢弃)。 话归原题，在驱动程序中我们指定了磁盘大小为16M，共包含16\*1024\*1024/512=32768个扇区。假设这块磁盘具有最大盘面和每磁道扇区数后，它的磁道数就是：32768/255/63=2。  我们看起开应该很happy，因为系统太看得起我们了，竟然把我们的块设备看成现代磁盘进行磁道数的换算处理。 不过我们也可能unhappy，因为这造成磁盘最大只能被分成2个区。(至于为什么分区以磁道作为边界，可以想象一下磁盘的结构) 但我们的磁盘只有区区16M啊，所以最好还是告诉系统我们的磁盘没有那么多的盘面数和每磁道扇区数，这将让磁道数来得多一些。  在下一章中，我们打算搞定这个问题。  **NOTE：实践证明目前磁盘分区划分已经与磁道数无关，是按照sector来划分的。** |

# 第5章 磁盘物理信息

|  |
| --- |
| 既然上一章结束时我们已经预告了本章的内容，那么本章中我们就让这个块设备有能力告知操作系统它的“物理结构”。  当然，对于基于内存的块设备来说，什么样的物理结构并不重要，这就如同从酒吧带mm回家时不需要打听她的姓名一样。但如果不幸遇到的是兼职，并且带她去不入流的招待所时，建议最好还是先串供一下姓名、生日和职业等信息，以便JJ查房时可以伪装成情侣。同样，如果要实现的是真实的物理块设备驱动，那么返回设备的物理结构时大概不能这么随意。  对于块设备驱动程序而言，我们现在需要关注那条目前只有一行的struct block\_device\_operations simp\_blkdev\_fops结构。到目前为止，它存在的目的仅仅是因为它必须存在，但马上我们将发现它存在的另一个目的：**为块设备驱动添加获得块设备物理结构的接口**。  对于具有极强钻研精神的极品读者来说，大概在第一章中就会自己去看struct block\_device\_operations结构，然后将发现这个结构其实还挺复杂： struct block\_device\_operations {         int (\*open) (struct block\_device \*, fmode\_t);         int (\*release) (struct gendisk \*, fmode\_t);         int (\*locked\_ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);         int (\*ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);         int (\*compat\_ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);         int (\*direct\_access) (struct block\_device \*, sector\_t,                                                 void \*\*, unsigned long \*);         int (\*media\_changed) (struct gendisk \*);         int (\*revalidate\_disk) (struct gendisk \*);         int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);         struct module \*owner; };  /\* 下面是3.13中的定义 \*/  struct block\_device\_operations {  int (\*open) (struct block\_device \*, fmode\_t);  void (\*release) (struct gendisk \*, fmode\_t);  int (\*ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);  int (\*compat\_ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);  int (\*direct\_access) (struct block\_device \*, sector\_t,  void \*\*, unsigned long \*);  unsigned int (\*check\_events) (struct gendisk \*disk,  unsigned int clearing);  /\* ->media\_changed() is DEPRECATED, use ->check\_events() instead \*/  int (\*media\_changed) (struct gendisk \*);  void (\*unlock\_native\_capacity) (struct gendisk \*);  int (\*revalidate\_disk) (struct gendisk \*);  int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);  /\* this callback is with swap\_lock and sometimes page table lock held \*/  void (\*swap\_slot\_free\_notify) (struct block\_device \*, unsigned long);  struct module \*owner;  }; 在前几章中，我们邂逅过其中的owner成员变量，它用于存储这个结构的所有者，也就是我们的模块，因此我们做了如下的赋值： .owner                = THIS\_MODULE, 而这一章中，我们将与它的同胞妹妹------getgeo也亲密接触一下。  我们要做的是： 1：在block\_device\_operations中增加getgeo成员变量初值的设定，指向我们的“获得块设备物理结构”函数。 2：实现我们的“获得块设备物理结构”函数。  第一步很简单，我们暂且为“获得块设备物理结构”函数取个名字叫simp\_blkdev\_getgeo()吧，也避免了在下文中把这么一大堆汉字拷来拷去。 在simp\_blkdev\_fops中添加.getgeo指向simp\_blkdev\_getgeo，也就是把simp\_blkdev\_fops结构改成这个样子： struct block\_device\_operations simp\_blkdev\_fops = {         .owner                = THIS\_MODULE,         .getgeo                = simp\_blkdev\_getgeo, };  第二步难一些，但也难不到哪去，在代码中的struct block\_device\_operations simp\_blkdev\_fops这行之前找个空点的场子，把如下函数插进去： static int simp\_blkdev\_getgeo(struct block\_device \*bdev,                 struct hd\_geometry \*geo) {         /\*          \* capacity        heads        sectors        cylinders          \* 0~16M        1        1        0~32768          \* 16M~512M        1        32        1024~32768          \* 512M~16G        32        32        1024~32768          \* 16G~...        255        63        2088~...          \*/         if (SIMP\_BLKDEV\_BYTES < 16 \* 1024 \* 1024) {                 geo->heads = 1;                 geo->sectors = 1;          } else if (SIMP\_BLKDEV\_BYTES < 512 \* 1024 \* 1024) {                 geo->heads = 1;                 geo->sectors = 32;         } else if (SIMP\_BLKDEV\_BYTES < 16ULL \* 1024 \* 1024 \* 1024) {                 geo->heads = 32;                 geo->sectors = 32;         } else {                 geo->heads = 255;                 geo->sectors = 63;         }          geo->cylinders = SIMP\_BLKDEV\_BYTES>>9/geo->heads/geo->sectors;          return 0; } 因为这里我们用到了struct hd\_geometry结构，所以还要增加一行#include <linux/hdreg.h>。  这个函数的目的，是选择适当的物理结构信息装入struct hd\_geometry \*geo结构。 当然，为了克服上一章中只能分成2个区的问题，我们应该尽可能增加磁道的数量。 希望读者不要理解成分几个区就需要几个磁道，这意味着一个磁道一个区，也意味着每个区必须一般大小。 由于分区总是以磁道为边界，尽可能增加磁道的数量不仅仅是为了让块设备容纳更多的分区， 更重要的是让分区的实际大小更接近于分区时的指定值，也就是提高实际做出的分区容量的精度。  不过对于设置的物理结构值，还存在一个限制，就是struct hd\_geometry中的数值上限。 我们看struct hd\_geometry的内容： struct hd\_geometry {         unsigned char heads;         unsigned char sectors;         unsigned short cylinders;         unsigned long start; }; unsigned char的磁头数和每磁道扇区数决定了其255的上限，同样，unsigned short的磁道数决定了其65535的上限。 这还不算，但在前一章中，我们知道对于现代硬盘，磁头数和每磁道扇区数通常取的值是255和63， 再组合上这里的65535的磁道数上限，hd\_geometry能够表示的最大块设备容量是255\*63\*65535\*512/1024/1024/1024=502G。 显然目前linux支持的最大硬盘容量大于502G，那么对于这类块设备，内核是如何通过hd\_geometry结构表示其物理结构的呢？ 诀窍不在内核，而在于用户态程序如fdisk等通过内核调用获得hd\_geometry结构后， 会舍弃hd\_geometry.cylinders内容，取而代之的是直接通过hd\_geometry中的磁头数和每磁道扇区数以及硬盘大小去计算磁道数。 因此对于超过502G的硬盘，由于用户程序得出的磁道数与hd\_geometry.cylinders无关，所以我们往往在fdisk中能看到这块硬盘的磁道数大于65535。  刚才扯远了，现在言归正题，我们决定让这个函数对于任何尺寸的块设备，总是试图返回比较漂亮的物理结构。 漂亮意味着返回的物理结构既要保证拥有足够多的磁道，也要保证磁头数和每磁道扇区数不超过255和63，同时最好使用程序员看起来比较顺眼的数字， 如：1、2、4、8、16、32、64等。 当然，我们也希望找到某个One Shot公式适用于所有大小的块设备，但很遗憾目前作者没找到，因此采用了分段计算的方法： 首先考虑容量很小的块设备：   即使磁头数和每磁道扇区数都是1，磁道数也不够多时，我们会将磁头数和每磁道扇区数都固定为1，以使磁道数尽可能多，以提高分区的精度。   因此磁道数随块设备容量而上升。   虽然我们已经知道了磁道数其实可以超过unsigned short的65535上限，但在这里却没有必要，因此我们要给磁道数设置一个上限。   因为不想让上限超过65535，同时还希望上限也是一个程序员喜欢的数字，因此这里选择了32768。   当然，当磁道数超过32768时，已经意味着块设备容量不那么小了，也就没有必要使用这种情况中如此苛刻的磁头数和每磁道扇区数了。   简单来说，当块设备容量小于1个磁头、每磁道1扇区和32768个磁道对应的容量--也就是16M时，我们将按照这种情况处理。 然后假设块设备容量已经大于16M了：   我们希望保证块设备包含足够多的磁道，这里我们认为1024个磁道应该不少了。   磁道的最小值发生在块设备容量为16M的时候，这时使用1024作为磁道数，可以计算出磁头数\*每磁道扇区数=32。   这里暂且把磁头数和每磁道扇区数固定为1和32，而让磁道数随着块设备容量的增大而增加。   同时，我们还是磁道的上限设置成32768，这时的块设备容量为512M。   总结来说，当块设备容量在16M和512M之间时，我们把磁头数和每磁道扇区数固定为1和32。 然后对于容量大于512M的块设备：   与上述处理相似，当块设备容量在512M和16G之间时，我们把磁头数和每磁道扇区数固定为32和32。 最后的一种情况：   块设备已经足够大了，大到即使我们使用磁头数和每磁道扇区数的上限，也能获得足够多的磁道数。这时把磁头数和每磁道扇区数固定为255和63。至于磁道数就算出多少是多少了，即使超过unsigned short的上限也无所谓，反正用不着。  随着这个函数解说到此结束，我们对代码的修改也结束了。  现在开始试验： 编译和加载： # make make -C /lib/modules/2.6.27.4/build SUBDIRS=/mnt/host\_test/simp\_blkdev/simp\_blkdev\_step05 modules make[1]: Entering directory `/mnt/ltt-kernel'   CC [M]  /mnt/host\_test/simp\_blkdev/simp\_blkdev\_step05/simp\_blkdev.o   Building modules, stage 2.   MODPOST 1 modules   CC      /mnt/host\_test/simp\_blkdev/simp\_blkdev\_step05/simp\_blkdev.mod.o   LD [M]  /mnt/host\_test/simp\_blkdev/simp\_blkdev\_step05/simp\_blkdev.ko make[1]: Leaving directory `/mnt/ltt-kernel' # insmod simp\_blkdev.ko # 用fdisk打开设备文件 # fdisk /dev/simp\_blkdev Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel Building a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.  Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)  Command (m for help): 看看设备的物理结构： Command (m for help): p  Disk /dev/simp\_blkdev: 16 MB, 16777216 bytes 1 heads, 32 sectors/track, 1024 cylinders Units = cylinders of 32 \* 512 = 16384 bytes             Device Boot      Start         End      Blocks   Id  System  Command (m for help): 我们发现，现在的设备有1个磁头、32扇区每磁道、1024个磁道。 这是符合代码中的处理的。  本章的内容也不是太难，连同上一章，我们已经休息2章了。 聪明的读者可能已经猜到作者打算说什么了。 不错，下一章会有一个surprise。 |

