Linux那些事儿

系列丛书

之

我是SCSI硬盘

SCSI disk驱动程序分析,

2.6.22.1,为SCSI子系统分析做铺垫.

[1] 原文为blog.csdn.net/fudan_abc 上的《linux 那些事儿之SCSI硬盘》，有闲情逸致的或者有批评建议的可以到上面做客，也可以email 到ilttv.cn@gmail.com

目录

引子

有一天身子问心:"我要是痛了,医生会给我治,你痛了谁给你治啊?"于是心说:"我只能自己给自己治."也许因为这样,每个人都有一个治疗自己心中伤痛的方法.喝酒,唱歌,发火,或哭或笑,跟朋友诉苦,共旅行,跑马拉松,最差的一种方法是逃避这种心痛.我的方法是写这种伪技术的文章.

但是写些什么呢?既然有人写了 USB,既然有人写了 PCI,那么如果不写 SCSI,恐怕是天理地理都难容了.

就说我们公司吧,机房里那么些服务器,哪台没有 SCSI 设备?SCSI 硬盘,SCSI 带库,各种各样的 SCSI 设备在机房里都能找到.机房里没有 SCSI 设备,就好比超级女声里没有张靓颖.不过我一直很好奇的想知道,之所以这些年 SCSI 总线这么火,是不是因为这总线和芙蓉姐姐的 S 线条一样迷人一样妩媚一样优雅?怀着这种好奇心,我开始了探索 SCSI 子系统的道路,不过让我们从简单的实例开始,这个简单的实例就是 SCSI 硬盘(SCSI DISK).

写 SCSI 硬盘驱动分析实际上也是对 usb-storage 的一个延续.SCSI 硬盘驱动对应于一个模块,sd_mod.o.usb-storage 要工作首先就得依赖于两个 scsi 的模块,一个是 scsi 核心模块 scsi_mod.o,一个就是这个 scsi 硬盘的驱动模块 sd_mod.o.

老套路,首先我们从 drivers/scsi 目录来看这个 Kconfig 文件中是如何描述 scsi disk 的.

```
58 config BLK_DEV_SD
59         tristate "SCSI disk support"
60         depends on SCSI
61         ---help---
62           If you want to use SCSI hard disks, Fibre Channel disks,
63           Serial ATA (SATA) or Parallel ATA (PATA) hard disks,
64           USB storage or the SCSI or parallel port version of
65           the IOMEGA ZIP drive, say Y and read the SCSI-HOWTO,
66           the Disk-HOWTO and the Multi-Disk-HOWTO, available from
67           <http://www.tldp.org/docs.html#howto>. This is NOT for SCSI
68           CD-ROMs.
69
70           To compile this driver as a module, choose M here and read
71           <file:Documentation/scsi/scsi.txt>.
72           The module will be called sd_mod.
73
74           Do not compile this driver as a module if your root file system
75           (the one containing the directory /) is located on a SCSI disk.
76           In this case, do not compile the driver for your SCSI host adapter
77           (below) as a module either.
```

这个”depends on SCSI”说的就是 scsi core.毫无疑问,scsi 跑得快,全凭 core 来带.所有的 scsi 模块都是基于 scsi core 的.

再来看 Makefile,drivers/scsi 目录下的 Makefile,洋洋洒洒 190 行,但真正引起我们注意的是下

面这几行,

```
140 obj-$(CONFIG_BLK_DEV_SD)            += sd_mod.o
141 obj-$(CONFIG_BLK_DEV_SR)            += sr_mod.o
142 obj-$(CONFIG_CHR_DEV_SG)            += sg.o
143 obj-$(CONFIG_CHR_DEV_SCH)           += ch.o
144
145 # This goes last, so that "real" scsi devices probe earlier
146 obj-$(CONFIG_SCSI_DEBUG)             += scsi_debug.o
147
148 obj-$(CONFIG_SCSI_WAIT_SCAN)       += scsi_wait_scan.o
149
150 scsi_mod-y                                  += scsi.o hosts.o scsi_ioctl.o constants.o \
151                                                  scsicam.o scsi_error.o scsi_lib.o \
152                                                  scsi_scan.o scsi_sysfs.o \
153                                                  scsi_devinfo.o
154 scsi_mod-$(CONFIG_SCSI_NETLINK) += scsi_netlink.o
155 scsi_mod-$(CONFIG_SYSCTL)            += scsi_sysctl.o
156 scsi_mod-$(CONFIG_SCSI_PROC_FS) += scsi_proc.o
157
158 scsi_tgt-y                                 += scsi_tgt_lib.o scsi_tgt_if.o
159
160 sd_mod-objs         := sd.o
```

140 行和 160 行,给了我们足够的惊喜.因为 SCSI Disk 的驱动只有一个文件,sd.c.咱们瞅一眼这个文件有多大,

```
localhost:/usr/src/linux-2.6.22.1/drivers/scsi # wc -l sd.c
1903 sd.c
```

区区 1903 行,当一个模块只有这么点长的时候,一路走来的兄弟们恐怕已经难以抑制内心那阵狂喜了吧.但我想提醒你的是,爱的魅力不在于对象的多寡,而在于程度的深浅;代码的魅力不在于行数的多寡,而在于背后哲学思想的深浅.