# 第6章 动态分配内存

|  |
| --- |
| 经历了内容极为简单的前两章的休息，现在大家一定感到精神百倍了。 作为已经坚持到现在的读者，对接下去将要面临的内容大概应该能够猜得八九不离十了，具体的内容猜不出来也无妨，但一定将是具有增加颅压功效的。  与物理块设备驱动程序的区别在于，我们的驱动程序使用内存来存储块设备中的数据。 到目前为止，我们一直都是使用这样一个静态数组来担负这一功能的： unsigned char simp\_blkdev\_data[SIMP\_BLKDEV\_BYTES];  如果读者懂得一些模块的知识，或者现在赶紧去临时抱佛脚google一些模块知识，应该知道模块其实是加载在**非线性映射区域**的。详细来说，在加载模块时，根据模块的ELF信息(天哪，又要去google elf了)，确定这个模块所需的静态内存大小。这些内存用来容纳模块的二进制代码，以及静态变量。然后申请容纳这些数据的一大堆页面。 当然，这些页面并不是连续的，而代码和变量却不可能神奇到即使被切成一块一块的也能正常工作。因此需要在非线性映射区域中找到一块连续的地址(现在又要去google非线性映射区域了)，用来将刚才申请到的一块一块的内存页映射到这个地址段中。 最后模块被请到这段区域中，然后执行模块的初始化函数......  现在看我们这个模块中的simp\_blkdev\_data变量，如果不是现在刻意关注，这个变量看起来显得那么得普通。 正如其它的一些名字原先也是那么的普通，但由于一些突发的事件受到大家的热烈关注， 比如一段视频让我们熟悉了kappa和陆佳妮，比如呼吸税让我们认识了蒋有绪。  现在我们开始关注simp\_blkdev\_data变量了，导火索是刚才介绍的非线性映射区域。 模块之所以被加载到非线性映射区域，是因为很难在线性映射区域中得到加载模块所需的连续的内存。 但使用非线性映射区域也并非只赚不赔的生意，至少在i386结构中，非线性映射区域实在是太小了。 在物理内存大于896M的i386系统中，整个非线性映射区域不会超过128M。 相反如果物理内存小于896M(不知道该算是幸运还是不幸)，非线性映射区域反而会稍微大一些，这种情况我想我们可以不用讨论了，毕竟不能为了加载一个模块去拔内存。 因此我们的结论是：非线性映射区域是很紧张的资源，我们要节约使用。 而像我们现在这个模块中的simp\_blkdev\_data却是个如假包换的反面典型，居然上来就吃掉了16M！这还是因为我们没有把SIMP\_BLKDEV\_BYTES定义得更大。  现在我们开始列举simp\_blkdev\_data的种种罪行： 1：剩余的非线性映射区域较小时导致模块加载失败 2：模块加载后占用了大量的非线性映射区域，导致其它模块加载失败。 3：模块加载后占用了大量的非线性映射区域，影响系统的正常运行。    这是因为不光模块，系统本身的很多功能也依赖非线性映射区域空间。 对于这样的害群之马，我们难道还有留下他的理由吗？ 本章的内容虽然麻烦一些，但想到能够一了百了地清除这个体大膘肥的simp\_blkdev\_data，倒也相当值得。 也希望今后能够看到在对贪官的处理上，能够也拿出这样的魄力和勇气。  现在在清除simp\_blkdev\_data的问题上，已经不存在什么悬念了，接下来我们需要关注的是将simp\_blkdev\_data碎尸万段后，拿出一个更恰当方法来代替它。 首先，我们决定不用静态声明的数组，而改用动态申请的内存。 其次，使用类似vmalloc()的函数可以动态申请大段内存，但其实这段内存占用的还是非线性映射区域，就好像用一个比较隐蔽的贪官来代替下马的贪官，我们不会愚蠢在这种地步。 剩下的，就是在线性映射区域申请很多个页的内存，然后自己去管理。这个方法一了百了地解决了使用大段非线性映射区域的问题，而唯一的问题是由于需要自己管理申请到的页面，使程序复杂了不少。 但为了整个系统的利益，这难道不是我们该做的吗？  申请一个内存页是很容易的，这里我们将采用所有容易的方法中最容易的那个： \_\_get\_free\_page函数，原型是： unsigned long \_\_get\_free\_page(gfp\_t gfp\_mask); 这个函数用来申请一个页面的内存。gfp\_mask包含一些对申请内存时的指定，比如，要在DMA区域中啦、必须清零等。 我们这里倒是使用最常见的\_\_get\_free\_page(GFP\_KERNEL)就可以了。  通过\_\_get\_free\_page申请到了一大堆内存页，新的问题来了，在读写块设备时，我们得到是块设备的偏移，如何快速地通过偏移找到对应的内存页呢？  最简单的方法是建立一个数组，用来存放偏移到内存的映射，数组中的每项对应一个一个页： 数组定义如下： void \*simp\_blkdev\_data[(SIMP\_BLKDEV\_BYTES + PAGE\_SIZE - 1) / PAGE\_SIZE]; PAGE\_SIZE是系统中每个页的大小，对i386来说，通常是4K，那堆加PAGE\_SIZE减1的代码是考虑到SIMP\_BLKDEV\_BYTES不是PAGE\_SIZE的整数倍时要让末尾的空间也能访问。 然后申请内存的代码大概是： for (i=0; i < (SIMP\_BLKDEV\_BYTES + PAGE\_SIZE - 1) / PAGE\_SIZE; i++) {         p = (void \*)\_\_get\_free\_page(GFP\_KERNEL);         simp\_blkdev\_data[i] = p; } 通过块设备偏移得到内存中的数据地址的代码大概是： mem\_addr = simp\_blkdev\_data[dev\_addr/PAGE\_SIZE] + dev\_addr % PAGE\_SIZE; 这种方法实现起来还是比较简单的，但缺点也不是没有：存放各个页面地址的数组虽然其体积比原先那个直接存放数据的数组已经缩小了很多， 但本身毕竟还是在非线性映射区域中。如果块设备大小为16M，在i386上，需要4096个页面，数组大小16K，这不算太大。 但如果某个疯子打算建立一个2G的虚拟磁盘，数组大小将达到2M，这就不算小了。  或者我们可以不用数组，而用链表来存储偏移到内存页的映射关系，这样可以回避掉数组存在的问题，但在链表中查找指定元素却不是一般的费时， 毕竟我们不希望用户觉得这是个软盘。  接下来作者不打断继续卖关子了，我们最终选择使用的是传说中的基树。 关于linux中基树细节的文档不多，特别是中文文档更少，更糟的是我们这篇文档也不打算作详细的介绍(因为作者建议去RTFSC)。 但总的来说，相对于二叉平衡树的红黑树来说，基树是一个n叉(一般为64叉)非平衡树，n叉减少了搜索的深度，非平衡减少了复杂的平衡操作。 当然，这两个特点也不是仅仅带来优点，但在这里我们就视而不见了，毕竟我们已经选择了基树，因此护短也是自认而然的事情，正如公仆护着王细牛一样。 从功能上来说，基树好像是为我们量身定做的一样，好用至极。 (其实我们也可以考虑选择红黑树和哈希表来实现这个功能，感兴趣的读者可以了解一下)  接下来的代码中，我们将要用到基树种的如下函数：   void INIT\_RADIX\_TREE((struct radix\_tree\_root \*root, gfp\_t mask);   用来初始化一个基树的结构，root是基树结构指针，mask是基树内部申请内存时使用的标志。    int radix\_tree\_insert(struct radix\_tree\_root \*root, unsigned long index, void \*item);   用来往基树中插入一个指针，index是指针的索引，item是指针，将来可以通过index从基树中快速获得这个指针的值。    void \*radix\_tree\_delete(struct radix\_tree\_root \*root, unsigned long index);   用来根据索引从基树中删除一个指针，index是指针的索引。    void \*radix\_tree\_lookup(struct radix\_tree\_root \*root, unsigned long index);   用来根据索引从基树中查找对应的指针，index是指针的索引。 其实基树的功能不仅限于此，比如，还可以给指针设定标志，详情还是请去读linux/lib/radix-tree.c  现在开始改造我们的代码： 首先删除那个无耻的数组： unsigned char simp\_blkdev\_data[SIMP\_BLKDEV\_BYTES]; 然后引入它的替代者--一个基树结构： static struct radix\_tree\_root simp\_blkdev\_data;  然后增加两个函数，用来申请和释放块设备的内存：  申请内存的函数如下： int alloc\_diskmem(void) {         int ret;         int i;         void \*p;          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          for (i = 0; i < (SIMP\_BLKDEV\_BYTES + PAGE\_SIZE - 1) >> PAGE\_SHIFT;                 i++) {                 p = (void \*)\_\_get\_free\_page(GFP\_KERNEL);                 if (!p) {                         ret = -ENOMEM;                         goto err\_alloc;                 }                  ret = radix\_tree\_insert(&simp\_blkdev\_data, i, p);                 if (IS\_ERR\_VALUE(ret))                         goto err\_radix\_tree\_insert;         }         return 0;  err\_radix\_tree\_insert:         free\_page((unsigned long)p); err\_alloc:         free\_diskmem();         return ret; } 先初始化基树结构，然后申请需要的每一个页面，按照每页面的次序作为索引，将指针插入基树。 代码中的“>> PAGE\_SHIFT”与“/ PAGE\_SIZE”作用相同， if (不明白为什么要这样)         do\_google()；  释放内存的函数如下： void free\_diskmem(void) {         int i;         void \*p;          for (i = 0; i < (SIMP\_BLKDEV\_BYTES + PAGE\_SIZE - 1) >> PAGE\_SHIFT;                 i++) {                 p = radix\_tree\_lookup(&simp\_blkdev\_data, i);                 radix\_tree\_delete(&simp\_blkdev\_data, i);                 /\* free NULL is safe \*/                 free\_page((unsigned long)p);         } } 遍历每一个索引，得到页面的指针，释放页面，然后从基树中释放这个指针。 由于alloc\_diskmem()函数在中途失败时需要释放申请过的页面，因此我们把free\_diskmem()函数设计成能够释放建立了一半的基树的形式。 对于只建立了一半的基树而言，有一部分索引对应的指针还没来得及插入基树，对于不存在的索引，radix\_tree\_delete()函数会返回NULL，幸运的是free\_page()函数能够忽略传入的NULL指针。  因为alloc\_diskmem()函数需要调用free\_diskmem()函数，在代码中需要把free\_diskmem()函数写在alloc\_diskmem()前面，或者在文件头添加函数的声明。  然后在模块的初始化和释放函数中添加对alloc\_diskmem()和free\_diskmem()的调用， 也就是改成这个样子： static int \_\_init simp\_blkdev\_init(void) {         int ret;          simp\_blkdev\_queue = blk\_alloc\_queue(GFP\_KERNEL);         if (!simp\_blkdev\_queue) {                 ret = -ENOMEM;                 goto err\_alloc\_queue;         }         blk\_queue\_make\_request(simp\_blkdev\_queue, simp\_blkdev\_make\_request);          simp\_blkdev\_disk = alloc\_disk(SIMP\_BLKDEV\_MAXPARTITIONS);         if (!simp\_blkdev\_disk) {                 ret = -ENOMEM;                 goto err\_alloc\_disk;         }          ret = alloc\_diskmem();         if (IS\_ERR\_VALUE(ret))                 goto err\_alloc\_diskmem;          strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);         simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;         simp\_blkdev\_disk->first\_minor = 0;         simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;         simp\_blkdev\_disk->queue = simp\_blkdev\_queue;         set\_capacity(simp\_blkdev\_disk, SIMP\_BLKDEV\_BYTES>>9);         add\_disk(simp\_blkdev\_disk);          return 0;  err\_alloc\_diskmem:         put\_disk(simp\_blkdev\_disk); err\_alloc\_disk:         blk\_cleanup\_queue(simp\_blkdev\_queue); err\_alloc\_queue:         return ret; }  static void \_\_exit simp\_blkdev\_exit(void) {         del\_gendisk(simp\_blkdev\_disk);         free\_diskmem();         put\_disk(simp\_blkdev\_disk);         blk\_cleanup\_queue(simp\_blkdev\_queue); }  最麻烦的放在最后： 我们需要修改simp\_blkdev\_make\_request()函数，让它适应新的数据结构。  原先的实现中，对于一个bio\_vec，我们找到对应的内存中数据的起点，直接传送bvec->bv\_len个字节就大功告成了，比如，读块设备时就是： memcpy(iovec\_mem, dsk\_mem, bvec->bv\_len); 但现在由于容纳数据的每个页面地址是不连续的，因此可能出现bio\_vec中的数据跨越页面边界的情况。 也就是说，一个bio\_vec中的数据的前半段在一个页面中，后半段在另一个页面中。 虽然这两个页面对应的块设备地址连续，但在内存中的地址不一定连续，因此像原先那样简单使用memcpy看样子是解决不了问题了。  实际上，虽然bio\_vec可能跨越页面边界，但它无论如何也不可能跨越2个以上的页面。 这是因为bio\_vec本身对应的数据最大长度只有一个页面。 因此如果希望做最简单的实现，只要在代码中做一个条件判断就OK了： if (没有跨越页面) {         1个memcpy搞定 } else {         /\* 肯定是跨越2个页面了 \*/         2个memcpy搞定 }  但为了表现出物理设备一次传送1个扇区数据的处理方式(这种情况下一个bio\_vec可能会跨越2个以上的扇区)，我们让代码支持2个以上页面的情况。 首先列出修改后的simp\_blkdev\_make\_request()函数： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         struct bio\_vec \*bvec;         int i;         unsigned long long dsk\_offset;          if ((bio->bi\_sector << 9) + bio->bi\_size > SIMP\_BLKDEV\_BYTES) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                 bio\_endio(bio, 0, -EIO); #else                 bio\_endio(bio, -EIO); #endif                 return 0;         }          dsk\_offset = bio->bi\_sector << 9;          bio\_for\_each\_segment(bvec, bio, i) {                 unsigned int count\_done, count\_current;                 void \*iovec\_mem;                 void \*dsk\_mem;                  iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                  count\_done = 0;                 while (count\_done < bvec->bv\_len) {                         count\_current = min(bvec->bv\_len - count\_done,                                 (unsigned int)(PAGE\_SIZE                                 - ((dsk\_offset + count\_done) & ~PAGE\_MASK)));                          dsk\_mem = radix\_tree\_lookup(&simp\_blkdev\_data,                                 (dsk\_offset + count\_done) >> PAGE\_SHIFT);                         if (!dsk\_mem) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": search memory failed: %llu\n",                                         (dsk\_offset + count\_done)                                         >> PAGE\_SHIFT);                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                         dsk\_mem += (dsk\_offset + count\_done) & ~PAGE\_MASK;                          switch (bio\_rw(bio)) {                         case READ:                         case READA:                                 memcpy(iovec\_mem + count\_done, dsk\_mem,                                         count\_current);                                 break;                         case WRITE:                                 memcpy(dsk\_mem, iovec\_mem + count\_done,                                         count\_current);                                 break;                         default:                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": unknown value of bio\_rw: %lu\n",                                         bio\_rw(bio));                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                         count\_done += count\_current;                 }                  kunmap(bvec->bv\_page);                 dsk\_offset += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0; }  看样子长了一些，但不要被吓着了，因为读的时候我们可以对代码做一些简化： 1：去掉乱七八糟的出错处理 2：无视每行80字符限制 3：把比特运算改成等价但更易读的乘除运算 4：无视碍眼的类型转换 5：假设内核版本大于2.6.24，以去掉判断版本的宏 就会变成这样了： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         struct bio\_vec \*bvec;         int i;         unsigned long long dsk\_offset;          dsk\_offset = bio->bi\_sector \* 512;          bio\_for\_each\_segment(bvec, bio, i) {                 unsigned int count\_done, count\_current;                 void \*iovec\_mem;                 void \*dsk\_mem;                  iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                  count\_done = 0;                 while (count\_done < bvec->bv\_len) {                         count\_current = min(bvec->bv\_len - count\_done, PAGE\_SIZE - (dsk\_offset + count\_done) % PAGE\_SIZE);                          dsk\_mem = radix\_tree\_lookup(&simp\_blkdev\_data, (dsk\_offset + count\_done) / PAGE\_SIZE);                         dsk\_mem += (dsk\_offset + count\_done) % PAGE\_SIZE;                          switch (bio\_rw(bio)) {                         case READ:                         case READA:                                 memcpy(iovec\_mem + count\_done, dsk\_mem, count\_current);                                 break;                         case WRITE:                                 memcpy(dsk\_mem, iovec\_mem + count\_done, count\_current);                                 break;                         }                         count\_done += count\_current;                 }                  kunmap(bvec->bv\_page);                 dsk\_offset += bvec->bv\_len;         }          bio\_endio(bio, 0);         return 0; } 是不是清楚多了？  dsk\_offset用来存储当前要处理的数据在块设备上的偏移，初始值是bio->bi\_sector \* 512，也就是起始扇区对应的偏移，也是第一个bio\_vec对应的块设备偏移。 每处理完成一个bio\_vec时，dsk\_offset值会被更新：dsk\_offset += bvec->bv\_len，以指向将要处理的数据在块设备上的偏移。  在bio\_for\_each\_segment()中代码的起始和末尾，执行kmap和kunmap开映射当前这个bio\_vec的内存，这个知识在前面的章节中已经提到了， 这个处理的结果是iovec\_mem指向当前的bio\_vec中的缓冲区。  现在在kmap和kunmap之间的代码的功能已经很明确了，就是完成块设备上偏移为dsk\_offset、长度为bvec->bv\_len的数据与iovec\_mem地址之间的传送。 假设不考虑bio\_vec跨越页面边界的情况，这段代码应该十分写意： dsk\_mem = radix\_tree\_lookup(&simp\_blkdev\_data, dsk\_offset / PAGE\_SIZE) + dsk\_offset % PAGE\_SIZE; switch (bio\_rw(bio)) { case READ: case READA:         memcpy(iovec\_mem, dsk\_mem, bvec->bv\_len);         break; case WRITE:         memcpy(dsk\_mem, iovec\_mem, bvec->bv\_len);         break; } 首先使用dsk\_offset / PAGE\_SIZE、也就是块设备偏移在内存中数据所位于的页面次序作为索引，查找该页的内存起始地址，然后加上块设备偏移在该页内的偏移、也就是dsk\_offset % PAGE\_SIZE，就得到了内存中数据的地址，然后就是简单的数据传送。 关于块设备偏移到内存地址的转换，我们举个例子： 假使模块加载时我们分配的第1个页面的地址为0xd0000000，用于存放块设备偏移为0~4095的数据 第2个页面的地址为0xd1000000，用于存放块设备偏移为4096~8191的数据 第3个页面的地址为0xc8000000，用于存放块设备偏移为8192~12287的数据 第4个页面的地址为0xe2000000，用于存放块设备偏移为12288~16383的数据 对于块设备偏移为9000的数据，首先通过9000 / PAGE\_SIZE确定它位于第3个页面中， 然后使用radix\_tree\_lookup(&simp\_blkdev\_data, 3)将查找出0xc8000000这个地址。 这是第3个页面的起始地址，这个地址的数据在块设备中的偏移是8192， 因此我们还要加上块设备偏移在页内的偏移量，也就是9000 % PAGE\_SIZE = 808， 得到的才是块设备偏移为9000的数据在内存中的数据地址。  当然，假设终归是假设，往往大多数情况下是自欺欺人的，就好像彩迷总喜欢跟女友说如果中了500万，就要怎么怎么对她好一样。 现在回到残酷的现实，我们还是要去考虑bio\_vec跨越页面边界的情况。 这意味着对于一个bio\_vec，我们将有可能传送多次。 为了记录前几次中已经完成的数据量，我们引入了一个新的变量，叫做count\_done。 在进行bio\_vec内的第一次传送前，count\_done的值是0，每完成一次传送，count\_done将加上这次完成的数据量。 当count\_done == bvec->bv\_len时，就是大功告成的之日。  接着就是边界的判断。 当前块设备偏移所在的内存页中，块设备偏移所在位置到页头的距离为： offset % PAGE\_SIZE 块设备偏移所在位置到页尾的距离为： PAGE\_SIZE - offset % PAGE\_SIZE 这个距离也就是不超越页边界时所能传送的数据的最大值。  因此在bio\_vec内的每一次中，我们使用 count\_current = min(bvec->bv\_len - count\_done, PAGE\_SIZE - (dsk\_offset + count\_done) % PAGE\_SIZE); 来确定这一次传送的数据量。 bvec->bv\_len - count\_done指的是余下需要传送的数据总量， PAGE\_SIZE - (dsk\_offset + count\_done) % PAGE\_SIZE指的是从当前块设备偏移开始、不超越页边界时所能传送的数据的最大值。 如果bvec->bv\_len - count\_done > PAGE\_SIZE - (dsk\_offset + count\_done) % PAGE\_SIZE，说明这一次将传送从当前块设备偏移到其所在内存页的页尾之间的数据， 余下的数据位于后续的页面中，将在接下来的循环中搞定， 如果bvec->bv\_len - count\_done <= PAGE\_SIZE - (dsk\_offset + count\_done) % PAGE\_SIZE，那么可喜可贺，这将是当前bio\_vec的最后一次传送，完成后就可以回家洗澡了。  结合以上的说明，我想应该不难看懂simp\_blkdev\_make\_request()的代码了，而我们的程序也已经大功告成。  现在总结一下修改的位置： 1：把unsigned char simp\_blkdev\_data[SIMP\_BLKDEV\_BYTES];换成static struct radix\_tree\_root simp\_blkdev\_data; 2：把本文中的free\_diskmem()和alloc\_diskmem()函数添加到代码中，虽然没有特别意义，但建议插在紧邻simp\_blkdev\_init()之前的位置。    但有特别意义的是free\_diskmem()和alloc\_diskmem()的顺序，如果读者到这里还打算提问是什么顺序，作者可要哭了。 3：把simp\_blkdev\_make\_request()、simp\_blkdev\_init()和simp\_blkdev\_exit()函数替换成文中的代码。    注意不要企图使用简化过的simp\_blkdev\_make\_request()函数，否则造成的后果：从程序编译失败到读者被若干美女轮奸，作者都概不负责。  从常理分析，在修改完程序后，我们都将试验一次修改的效果。 这次也不例外，因为审判彭宇的王法官也是这么推断的。  首先证明我们的模块至今为止仍然经得起编译、能够加载： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step06 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step06/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step06/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step06/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # insmod simp\_blkdev.ko #  看看模块的加载时分配的非线性映射区域大小： # lsmod Module                  Size  Used by simp\_blkdev             8212  0 ... # 如果这个Size一栏的数字没有引起读者的足够重视的话，我们拿修改前的模块来对比一下： # lsmod Module                  Size  Used by simp\_blkdev         16784392  0 看出区别了没？  如果本章到这里还不结束的话，估计读者要开始闪人了。 好的，我们马上就结束，希望在这之前闪掉的读者不要太多。 由于还没有来得及闪掉而看到这段话的读者们，作者相信你们具有相当的毅力。 学习是需要毅力的，这时的作者同样也需要毅力来坚持完成这本教程。  最后还是希望读者坚持，坚持看完所有的章节，坚持在遇到每一个不明白的问题时都努力寻求答案， 坚持在发现自己感兴趣的内容时能够深入地去了解、探寻、思考。 |