简简单单初始化

在那茫茫人海中,我找到了这两行,

```
1886 module_init(init_sd);
1887 module_exit(exit_sd);
```

不要问我它们来自哪里,咱们整个故事就是围绕着 drivers/sd.c 这么一个文件展开,所以除非特别声明的之外,都是来自这个文件.

```
1831 /**
1832  *      init_sd - entry point for this driver (both when built in or when
1833  *      a module).
1834  *
1835  *      Note: this function registers this driver with the scsi mid-level.
```

```
1836  **/
1837 static int __init init_sd(void)
1838 {
1839          int majors = 0, i, err;
1840
1841          SCSI_LOG_HLQUEUE(3, printk("init_sd: sd driver entry point\n"));
1842
1843          for (i = 0; i < SD_MAJORS; i++)
1844                  if (register_blkdev(sd_major(i), "sd") == 0)
1845                          majors++;
1846
1847          if (!majors)
1848                  return -ENODEV;
1849
1850          err = class_register(&sd_disk_class);
1851          if (err)
1852                  goto err_out;
1853
1854          err = scsi_register_driver(&sd_template.gendrv);
1855          if (err)
1856                  goto err_out_class;
1857
1858          return 0;
1859
1860 err_out_class:
1861          class_unregister(&sd_disk_class);
1862 err_out:
1863          for (i = 0; i < SD_MAJORS; i++)
1864                  unregister_blkdev(sd_major(i), "sd");
1865          return err;
1866 }
1867
1868 /**
1869  *      exit_sd - exit point for this driver (when it is a module).
1870  *
1871  *      Note: this function unregisters this driver from the scsi mid-level.
1872  **/
1873 static void __exit exit_sd(void)
1874 {
1875          int i;
1876
1877          SCSI_LOG_HLQUEUE(3, printk("exit_sd: exiting sd driver\n"));
1878
1879          scsi_unregister_driver(&sd_template.gendrv);
```

```
1880            class_unregister(&sd_disk_class);
1881
1882            for (i = 0; i < SD_MAJORS; i++)
1883                    unregister_blkdev(sd_major(i), "sd");
1884 }
```

没什么特别的,一串的注册注销函数.

首先,register_blkdev,注册一个块设备.这个函数也算是骨灰级的了,N 年前就有这个函数了.那时候我曾天真的以为这个世界上只有三种设备,块设备,字符设备,网络设备.后来发现世界并非那么简单,生活也并非那么简单,尽管,生,很简单,活,很简单,但生活却不简单.

我们来看一下这个函数的效果,加载 sd_mod 之前,

```
localhost:~ # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
128 ptm
136 pts

Block devices:
  1 ramdisk
  3 ide0
  7 loop
  9 md
253 device-mapper
254 mdp
```

而通过下面两条命令加载了 scsi_mod 和 sd_mod 之后,

```
localhost:~ # modprobe scsi_mod
localhost:~ # modprobe sd_mod
localhost:~ # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
```

```
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
128 ptm
136 pts

Block devices:
  1 ramdisk
  3 ide0
  7 loop
  8 sd
  9 md
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
253 device-mapper
254 mdp
```

可以看到,多了一个叫做 sd 的家伙.

这里出现的宏 SD_MAJORS 实际上被定义为 16.所以经过 16 次循环之后,我们看到这里叫 sd 的有 16 个.

至于你说这些号码是怎么来的,就像八国联军在中国瓜分势力范围一样,每个国家分一片地,而 Linux 中所有的主设备号也是被各种各样的设备所瓜分.其中,8,65-71,136-143 这么个 16 个号码就被 scsi disk 所霸占了.sd_major()函数的返回值就是这 16 个数字.每个主设备号可以带 256 个次设备号.

1850 行, class_register,这行的效果就是,

```
localhost:~ # ls /sys/class/
```

backlight dma graphics input mem misc net pci_bus scsi_device scsi_disk scsi_host spi_master tty vc vtconsole

看到其中那项 scsi_disk 了么?这就是 class_register 这句干的好事.

而 1854 行,scsi_register_driver 则是赤裸裸的注册一个 scsi 设备驱动.伟大的设备模型告诉我们对于每个设备驱动,有一个与之对应的 struct device_driver 结构体,而为了体现各类设备驱动自身的特点,各个子系统可以定义自己的结构体,然后把 struct device_driver 包含进来如 C++中基类和扩展类一样.对于 scsi 子系统,这个基类就是 struct scsi_driver,这个结构体本身定义于 include/scsi/scsi_driver.h:

```
10 struct scsi_driver {
11          struct module               *owner;
12          struct device_driver        gendrv;
13
14          int (*init_command)(struct scsi_cmnd *);
15          void (*rescan)(struct device *);
16          int (*issue_flush)(struct device *, sector_t *);
17          int (*prepare_flush)(struct request_queue *, struct request *);
18 };
```

而咱们也自然定义了一个 scsi_driver 的结构体实例.它的名字叫做 sd_template.

```
232 static struct scsi_driver sd_template = {
233         .owner                      = THIS_MODULE,
234         .gendrv = {
235                 .name               = "sd",
236                 .probe              = sd_probe,
237                 .remove             = sd_remove,
238                 .suspend            = sd_suspend,
239                 .resume             = sd_resume,
240                 .shutdown           = sd_shutdown,
241         },
242         .rescan                     = sd_rescan,
243         .init_command               = sd_init_command,
244         .issue_flush                = sd_issue_flush,
245 };
```

这其中,gendrv 就是 struct device_driver 的结构体变量.咱们这么一注册,其直观效果就是:

```
localhost:~ # ls /sys/bus/scsi/drivers/
sd
```

而与以上三个函数相反的就是 exit_sd()中的另外仨函数

,scsi_unregister_driver,class_unregister,unregister_blkdev.这点我想不用我多说,阜成门外华联商厦门口卖盗版光盘那几位哥们儿也能明白怎么回事.

Okay 了,这个初始化就这么简单,就这么结束了.相比 uhci-hcd 的那个初始化,这里的确简单的不得了.实话实说,SCSI 硬盘驱动确实是挺简单的.下一步我们就从 sd_probe 函数看起,某种意义来说,读 sd_mod 的代码就算是对 scsi 子系统的入门.

依然 probe

虽然 scsi disk 不难,但是如果你以为 scsi disk 这个模块每个函数都像 init_sd()一样简单,那么我只能说你属于那种被蜘蛛咬了就以为自己是蜘蛛侠,被雷电劈了就以为自己是闪电侠,摸了一次高压电就以为自己是沈殿霞.你不服?咱们来看 sd_probe,这个函数就不是那么简单.

```
1566 /**
1567  *      sd_probe - called during driver initialization and whenever a
1568  *      new scsi device is attached to the system. It is called once
1569  *      for each scsi device (not just disks) present.
1570  *      @dev: pointer to device object
1571  *
1572  *      Returns 0 if successful (or not interested in this scsi device
1573  *      (e.g. scanner)); 1 when there is an error.
1574  *
1575  *      Note: this function is invoked from the scsi mid-level.
1576  *      This function sets up the mapping between a given
1577  *      <host,channel,id,lun> (found in sdp) and new device name
1578  *      (e.g. /dev/sda). More precisely it is the block device major
1579  *      and minor number that is chosen here.
1580  *
1581  *      Assume sd_attach is not re-entrant (for time being)
1582  *      Also think about sd_attach() and sd_remove() running coincidentally.
1583  **/
1584 static int sd_probe(struct device *dev)
1585 {
1586         struct scsi_device *sdp = to_scsi_device(dev);
1587         struct scsi_disk *sdkp;
1588         struct gendisk *gd;
1589         u32 index;
1590         int error;
1591
1592         error = -ENODEV;
1593             if (sdp->type != TYPE_DISK && sdp->type != TYPE_MOD &&
sdp->type != TYPE_RBC)
1594                 goto out;
1595
1596         SCSI_LOG_HLQUEUE(3, sdev_printk(KERN_INFO, sdp,
1597                                         "sd_attach\n"));
1598
1599         error = -ENOMEM;
1600         sdkp = kzalloc(sizeof(*sdkp), GFP_KERNEL);
1601         if (!sdkp)
1602                 goto out;
```

```
1603
1604            gd = alloc_disk(16);
1605            if (!gd)
1606                    goto out_free;
1607
1608            if (!idr_pre_get(&sd_index_idr, GFP_KERNEL))
1609                    goto out_put;
1610
1611            spin_lock(&sd_index_lock);
1612            error = idr_get_new(&sd_index_idr, NULL, &index);
1613            spin_unlock(&sd_index_lock);
1614
1615            if (index >= SD_MAX_DISKS)
1616                    error = -EBUSY;
1617            if (error)
1618                    goto out_put;
1619
1620            sdkp->device = sdp;
1621            sdkp->driver = &sd_template;
1622            sdkp->disk = gd;
1623            sdkp->index = index;
1624            sdkp->openers = 0;
1625
1626            if (!sdp->timeout) {
1627                    if (sdp->type != TYPE_MOD)
1628                            sdp->timeout = SD_TIMEOUT;
1629                    else
1630                            sdp->timeout = SD_MOD_TIMEOUT;
1631            }
1632
1633            class_device_initialize(&sdkp->cdev);
1634            sdkp->cdev.dev = &sdp->sdev_gendev;
1635            sdkp->cdev.class = &sd_disk_class;
1636            strncpy(sdkp->cdev.class_id, sdp->sdev_gendev.bus_id, BUS_ID_SIZE);
1637
1638            if (class_device_add(&sdkp->cdev))
1639                    goto out_put;
1640
1641            get_device(&sdp->sdev_gendev);
1642
1643            gd->major = sd_major((index & 0xf0) >> 4);
1644            gd->first_minor = ((index & 0xf) << 4) | (index & 0xfff00);
1645            gd->minors = 16;
1646            gd->fops = &sd_fops;
```

```
1647
1648            if (index < 26) {
1649                    sprintf(gd->disk_name, "sd%c", 'a' + index % 26);
1650            } else if (index < (26 + 1) * 26) {
1651                    sprintf(gd->disk_name, "sd%c%c",
1652                            'a' + index / 26 - 1,'a' + index % 26);
1653            } else {
1654                    const unsigned int m1 = (index / 26 - 1) / 26 - 1;
1655                    const unsigned int m2 = (index / 26 - 1) % 26;
1656                    const unsigned int m3 =    index % 26;
1657                    sprintf(gd->disk_name, "sd%c%c%c",
1658                            'a' + m1, 'a' + m2, 'a' + m3);
1659            }
1660
1661            gd->private_data = &sdkp->driver;
1662            gd->queue = sdkp->device->request_queue;
1663
1664            sd_revalidate_disk(gd);
1665
1666            gd->driverfs_dev = &sdp->sdev_gendev;
1667            gd->flags = GENHD_FL_DRIVERFS;
1668            if (sdp->removable)
1669                    gd->flags |= GENHD_FL_REMOVABLE;
1670
1671            dev_set_drvdata(dev, sdkp);
1672            add_disk(gd);
1673
1674            sd_printk(KERN_NOTICE, sdkp, "Attached SCSI %sdisk\n",
1675                      sdp->removable ? "removable " : "");
1676
1677            return 0;
1678
1679    out_put:
1680            put_disk(gd);
1681    out_free:
1682            kfree(sdkp);
1683    out:
1684            return error;
1685 }
```

如果我们不看新闻联播,我们又怎么知道自己生活在幸福中呢?如果我们不看 probe,我们又怎么知道设备驱动的故事是如何展开的呢?