# 第7章 了解伙伴系统，分配多页内存

|  |
| --- |
| 上一章中我们对驱动程序做了很大的修改，单独分配每一页的内存，然后使用基树来进行管理。 这使得驱动程序占用的非线性映射区域大大减少，让它看起来朝优秀的代码又接近了一些。 因为优秀的代码是相似的，糟糕的代码却各有各的糟糕之处。  本章中我们将讨论一些细枝末节的问题，算是对上一章中内容的巩固，也是为后面的章节作一些铺垫。  首先聊一聊低端内存、高端内存和非线性映射区域的问题： 在i386结构中，由于任务使用32位寄存器表示地址，这造成每个任务的最大寻址范围是4G。无论任务对应的是用户程序还是内核代码，都逃脱不了这个限制。  让问题更糟糕的是，普通的linux内核又将4G的地址划分为2个部分，前3G让用户空间程序使用，后1G由内核本身使用。这又将内核实际使用的空间压缩了4倍。 不过linux采用这样的方案倒也不是由于开发者脑瘫，因为这样一来，内核可以与用户进程共用同一个页表，因而在进行用户态和内核态的切换时不必刷新页表，提高了系统的效率。而带来的麻烦就是内核只有1G的地址范围可用。 其实也有一个相当出名的4G+4G的patch，就是采用上述相反的方法，让内核与用户进程使用独立的地址空间，其优缺点也正好与现在的实现相反。但这毕竟不是标准内核的情况，对大多数系统而言，我们不得不接受内核只有1G的地址范围可用的现实。  然后我们再来看内核如何使用这1G的地址范围。 作为内核，当然需要有能力访问到所有的物理内存，而在保护模式下，内存需要通过页表映射到一个虚拟地址上，再进行访问。虽然内核可以在访问任何物理内存时都采用映射->访问->取消映射的方法，但这很可能将任意一台机器彻底变成386的速度。因此，内核一般把尽可能多的物理内存事先映射到它的地址空间中去，这里的“尽可能多”指的是896M。 原因是内核手头只有1G的地址空间，而其中的128M还需要留作非线性映射空间。 这样一来，内核地址空间中的3G~3G+896M便映射了0~896M范围的物理内存。这个映射关系在启动系统时完成，并且在系统启动后不会改变。 物理内存中0~896M的这段空间是幸运的，因为它们在内核空间中有固定的住所，这也使它们能够方便、快速地被访问。相对896M以上的物理内存，它们地址是比较低的，正因为此，我们通常把这部分内存区域叫做低端内存。  但地址高于896M的物理内存就没这么幸运了。由于它们没有在启动时被固定映射到内核空间的地址空间中，我们需要在访问之前对它们进行映射。但映射到哪里呢？幸好内核没有把整个1G的地址空间都用作映射上面所说的低端内存，好歹还留下128M。 其实这128M还是全都能用，在其开头和结尾处还有一些区域拿去干别的事情了(希望读者去详细了解一下)， 所以我们可以用这剩下的接近128M的区域来映射高于896M的物理内存。明显可以看出这时是僧多粥少，所以这部分区域最好应该节约使用。但希望读者不要把访问高于896M的物理内存的问题想得过于严重，因为一般来说，内核会倾向于把这部分内存分配给用户进程使用，而这是不需要占用内核空间地址的。  其实非线性映射区域还有另一个作用，就是用来作连续地址的映射。 内核采用伙伴系统管理内存，这使得内核程序可以一次申请2的n次幂个页面。但如果n比较大时，申请失败的风险也会随之增加。正如桑拿时遇到双胞胎的机会很少、遇到三胞胎的机会更少一样，获得地址连续的空闲页面的机会总是随着连续地址长度的增加而减少。另外，即使能够幸运地得到地址连续的空闲页面，可能产生的浪费问题也是不能回避的。 比如我们需要申请地址连续513K的内存，从伙伴系统中申请时，由于只能选择申请2的n次幂个页面，因此我们不得不去申请1M内存。 不过这两个问题倒是都能够通过使用非线性映射区域来解决。 我们可以从伙伴系统中申请多个小段的内存，然后把它们映射到非线性映射区域中的连续区域中访问。 内核中与此相关的函数有vmalloc、vmap等。  其实80前的作者很羡慕80后和90后的新一代，不仅因为可以在上中学时谈恋爱，还因为随着64位系统的流行，上面这些与32位系统如影随形的问题都将不复存在。关于64位系统中的内存区域问题就留给有兴趣的读者去钻研了。  然后我们再谈谈linux中的伙伴系统。 伙伴系统总是分配出2的n次幂个连续页面，并且首地址以其长度为单位对齐。这增大了将回收的页与其它空白页合并的可能性，也就是减少了内存碎片。  我们的块设备驱动程序需要从伙伴系统中获得所需的内存。目前的做法是每次获得1个页面，也就是分配页面时，把2的n次幂中的n指定为0。这样做的好处是只要系统中存在空闲的页面，不管空闲的页面是否连续，分配总是能成功。但坏处是增加了造就页面碎片的几率。当系统中没有单独的空闲页面时，伙伴系统就不得不把原先连续的空闲页面拆开，再把其中的1个页面返回给我们的程序。同时，在伙伴系统中需要使用额外的内存来管理每一组连续的空闲页面，因此增大页面碎片也意味着需要更多的内存来管理这些碎片。 这还不算，如果系统中的空闲页面都以碎片方式存在，那么真正到了需要分配连续页面的时候，即使存在空闲的内存，也会因为这些内存不连续而导致分配失败。 除了对系统的影响以外，对我们的驱动程序本身而言，由于使用了基树来管理每一段内存，将内存段定义得越短，意味着需要管理更多的段数，也意味着更大的基树结构和更慢的操作。  因此我们打算增加单次从伙伴系统中获得连续内存的长度，比如，每次分配2个、4个、或者8个甚至64个页，来避免上述的问题。每次分配更大的连续页面很明显拥有不少优势，但其劣势也同样明显： 当系统中内存碎片较多时，吃亏的就是咱们的驱动程序了。原本分很多次一点一点去系统讨要，最终可以要到足够的内存，但像现在这样子狮子大开口，却反而要不到了。 还有就是如果系统中原先就存在不少碎片，原先的分配方式倒是可以把碎片都利用起来，而现在这种挑肥捡瘦的分配会同样无视那些更小的不连续页面，反而可能企图去拆散那些更大的连续页面。  折中的做法大概就是选择每次分配一块不大不小的连续的页，暂且我们选择每次分配连续的4个页。 现在开始修改代码： 为简单起见，我们了以下的4个宏： #define SIMP\_BLKDEV\_DATASEGORDER        (2) #define SIMP\_BLKDEV\_DATASEGSHIFT        (PAGE\_SHIFT + SIMP\_BLKDEV\_DATASEGORDER) #define SIMP\_BLKDEV\_DATASEGSIZE                (PAGE\_SIZE << SIMP\_BLKDEV\_DATASEGORDER) #define SIMP\_BLKDEV\_DATASEGMASK                (~(SIMP\_BLKDEV\_DATASEGSIZE-1)) SIMP\_BLKDEV\_DATASEGORDER表示我们从伙伴系统中申请内存时使用的order值，把这个值设置为2时，每次将从伙伴系统中申请连续的4个页面。 我们暂且把这样的连续页面叫做内存段，这样一来，在i386结构中，每个内存段的大小为16K，假设块设备大小还是16M，那么经历了本章的修改后，驱动程序所使用的内存段数量将从原先的4096个减少为现在的1024个。 SIMP\_BLKDEV\_DATASEGSHIFT是在偏移量和内存段之间相互转换时使用的移位值，类似于页面处理中的PAGE\_SHIFT。这里就不做更详细地介绍了，毕竟这不是C语言教程。 SIMP\_BLKDEV\_DATASEGSIZE是以字节为单位的内存段的长度，在i386和SIMP\_BLKDEV\_DATASEGORDER=2时它的值是16384。 SIMP\_BLKDEV\_DATASEGMASK是内存段的屏蔽位，类似于页面处理中的PAGE\_MASK。 其实对于功能而言，我们只需要SIMP\_BLKDEV\_DATASEGORDER和SIMP\_BLKDEV\_DATASEGSIZE就足够了，其它的宏用于快速的乘除和取模等计算。如果读者对此感到有些迷茫的话，建议最好还是搞明白，因为在linux内核的世界中这一类的位操作将随处可见。  然后要改的是申请和释放内存代码。 原先我们使用的是\_\_get\_free\_page()和free\_page()函数，这一对函数用来申请和释放一个页面。 这显然不能满足现在的要求，我们改用它们的大哥：\_\_get\_free\_pages()和free\_pages()。 它们的原型是： unsigned long \_\_get\_free\_pages(gfp\_t gfp\_mask, unsigned int order); void free\_pages(unsigned long addr, unsigned int order); 可以注意到与\_\_get\_free\_page()和free\_page()函数相比，他们多了个order参数，正是用于指定返回2的多少次幂个连续的页。 因此原先的free\_diskmem()和alloc\_diskmem()函数将改成以下这样： void free\_diskmem(void) {         int i;         void \*p;          for (i = 0; i < (SIMP\_BLKDEV\_BYTES + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 p = radix\_tree\_lookup(&simp\_blkdev\_data, i);                 radix\_tree\_delete(&simp\_blkdev\_data, i);                 /\* free NULL is safe \*/                 free\_pages((unsigned long)p, SIMP\_BLKDEV\_DATASEGORDER);         } }  int alloc\_diskmem(void) {         int ret;         int i;         void \*p;          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          for (i = 0; i < (SIMP\_BLKDEV\_BYTES + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 p = (void \*)\_\_get\_free\_pages(GFP\_KERNEL,                         SIMP\_BLKDEV\_DATASEGORDER);                 if (!p) {                         ret = -ENOMEM;                         goto err\_alloc;                 }                  ret = radix\_tree\_insert(&simp\_blkdev\_data, i, p);                 if (IS\_ERR\_VALUE(ret))                         goto err\_radix\_tree\_insert;         }         return 0;  err\_radix\_tree\_insert:         free\_pages((unsigned long)p, SIMP\_BLKDEV\_DATASEGORDER); err\_alloc:         free\_diskmem();         return ret; } 除了用\_\_get\_free\_pages()和free\_pages()代替了原先的\_\_get\_free\_page()和free\_page()函数以外，还使用刚刚定义的那几个宏代替了原先的PAGE宏。 这样一来，所需内存段数的计算方法也完成了修改。  剩下的就是使用内存段的simp\_blkdev\_make\_request()代码。 实际上，我们只要用刚才定义的SIMP\_BLKDEV\_DATASEGSIZE、SIMP\_BLKDEV\_DATASEGMASK和SIMP\_BLKDEV\_DATASEGSHIFT替换原先代码中的PAGE\_SIZE、PAGE\_MASK和PAGE\_SHIFT就大功告成了， 当然，这个结论是作者是经过充分检查和实验后才得出的，希望不要误认为编程时可以大大咧咧地随心所欲。作为程序员，严谨的态度永远都是需要的。  现在，我们的simp\_blkdev\_make\_request()函数变成了这样： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         struct bio\_vec \*bvec;         int i;         unsigned long long dsk\_offset;          if ((bio->bi\_sector << 9) + bio->bi\_size > SIMP\_BLKDEV\_BYTES) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                 bio\_endio(bio, 0, -EIO); #else                 bio\_endio(bio, -EIO); #endif                 return 0;         }          dsk\_offset = bio->bi\_sector << 9;          bio\_for\_each\_segment(bvec, bio, i) {                 unsigned int count\_done, count\_current;                 void \*iovec\_mem;                 void \*dsk\_mem;                  iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                  count\_done = 0;                 while (count\_done < bvec->bv\_len) {                         count\_current = min(bvec->bv\_len - count\_done,                                 (unsigned int)(SIMP\_BLKDEV\_DATASEGSIZE                                 - ((dsk\_offset + count\_done) &                                 ~SIMP\_BLKDEV\_DATASEGMASK)));                          dsk\_mem = radix\_tree\_lookup(&simp\_blkdev\_data,                                 (dsk\_offset + count\_done)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT);                         if (!dsk\_mem) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": search memory failed: %llu\n",                                         (dsk\_offset + count\_done)                                         >> SIMP\_BLKDEV\_DATASEGSHIFT);                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                         dsk\_mem += (dsk\_offset + count\_done)                                 & ~SIMP\_BLKDEV\_DATASEGMASK;                          switch (bio\_rw(bio)) {                         case READ:                         case READA:                                 memcpy(iovec\_mem + count\_done, dsk\_mem,                                         count\_current);                                 break;                         case WRITE:                                 memcpy(dsk\_mem, iovec\_mem + count\_done,                                         count\_current);                                 break;                         default:                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": unknown value of bio\_rw: %lu\n",                                         bio\_rw(bio));                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                         count\_done += count\_current;                 }                  kunmap(bvec->bv\_page);                 dsk\_offset += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0; }  本章的到这里就完成了，接下去我们还是打算试验一下效果。 其实这个实验不太好做，因为linux本身也会随时分配和释放页面，这会影响我们看到的结果。 如果读者看到的现象与预期不同，这也属于预期。  不过为了降低试验受到linux自身活动影响的可能性，建议试验开始之前尽可能关闭系统中的服务、不要同时做其它的操作、不要在xwindows中做。 然后我们开始试验： 先编译模块： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step07 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step07/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step07/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step07/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' #  现在看看伙伴系统的情况： # cat /proc/buddyinfo Node 0, zone      DMA    288     63     34      0      0      0      0      1      1      1      0 Node 0, zone   Normal   9955   1605     24      1      0      1      1      0      0      0      1 Node 0, zone  HighMem   2036    544     13      6      2      1      1      0      0      0      0 #  加载模块后再看看伙伴系统的情况： # insmod simp\_blkdev.ko # cat /proc/buddyinfo Node 0, zone      DMA    337    140      1      1      1      0      0      0      1      0      0 Node 0, zone   Normal  27888   8859     18      0      0      1      0      0      1      0      0 Node 0, zone  HighMem   1583    544     13      6      2      1      1      0      0      0      0 #  释放模块后再看看伙伴系统的情况： # rmmod simp\_blkdev # cat /proc/buddyinfo Node 0, zone      DMA    337    140     35      0      0      0      0      1      1      1      0 Node 0, zone   Normal  27888   8860    632      7      0      1      1      0      0      0      1 Node 0, zone  HighMem   1583    544     13      6      2      1      1      0      0      0      0 #  首先补充说明一下伙伴系统对每种类型的内存区域分别管理，这在伙伴系统中称之为zone。 在i386中，常见的zone有DMA、Normal和HighMem，分别对应0~16M、16~896M和896M以上的物理内存。 DMA zone的特点是老式ISA设备只能使用这段区域进行DMA操作。 Normal zone的特点它被固定映射在内核的地址空间中，我们可以直接使用指针访问这段内存。(不难看出，DMA zone也有这个性质) HighMem zone的特点它没有以上两种zone的特点。 其实我们在上文中讲述的低端内存区域是这里的DMA和Normal zone，而高端内存区域是这里的HighMem zone。  /proc/buddyinfo用于显示伙伴系统的各个zone中剩余的各个order的内存段个数。  我们的模块目前使用低端内存来存储数据，而一般情况下系统会尽可能保留DMA zone的空域内存不被分配出去， 因此我们主要关注/proc/buddyinfo中的Normal行。 行中的各列中的数字表示伙伴系统的这一区域中每个order的剩余内存数量。 比如： Node 0, zone   Normal   9955   1605     24      1      0      1      1      0      0      0      1 这一行表示Normal zone中剩余9955个独立的内存页、1605个连续2个页的内存、24连续4个页的内存等。  由于我们现在每次申请4个页的内存，因此最关注的Normal行的第3列。  首先看模块加载前，Normal行的第3列数字是24，表示系统中剩余24个连续4页的内存区域。 然后我们看模块加载之后的情况，Normal行的第3列从24变为了18，减少了6个连续4页的内存区域。 这说明我们的程序只用掉了6个连续4页的内存区域------明显不可能。 因为作为模块编者，我们很清楚程序需要使用1024个连续4页的内存区域。  继续看这一行的后面，原先处在最末尾的1便成了0。 我们可以数出来最末尾的数字对应order为10的连续页面，也就是连续4M的页面，原来是空闲的，而现在被拆散用掉了。 但即使它被用掉了，也不够我们的的16M空间，数字的分析变得越来越复杂，是坚持下去还是就此停止？ 这一次我们决定停止，因为真相是现在进行的模块加载前后的剩余内存对比确实产生不了什么结论。  详细解释一下，其实我们可以看出在模块加载之前，Normal区域中order>=2的全部空闲内存加起来也不够这个模块使用。 甚至加上DMA区域中order>=2的全部空闲内存也不够。 虽然剩余的order<2的一大堆页面凑起来倒是足够，但谁让我们的模块挑食，只要order=2的页面呢。 因此这时候系统会试图释放出空闲内存。比如：释放一些块设备缓冲页面，或者将用户进程的内存转移到swap中，以获得更多的空闲内存。 很幸运，系统通过释放内存操作拿到了足够的空闲内存使我们的模块得以顺利加载， 但同时由于额外增加出的空闲内存使我们对比模块加载前后的内存差别失去了意义。  其实细心一些的话，刚才的对比中，我们还是能够得到一些结论的，比如， 我们可以注意到模块加载后order为0和1的两个数字的暴增，这就是系统释放页面的证明。 详细来说，系统释放出的页面既包含order<2的，也包含order>=2的，但由于其中order>=2的页面多半被我们的程序拿走了， 这就造成模块加载后的空闲页面中大量出现order<2的页面。  既然我们没有从模块加载前后的空闲内存变化中拿到什么有意义的结论， 我们不妨换条路走，去看看模块释放前后空闲内存的变化情况：  首先还是看Normal区域： order为0和1的页面数目基本没有变化，这容易解释，因为我们释放出的都是order=2的连续页面。 order=2的连续页面从18增加到632，增加了614个。这应该是模块卸载时所释放的内存的一部分。 由于这个模块在卸载时，会释放1024个order=2的连续页面，那么我们还要继续找出模块释放的内存中其他部分的行踪。 也就是1024-614=410个order=2的连续页到哪去了。  回顾上文中的伙伴系统说明，伙伴系统会适时地合并连续页面，那么我们假设一部分模块释放出的页面被合并成更大order的连续页面了。 让我们计算一下order>2的页面的增加情况： order=3的页面增加了7个，order=6的页面增加了1个，order=8的页面减少了1个，order=10的页面增加了1个。 这分别相当于order=2的页面增加14个、增加16、减少64个、增加256个，综合起来就是增加222个。 这就又找到了一部分，剩下的行踪不明的页面还有410-222=188个。  我们继续追查，现在DMA zone区域。 我们的程序所使用的是低端内存，其实也包含0~16M之间的DMA zone。 刚才我们说过，系统会尽可能不把DMA区域的内存分配出去，以保证真正到必须使用这部分内存时，能够拿得出来。 但“尽可能”不代表“绝对不”，如果出现内存不足的情况，DMA zone的空闲内存也很难幸免。 但刚才我们的试验中，已经遇到了Normal区域内存不足情况，这时把DMA zone中的公主们拿去充当Normal zone的军妓也是必然的了。 因此我们继续计算模块释放后DMA区域的内存变化。在DMA区域： order=2的页面增加了34个，order=3的页面减少了1个，order=4的页面减少了1个，order=7的页面增加了1个，order=9的页面增加了1个。 这分别相当于order=2的页面增加34个、减少2、减少4个、增加32个，增加128个，综合起来就是增加188个。  数字刚好吻合，我们就找到了模块释放出的全部页面的行踪。 这也验证了本章中改动的功能符合预期。  然后我们再一次加载和卸载模块，同时查看伙伴系统中空闲内存的变化： # insmod simp\_blkdev.ko # cat /proc/buddyinfo Node 0, zone      DMA    336    141      0      0      0      1      1      0      1      0      0 Node 0, zone   Normal  27781   8866      0      1      0      1      0      0      1      0      0 Node 0, zone  HighMem   1459    544     13      6      2      1      1      0      0      0      0 # # rmmod simp\_blkdev # cat /proc/buddyinfo Node 0, zone      DMA    336    141     35      0      0      0      0      1      1      1      0 Node 0, zone   Normal  27781   8867    633      7      0      1      1      0      0      0      1 Node 0, zone  HighMem   1459    544     13      6      2      1      1      0      0      0      0 #  我们可以发现这一次模块加载前后的内存变化情况与上一轮有些不同，而分析工作就留给有兴趣的读者了。  本章对代码的改动量不大，主要说明一下与我们程序中出现的linux内存管理知识。 其实上一章的改动中已经涉及到了这部分知识，只是因为那时的重点不在这个方面，并且作者也不希望在同一章中加入过多的内容，因此在本章中做个补足。同时，本章中的说明也给后续章节中将要涉及到的内容做个准备，这样读者在将来也可以惬意一些。  不过在开始写这一章时，作者曾反复考虑该不该这样组织本章，正如我们曾经说过的，希望读者在遇到不明白的地方时主动去探索教程之外更多的知识，而不是仅仅读完这个教程本身。 本教程的目的是牵引出通过实现一个块设备驱动程序来牵引出相关的linux的各个知识点，让读者们以此为契机，通过寻求疑问的答案、通过学习更细节的知识来提高自己的能力。 因此教程中对于不少涉及到的知识点仅仅给出简单的介绍，因为读者完全有能力通过google了解更详细的内容， 这也是作者建议的看书方法。 不过本章是个例外，因为作者最终认为对这些知识的介绍对于这部教程的整体性是有帮助的。 但这里的介绍其实仍然只属于皮毛，因此还是希望读者进一步了解教程以外的更多知识。 |

# 第8章 安装时设定磁盘大小

|  |
| --- |
| 本章的目的是让读者继续休息，因此决定仍然搞一些简单的东西。 比如：给我们的驱动程序模块加上模块参数，这样在加载模块时，可以通过参数设定块设备的大小。  给我们模块加参数的工作不难，这牵涉到1个宏： module\_param\_named(name, value, type, perm)         name是参数的名称         value是参数在模块中对应的变量         type是参数的类型         perm是参数的权限 如，在模块中添加 int disk\_size = 1024; module\_param\_named(size, disk\_size, int, S\_IRUGO); 可以给模块加上名称为"size"的参数，如果在加载模块是使用insmod thismodule size=100，那么在模块代码中disk\_size的值就是100。 相反，如果加载模块时没有指定参数，那么模块代码中disk\_size的值仍是默认的1024。 S\_IRUGO指定了这个参数的值在模块加载以后可以被所有人通过/sys/module/[module\_name]/parameters/看到，但无法修改。 好了，有关module\_param\_named就介绍到这里，细节可以google或者看linux/include/linux/moduleparam.h。  然后我们就要给这个模块加个参数，用来在加载时指定块设备的大小。 参数的名字都已经想好了，就叫size吧，类型嘛，32位无符号整数最大能设定到4G，而我们的野心看起来可能更大一些，为了让这个模块支持4G以上的虚拟磁盘(当然是内存足够的情况下)，我们打算使用64位无符号整型。这样能够设定的最大值为16777216T，应该够了吧。  然后我们试图找出module\_param\_named的参数中与unsigned long long对应的type来。 结果是：google了，没找到；看linux/include/linux/moduleparam.h了，还是没找到。 结论是：目前的linux(2.6.28)还不支持unsigned long long类型的模块参数。 更新一些的内核中会不会有是将来的事，尽快搞定这一章的功能却是现在面临的问题。  然后我们就开始找解决方案： 1：给内核打个补丁，看样子不错，但至少今天之类完成不了我们的程序了    并且这样一来，我们的程序只能在今后的内核中运行，而失去对旧版linux的兼容性。 2：指定设置磁盘大小的单位为M。这样可设置的最大的数字就成了4G\*1M，也就是4096T。    这个主意看似不错。而且看样子10年内机器的内存应该到不了这个容量。 3：用字符串来指定大小    这倒是可以解决所有问题，并且我们可以支持16M、1G之类的设定，让我们的程序看起来比较花哨。    缺点应该是我们需要在程序中自己去解析传入的字符串了，幸运的是，实际的解析代码比想象的容易一些。 因此，我们采用第3个方案，向模块中添加一个名称为size、类型为字符串的参数，并且支持解析以K,M,G,T为单位的设定。  第1步：   向程序中添加以下参数申明。   static char \*simp\_blkdev\_param\_size = "16M";   module\_param\_named(size, simp\_blkdev\_param\_size, charp, S\_IRUGO);   char \*simp\_blkdev\_param\_size用于存储设定的磁盘大小，我们把磁盘大小的默认值指定为16M。   目前我们不允许用户在模块加载后改变磁盘大小，将来嘛，有可能增加这一功能，看起来很眩。 第2步：   原来的程序使用   #define SIMP\_BLKDEV\_BYTES      (16\*1024\*1024)   定义磁盘大小，而现在我们不需要这一行了。   同时，我们需要一个unsigned long long变量来存储用户设定的磁盘大小，因此我们增加这个变量：   static unsigned long long simp\_blkdev\_bytes;   然后把程序中所有使用SIMP\_BLKDEV\_BYTES的位置换成使用simp\_blkdev\_bytes变量。 第3步：   在模块加载时对模块参数进行解析，设置simp\_blkdev\_bytes变量的值。   我们增加一个函数进行解析工作： int getparam(void) {         char unit;         char tailc;          if (sscanf(simp\_blkdev\_param\_size, "%llu%c%c", &simp\_blkdev\_bytes,                 &unit, &tailc) != 2) {                 return -EINVAL;         }          if (!simp\_blkdev\_bytes)                 return -EINVAL;          switch (unit) {         case 'g':         case 'G':                 simp\_blkdev\_bytes <<= 30;                 break;         case 'm':         case 'M':                 simp\_blkdev\_bytes <<= 20;                 break;         case 'k':         case 'K':                 simp\_blkdev\_bytes <<= 10;                 break;         case 'b':         case 'B':                 break;         default:                 return -EINVAL;         }          /\* make simp\_blkdev\_bytes fits sector's size \*/         simp\_blkdev\_bytes = (simp\_blkdev\_bytes + (1<<9) - 1) & ~((1ULL<<9) - 1);          return 0; } 然后在simp\_blkdev\_init()中调用这个函数： ret = getparam(); if (IS\_ERR\_VALUE(ret))         goto err\_getparam; 当然，err\_getparam的位置读者应该能猜出来了。  这样一来，工作大概就完成了，让我们看看结果： 使用默认值： # insmod simp\_blkdev.ko # fdisk /dev/simp\_blkdev Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel Building a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.  Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)  Command (m for help): p  Disk /dev/simp\_blkdev: 16 MB, 16777216 bytes 1 heads, 32 sectors/track, 1024 cylinders Units = cylinders of 32 \* 512 = 16384 bytes             Device Boot      Start         End      Blocks   Id  System  Command (m for help): q  # 设定成20M： # rmmod simp\_blkdev # insmod simp\_blkdev.ko size=20M # fdisk /dev/simp\_blkdev Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel Building a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.   The number of cylinders for this disk is set to 1280. There is nothing wrong with that, but this is larger than 1024, and could in certain setups cause problems with: 1) software that runs at boot time (e.g., old versions of LILO) 2) booting and partitioning software from other OSs    (e.g., DOS FDISK, OS/2 FDISK) Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)  Command (m for help): p  Disk /dev/simp\_blkdev: 20 MB, 20971520 bytes 1 heads, 32 sectors/track, 1280 cylinders Units = cylinders of 32 \* 512 = 16384 bytes             Device Boot      Start         End      Blocks   Id  System  Command (m for help): q  # 变态一下，还是设定成20M，但用k作单位： # rmmod simp\_blkdev # insmod simp\_blkdev.ko size=20480k # fdisk /dev/simp\_blkdev Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel Building a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.   The number of cylinders for this disk is set to 1280. There is nothing wrong with that, but this is larger than 1024, and could in certain setups cause problems with: 1) software that runs at boot time (e.g., old versions of LILO) 2) booting and partitioning software from other OSs    (e.g., DOS FDISK, OS/2 FDISK) Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)  Command (m for help): p  Disk /dev/simp\_blkdev: 20 MB, 20971520 bytes 1 heads, 32 sectors/track, 1280 cylinders Units = cylinders of 32 \* 512 = 16384 bytes             Device Boot      Start         End      Blocks   Id  System  Command (m for help): q  #  看样子结果不错。 这一章中基本上没有提到什么比较晦涩的知识，而且看样子通过这一章的学习，大家也应该休息好了。 如果读者现在感觉到精神百倍，那么这一章的目的应该就达到了。 |

# 第9章 数据安全

|  |
| --- |
| 在本章中我们来讨论一下这个驱动程序的数据安全，因为最近的一些事情让作者愈发地感觉到数据泄漏对当事人来说是麻烦的。  我们开门见山的解释一下数据安全问题： 内核常常会向用户态传递数据，而作为内核程序的开发者，我们必须意识到不能把包含意料内容之外的数据随便透露给用户态，因为如果这些数据不巧被别有用心者利用，就会带来不少麻烦。 比如陈冠希就犯了这样的错误。新余市出国考察团也没有在陈冠希身上吸取教训，把单据也不当回事儿。 单据对于考察团而言并不是什么重要的玩意儿，但一旦落到“别有用心”的人手中被加以利用，就不得不当一回事了。 由此我们发现了单据的商业价值。今后在旅游公司干过的员工拿着手头攒到的大量单据，可能会比KIRA更有前途。因此公务员确实属于高风险职业，加薪也是情理当中的了。  对于内核而言，其中的数据也是如此。 即使一些数据对内核而言没有价值，但也不能随意地向用户态传递，因为这段内存中可能不巧包含了不能随意让用户获取的数据，比如用户A使用linux整理他女友的裸照文件，裸照的数据很可能存在于用户A的进程的虚存中，也可能还存在于文件缓存中，A的进程结束后，系统回收了进程的内存，这时内存中的数据被系统认定为无效数据，但系统并没有清空这段数据。A打开的文件的缓存也类似，缓存被系统回收后，内存中的数据并没有被清除。 随后用户B使用了我们的块设备驱动程序。驱动程序初始化时需要获取足够的内存以存储块设备中的数据，系统很可能将用户A使用过的那段包含裸照数据的内存分配给我们的块设备驱动程序。 这时如果用户B老老实实分区、创建文件系统、写入文件，这当然没事，但如果用户B别有用心的上来就直接去读块设备中的数据，那么他可能很幸运的看到不该看的东西。  因此我们咬牙切齿，嫉妒心促使我们修改这个块设备驱动，我们都没遇到的好事儿，也决不允许用户B遇到。 修改的方法很简单，我们申请内存时使用了\_\_get\_free\_pages()函数， 这个函数的第一个参数是gfp\_mask，原先我们传递的是GFP\_KERNEL，表示用于内核中的一般情况。 现在我们只要向gfp\_mask中添加\_\_GFP\_ZERO标志，以提示需要申请清0后的内存。 这样驱动程序加载后，块设备中数据的初始值全为0，这就避免了上文中提到的安全问题。 详细来说，就是把alloc\_diskmem()函数中的 p = (void \*)\_\_get\_free\_pages(GFP\_KERNEL, 这一行改成 p = (void \*)\_\_get\_free\_pages(GFP\_KERNEL | \_\_GFP\_ZERO,  安全方面的改动已经完成了，但为了避免读者认为本章偷工减料，我们再多改一些代码。  块设备中每扇区的数据长度为512字节，我们在驱动程序经常遇到与此相关的转换。为了快速运算，我们经常用到9这个常数，比如：乘以512就是左移9、除以512就是右移9、除以512的余数就是& ((1ULL<<9) - 1)、 向上对齐到512的倍数就是加上(1<<9) - 1再& ~((1ULL<<9) - 1)。  不过现在我们决定通过定义几个宏来吧这些操作写得好看一些。 先定义： #define SIMP\_BLKDEV\_SECTORSHIFT        (9) #define SIMP\_BLKDEV\_SECTORSIZE        (1ULL<<SIMP\_BLKDEV\_SECTORSHIFT) #define SIMP\_BLKDEV\_SECTORMASK        (~(SIMP\_BLKDEV\_SECTORSIZE-1))  然后使用这几个宏来进行扇区相关的转换工作。  详细来说，就是把simp\_blkdev\_make\_request()函数中的： if ((bio->bi\_sector << 9) + bio->bi\_size > simp\_blkdev\_bytes) { 改成 if ((bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT) + bio->bi\_size         > simp\_blkdev\_bytes) {  dsk\_offset = bio->bi\_sector << 9; 改成 dsk\_offset = bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT;  把simp\_blkdev\_getgeo()函数中的： geo->cylinders = simp\_blkdev\_bytes>>9/geo->heads/geo->sectors; 改成 geo->cylinders = simp\_blkdev\_bytes >> SIMP\_BLKDEV\_SECTORSHIFT         / geo->heads / geo->sectors;  把getparam()函数中的： simp\_blkdev\_bytes = (simp\_blkdev\_bytes + (1<<9) - 1) & ~((1ULL<<9) - 1); 改成 simp\_blkdev\_bytes = (simp\_blkdev\_bytes + SIMP\_BLKDEV\_SECTORSIZE - 1)         & SIMP\_BLKDEV\_SECTORMASK;  把simp\_blkdev\_init()函数中的： set\_capacity(simp\_blkdev\_disk, simp\_blkdev\_bytes>>9); 改成 set\_capacity(simp\_blkdev\_disk,         simp\_blkdev\_bytes >> SIMP\_BLKDEV\_SECTORSHIFT);  如果运气不算太背的话，程序应该是能够运行的，让我们试试： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step09 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step09/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step09/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step09/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # insmod simp\_blkdev.ko # 看一看驱动程序刚刚加载时里面的数据： # hexdump /dev/simp\_blkdev -vn512 0000000 0000 0000 0000 0000 0000 0000 0000 0000 0000010 0000 0000 0000 0000 0000 0000 0000 0000 0000020 0000 0000 0000 0000 0000 0000 0000 0000 ........ 00001d0 0000 0000 0000 0000 0000 0000 0000 0000 00001e0 0000 0000 0000 0000 0000 0000 0000 0000 00001f0 0000 0000 0000 0000 0000 0000 0000 0000 0000200 # 对比一下修改前的效果： # hexdump /dev/simp\_blkdev -vn512 0000000 f300 0800 1200 0000 b804 1200 0000 0500 0000010 501a 6930 1806 246a bf0a 7700 256a bf0b ........ 00001e0 1186 296e bf0f 14a6 2a6e bf0f 05c6 ffff 00001f0 00ff 16d6 2a6f bf10 05f6 286d bf0e 0007 0000200 #  本章到此结束，读者是不是感觉我们的教程越来越简单了？ |