首先,我们为 scsi device 准备一个指针,struct scsi_device *sdp,为 scsi disk 准备一个指针,struct scsi_disk *sdkp,此外,甭管是 scsi 硬盘还是 ide 硬盘,都少不了一个结构体 struct gendisk,这里咱们准备了一个指针 struct gendisk *gd.

一路走来的兄弟们一定知道,sd_probe 将会由 scsi 核心层调用,或者也叫 scsi mid-level 来调用.scsi mid-level 在调用 sd_probe 之前,已经为这个 scsi 设备准备好了 struct device,struct scsi_device,已经为它们做好了初始化,所以这里 struct device *dev 作为参数传递进来咱们就可以直接引用它的成员了.

这不,1593 行,就开始判断 sdp->type,这是 struct scsi_device 结构体中的成员 char type,它用来表征这个 scsi 设备是哪种类型的,scsi 设备五花八门,而只有这里列出来的这三种是 sd_mod 所支持的.这其中我们最熟悉的当属 TYPE_DISK,它就是普通的 scsi 磁盘,而 TYPE_MOD 表示的是磁光盘(Magneto-Optical disk),一种采用激光和磁场共同作用的磁光方式存储技术实现的介质,外观和 3.5 英寸软盘相似,量你也不知道,所以不多说了.另外,TYPE_RBC 也算在咱们名下,RBC 表示 Reduced Block Commands,中文叫命令集,这个也不必多说.

1600 行,为 sdkp 申请内存.struct scsi_disk 定义于 include/scsi/sd.h:

```
 34 struct scsi_disk {
 35         struct scsi_driver *driver;     /* always &sd_template */
 36         struct scsi_device *device;
 37         struct class_device cdev;
 38         struct gendisk    *disk;
 39         unsigned int       openers;     /* protected by BKL for now, yuck */
 40         sector_t           capacity;    /* size in 512-byte sectors */
 41         u32                index;
 42         u8                 media_present;
 43         u8                 write_prot;
 44         unsigned           WCE : 1;     /* state of disk WCE bit */
 45         unsigned           RCD : 1;     /* state of disk RCD bit, unused */
 46         unsigned           DPOFUA : 1;  /* state of disk DPOFUA bit */
 47 };
```

看起来,似乎描述一个 scsi disk 很简单,其实你不要忘了,前面我们还提到另一个结构体 struct gendisk,这个结构体来自一个神秘的地方,include/linux/genhd.h:

```
113 struct gendisk {
114         int major;                      /* major number of driver */
115         int first_minor;
116         int minors;                     /* maximum number of minors, =1 for
117                                          * disks that can't be partitioned. */
118         char disk_name[32];             /* name of major driver */
119         struct hd_struct **part;        /* [indexed by minor] */
120         int part_uevent_suppress;
121         struct block_device_operations *fops;
122         struct request_queue *queue;
123         void *private_data;
124         sector_t capacity;
125
126         int flags;
127         struct device *driverfs_dev;
128         struct kobject kobj;
129         struct kobject *holder_dir;
```

```
130             struct kobject *slave_dir;
131
132             struct timer_rand_state *random;
133             int policy;
134
135             atomic_t sync_io;               /* RAID */
136             unsigned long stamp;
137             int in_flight;
138 #ifdef    CONFIG_SMP
139             struct disk_stats *dkstats;
140 #else
141             struct disk_stats dkstats;
142 #endif
143             struct work_struct async_notify;
144 };
```

于是,struct scsi_disk 和 struct gendisk 联手来为我们描述一块磁盘,scsi_disk 是 scsi 专用,而 gendisk 中的 gen 表示 general,过了英语四级的都知道,这表示通用,即 scsi 呀,ide 呀,大家伙都能利用的.

于是 1604 行,alloc_disk 就是为我们分配一个 gendisk.

但是接下来 1608 行的 sd_index_idr 就有些学问了.

下面我们必须用专门一段文字来描述 idr 了.首先在 89 行我们看到下面这么一句,

```
89 static DEFINE_IDR(sd_index_idr);
```

这被叫做定义一个 IDR.印象中大四上刚开学的时候,江湖中开始流传一篇文章叫做”idr”-integer ID management,专门对 idr 进行了一些介绍,这篇文章最早是发表在 LWN(Linux Weekly News)上面.怎奈少不更事的我一直沉迷于上网,聊天,灌水,玩游戏,所以直到今天,依然不知道为什么这玩意儿叫做 idr,只是懵懵懂懂的感觉它是一个用来管理一些小整数的工具,具体来说,就是内核中定义了一些函数,几乎所有的函数都被定义在一个文件中,即 lib/idr.c,关于它的实现咱们自然不必多说,说多了就未免喧宾夺主了,我们只看它的实际效果.

实际上我们一共调用了三个来自 lib/idr.c 的函数,或者更确切的说是四个,因为上面这个宏 DEFINE_IDR 也是一个函数的包装,总的来说,如果我们需要使用 idr 工具,我们就需要首先调用 idr_init 函数,或者使用它的马夹 DEFINE_IDR,这算是初始化,也叫做创建一个 idr 对象,其实就是申请一个 struct idr 结构体变量.然后使用两个函数,一个是 idr_pre_get(),一个是 idr_get_new(),当我们日后觉得这个 idr 已经没有利用价值了,我们则可以调用另一个函数,idr_remove()来完成过河拆桥的工作.

我们看到 1608 行调用 idr_pre_get(),其第一个参数就是我们之前初始化的&sd_index_idr,第二个参数是一个掩码,和我们以往每一次申请内存时一样,通常传递的就是 GFP_KERNEL.这个函数有点与众不同的是,它返回 0 表示出错,返回非 0 才表示正常,典型的抽疯式函数.

而 1612 行,idr_get_new(),就是获得下一个 available 的 ID 号,保存在第三个参数中,即我们这里的 index,第二个参数不是太常用,传递个 NULL 就可以了.一切正常将返回 0.

而 index 必须小于 SD_MAX_DISKS,这个宏定义于 include/scsi/sd.h:

```
12  * This is limited by the naming scheme enforced in sd_probe,
13  * add another character to it if you really need more disks.
14  */
15 #define SD_MAX_DISKS      (((26 * 26) + 26 + 1) * 26)
```

比这个宏还大就肯定出错了.关于这个宏,曾几何时,我也和你一样,丈二和尚摸不着头脑,我也曾彷徨,也曾犹豫,也曾困惑,后来有一天我终于明白了,26 代表的是英文字母的个数,而下面我们马上就能看到,Linux 中对 scsi disk 的命名规则正是利用了 26 个英文字母.
不信你就看 1643 到 1659 行,这一段同时也正是 idr 的精华.最终你会发现,gd->disk_name 一定是在 sda-sdz 之间,或者是在 sdaa 到 sdzz 之间,或者是在 sdaaa 到 sdzzz 之间.算一下,是不是正好数量为 SD_MAX_DISKS 个.而 index 的取值范围则是[0,SD_MAX_DISKS)之间,只取整数.
举例来说,如果你只有一块硬盘,那么你能看到的是,/dev/sda,如果你有多块硬盘,比如像我下面这个例子中的一样,
localhost:~ # fdisk -l

Disk /dev/sda: 146.1 GB, 146163105792 bytes
255 heads, 63 sectors/track, 17769 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1		1	266	2136613+	83	Linux
/dev/sda2		2879	17769	119611957+	83	Linux
/dev/sda3	*	267	1572	10490445	83	Linux
/dev/sda4		1573	2878	10490445	82	Linux swap / Solaris

Partition table entries are not in disk order

Disk /dev/sdb: 5368 MB, 5368709120 bytes
166 heads, 62 sectors/track, 1018 cylinders
Units = cylinders of 10292 * 512 = 5269504 bytes

Device	Boot	Start	End	Blocks	Id	System

Disk /dev/sdc: 5368 MB, 5368709120 bytes
166 heads, 62 sectors/track, 1018 cylinders
Units = cylinders of 10292 * 512 = 5269504 bytes

Device	Boot	Start	End	Blocks	Id	System

Disk /dev/sdd: 5368 MB, 5368709120 bytes
166 heads, 62 sectors/track, 1018 cylinders
Units = cylinders of 10292 * 512 = 5269504 bytes

Device	Boot	Start	End	Blocks	Id	System

Disk /dev/sde: 5368 MB, 5368709120 bytes
166 heads, 62 sectors/track, 1018 cylinders
Units = cylinders of 10292 * 512 = 5269504 bytes

```
   Device Boot      Start         End      Blocks   Id  System
```

这个例子中我的机器里是 5 块 scsi 硬盘,那么它们的名字分别是 sda,sdb,sdc,sdd,sde,但是如果你他妈的比较变态,一台机器里接了 30 块硬盘,那么没啥说的,它们就会依次被命名为 sda,sdb,…,sdx,sdy,sdz,这还不够,只有 26 块,接下来的硬盘名称就叫做 sdaa,sdab,sdac,sdad,总共凑满 30 块.但如果你觉得这还不够变态,你非要挑战极限,你非要 play zhuangbility,那么在用完了 sdaa 到 sdzz 之后,Linux 还允许你用 sdaaa,sdaab,…,一直到 sdzzz.总之,在 Linux 中,你最多可以使用的硬盘数撑死就是 SD_MAX_DISKS 个.当然,我还是奉劝你,别这么干,毕竟孔子曾经曰过:”莫装 B,装 B 遭雷劈!”