# 第10章 分配高端内存

|  |
| --- |
| 如果你的linux系统是x86平台，并且内存大于896M，那么恭喜你，我们大概可以在这个实验中搞坏你的系统。 反之如果你的系统不符合这些条件，也不用为无法搞坏系统而感到失望，本章的内容同样适合你。 这时作者自然也要申明一下对读者产生的任何损失概不负责，因为这年头一不小心就可能差点成了被告，比如南京的彭宇和镇江花山湾的小许姑娘。  在实验看到的情况会因为系统的实际状况不同而稍有区别，但我们需要说明的问题倒是相似的。 但希望读者不要把这种相似理解成了ATM机取款17.5万和贪污2.6亿在判决上的那种相似。  首先我们来看看目前系统的内存状况： # cat /proc/meminfo MemTotal:      1552532 kB MemFree:       1529236 kB Buffers:          2716 kB Cached:          10124 kB SwapCached:          0 kB Active:           8608 kB Inactive:         7664 kB **HighTotal**:      655296 kB 64位系统已经不存在高端内存和低端内存的问题 **HighFree**:       640836 kB **LowTotal**:       897236 kB **LowFree**:        888400 kB SwapTotal:      522104 kB SwapFree:       522104 kB Dirty:              44 kB Writeback:           0 kB AnonPages:        3440 kB Mapped:           3324 kB Slab:             2916 kB SReclaimable:      888 kB SUnreclaim:       2028 kB PageTables:        272 kB NFS\_Unstable:        0 kB Bounce:              0 kB WritebackTmp:        0 kB CommitLimit:   1298368 kB Committed\_AS:    10580 kB VmallocTotal:   114680 kB VmallocUsed:       392 kB VmallocChunk:   114288 kB HugePages\_Total:     0 HugePages\_Free:      0 HugePages\_Rsvd:      0 HugePages\_Surp:      0 Hugepagesize:     4096 kB DirectMap4k:     12288 kB DirectMap4M:    905216 kB # 输出很多，但我们只关心这几行： MemFree:       1529236 kB --这说明系统中有接近1.5G的空闲内存 HighFree:       640836 kB --这说明空闲内存中，处在高端的有600M左右 LowFree:        888400 kB --这说明空闲内存中，处在低端的有800M左右  现在加载上一章完成的模块，我们指定创建800M的块设备： # insmod simp\_blkdev.ko size=800M # 成功了，我们再看看内存状况： # cat /proc/meminfo MemFree:        708812 kB HighFree:       640464 kB LowFree:         68348 kB ... # 我们发现高端内存没怎变，低端内存却已经被耗得差不多了。 我们一不做二不休，继续加大块设备的容量，看看极限能到多少： # rmmod simp\_blkdev # insmod simp\_blkdev.ko size=860M # cat /proc/meminfo MemFree:        651184 kB HighFree:       641972 kB LowFree:          9212 kB ... # 系统居然还没事，这时虽然高端内存还是没怎么变，但低端内存剩下的得已经很可怜了。 然后进一步加大块设备的容量： # rmmod simp\_blkdev # insmod simp\_blkdev.ko size=870M ... 这里不用再cat /proc/meminfo了，因为系统已经完蛋了。 如果有些读者嗜好独特，对出错信息情有独钟的话，在这里也满足一下： kernel: [ 3588.769050] insmod invoked oom-killer: gfp\_mask=0x80d0, order=2, oomkilladj=0 kernel: [ 3588.769516] Pid: 4236, comm: insmod Tainted: G        W 2.6.27.4 #53 kernel: [ 3588.769868]  [<c025e61e>] oom\_kill\_process+0x42/0x183 kernel: [ 3588.771041]  [<c025ea5c>] out\_of\_memory+0x157/0x188 kernel: [ 3588.771306]  [<c0260a5c>] \_\_alloc\_pages\_internal+0x2ab/0x360 kernel: [ 3588.771500]  [<c0260b25>] \_\_get\_free\_pages+0x14/0x24 kernel: [ 3588.771679]  [<f8865204>] alloc\_diskmem+0x45/0xb5 [simp\_blkdev] kernel: [ 3588.771899]  [<f8867054>] simp\_blkdev\_init+0x54/0xc6 [simp\_blkdev] kernel: [ 3588.772217]  [<c0201125>] \_stext+0x3d/0xff kernel: [ 3588.772393]  [<f8867000>] ? simp\_blkdev\_init+0x0/0xc6 [simp\_blkdev] kernel: [ 3588.772599]  [<c0235f2f>] ? \_\_blocking\_notifier\_call\_chain+0x40/0x4c kernel: [ 3588.772845]  [<c0241771>] sys\_init\_module+0x87/0x19d kernel: [ 3588.773250]  [<c02038cd>] sysenter\_do\_call+0x12/0x21 kernel: [ 3588.773884]  ======================= kernel: [ 3588.774237] Mem-Info: kernel: [ 3588.774241] DMA per-cpu: kernel: [ 3588.774404] CPU    0: hi:    0, btch:   1 usd:   0 kernel: [ 3588.774582] Normal per-cpu: kernel: [ 3588.774689] CPU    0: hi:  186, btch:  31 usd:   0 kernel: [ 3588.774870] HighMem per-cpu: kernel: [ 3588.778602] CPU    0: hi:  186, btch:  31 usd:   0 ...  搞坏系统就当是交学费了，但交完学费我们总要学到些东西。 虽然公款出国考察似乎已经斯通见惯，但至少在我们的理解中，学费不是旅游费，更不是家属的旅游费。  我们通过细心观察、周密推理后得出的结论是： 目前的块设备驱动程序会一根筋地使用低端内存，即使系统中低端内存很紧缺的时候，也会直道把系统搞死却不去动半点的高端内存，这未免也太挑食了，因此在本章和接下来的几章中，我们将帮助驱动程序戒掉对低端内存的瘾。  相对高端内存而言，低端内存是比较宝贵的，这是因为它不需要影射就能直接被内核访问的特性。 而内核中的不少功能都直接使用低端内存，以保证访问的速度和简便，但换句话来说，如果低端内存告急，那么系统可能离Panic也不远了。 因此总的来说，对低端内存的使用方法大概应该是：除非有足够理由，否则就别乱占着。 详细来说，就是： 1：不需要使用低端内存的“在内核中不需要映射就能直接访问”这个特性的功能，应该优先使用高端内存    如：分配给用户态进程的内存，和vmalloc的内存 2：需要占用大量内存的功能，并且也可以通过高端内存实现的，应该优先使用高端内存    如：我们的程序  与内存有关的知识我们在以前的章节中已经谈到，因此这里不再重复了，但需要说明的是在高端内存被映射之前，我们是无法通过指针来指向它的。因为它不在内核空间的地址范围以内。  虽然如此，我们却无论如何都需要找出一种方法来指定一个没有被映射的高端内存，这是由于至少在进行映射操作时，我们需要指定去映射谁。这就像为一群猴子取名的时候，如何来说明是正在给哪只猴子取名一样。  虽然给猴子取名的问题可能比较容易解决，比如我们可以说，给哪只红屁股的公猴取名叫齐天大圣、给那只瘦瘦的母猴取名叫白晶晶，但可惜一块高端内存即没有红屁股，又没有胖瘦之分，它们唯一有的就是地址，因此我们也必须通过地址来指定这段高端内存。  刚才说过，在高端内存被映射之前，他在内核的地址空间中是不存在的，但虽然如此，它至少存在其物理地址，而我们正是可以通过它的物理地址来指定它。是的，本质上是这样的，但在linux中，我们还需要再绕那么一丁点： linux在启动阶段为全部物理内存按页为单位建立了的对应的struct page结构，用来管理这些物理内存，也就是，每个页的物理内存，都有着1对1的struct page结构，而这些struct page结构是位于低端内存中的，我们只要使用指向某个struct page结构的指针，就能指定物理内存中的一个页。 因此，对于没有被映射到内核空间中的高端内存，我们可以通过对应的struct page结构来指定它。 (如果读者希望了解更详细的知识，可以考虑从virt\_to\_page函数一路google下去)  我们在这里大肆谈论高端内存的表示方法，因为这是让我们的模块使用高端内存的前提。  我们的驱动程序使用多段内存来存储块设备中的数据。 原先的程序中，我们使用指向这些内存段的指针来指定这些数据的位置，这是没有问题的， 因为当时我们是使用\_\_get\_free\_pages()来申请内存，\_\_get\_free\_pages()函数只能用来申请低端内存， 因为这个函数返回的是申请到的内存的指针，而上文中说过，高端内存是不能用这样的指针表示的。 要申请高端内存，明显不能使用这样的函数，因此我们隆重介绍它的代替者出场： struct page \*alloc\_pages(gfp\_t gfp\_mask, unsigned int order); 这个函数的参数与\_\_get\_free\_pages()相同，但区别在于，它返回指向struct page的指针， 这个我们在上文中介绍过的指针赋予了alloc\_pages()函数申请高端内存的能力。 其实申请一块高端内存并不难，只要使用\_\_GFP\_HIGHMEM参数调用alloc\_pages()函数， 就可能返回一块高端内存，之所以说是“可能”，使因为在某些情况下，比如高端内存不够或不存在时，也会但会低端内存充数。  我们的现在的目标是让驱动程序使用高端内存，这需要： 1：让驱动程序申请高端内存 2：让驱动程序使用高端内存  但在这一章中，我们要做的即不是1，也不是2，而是1之前的准备工作。 因为1和2必须一气呵成地改完，而为了让一气呵成的时候不要再面临其他插曲，我们需要做好充足的准备工作，就像ml前尿尿一样。对应到程序的修改工作上，我们打算先让程序使用struct page \*来指定申请到的内存。  要实现这个目的，我们先要改申请内存的函数，也就是alloc\_diskmem()。刚才我们介绍过alloc\_pages()，现在就要用它了：  首先把函数中定义的 void \*p; 改成 struct page \*page; 因为我们要使用struct page \*来指定申请到的内存，而不是地址了。  然后把 p = (void \*)\_\_get\_free\_pages(GFP\_KERNEL | \_\_GFP\_ZERO, SIMP\_BLKDEV\_DATASEGORDER); 改成 page = alloc\_pages(GFP\_KERNEL | \_\_GFP\_ZERO, SIMP\_BLKDEV\_DATASEGORDER); 这一行改动的原因大概已经说得很详细了。  还有那个if(!p)改成if (!page)  然后就是把指针加入基树的那一行： ret = radix\_tree\_insert(&simp\_blkdev\_data, i, p); 改成 ret = radix\_tree\_insert(&simp\_blkdev\_data, i, page);  由于我们使用了struct page \*来指定申请到的内存，因此错误处理部分也要小改一下： free\_pages((unsigned long)p, SIMP\_BLKDEV\_DATASEGORDER); 改成 \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER); 这里补充介绍一下\_\_free\_pages()函数，可能大家已经猜到其作用了， 其实与我们原先使用的free\_pages()函数相似，都是用来释放一段内存， 但\_\_free\_pages()使用struct page \*来指定要释放的内存，这也意味着它能够用来释放高端内存。  大家应该已经发现我们虽然改用alloc\_pages()函数来申请内存，但并没有指定\_\_GFP\_HIGHMEM参数， 这时申请到的仍然是低端内存，因此避免了在这一章中对访问内存那部分代码的大肆改动。  改动过的alloc\_pages()函数是这样的： int alloc\_diskmem(void) {         int ret;         int i;         struct page \*page;          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          for (i = 0; i < (simp\_blkdev\_bytes + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 page = alloc\_pages(GFP\_KERNEL | \_\_GFP\_ZERO,                         SIMP\_BLKDEV\_DATASEGORDER);                 if (!page) {                         ret = -ENOMEM;                         goto err\_alloc;                 }                  ret = radix\_tree\_insert(&simp\_blkdev\_data, i, page);                 if (IS\_ERR\_VALUE(ret))                         goto err\_radix\_tree\_insert;         }         return 0;  err\_radix\_tree\_insert:         \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER); err\_alloc:         free\_diskmem();         return ret; }  相应的，释放内存用的free\_diskmem()函数也需要一些更改， 为了避免有人说作者唐僧，列出修改后的样子应该已经足够了： void free\_diskmem(void) {         int i;         struct page \*page;          for (i = 0; i < (simp\_blkdev\_bytes + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 page = radix\_tree\_lookup(&simp\_blkdev\_data, i);                 radix\_tree\_delete(&simp\_blkdev\_data, i);                 /\* free NULL is safe \*/                 \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER);         } }  随后是simp\_blkdev\_make\_request()函数：  首先我们不是把void \*dsk\_mem改成struct page \*dsk\_page，而是增加一个 struct page \*dsk\_page; 变量，因为在访问内存时，我们还是需要用到dsk\_mem变量的。  然后是从基数中获取指针的代码，把原先的 dsk\_mem = radix\_tree\_lookup(&simp\_blkdev\_data, (dsk\_offset + count\_done) >> SIMP\_BLKDEV\_DATASEGSHIFT); 改成 dsk\_page = radix\_tree\_lookup(&simp\_blkdev\_data, (dsk\_offset + count\_done) >> SIMP\_BLKDEV\_DATASEGSHIFT); 虽然看起来没什么太大变化，但我们需要知道，这时基树返回的指针已经不是直接指向数据所在的内存了。  还有那个判断是否从基树中获取成功的 if (!dsk\_mem) { 用脚丫子也能想得出应该改成这样： if (!dsk\_page) {  还有就是我们需要首先将struct page \*dsk\_page地址转换成内存的地址后，才能对这块内存进行访问。 这里我们使用了page\_address()函数。 这个函数可以获得struct page数据结构所对应内存的地址。 这时可能有读者要问了，如果这个struct page对应的是高端内存，那么如何返回地址呢？ 实际上，这种情况下如果高端内存中的页面已经被映射到内核的地址空间，那么函数会返回映射到内核空间中的地址，而如果没有映射的话，函数将返回0。 对于我们目前的程序而言，由于使用的是低端内存，因此struct page对应的内存总是处于内核地址空间中的。  对应到代码中，我们需要在使用dsk\_mem之前，也就是 dsk\_mem += (dsk\_offset + count\_done) & ~SIMP\_BLKDEV\_DATASEGMASK; 这条语句之前，让dsk\_mem指向struct page \*dsk\_page对应的内存的实际地址。 这是通过如下代码实现的： dsk\_mem = page\_address(dsk\_page); if (!dsk\_mem) {         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                 ": get page's address failed: %p\n",                 dsk\_page);         kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, 0, -EIO); #else         bio\_endio(bio, -EIO); #endif  总的来说，修改后的simp\_blkdev\_make\_request()函数是这样的： static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         struct bio\_vec \*bvec;         int i;         unsigned long long dsk\_offset;          if ((bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT) + bio->bi\_size                 > simp\_blkdev\_bytes) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                 bio\_endio(bio, 0, -EIO); #else                 bio\_endio(bio, -EIO); #endif                 return 0;         }          dsk\_offset = bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT;          bio\_for\_each\_segment(bvec, bio, i) {                 unsigned int count\_done, count\_current;                 void \*iovec\_mem;                 struct page \*dsk\_page;                 void \*dsk\_mem;                  iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                  count\_done = 0;                 while (count\_done < bvec->bv\_len) {                         count\_current = min(bvec->bv\_len - count\_done,                                 (unsigned int)(SIMP\_BLKDEV\_DATASEGSIZE                                 - ((dsk\_offset + count\_done) &                                 ~SIMP\_BLKDEV\_DATASEGMASK)));                          dsk\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                                 (dsk\_offset + count\_done)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT);                         if (!dsk\_page) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": search memory failed: %llu\n",                                         (dsk\_offset + count\_done)                                         >> SIMP\_BLKDEV\_DATASEGSHIFT);                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                          dsk\_mem = page\_address(dsk\_page);                         if (!dsk\_mem) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": get page's address failed: %p\n",                                         dsk\_page);                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                         }                          dsk\_mem += (dsk\_offset + count\_done)                                 & ~SIMP\_BLKDEV\_DATASEGMASK;                          switch (bio\_rw(bio)) {                         case READ:                         case READA:                                 memcpy(iovec\_mem + count\_done, dsk\_mem,                                         count\_current);                                 break;                         case WRITE:                                 memcpy(dsk\_mem, iovec\_mem + count\_done,                                         count\_current);                                 break;                         default:                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": unknown value of bio\_rw: %lu\n",                                         bio\_rw(bio));                                 kunmap(bvec->bv\_page); #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)                                 bio\_endio(bio, 0, -EIO); #else                                 bio\_endio(bio, -EIO); #endif                                 return 0;                         }                         count\_done += count\_current;                 }                  kunmap(bvec->bv\_page);                 dsk\_offset += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0; }  通过对这3个函数的更改，代码可以使用struct page \*来定位存储块设备数据的内存了。 这也为将来使用高端内存做了一部分准备。  因为本章修改的代码在外部功能上没有发生变动，所以我们就不在这里尝试编译了运行代码了。 不过感兴趣的读者不妨试一试这段代码能不能进行编译和会不会引起死机。 |

# 第11章 使用高端内存前的准备

|  |
| --- |
| 本章中我们仍然为块设备驱动程序使用高端内存做准备工作。 这里要进行的准备工作并不意味着要增加或改变什么功能，而是要收拾一部分代码，因为它们看起来已经有点复杂了。  有编程经验的读者大概能够意识到，编程时最常做的往往不是输入程序，而是拷贝-粘贴。 这是由于我们在编程时可能会不断地发现设计上的问题，或意识到还可以采用更好的结构，然后当然是实现它。 当然，更理想的情况大概是在一开始规划时就确定一个最佳的结构，以避免将来的更改，但事实往往会与理想背道而驰，但关键是我们发现这种苗头时要及时纠正，而不是像某些部门一样去得过且过大事化小来掩盖问题。 要知道，酒是越陈越香，而垃圾却是越捂越臭，如果我们无法在最初做出完美的设计，至少我们还拥有纠正的勇气。  这里读者可能已经感觉到了，这里我们将要修改simp\_blkdev\_make\_request()函数，因为它显得有些大了， 以至于在前几章中对其进行修改时，不得不列出大段的代码来展示修改结果。不过这不是主要原因，相对于缩短函数长度来说，我们分割函数时可能更加在意的是提高代码的可读性。  其实这里分割simp\_blkdev\_make\_request()也是为了将来实现对高端内存的支持，因为访问高端内存无疑将牵涉到页面映射问题，而页面映射的处理又牵涉到了这个函数，因此我们也希望把这部分功能独立出来，以免动戳就改动这个大函数，也可能是为了作者的偏好，因为作者作者哪怕是改动函数中的一个字符，也会把整个函数从头到尾检查一番，以确定这次改动不会产生其他影响，这就解释了作者为什么更加偏爱简单一些的函数了。 当然这种偏好也不一定完全是好事，比如前两天选择液晶电视时，作者就趋向于显示器+机顶盒...  对于一直坚持到这一章的读者而言，应该对simp\_blkdev\_make\_request()函数的功能烂熟于心了， 因此我们直接列出修改后的代码： static int simp\_blkdev\_trans\_oneseg(struct page \*start\_page,                 unsigned long offset, void \*buf, unsigned int len, int dir) {         void \*dsk\_mem;          dsk\_mem = page\_address(start\_page);         if (!dsk\_mem) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": get page's address failed: %p\n", start\_page);                 return -ENOMEM;         }         dsk\_mem += offset;          if (!dir)                 memcpy(buf, dsk\_mem, len);         else                 memcpy(dsk\_mem, buf, len);          return 0; }  static int simp\_blkdev\_trans(unsigned long long dsk\_offset, void \*buf,                 unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_first\_page;         unsigned int this\_off;         unsigned int this\_cnt;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each data segment \*/                 this\_off = (dsk\_offset + done\_cnt) & ~SIMP\_BLKDEV\_DATASEGMASK;                 this\_cnt = min(len - done\_cnt,                         (unsigned int)SIMP\_BLKDEV\_DATASEGSIZE - this\_off);                  this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT);                 if (!this\_first\_page) {                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": search memory failed: %llu\n",                                 (dsk\_offset + done\_cnt)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT);                         return -ENOENT;                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans\_oneseg(this\_first\_page,                         this\_off, buf + done\_cnt, this\_cnt, dir)))                         return -EIO;                  done\_cnt += this\_cnt;         }          return 0; }  static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         int dir;         unsigned long long dsk\_offset;         struct bio\_vec \*bvec;         int i;         void \*iovec\_mem;          switch (bio\_rw(bio)) {         case READ:         case READA:                 dir = 0;                 break;         case WRITE:                 dir = 1;                 break;         default:                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": unknown value of bio\_rw: %lu\n", bio\_rw(bio));                 goto bio\_err;         }          if ((bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT) + bio->bi\_size                 > simp\_blkdev\_bytes) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size);                 goto bio\_err;         }          dsk\_offset = bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT;          bio\_for\_each\_segment(bvec, bio, i) {                 iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                 if (!iovec\_mem) {                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": map iovec page failed: %p\n", bvec->bv\_page);                         goto bio\_err;                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans(dsk\_offset, iovec\_mem,                         bvec->bv\_len, dir)))                         goto bio\_err;                  kunmap(bvec->bv\_page);                  dsk\_offset += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0;  bio\_err: #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, 0, -EIO); #else         bio\_endio(bio, -EIO); #endif         return 0; }  代码在功能上与原先没什么不同，我们只是从中抽象出处理块设备与一段连续内存之间数据传输的simp\_blkdev\_trans()函数，和同样功能的、但数据长度符合块设备数据块长度限制的simp\_blkdev\_trans\_oneseg()函数。  这样一来，程序的结构就比较明显了： simp\_blkdev\_make\_request()负责决定数据传输方向、检查bio请求是否合法、遍历bio中的每个bvec、映射bvec中的内存页，然后把剩余的工作扔给simp\_blkdev\_trans()，而simp\_blkdev\_trans()函数通过分割请求数据搞定了数跨越多个块设备数据块的问题，并且顺便把块设备数据块的第一个page给找了出来，然后邀请simp\_blkdev\_trans\_oneseg()函数出场。 simp\_blkdev\_trans\_oneseg()函数是幸运的，因为前期的大多数铺垫工作已经做完了，而它只要像领导种树一样装模作样的添上最后一铲土，就可以迎来开热烈的掌声。实际上，simp\_blkdev\_trans\_oneseg()拿到page指针对应的内存，然后根据给定的数据方向执行指定长度的数据传输。 simp\_blkdev\_trans\_oneseg()不需要关心数据长度是否超出块设备数据块边界的问题，正如领导也不会去管那棵树的死活一样。  本章的代码也同样不做实验，因为我们确实也没什么好做的。至于能不能通过编译，作者已经试过了，有兴趣的读者大概可以验证一下前一句话是不是真的。  作为支持高端内存的前奏，前一章和本章中做了一些可能让人觉得莫名其妙的改动。不过到此为止，准备工作已经做得差不多了，我们的程序已经为支持高端内存打下坚实的基础。 下一章将进入正题，我们将实现这一期盼已久的功能。 |

# 第12章 使用高端内存

|  |
| --- |
| 本章中我们将实现对高端内存的支持。  女孩子相处时，和她聊天，逛街，爬山，看电影，下棋中的每一件事情好像都与结婚扯不上太大的关系，但经过天天年年的日积月累后，女孩子在潜意识中可能已经把你看成了她生活的一部分，最终的结果显得是那么的自然，甚至连求婚都有些多余了。  学习也很相似，我们认真学习的的每一样知识，努力寻求的每一个答案就其本身而言，都不能让自己成为专家，但专家却无一不是经历了长时间的认真学习，努力钻研和细致思考的结果。  正如我们的程序，经历了前几章中的准备工作，离目标功能的距离大概也不算太远了。而现在我们要做得就是实现它。  首先改动alloc\_diskmem()函数，给这个函数中申请内存的语句、也就是alloc\_pages()的gfp\_mask中加上\_\_GFP\_HIGHMEM标志， 这使得申请块设备的内存块时，会优先考虑使用高端内存。 修改过的函数如下： int alloc\_diskmem(void) {         int ret;         int i;         struct page \*page;          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          for (i = 0; i < (simp\_blkdev\_bytes + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 page = alloc\_pages(GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                         SIMP\_BLKDEV\_DATASEGORDER);                 if (!page) {                         ret = -ENOMEM;                         goto err\_alloc;                 }                  ret = radix\_tree\_insert(&simp\_blkdev\_data, i, page);                 if (IS\_ERR\_VALUE(ret))                         goto err\_radix\_tree\_insert;         }         return 0;  err\_radix\_tree\_insert:         \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER); err\_alloc:         free\_diskmem();         return ret; }  不过事情还没有全部做完，拿到了高端内存，我们还要有能力使用它才行。 这就如同带回一个身材火爆的mm仅仅是个开始，更关键的还在于如何不让人家半小时后怒火冲天摔门而归。 因此我们要继续改造使用内存处的代码，也就是simp\_blkdev\_trans\_oneseg()函数。  在此之前这个函数很简单，由于申请的是低端内存，这就保证了这些内存一直是被映射在内核的地址空间中的。 因此只要使用一个page\_address()函数就完成了page指针到内存指针的转换问题。 但对于高端内存就没有这样简单了。  首先，高端内存需要在进行访问之前被映射到非线性映射区域，还要在访问之后解除这个映射以免人家骂我们的程序像公仆欠白条，我们可以使用kmap()和kunmap()函数解决这个问题。  然后我们还要考虑另一个边界问题，也就是页面边界。 由于我们使用的kmap()函数一次只能映射一个物理页面，当需要访问的数据在块设备的内存块中跨越页面边界时， 我们就需要识别这样的情况，并做出相应的处理，也就是多次调用kmap()和kunmap()函数对依次每个页面进行访问。我们可以采用与先前章节中处理被访问数据跨越多个块设备内存块相似的方法来应对这种情况。  其实对于这种情况，我们还可以选择另一个方案，就是使用vmap()函数。我们可以使用它把地址分散的多个物理页面映射到一段地址连续的区域中，当然对我们正在用作块设备存储空间的这些地址连续的物理页面更没有问题。 但问题在于vmap()函数的内部处理比较复杂，这也意味着vmap()函数需要耗费更多的CPU时间，并且使用vmap()函数时，我们需要一次性映射相当于内存块长度的所有页面，但我们往往不会访问全部的这些页面，这意味着另一方面的性能损失。因此，我们决定选择使用kmap()函数，而让程序自己去处理跨页面的访问问题。  参照以上的思路，我们写出了新的simp\_blkdev\_trans\_oneseg()函数： static int simp\_blkdev\_trans\_oneseg(struct page \*start\_page,                 unsigned long offset, void \*buf, unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_page;         unsigned int this\_off;         unsigned int this\_cnt;         void \*dsk\_mem;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each page \*/                 this\_page = start\_page + ((offset + done\_cnt) >> PAGE\_SHIFT);                 this\_off = (offset + done\_cnt) & ~PAGE\_MASK;                 this\_cnt = min(len - done\_cnt, (unsigned int)PAGE\_SIZE                         - this\_off);                  dsk\_mem = kmap(this\_page);                 if (!dsk\_mem) {                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": map device page failed: %p\n", this\_page);                         return -ENOMEM;                 }                 dsk\_mem += this\_off;                  if (!dir)                         memcpy(buf + done\_cnt, dsk\_mem, this\_cnt);                 else                         memcpy(dsk\_mem, buf + done\_cnt, this\_cnt);                  kunmap(this\_page);                  done\_cnt += this\_cnt;         }          return 0; }  其核心是使用kmap()函数将内存页面映射到内核空间然后再进行访问，以实现对高端内存的操作。  到此为止，经历了若干章的问题就这样被解决了。 通过这样的改变，我们至少得到了两个好处： 1：避免了争抢宝贵的低端内存    作为内存消耗大户，霸占低端内存的行为不可容忍，其理由我们在前些章节中已经论述过。    今后我们的程序至少不会在这一方面被人鄙视了。 2：增加了块设备的最大容量    使用原先的程序，在i386中无论如何也无法建立容量超过896M的块设备，实际上更小，这是由于低端内存不可能全部拿来放块设备的数据，而现在的程序可以使用包括高端内存在内的所有空闲内存，这无疑大大增加了块设备的最大容量。  前些章中没有进行的试验憋到现在终于可以开始了。  首先证明这个程序经过了这么多个章节的折腾后仍然是能编译的： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step12 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step12/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step12/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step12/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' #  然后瞧瞧目前的内存状况： # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:       509320 kB LowTotal:       896356 kB LowFree:        872612 kB ... # 我们看到高端内存与低端内存分别剩余509M和872M。  然后加载现在的模块，为了让模块吃内存的行为表现得更加显眼一些， 我们使用size参数指定了更大的块设备容量： # insmod simp\_blkdev.ko size=500M #  现在看看内存的变化情况： # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:         1652 kB LowTotal:       896356 kB LowFree:        863696 kB ... # 结果显示模块如我们所料的吃掉了500M左右的高端内存。 虽然低端内存看样子也少了一些，我们却不能用模块本身占用的内存空间来解释这一现象，因为模块的代码和静态数据占用的内存无论如何也到不了8.9M，或许我们解释为用作一些文件操作的缓存了，还有就是基树结构占用的内存，这个结构占用的内存会随着块设备容量的增大而增加，或者我们可以计算一下...... 不过现在我们并不打算对这个小问题做过多的关注，因为这是扯淡，正如闹得沸沸扬扬的周久耕事件的最后调查结果居然仅仅只是公款买烟。因此我们不会纠缠在这8.9M的问题中，因为很明显大头是在减少的500多兆高端内存上，这减少的500M高端内存已经足以证明这几章中的修改结果了。  我们再移除这个模块后看看内存的状况： # rmmod simp\_blkdev # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:       504684 kB LowTotal:       896356 kB LowFree:        868480 kB ... # 刚才被占用的高端内存又回来了，一切都显得如此的和谐。  作为最后一步的测试，我们做一件本章之前做不到的事情，就是申请大于896M的内存。 刚才我们看到剩余的低端内存和高端内存总共达到了1.37G，好吧，我们就申请1.3G： # insmod simp\_blkdev.ko size=1300M # 这时我们惊喜地发现系统没有DOWN掉。  再看看这时的内存情况： # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        41204 kB LowTotal:       896356 kB LowFree:         48284 kB ... # 高端内存与低端内存中的大头基本上都被吃掉了，数量上也差不多是1.3G，这符合我们的预期。  老让模块占用着这么多的内存也不是什么好主意， 我们放掉： # rmmod simp\_blkdev. #  随着本章的结束，围绕高端内存的讨论也终于修成正果了。 不过我们对这个驱动程序的改进还没有完，因为我们要发扬做精每一样事情的精神，一个民族的振兴，不是靠对小学生进行填鸭式的政治思想教育，也不是靠官员及家属的出国考察，更不是靠公仆们身先士卒、前仆后继、以自己的健康为代价大吃大喝以创造9000亿的GDP，而是靠每一个屁民们的诚实、认真、勤劳、勇敢、创造、奉献与精益求精。 |

# 第13章 代码优化-free\_diskmem

|  |
| --- |
| 没有最好的代码，是因为我们总能把代码改得更好。 因此我们现在打算做一个小的性能改进，这次我们准备拿free\_diskmem()函数下刀。  本质上说，这个改进的意义不大，这是因为free\_diskmem()函数仅仅是在模块卸载时被调用，而对这种执行次数即少又不在关键路径上的函数来说，最好是尽量让他简单以增加可靠性和可读性，除非它的耗时已经慢到能让人有所感觉，否则0.01秒和0.00001秒是差不多的，毕竟在现实中尼奥不太可能用我们的程序。  但我们仍然打算继续这一改进，一是为了示范什么是没有意义的改进，二是为了通过这一改进示范使用radix\_tree\_gang\_lookup()函数和page->index的技巧。  首先我们看看原先的free\_diskmem()函数： void free\_diskmem(void) {         int i;         struct page \*page;          for (i = 0; i < (simp\_blkdev\_bytes + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 page = radix\_tree\_lookup(&simp\_blkdev\_data, i);                 radix\_tree\_delete(&simp\_blkdev\_data, i);                 /\* free NULL is safe \*/                 \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER);         } }  它遍历所有的内存块索引，在基树中找到这个内存块的page指针，然后释放内存，顺带着释放掉基数中的这个节点。 考虑到这个函数不仅会在模块卸载时被调用，也会在模块加载时、申请内存中途掉链子时用来擦屁股，因此也需要考虑内存没有完全申请的情况。 所幸的是这种情况下radix\_tree\_lookup()函数会返回NULL指针，而radix\_tree\_delete()和\_\_free\_pages()函数都能对NULL指针做出我们最期待的处理：就是什么也不做。  这段代码很小很直接，逻辑简单而清晰，性能也差不到哪里去，完全符合设计要求， 不幸的是我们还是打算做一些没必要的优化，借此还可以顺便读一读基树的内核代码。 首先看radix\_tree\_lookup()函数，它在基数中查找指定索引对应的指针，为了获得这一指针的值，基本上它需要把基树从上到下找一遍。 而对于free\_diskmem()函数而言，我们仅仅是需要遍历基树中的所有节点，使用逐一查找的方法进行遍历未免代价太大了。 就像是我们要给在场的所有同学每人发一个糖果，只需要让他们排好队，每人领一个即可，而不需要按照名单找出每个人再发。 为了实现这一思想，我们跑到linux/lib/radix-tree.c中找函数，找啊找，找到了radix\_tree\_gang\_lookup()函数。  radix\_tree\_gang\_lookup()函数虽然不是我们理想中的遍历函数，但也有了八九不离十的功能。 就像在酒吧里找不到D Cup，带回去个C Cup也总比看A片强。 通过radix\_tree\_gang\_lookup()函数，我们可以一次从基树中获取多个节点的信息： unsigned int radix\_tree\_gang\_lookup(struct radix\_tree\_root \*root, void \*\*results,  unsigned long first\_index, unsigned int max\_items); 具体的参数嘛，RTFSC吧。  这是我们注意到使用这个函数时顾此失彼的一面，虽然我们获得了一组需要释放的指针，但却无法获得这些指针的索引。而执行释放基树中节点的操作时却恰恰需要使用索引作参数。 然后就是一个技巧了，我们借用page结构的index成员来存储这一索引。 之所以可以这样用，是因为page结构的index成员在该页用作页高速缓存时存储相对文件起始处的以页大小为单位的偏移，而我们所使用的页面不会被同时用作页高速缓存，因此这里可以借用page.index成员。 按照以上思路，我们写出了修改后的代码： void free\_diskmem(void) {         unsigned long long next\_seg;         struct page \*seglist[64];         int listcnt;         int i;          next\_seg = 0;         do {                 listcnt = radix\_tree\_gang\_lookup(&simp\_blkdev\_data,                         (void \*\*)seglist, next\_seg, ARRAY\_SIZE(seglist));                  for (i = 0; i < listcnt; i++) {                         next\_seg = seglist[i]->index;                         radix\_tree\_delete(&simp\_blkdev\_data, next\_seg);                         \_\_free\_pages(seglist[i], SIMP\_BLKDEV\_DATASEGORDER);                 }                  next\_seg++;         } while (listcnt == ARRAY\_SIZE(seglist)); }  当然，alloc\_diskmem()函数中也需要加上page->index = i这一行，用于把基树的索引存入page.index，修改后的代码如下： int alloc\_diskmem(void) {         int ret;         int i;         struct page \*page;          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          for (i = 0; i < (simp\_blkdev\_bytes + SIMP\_BLKDEV\_DATASEGSIZE - 1)                 >> SIMP\_BLKDEV\_DATASEGSHIFT; i++) {                 page = alloc\_pages(GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                         SIMP\_BLKDEV\_DATASEGORDER);                 if (!page) {                         ret = -ENOMEM;                         goto err\_alloc;                 }                  page->index = i;                 ret = radix\_tree\_insert(&simp\_blkdev\_data, i, page);                 if (IS\_ERR\_VALUE(ret))                         goto err\_radix\_tree\_insert;         }         return 0;  err\_radix\_tree\_insert:         \_\_free\_pages(page, SIMP\_BLKDEV\_DATASEGORDER); err\_alloc:         free\_diskmem();         return ret; }  现在试验一下修改后的代码，先看看能不能编译： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step13 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step13/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step13/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step13/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' #  看看当前系统的内存情况： # cat /proc/meminfo HighTotal:     1146816 kB HighFree:       339144 kB LowTotal:       896356 kB LowFree:        630920 kB ... # 这里显示现在剩余339M高端内存和630M低端内存。  然后加载我们的模块，让它吃掉300M内存： # insmod simp\_blkdev.ko size=300M # cat /proc/meminfo HighTotal:     1146816 kB HighFree:       137964 kB LowTotal:       896356 kB LowFree:        523900 kB ... # 正如我们的预期，剩余内存减少300M左右。  然后看看卸载模块后的内存情况： # rmmod simp\_blkdev # cat /proc/meminfo HighTotal:     1146816 kB HighFree:       338028 kB LowTotal:       896356 kB LowFree:        631044 kB ... # 我们发现剩余内存增加了300M，这意味着模块已经把吃掉的内存吐回来了， 从而可以推断出我们修改过的free\_diskmem()函数基本上是能够工作的。  本章的改动不大，就算是暂作休整，以留住忍耐至今忍无可忍认为无需再忍而开始打包收拾行李准备溜之大吉的读者们。 不过下一章中倒是预备了一个做起来让人比较有成就感的功能。 |