算了我们言归正传,1643 行,gd->major 被赋了值,自从在张江软件园某不知名的小公司里笔试过那道”不用临时变量交换两个变量的值”之后,我曾深刻的反省自己为何当初没有好好学习谭浩强老师那本<<C 程序设计>>中的位运算.痛定思痛之后,我终于能看懂眼前这代码了,当然首先我们得明白这个 sd_major 为何物?

```
247 /*
248  * Device no to disk mapping:
249  *
250  *       major         disc2     disc  p1
251  *    |............|.............|....|....| <- dev_t
252  *      31          20 19          8 7   4 3   0
253  *
254  * Inside a major, we have 16k disks, however mapped non-
255  * contiguously. The first 16 disks are for major0, the next
256  * ones with major1, ... Disk 256 is for major0 again, disk 272
257  * for major1, ...
258  * As we stay compatible with our numbering scheme, we can reuse
259  * the well-know SCSI majors 8, 65--71, 136--143.
260  */
261 static int sd_major(int major_idx)
262 {
263         switch (major_idx) {
264         case 0:
265                 return SCSI_DISK0_MAJOR;
266         case 1 ... 7:
267                 return SCSI_DISK1_MAJOR + major_idx - 1;
268         case 8 ... 15:
269                 return SCSI_DISK8_MAJOR + major_idx - 8;
270         default:
271                 BUG();
272                 return 0;       /* shut up gcc */
273         }
274 }
```

看起来挺复杂,其实不然,我们前面说过,scsi disk 的主设备号是已经固定好了的,它就是瓜分了 8,65-71,128-135 这几个号,这里 SCSI_DISK0_MAJOR 就是 8,SCSI_DISK1_MAJOR 就是 65,SCSI_DISK8_MAJOR 就是 128.sd_major()接受的参数就是 index 的 bit4 到 bit7,而它取值范围自然就是 0 到 15,这也正是 sd_major()中 switch/case 语句判断的范围,即实际上 major_idx

就是主设备号的一个索引,就说是在这个16个主设备号中它算老几.而first_minor就是对应于本 index 的第一个次设备号,我们可以用代入法得到,当 index 为 0,则 first_minor 为 0,当 index 为 1,则 first_minor 为 16,当 index 为 2,则 first_minor 为 32.另一方面,minor 本身表示本 index 下有多少个次设备号,这个大家都是一样的,都是 16.我们通过下面这个例子也能看到:

```
[root@localhost ~]# cat /proc/partitions
major minor  #blocks  name

   8        0  285474816 sda
   8        1    2104483 sda1
   8        2   16779892 sda2
   8        3          1 sda3
   8        5   20972826 sda5
   8        6   20482843 sda6
   8        7   20482843 sda7
   8        8   10241406 sda8
   8        9   20482843 sda9
   8       10   20482843 sda10
   8       11   20482843 sda11
   8       12   20482843 sda12
   8       13   20482843 sda13
   8       14   20482843 sda14
   8       15   20482843 sda15
   8       16    5242880 sdb
   8       32    5242880 sdc
   8       48    5242880 sdd
   8       64    5242880 sde
```

很显然,对于 sda,其次设备是从 0 开始,对于 sdb,次设备号从 16 开始,对于 sdc,则从 32 开始,sdd 则从 48 开始,每个 index 或者说每个 disk_name 下面有 16 个次设备号.也因此一块 SCSI 硬盘就是最多 15 个分区.

除此之外,sd_probe 中主要就是些简单的赋值了.当然也不全是,比如 1633 行到 1639 行这一段,它的效果就是让这个设备出现在了 sysfs 中 class 子目录下面,比如下面这个例子:

```
localhost:~ # cat /sys/class/scsi_device/
0:0:8:0/ 0:2:0:0/ 1:0:0:0/ 1:0:0:1/ 1:0:0:2/ 1:0:0:3/
```

每一个 scsi 设备都在这个占了一个子目录.

1641 行这个 get_device 不用多说,访问一个 struct device 的第一步,增加引用计数.以后不用的时候自然会有一个相对的函数 put_device 被调用.

1671 行,dev_set_drvdata,就是设置 dev->driver_data 等于 sdkp,即让 struct device 的指针 dev 和 struct scsi_disk 的指针 sdkp 给联系起来.这就是所谓的朋友多了路好走,关系网建立得越大,日后使用起来就越方便.

最后,特别提醒两个赋值.第一个是,1621 行,让 sdkp->driver 等于&sd_template,另一个是 1646 行,让 gd->fops 等于&sd_fops.这两行赋值对咱们整个故事的意义,不亚于 1979 年那个春天有一位老人在中国的南海边划了一个圈对中国的重大意义.在咱们讲完 sd_probe 之后,这两个赋值将引领我们展开下面的故事.

关于 sd_probe,眼瞅着就要完了,但是很显然,有两个函数我们还没有提到,它们就是 1664 行的

sd_revalidate_disk()以及 1672 行的 add_disk(),这两个函数是如此的重要,以至于我们有必要在下一节专门来讲述它们.

磁盘磁盘你动起来!

首先我们看 sd_revalidate_disk(),这个函数很重要,一定程度上来说,正是这个函数从硬件和软件两个方面掀起了我们了解 scsi 磁盘的性高潮.这个函数它不是一个函数在战斗,它完全是贾宝玉林黛玉方世玉附体,由这一个函数可以牵连出 N 个函数.而这 N 个函数中的一些函数本身又有好几百行,所以我们算是陷进去了.

```
1496 /**
1497  *      sd_revalidate_disk - called the first time a new disk is seen,
1498  *      performs disk spin up, read_capacity, etc.
1499  *      @disk: struct gendisk we care about
1500  **/
1501 static int sd_revalidate_disk(struct gendisk *disk)
1502 {
1503         struct scsi_disk *sdkp = scsi_disk(disk);
1504         struct scsi_device *sdp = sdkp->device;
1505         unsigned char *buffer;
1506         unsigned ordered;
1507
1508         SCSI_LOG_HLQUEUE(3, sd_printk(KERN_INFO, sdkp,
1509                                       "sd_revalidate_disk\n"));
1510
1511         /*
1512          * If the device is offline, don't try and read capacity or any
1513          * of the other niceties.
1514          */
1515         if (!scsi_device_online(sdp))
1516                 goto out;
1517
1518         buffer = kmalloc(SD_BUF_SIZE, GFP_KERNEL | __GFP_DMA);
1519         if (!buffer) {
1520                 sd_printk(KERN_WARNING, sdkp, "sd_revalidate_disk: Memory "
1521                           "allocation failure.\n");
1522                 goto out;
1523         }
1524
1525         /* defaults, until the device tells us otherwise */
1526         sdp->sector_size = 512;
1527         sdkp->capacity = 0;
1528         sdkp->media_present = 1;
```

```
1529            sdkp->write_prot = 0;
1530            sdkp->WCE = 0;
1531            sdkp->RCD = 0;
1532
1533            sd_spinup_disk(sdkp);
1534
1535            /*
1536             * Without media there is no reason to ask; moreover, some devices
1537             * react badly if we do.
1538             */
1539            if (sdkp->media_present) {
1540                    sd_read_capacity(sdkp, buffer);
1541                    sd_read_write_protect_flag(sdkp, buffer);
1542                    sd_read_cache_type(sdkp, buffer);
1543            }
1544
1545            /*
1546             * We now have all cache related info, determine how we deal
1547             * with ordered requests.  Note that as the current SCSI
1548             * dispatch function can alter request order, we cannot use
1549             * QUEUE_ORDERED_TAG_* even when ordered tag is supported.
1550             */
1551            if (sdkp->WCE)
1552                    ordered = sdkp->DPOFUA
1553                                            ? QUEUE_ORDERED_DRAIN_FUA :
QUEUE_ORDERED_DRAIN_FLUSH;
1554            else
1555                    ordered = QUEUE_ORDERED_DRAIN;
1556
1557            blk_queue_ordered(sdkp->disk->queue, ordered, sd_prepare_flush);
1558
1559            set_capacity(disk, sdkp->capacity);
1560            kfree(buffer);
1561
1562    out:
1563            return 0;
1564 }
```

用我们经常用错的一个成语来说,就是首当其冲的函数便是 sd_spinup_disk().

```
1005 /*
1006  * spinup disk - called only in sd_revalidate_disk()
1007  */
1008 static void
1009 sd_spinup_disk(struct scsi_disk *sdkp)
1010 {
```

```
1011            unsigned char cmd[10];
1012            unsigned long spintime_expire = 0;
1013            int retries, spintime;
1014            unsigned int the_result;
1015            struct scsi_sense_hdr sshdr;
1016            int sense_valid = 0;
1017
1018            spintime = 0;
1019
1020            /* Spin up drives, as required.    Only do this at boot time */
1021            /* Spinup needs to be done for module loads too. */
1022            do {
1023                    retries = 0;
1024
1025                    do {
1026                            cmd[0] = TEST_UNIT_READY;
1027                            memset((void *) &cmd[1], 0, 9);
1028
1029                            the_result = scsi_execute_req(sdkp->device, cmd,
1030                                                          DMA_NONE,
NULL, 0,
1031                                                          &sshdr,
SD_TIMEOUT,
1032
SD_MAX_RETRIES);
1033
1034                            /*
1035                             * If the drive has indicated to us that it
1036                             * doesn't have any media in it, don't bother
1037                             * with any more polling.
1038                             */
1039                            if (media_not_present(sdkp, &sshdr))
1040                                    return;
1041
1042                            if (the_result)
1043                                    sense_valid = scsi_sense_valid(&sshdr);
1044                            retries++;
1045                    } while (retries < 3 &&
1046                             (!scsi_status_is_good(the_result) ||
1047                              ((driver_byte(the_result) & DRIVER_SENSE) &&
1048                                      sense_valid && sshdr.sense_key ==
UNIT_ATTENTION)));
1049
1050                    if ((driver_byte(the_result) & DRIVER_SENSE) == 0) {
```

```
1051                    /* no sense, TUR either succeeded or failed
1052                     * with a status error */
1053                    if(!spintime && !scsi_status_is_good(the_result)) {
1054                            sd_printk(KERN_NOTICE, sdkp, "Unit Not
Ready\n");
1055                            sd_print_result(sdkp, the_result);
1056                    }
1057                    break;
1058            }
1059
1060            /*
1061             * The device does not want the automatic start to be issued.
1062             */
1063            if (sdkp->device->no_start_on_add) {
1064                    break;
1065            }
1066
1067            /*
1068             * If manual intervention is required, or this is an
1069             * absent USB storage device, a spinup is meaningless.
1070             */
1071            if (sense_valid &&
1072                sshdr.sense_key == NOT_READY &&
1073                sshdr.asc == 4 && sshdr.ascq == 3) {
1074                    break;          /* manual intervention required */
1075
1076            /*
1077             * Issue command to spin up drive when not ready
1078             */
1079            } else if (sense_valid && sshdr.sense_key == NOT_READY) {
1080                    if (!spintime) {
1081                            sd_printk(KERN_NOTICE, sdkp, "Spinning up
disk...");
1082                            cmd[0] = START_STOP;
1083                            cmd[1] = 1;     /* Return immediately */
1084                            memset((void *) &cmd[2], 0, 8);
1085                            cmd[4] = 1;     /* Start spin cycle */
1086                                    scsi_execute_req(sdkp->device, cmd,
DMA_NONE,
1087                                                     NULL, 0, &sshdr,
1088                                                             SD_TIMEOUT,
SD_MAX_RETRIES);
1089                            spintime_expire = jiffies + 100 * HZ;
1090                            spintime = 1;
```

```
1091                                    }
1092                                    /* Wait 1 second for next try */
1093                                    msleep(1000);
1094                                    printk(".");
1095
1096                            /*
1097                              * Wait for USB flash devices with slow firmware.
1098                              * Yes, this sense key/ASC combination shouldn't
1099                              * occur here.    It's characteristic of these devices.
1100                              */
1101                            } else if (sense_valid &&
1102                                            sshdr.sense_key == UNIT_ATTENTION &&
1103                                            sshdr.asc == 0x28) {
1104                                    if (!spintime) {
1105                                            spintime_expire = jiffies + 5 * HZ;
1106                                            spintime = 1;
1107                                    }
1108                                    /* Wait 1 second for next try */
1109                                    msleep(1000);
1110                            } else {
1111                                    /* we don't understand the sense code, so it's
1112                                      * probably pointless to loop */
1113                                    if(!spintime) {
1114                                              sd_printk(KERN_NOTICE, sdkp, "Unit Not
Ready\n");
1115                                            sd_print_sense_hdr(sdkp, &sshdr);
1116                                    }
1117                                    break;
1118                            }
1119
1120                    } while (spintime && time_before_eq(jiffies, spintime_expire));
1121
1122                    if (spintime) {
1123                            if (scsi_status_is_good(the_result))
1124                                    printk("ready\n");
1125                            else
1126                                    printk("not responding...\n");
1127                    }
1128 }
```