# 第14章 延迟分配内存

|  |
| --- |
| 在本章中我们要做一个比较大的改进，就是实现内存的推迟分配。  这意味着我们并不是在驱动程序加载时就分配用于容纳数据的全部内存，而是推迟到真正需要用到某块内存时再进行分配。  详细来说，我们将在块设备的某个区域上发生第一次写请求时分配用于容纳被写入数据的内存，如果读者在之前章节的熏陶下养成了细致的作风和勤于思考的习惯，应该能发现这里提到的分配内存的时机是第一次写，而不是第一次读写。 现在可能有些读者已经悟出了这样做的道理，让我们无视他们，依然解释一下这样做的目的。 对块设备而言，只要保证读出的数据是最近一次写进的即可。 如果在读数据之前从来没有往块设备的同一块区域中写入数据，那么这时返回任何随机数据都是正确的。 这意味着对于第一次读，我们完全可以返回任意的数据给用户，这时并不需要分配某段内存来存储它。 对真实的物理设备而言，就像我们买回的新硬盘，出厂时盘片中的数据内容是什么都无所谓。 在具体的实现中，我们可以不对用以接收被读出数据的内存进行任何填充，直接告诉上层“已经读好了”， 这样做无疑会更加快速，但这会造成2个问题： 1：这块内存原先的内容最终将被传送到用户程序中，这将造成数据安全问题 2：违背了真实设备的一个潜特性，就是即使这个设备没有写入任何内容，对同一区域的多次读操作返回的内容相同。 因此，我们将向接收数据的内存中写些什么，最简单的就是用全0填充了。  实现这一功能的优点在于，块设备不需要在一开始加载时就占用全部的内存，这优化了系统资源的使用率。 让我们假设块设备自始至终没有被全部填满时，通过本章的功能，将占用更少的内存。 另外，我们甚至可以创建容量远远大于机器物理内存的块设备，只要在随后的使用中不往这个块设备中写入过多的内容即可。  在linux中，类似的思想被广泛应用。 比如对进程的内存区而言，并不是一开始就为这段内存区申请和映射全部需要的物理内存，又如在不少文件系统中，也不会给没有写入内容的文件部分分配磁盘的。  现在我们就实现这一功能。 分析代码，我们发现不太容易找到往什么地方加代码。 往往在这种情况下，不如首先看看可以剥掉哪部分不需要的代码，正如初次跟一个mm时，如果两个人都有些害羞，不知道从哪开始、或者正在期待对方打开局面时，不如先脱下该脱的东西，然后的事情基本上就比较自然了。  现在的代码中，明显可以砍掉的是在驱动程序加载时用于申请容纳数据的内存的代码，也就是alloc\_diskmem()函数，把它砍了，没错，是全砍了。  还有调用它的代码，在simp\_blkdev\_init()函数里面的这几行： ret = alloc\_diskmem(); if (IS\_ERR\_VALUE(ret))         goto err\_alloc\_diskmem; 是的，也砍了。  还没完，既然这个函数的调用都没了，那么调用这个函数失败时的出错处理也没用了，也就是： err\_alloc\_diskmem:         put\_disk(simp\_blkdev\_disk); 这两句，不用犹豫了，砍掉。  经过刚才的大刀阔斧后，我们发现......刚才由于砍上瘾了，不小心多砍了一条语句，就是对基树的初始化语句： INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL); 原来它是在alloc\_diskmem()函数里面的，现在alloc\_diskmem()函数不在了，我们索性把它放到初始化模块的simp\_blkdev\_init()函数中， 放到刚才原来调用alloc\_diskmem()函数的位置就行了。 (注： 其实这里不添加INIT\_RADIX\_TREE()宏也行，直接在定义基树结构时顺便初始化掉就行了，也就是把 static struct radix\_tree\_root simp\_blkdev\_data; 改成 static struct radix\_tree\_root simp\_blkdev\_data = RADIX\_TREE\_INIT(GFP\_KERNEL); 就行了，或者改成让人更加撞墙的形式： static RADIX\_TREE(simp\_blkdev\_data, GFP\_KERNEL); 也可以，但我们这里的代码中，依然沿用原先的方式。 )  这样一来，simp\_blkdev\_init()函数变成了这个样子： static int \_\_init simp\_blkdev\_init(void) {         int ret;          ret = getparam();         if (IS\_ERR\_VALUE(ret))                 goto err\_getparam;          simp\_blkdev\_queue = blk\_alloc\_queue(GFP\_KERNEL);         if (!simp\_blkdev\_queue) {                 ret = -ENOMEM;                 goto err\_alloc\_queue;         }         blk\_queue\_make\_request(simp\_blkdev\_queue, simp\_blkdev\_make\_request);          simp\_blkdev\_disk = alloc\_disk(SIMP\_BLKDEV\_MAXPARTITIONS);         if (!simp\_blkdev\_disk) {                 ret = -ENOMEM;                 goto err\_alloc\_disk;         }          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);         simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;         simp\_blkdev\_disk->first\_minor = 0;         simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;         simp\_blkdev\_disk->queue = simp\_blkdev\_queue;         set\_capacity(simp\_blkdev\_disk,                 simp\_blkdev\_bytes >> SIMP\_BLKDEV\_SECTORSHIFT);         add\_disk(simp\_blkdev\_disk);          return 0;  err\_alloc\_disk:         blk\_cleanup\_queue(simp\_blkdev\_queue); err\_alloc\_queue: err\_getparam:         return ret; }  淋漓尽致地大砍一番之后，我们发现下一步的工作清晰多了。 现在在模块加载时，已经不会申请所需的内存，而我们需要做的就是，在处理块设备读写操作时，添加不存在相应内存时的处理代码。  在程序中，查找基数中的一个内存块是在simp\_blkdev\_trans()函数内完成的，目前的处理是： this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT); if (!this\_first\_page) {         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                 ": search memory failed: %llu\n",                 (dsk\_offset + done\_cnt)                 >> SIMP\_BLKDEV\_DATASEGSHIFT);         return -ENOENT; } 也就是找不到内存块时直接看作错误。 在以前这是正确的，因为所有的内存块都在初始化驱动程序时申请了，因此除非电脑的脑子进水了，运行错了指令，或者人脑的脑子进水了，编错了代码，否则不会发生这种情况。  但现在情况不同了，这时找不到内存块是正常的，这意味着该位置的数据从未被写入过，因此我们需要在这里做出合理的动作。也就是在本章开始时所说的，对于读处理返回全0，对于写处理给块设备的这段空间申请内存，并写入数据。因此我们把上段代码改成了这个样子： this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT); if (!this\_first\_page) {         if (!dir) {                 memset(buf + done\_cnt, 0, this\_cnt);                 goto trans\_done;         }          /\* prepare new memory segment for write \*/         this\_first\_page = alloc\_pages(                 GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                 SIMP\_BLKDEV\_DATASEGORDER);         if (!this\_first\_page) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": allocate page failed\n");                 return -ENOMEM;         }          this\_first\_page->index = (dsk\_offset + done\_cnt)                 >> SIMP\_BLKDEV\_DATASEGSHIFT;          if (IS\_ERR\_VALUE(radix\_tree\_insert(&simp\_blkdev\_data,                 this\_first\_page->index, this\_first\_page))) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": insert page to radix\_tree failed"                         " seg=%lu\n", this\_first\_page->index);                 \_\_free\_pages(this\_first\_page,                         SIMP\_BLKDEV\_DATASEGORDER);                 return -EIO;         } } 对这段代码的流程几乎不要解释了，因为代码本身就是最好的说明。 唯一要提一下的就是goto trans\_done这句话，因为前一条语句实质上已经完成了数据读取， 因此需要直接跳转到该段数据处理完成的位置，也就是函数中的done\_cnt += this\_cnt语句之前。 说到这里猴急的读者可能已经在done\_cnt += this\_cnt语句之前添加 trans\_done: 这一行了，不错，正是要加这一行。  改过的simp\_blkdev\_trans()函数变成了这个样子： static int simp\_blkdev\_trans(unsigned long long dsk\_offset, void \*buf,                 unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_first\_page;         unsigned int this\_off;         unsigned int this\_cnt;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each data segment \*/                 this\_off = (dsk\_offset + done\_cnt) & ~SIMP\_BLKDEV\_DATASEGMASK;                 this\_cnt = min(len - done\_cnt,                         (unsigned int)SIMP\_BLKDEV\_DATASEGSIZE - this\_off);                  this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT);                 if (!this\_first\_page) {                         if (!dir) {                                 memset(buf + done\_cnt, 0, this\_cnt);                                 goto trans\_done;                         }                          /\* prepare new memory segment for write \*/                         this\_first\_page = alloc\_pages(                                 GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                                 SIMP\_BLKDEV\_DATASEGORDER);                         if (!this\_first\_page) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": allocate page failed\n");                                 return -ENOMEM;                         }                          this\_first\_page->index = (dsk\_offset + done\_cnt)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT;                          if (IS\_ERR\_VALUE(radix\_tree\_insert(&simp\_blkdev\_data,                                 this\_first\_page->index, this\_first\_page))) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": insert page to radix\_tree failed"                                         " seg=%lu\n", this\_first\_page->index);                                 \_\_free\_pages(this\_first\_page,                                         SIMP\_BLKDEV\_DATASEGORDER);                                 return -EIO;                         }                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans\_oneseg(this\_first\_page,                         this\_off, buf + done\_cnt, this\_cnt, dir)))                         return -EIO;  trans\_done:                 done\_cnt += this\_cnt;         }          return 0; }  代码就这样被莫名其妙地改完了，感觉这次的改动比预想的少，并且也比较集中，这其实还是托了前些章的福，正是在此之前对程序结构的规划调整，在增加可读性的同时，也给随后的维护带来方便。处于良好维护下的程序代码结构应该越维护越让人赏心悦目，而不是越维护越混乱不堪。  现在我们来试验一下这次修改的效果： 先编译： # make make -C /lib/modules/2.6.18-53.el5/build SUBDIRS=/root/test/simp\_blkdev/simp\_blkdev\_step14 modules make[1]: Entering directory `/usr/src/kernels/2.6.18-53.el5-i686'   CC [M]  /root/test/simp\_blkdev/simp\_blkdev\_step14/simp\_blkdev.o   Building modules, stage 2.   MODPOST   CC      /root/test/simp\_blkdev/simp\_blkdev\_step14/simp\_blkdev.mod.o   LD [M]  /root/test/simp\_blkdev/simp\_blkdev\_step14/simp\_blkdev.ko make[1]: Leaving directory `/usr/src/kernels/2.6.18-53.el5-i686' # 没发现问题。  然后看看目前的内存状况： # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        87920 kB LowTotal:       896356 kB LowFree:        791920 kB ... # 可以看出高端和低端内存分别剩余87M和791M。  然后指定size=50M加载模块后看看内存变化： # insmod simp\_blkdev.ko size=50M # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        86804 kB LowTotal:       896356 kB LowFree:        791912 kB ... # 在这里我们发现剩余内存的变化不大， 这也证明了这次修改的效果，因为加载模块时不会申请用于存储数据的全部内存。 而在原先的代码中，这一步骤将使机器减少大约50M的剩余空间。  然后我们来验证读取块设备时也不会导致分配内存： # dd if=/dev/simp\_blkdev of=/dev/null 102400+0 records in 102400+0 records out 52428800 bytes (52 MB) copied, 0.376118 seconds, 139 MB/s # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        85440 kB LowTotal:       896356 kB LowFree:        791888 kB ... # 剩余内存几乎没有变化，这证明了我们的设想。  然后是写设备的情况： # dd if=/dev/zero of=/dev/simp\_blkdev dd: writing to `/dev/simp\_blkdev': No space left on device 102401+0 records in 102400+0 records out 52428800 bytes (52 MB) copied, 0.542117 seconds, 96.7 MB/s # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        34116 kB LowTotal:       896356 kB LowFree:        791516 kB ... # 这时剩余内存终于减少了大约50M， 这意味着驱动程序申请了大约50M的内存用于存储写入的数据。  如果向已写入的位置再次写入数据，理论上不应该造成再一次的分配， 让我们试试： # dd if=/dev/zero of=/dev/simp\_blkdev dd: writing to `/dev/simp\_blkdev': No space left on device 102401+0 records in 102400+0 records out 52428800 bytes (52 MB) copied, 0.644972 seconds, 81.3 MB/s # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        33620 kB LowTotal:       896356 kB LowFree:        791516 kB ... # 结果与预想一致。  现在卸载模块： # rmmod simp\_blkdev # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        84572 kB LowTotal:       896356 kB LowFree:        791640 kB ... # 我们发现被驱动程序使用的内存被释放回来了。  如果以上的实验没有让读者过瘾的话，我们来继续一个过分一些的， 也就是创建空间远远大于机器物理内存的块设备。 首先我们看看目前的系统内存状况： # cat /proc/meminfo ...  HighTotal:     1146816 kB HighFree:        77688 kB LowTotal:       896356 kB LowFree:        783296 kB ... # 机器的总内存是2G，目前剩余的高、低端内存加起来是860M左右。  然后我们加载模块，注意一下size参数的值： # insmod simp\_blkdev.ko size=10000G # 命令成功返回，而如果换作原先的代码， 命令出错返回......是不太可能的， 最可能的大概是内核直接panic。 这是因为申请光全部内存的操作将导致申请出错时运行的用于释放内存的代码所需要的内存都无法满足。  无论我们设置多大的块设备容量，模块加载后只要不执行写操作， 驱动程序都不会申请存储数据的内存。而这个测试： # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        75208 kB LowTotal:       896356 kB LowFree:        783132 kB ... # 也证明了这一点。  现在我们看看这时的块设备情况： # fdisk -l /dev/simp\_blkdev  Disk /dev/simp\_blkdev: 10737.4 GB, 10737418240000 bytes 255 heads, 63 sectors/track, 1305416 cylinders Units = cylinders of 16065 \* 512 = 8225280 bytes  Disk /dev/simp\_blkdev doesn't contain a valid partition table # 果然是10000G，这可以通过换算10737418240000 bytes得到。 而fdisk显示10737.4 GB是因为它是按照1k=1000字节、1M=1000K、1G=1000M来算的， 这种流氓的算法给硬盘厂商的缺斤少两行为提供了极好的借口。  这里省略fdisk、mkfs、mount、cp等操作， 直接用dd往这个"10000G磁盘"中写入50M的数据： # dd if=/dev/zero of=/dev/simp\_blkdev bs=1M count=50 50+0 records in 50+0 records out 52428800 bytes (52 MB) copied, 0.324054 seconds, 162 MB/s # cat /proc/meminfo ... HighTotal:     1146816 kB HighFree:        23512 kB LowTotal:       896356 kB LowFree:        782884 kB ... # 现在的内存情况证明我们的"10000G磁盘"为这些数据申请了50M的内存。  实验差不多了，我们卸载模块： # rmmod simp\_blkdev. #  做完以上的实验，读者可能会有一个疑问，如果我们真的向那个"10000G磁盘"中写入了10000G的数据怎么样呢？ 回答可能不太如人意，就是系统很可能会panic。 因为这个操作将迫使驱动程序吃掉全部可能获得的物理内存，并且在吃光最后那么一丁点内存之前不会发生错误， 这也意味着走到出错处理这一步的时候，系统已经几乎无可救药了。其实在此之前系统就会一次进行： 释放缓存、试图把所有的用户进程的内存换出、杀死全部能够杀死的进程等操作。 而我们的驱动程序由于被看作是内核的一部分，却不会被停止，而是在继续不停的吃掉通过上述方式释放出的可怜的内存。试想，一个已经走到这一步的系统还有什么继续运行的可能呢？  因此，我们的程序确实需要改善以解决这个问题，因为世界上总是有一些疯狂的人在想各种办法虐待电脑。 但我们并不打算在本教程中解决它，因为这个教程中的每一章都企图为读者说明一类知识或一种方法，而不是仅仅为了这个示例性质的程序的功能本身。所以这一项改善就当作是留给读者的练习了。  本章通过改善块设备驱动程序实现了内存的滞后申请，其目的在于介绍这种方法，以使它在其他的相似程序中也得以实现。不过，这并不意味着作者希望读者把这种方法过分引用， 比如引用成平时不学习，考试前临时抱佛脚。 |

# 第15章(最终章)

|  |
| --- |
| 在上一章中我们对这个块设备驱动所作的更改使它具备了动态申请内存的能力，但实际上同时也埋下一个隐患，就是数据访问冲突。  这里我们顺便唠叨一下内核开发中的同步问题。 提到数据访问同步，自然而然会使人想到多进程、多线程、加锁、解锁、信号量、synchronized关键字等东西，然后就很头疼。 对于用户态程序，网上大量的解释数据同步概念和方法的文章给人的印象大概是： 同步很危险，编程要谨慎， 处处有机关，问题很难找。  对于第一次进行多线程时编程的人来说，感觉可能是以下两种： 一种是觉得程序中处处都会有问题，任何一条访问数据的指令都不安全，恨不得把程序中所有的数据都加上锁，甚至打算给锁本身的数据再加个锁，另一种是没觉得有什么困难，根本不去理什么都互斥不互斥，就按原先的来，编出的程序居然也运行得很顺。 然后怀着这两种想法人通过不断的学习和实践掌握了数据同步的知识后认识到，数据同步其实并不像前一种想法那样危险，也不像后一种想法那样简单。  所幸的是对于不少用户态程序来说，倒是可以不用考虑数据同步问题。 至少当我们刚开始写HelloWorld时不用去理这个麻烦。  而对于内核态代码而言，很不幸，整个儿几乎都相当于用户态的多线程。 其实事情也并非原本就是这么糟的。 在很久很久以前，山是青的，草是绿的，牛奶是能喝的，见到老人摔跤是敢扶的，作者是纯情的，电脑也是单CPU的。那时的内核环境很静，很美。除了中断会时不时地捣捣乱，其余的都挺诗意。 代码独个儿在跑，就像是一辆汽车在荒漠上奔驰，因为没有其他妨碍，几乎可以毫无顾忌地访问数据，而不用考虑什么万恶的访问冲突。唯一要考虑的从天而降的中断奥特曼，解决的方法倒也不难，禁用了中断看你还能咋的。  然后随着作者的成长，目光从书本转向了美眉，计算机也由单CPU发展成了多CPU。 内核代码的执行环境终于开始热闹起来，由于每个CPU上都在执行任务，这些任务进入到对应的内核态时会出现多条内核指令流同时执行，这些指令流对全局数据的访问很明显就牵涉到了同步问题，这是开端。 从那时起编程时要考虑其他CPU上的事情了。  然后随着作者的进一步成长，目光从美眉的脸转向了胸，CPU制造商为了贯彻给程序员找麻烦的精神，搞出了乱序执行。这一创举惊醒了多年来还在梦中的诸多程序员，原来，程序不是按程序执行的啊。 正如林高官说的：“我是交通部派来的，级别和你们市长一样高，敢跟我斗，你们这些人算个屁呀！”原来，无职无权的平民百姓就是屁啊。正当程序员从睡梦中惊醒还没缓过神时，编译器又跟着捣乱， “你CPU都能乱序了，凭什么不让我乱序？”然后热闹了，好在我们还有mb()、rmb()、wmb()、barrier()这几根救命稻草，事情倒是没变得太糟。  然后随着作者的进一步成长，目光从美眉的胸转向了臀，内核也从一开始时被动的为了适应多CPU而不得已半推半就支持多任务并行，转向了主动掀起裙角管它一个还是几个CPU都去多任务了。 从技术面解释，这就是大名鼎鼎的内核抢占。 内核的程序员从此不仅要考虑其他CPU，好要提妨自个儿的CPU，因为执行代码的CPU说不定什么时候就莫名其妙的被调度执行别的任务了。  如果以作者的成长历程为主线解释内核的演化还不至于太混乱的话， 我们还可以考虑再介绍一下spin\_lock, mutex\_lock, preempt\_disable,atomic\_t和rcu等函数，不过作者忍住了这一冲动，还是让读者去google吧。  然后回到我们的代码，现在的代码是有问题的。 比如simp\_blkdev\_trans()函数中，假设2个任务同时向块设备的同一区域写数据，而这块区域在这之前没有被写过，也就是说还没有申请内存，那么如果运气够好的话，这两个进程可能几乎同时运行到： this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT); 这句，很明显这两个任务得到的this\_first\_page都是NULL，然后它们争先恐后的执行 if (!this\_first\_page) 判断，从而进入之后的alloc\_pages，随后它们都会为这个块设备区域申请内存，并加入基树结构。 如果运气爆发的话，这两个任务radix\_tree\_insert()的代码中将有机会近乎同时越过 if (slot != NULL)         return -EEXIST; 的最后防线，先后将新申请的内存指针赋值给基树结点。 虽然x86的多处理器对同一块内存的写操作是原子的，这样至少不会因为这两个任务同时赋值基树指针造成指针指向莫名其妙的值，但这仍然也解决不了我们的问题，后一个赋值操作将覆盖前一个操作的结果，基数节点最终将指向稍后一点执行赋值操作的任务。 这两个任务最终将运行到radix\_tree\_insert()函数的结尾，而函数的返回值都是漂亮的0。 剩下的事情扳脚丫子大概也能想出来了，这两个任务都将自欺欺人地认为自己正确而成功地为块设备分配了内存， 而真相是其中一个任务拿走的内存却再也没有机会拿回来了。  至于解决方法嘛，当然是加锁。 只要我们让“查找基数中有没有这个节点”到“分配内存并插入这节点”的过程中没有其他任务的打搅， x就自然的解决了这个问题。  首先定义一个锁，因为是用来锁simp\_blkdev\_data的， 就放在static struct radix\_tree\_root simp\_blkdev\_data;后面吧： DEFINE\_MUTEX(simp\_blkdev\_datalock); /\* protects the disk data op \*/  然后根据刚才的思想给对simp\_blkdev\_trans()函数中的simp\_blkdev\_datalock的操作加锁， 也就是在 this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT); 语句之前添加： mutex\_lock(&simp\_blkdev\_datalock);  操作结束后被忘了把锁还回去，否则下次再操作时就成死锁了，因此在 trans\_done: 后面加上 mutex\_unlock(&simp\_blkdev\_datalock); 这一行。  完成了吗？细心看看就知道还没完。 simp\_blkdev\_trans()函数中有一些判断异常的代码，这些代码大多是扔出一条printk就直接return的。 这样可不行，可千万别让它们临走时把锁也顺回去了。 这意味着我们要在simp\_blkdev\_trans()函数中的3个故障时return的代码前完成锁的释放。 因此simp\_blkdev\_trans()函数最后就成了这样： static int simp\_blkdev\_trans(unsigned long long dsk\_offset, void \*buf,                 unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_first\_page;         unsigned int this\_off;         unsigned int this\_cnt;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each data segment \*/                 this\_off = (dsk\_offset + done\_cnt) & ~SIMP\_BLKDEV\_DATASEGMASK;                 this\_cnt = min(len - done\_cnt,                         (unsigned int)SIMP\_BLKDEV\_DATASEGSIZE - this\_off);                  mutex\_lock(&simp\_blkdev\_datalock);                  this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT);                 if (!this\_first\_page) {                         if (!dir) {                                 memset(buf + done\_cnt, 0, this\_cnt);                                 goto trans\_done;                         }                          /\* prepare new memory segment for write \*/                         this\_first\_page = alloc\_pages(                                 GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                                 SIMP\_BLKDEV\_DATASEGORDER);                         if (!this\_first\_page) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": allocate page failed\n");                                 mutex\_unlock(&simp\_blkdev\_datalock);                                 return -ENOMEM;                         }                          this\_first\_page->index = (dsk\_offset + done\_cnt)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT;                          if (IS\_ERR\_VALUE(radix\_tree\_insert(&simp\_blkdev\_data,                                 this\_first\_page->index, this\_first\_page))) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": insert page to radix\_tree failed"                                         " seg=%lu\n", this\_first\_page->index);                                 \_\_free\_pages(this\_first\_page,                                         SIMP\_BLKDEV\_DATASEGORDER);                                 mutex\_unlock(&simp\_blkdev\_datalock);                                 return -EIO;                         }                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans\_oneseg(this\_first\_page,                         this\_off, buf + done\_cnt, this\_cnt, dir))) {                         mutex\_unlock(&simp\_blkdev\_datalock);                         return -EIO;                 } trans\_done:                 mutex\_unlock(&simp\_blkdev\_datalock);                 done\_cnt += this\_cnt;         }          return 0; }  这个函数差不多了。 我们再看看代码中还有什么地方也对simp\_blkdev\_data进行操作来着，别漏掉了这些小王八蛋。 查找一下代码，我们发现free\_diskmem()函数中也进行了操作。  其实从理论上说，这里不加锁是不会产生问题的，因为对内核在执行对块设备设备时，会锁住这个设备对应的模块(天哪，又是锁，这一章和锁彪上了)，其结果是在simp\_blkdev\_trans()函数操作simp\_blkdev\_data的过程中，该模块无法卸载，从而无法不会运行到free\_diskmem()函数。  那么如果同时卸载这个模块呢，回答是也没有问题，英勇的模块锁也会搞掂这种情况。  这一章由于没有进行功能增加，就不列出修改后模块的测试经过了，不过作为对读者的安慰，我们将列出到目前为止经历了大大小小修改后的全部模块代码。 看到这些代码，我们能历历在目的回忆出读这篇教程到现在为止所经受的全部折磨和苦难。 当然也能感受到坚持到现在所得到的知识和领悟。  对于Linux而言，甚至仅仅对于块设备驱动程序而言，这部教程揭开的也仅仅是冰山一角。而更多的知识其实离我们很近，在google上，在代码中，在心中。 学习，是要用心，不断地去想，同时要有恒心、耐心、要细心， 人应该越学越谦虚，问题应该越学越多，这大概就是作者通过这部教程最想告诉读者的。  #include <linux/module.h> #include <linux/blkdev.h> #include <linux/hdreg.h> #include <linux/version.h>  /\* \* A simple block device driver based on memory \* \* Copyright 2008 - \*        Zhaolei <zhaolei@cn.fujitsu.com> \* \* Sample for using: \*   Create device file (first time only): \*     Note: If your system have udev, it can create device file for you in time \*           of lsmod and fdisk automatically. \*           Otherwise you need to create them yourself by following steps. \*     mknod /dev/simp\_blkdev  b 72 0 \*     mknod /dev/simp\_blkdev1 b 72 1 \*     mknod /dev/simp\_blkdev2 b 72 2 \* \*   Create dirs for test (first time only): \*     mkdir /mnt/temp1/ # first time only \*     mkdir /mnt/temp2/ # first time only \* \*   Run it: \*     make \*     insmod simp\_blkdev.ko \*     # or insmod simp\_blkdev.ko size=numK/M/G/T \*     fdisk /dev/simp\_blkdev # create 2 patitions \*     mkfs.ext3 /dev/simp\_blkdev1 \*     mkfs.ext3 /dev/simp\_blkdev2 \*     mount /dev/simp\_blkdev1 /mnt/temp1/ \*     mount /dev/simp\_blkdev2 /mnt/temp2/ \*     # play in /mnt/temp1/ and /mnt/temp2/ \*     umount /mnt/temp1/ \*     umount /mnt/temp2/ \*     rmmod simp\_blkdev.ko \* \*/  #define SIMP\_BLKDEV\_DEVICEMAJOR        COMPAQ\_SMART2\_MAJOR #define SIMP\_BLKDEV\_DISKNAME        "simp\_blkdev"  #define SIMP\_BLKDEV\_SECTORSHIFT        (9) #define SIMP\_BLKDEV\_SECTORSIZE        (1ULL<<SIMP\_BLKDEV\_SECTORSHIFT) #define SIMP\_BLKDEV\_SECTORMASK        (~(SIMP\_BLKDEV\_SECTORSIZE-1))  /\* usable partitions is SIMP\_BLKDEV\_MAXPARTITIONS - 1 \*/ #define SIMP\_BLKDEV\_MAXPARTITIONS        (64)  #define SIMP\_BLKDEV\_DATASEGORDER        (2) #define SIMP\_BLKDEV\_DATASEGSHIFT        (PAGE\_SHIFT + SIMP\_BLKDEV\_DATASEGORDER) #define SIMP\_BLKDEV\_DATASEGSIZE                (PAGE\_SIZE << SIMP\_BLKDEV\_DATASEGORDER) #define SIMP\_BLKDEV\_DATASEGMASK                (~(SIMP\_BLKDEV\_DATASEGSIZE-1))  static struct request\_queue \*simp\_blkdev\_queue; static struct gendisk \*simp\_blkdev\_disk;  static struct radix\_tree\_root simp\_blkdev\_data; DEFINE\_MUTEX(simp\_blkdev\_datalock); /\* protects the disk data op \*/  static char \*simp\_blkdev\_param\_size = "16M"; module\_param\_named(size, simp\_blkdev\_param\_size, charp, S\_IRUGO);  static unsigned long long simp\_blkdev\_bytes;  static int simp\_blkdev\_trans\_oneseg(struct page \*start\_page,                 unsigned long offset, void \*buf, unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_page;         unsigned int this\_off;         unsigned int this\_cnt;         void \*dsk\_mem;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each page \*/                 this\_page = start\_page + ((offset + done\_cnt) >> PAGE\_SHIFT);                 this\_off = (offset + done\_cnt) & ~PAGE\_MASK;                 this\_cnt = min(len - done\_cnt, (unsigned int)PAGE\_SIZE                         - this\_off);                  dsk\_mem = kmap(this\_page);                 if (!dsk\_mem) {                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": map device page failed: %p\n", this\_page);                         return -ENOMEM;                 }                 dsk\_mem += this\_off;                  if (!dir)                         memcpy(buf + done\_cnt, dsk\_mem, this\_cnt);                 else                         memcpy(dsk\_mem, buf + done\_cnt, this\_cnt);                  kunmap(this\_page);                  done\_cnt += this\_cnt;         }          return 0; }  static int simp\_blkdev\_trans(unsigned long long dsk\_offset, void \*buf,                 unsigned int len, int dir) {         unsigned int done\_cnt;         struct page \*this\_first\_page;         unsigned int this\_off;         unsigned int this\_cnt;          done\_cnt = 0;         while (done\_cnt < len) {                 /\* iterate each data segment \*/                 this\_off = (dsk\_offset + done\_cnt) & ~SIMP\_BLKDEV\_DATASEGMASK;                 this\_cnt = min(len - done\_cnt,                         (unsigned int)SIMP\_BLKDEV\_DATASEGSIZE - this\_off);                  mutex\_lock(&simp\_blkdev\_datalock);                  this\_first\_page = radix\_tree\_lookup(&simp\_blkdev\_data,                         (dsk\_offset + done\_cnt) >> SIMP\_BLKDEV\_DATASEGSHIFT);                 if (!this\_first\_page) {                         if (!dir) {                                 memset(buf + done\_cnt, 0, this\_cnt);                                 goto trans\_done;                         }                          /\* prepare new memory segment for write \*/                         this\_first\_page = alloc\_pages(                                 GFP\_KERNEL | \_\_GFP\_ZERO | \_\_GFP\_HIGHMEM,                                 SIMP\_BLKDEV\_DATASEGORDER);                         if (!this\_first\_page) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": allocate page failed\n");                                 mutex\_unlock(&simp\_blkdev\_datalock);                                 return -ENOMEM;                         }                          this\_first\_page->index = (dsk\_offset + done\_cnt)                                 >> SIMP\_BLKDEV\_DATASEGSHIFT;                          if (IS\_ERR\_VALUE(radix\_tree\_insert(&simp\_blkdev\_data,                                 this\_first\_page->index, this\_first\_page))) {                                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                         ": insert page to radix\_tree failed"                                         " seg=%lu\n", this\_first\_page->index);                                 \_\_free\_pages(this\_first\_page,                                         SIMP\_BLKDEV\_DATASEGORDER);                                 mutex\_unlock(&simp\_blkdev\_datalock);                                 return -EIO;                         }                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans\_oneseg(this\_first\_page,                         this\_off, buf + done\_cnt, this\_cnt, dir))) {                         mutex\_unlock(&simp\_blkdev\_datalock);                         return -EIO;                 } trans\_done:                 mutex\_unlock(&simp\_blkdev\_datalock);                 done\_cnt += this\_cnt;         }          return 0; }  static int simp\_blkdev\_make\_request(struct request\_queue \*q, struct bio \*bio) {         int dir;         unsigned long long dsk\_offset;         struct bio\_vec \*bvec;         int i;         void \*iovec\_mem;          switch (bio\_rw(bio)) {         case READ:         case READA:                 dir = 0;                 break;         case WRITE:                 dir = 1;                 break;         default:                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": unknown value of bio\_rw: %lu\n", bio\_rw(bio));                 goto bio\_err;         }          if ((bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT) + bio->bi\_size                 > simp\_blkdev\_bytes) {                 printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                         ": bad request: block=%llu, count=%u\n",                         (unsigned long long)bio->bi\_sector, bio->bi\_size);                 goto bio\_err;         }          dsk\_offset = bio->bi\_sector << SIMP\_BLKDEV\_SECTORSHIFT;          bio\_for\_each\_segment(bvec, bio, i) {                 iovec\_mem = kmap(bvec->bv\_page) + bvec->bv\_offset;                 if (!iovec\_mem) {                         printk(KERN\_ERR SIMP\_BLKDEV\_DISKNAME                                 ": map iovec page failed: %p\n", bvec->bv\_page);                         goto bio\_err;                 }                  if (IS\_ERR\_VALUE(simp\_blkdev\_trans(dsk\_offset, iovec\_mem,                         bvec->bv\_len, dir)))                         goto bio\_err;                  kunmap(bvec->bv\_page);                  dsk\_offset += bvec->bv\_len;         }  #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, bio->bi\_size, 0); #else         bio\_endio(bio, 0); #endif          return 0;  bio\_err: #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 24)         bio\_endio(bio, 0, -EIO); #else         bio\_endio(bio, -EIO); #endif         return 0; }  static int simp\_blkdev\_getgeo(struct block\_device \*bdev,                 struct hd\_geometry \*geo) {         /\*          \* capacity        heads        sectors        cylinders          \* 0~16M        1        1        0~32768          \* 16M~512M        1        32        1024~32768          \* 512M~16G        32        32        1024~32768          \* 16G~...        255        63        2088~...          \*/         if (simp\_blkdev\_bytes < 16 \* 1024 \* 1024) {                 geo->heads = 1;                 geo->sectors = 1;          } else if (simp\_blkdev\_bytes < 512 \* 1024 \* 1024) {                 geo->heads = 1;                 geo->sectors = 32;         } else if (simp\_blkdev\_bytes < 16ULL \* 1024 \* 1024 \* 1024) {                 geo->heads = 32;                 geo->sectors = 32;         } else {                 geo->heads = 255;                 geo->sectors = 63;         }          geo->cylinders = simp\_blkdev\_bytes >> SIMP\_BLKDEV\_SECTORSHIFT                 / geo->heads / geo->sectors;          return 0; }  struct block\_device\_operations simp\_blkdev\_fops = {         .owner                = THIS\_MODULE,         .getgeo                = simp\_blkdev\_getgeo, };  void free\_diskmem(void) {         unsigned long long next\_seg;         struct page \*seglist[64];         int listcnt;         int i;          next\_seg = 0;         do {                 listcnt = radix\_tree\_gang\_lookup(&simp\_blkdev\_data,                         (void \*\*)seglist, next\_seg, ARRAY\_SIZE(seglist));                  for (i = 0; i < listcnt; i++) {                         next\_seg = seglist[i]->index;                         radix\_tree\_delete(&simp\_blkdev\_data, next\_seg);                         \_\_free\_pages(seglist[i], SIMP\_BLKDEV\_DATASEGORDER);                 }                  next\_seg++;         } while (listcnt == ARRAY\_SIZE(seglist)); }  int getparam(void) {         char unit;         char tailc;          if (sscanf(simp\_blkdev\_param\_size, "%llu%c%c", &simp\_blkdev\_bytes,                 &unit, &tailc) != 2) {                 return -EINVAL;         }          if (!simp\_blkdev\_bytes)                 return -EINVAL;          switch (unit) {         case 'g':         case 'G':                 simp\_blkdev\_bytes <<= 30;                 break;         case 'm':         case 'M':                 simp\_blkdev\_bytes <<= 20;                 break;         case 'k':         case 'K':                 simp\_blkdev\_bytes <<= 10;                 break;         case 'b':         case 'B':                 break;         default:                 return -EINVAL;         }          /\* make simp\_blkdev\_bytes fits sector's size \*/         simp\_blkdev\_bytes = (simp\_blkdev\_bytes + SIMP\_BLKDEV\_SECTORSIZE - 1)                 & SIMP\_BLKDEV\_SECTORMASK;          return 0; }  static int \_\_init simp\_blkdev\_init(void) {         int ret;          ret = getparam();         if (IS\_ERR\_VALUE(ret))                 goto err\_getparam;          simp\_blkdev\_queue = blk\_alloc\_queue(GFP\_KERNEL);         if (!simp\_blkdev\_queue) {                 ret = -ENOMEM;                 goto err\_alloc\_queue;         }         blk\_queue\_make\_request(simp\_blkdev\_queue, simp\_blkdev\_make\_request);          simp\_blkdev\_disk = alloc\_disk(SIMP\_BLKDEV\_MAXPARTITIONS);         if (!simp\_blkdev\_disk) {                 ret = -ENOMEM;                 goto err\_alloc\_disk;         }          INIT\_RADIX\_TREE(&simp\_blkdev\_data, GFP\_KERNEL);          strcpy(simp\_blkdev\_disk->disk\_name, SIMP\_BLKDEV\_DISKNAME);         simp\_blkdev\_disk->major = SIMP\_BLKDEV\_DEVICEMAJOR;         simp\_blkdev\_disk->first\_minor = 0;         simp\_blkdev\_disk->fops = &simp\_blkdev\_fops;         simp\_blkdev\_disk->queue = simp\_blkdev\_queue;         set\_capacity(simp\_blkdev\_disk,                 simp\_blkdev\_bytes >> SIMP\_BLKDEV\_SECTORSHIFT);         add\_disk(simp\_blkdev\_disk);          return 0;  err\_alloc\_disk:         blk\_cleanup\_queue(simp\_blkdev\_queue); err\_alloc\_queue: err\_getparam:         return ret; }  static void \_\_exit simp\_blkdev\_exit(void) {         del\_gendisk(simp\_blkdev\_disk);         free\_diskmem();         put\_disk(simp\_blkdev\_disk);         blk\_cleanup\_queue(simp\_blkdev\_queue); }  module\_init(simp\_blkdev\_init); module\_exit(simp\_blkdev\_exit);  MODULE\_LICENSE("GPL");  追记：偶然看到刚才的代码首部注释，Copyright后面还是2008年。 大概是从第一章开始一直这样拷贝过来的。 这部教程从2008年11月断断续续的写到了2009年3月，终于功德圆满了。 作为作者写的第一个如此长度篇幅的教程，炸一眼瞟过来，倒也还像个样子，看来写教程并不是太难高攀的事情，因此如果读者也时不时地有一些写起来的冲动，就不妨开始吧: )  本章以块设备驱动程序的代码为例，说明了内核中的同步概念，当然，在不少情况下，程序员遇到的同步问题比这里的要复杂的多，内核中也采用了很多方法和技巧来处理同步，了解和学习这些知识，收获的不仅是数据同步本身的解决方法，更是一种思路，这对于更一般的程序设计都是有很大帮助的，因此有空时google一下， 总能找到自己想了解的知识。  <--全文完，赵磊出品，必属精品--> |