顾名思义,spinup_disk 就是让磁盘转起来.然而,要看明白这个函数,你就不得不对 SCSI spec 有一定了解了.

这个函数虽然复杂,但是我们本着擒贼先擒王的思想,重点关注这个函数中最有价值的那行代码,没错,即使是曲阳路易买得超市门口看自行车的大妈都知道,这个函数中最有价值的那行代码一定是 1029 行,scsi_execute_req()函数的调用.这个函数算是 scsi 核心层提供的,咱们只管

调用不用管实现.我们在 include/scsi/scsi_device.h 中能找到它的声明:

```
297 extern int scsi_execute_req(struct scsi_device *sdev, const unsigned char *cmd,
298     int data_direction, void *buffer, unsigned bufflen,
299     struct scsi_sense_hdr *, int timeout, int retries);
```

和 usb 核心层一样,scsi 核心层也提供了大量的函数让我们调用,这些函数极大的便利了我们编写 scsi 设备驱动程序.我们只要准备好参数传递给这个函数,然后就万事大吉了,等着判断函数返回值就是了,至于需要传递的数据,则已经被填充在我们的参数中的 buffer 里边了.这就好比我每天上班的时候把自行车停在西直门城铁站外,到了晚上下班回来的时候,自行车框里自然而然的就被填充满了,什么都有,香烟盒,卫生纸,吃剩的苹果,嚼过的口香糖,偶尔还有用过的避孕套,总而言之,首都人民的热情一次次的让我感动得泪流满面,让我觉得北漂的日子并不孤独.

这个函数说白了就是执行一个 scsi 命令,其第一个参数不必多说,就是我们的 struct scsi_device 的结构体指针,咱们这个故事里就这么一个.第二个参数则是代表着命令,cmd 嘛,就是 command.其实每一个参数的意思都很明了.

咱们结合我们的代码来看我们具体传递了怎样的参数.第一个 sdkp->device 这没得说,第二个,cmd,咱们在 1011 行申请的一个 unsigned char 类型的数组,总共 10 个元素,1026 行给赋了值为 TEST_UNIT_READY.Test Unit Ready 就是一个很基本的 SCSI 命令.DMA_NONE 代表传输方向,buffer 和 bufflen 咱们用不上,因为这个命令就是测试设备准备好了没有,不需要传递什么数据.

所以正常来讲,咱们这么一调用 scsi_execute_req()以执行这个 Test Unit Ready 命令,返回的结果基本上都是好的,除非设备真的有毛病.

当然你要说有没有出错的时候,那当然也是有的.比如下面这个例子,

```
[root@localhost dev]# ls sd*
sda  sda1  sda10  sda11  sda12  sda13  sda14  sda2  sda3  sda5  sda6  sda7  sda8
sda9  sdb  sdc  sdd  sde  sdf
[root@localhost ~]# sg_turs /dev/sda
Completed 1 Test Unit Ready commands with 0 errors
[root@localhost ~]# sg_turs /dev/sdb
Completed 1 Test Unit Ready commands with 0 errors
[root@localhost ~]# sg_turs /dev/sdc
Completed 1 Test Unit Ready commands with 0 errors
[root@localhost ~]# sg_turs /dev/sde
Completed 1 Test Unit Ready commands with 0 errors
[root@localhost ~]# sg_turs /dev/sdf
test unit ready:    Fixed format, current;    Sense key: Not Ready
  Additional sense: Medium not present
Completed 1 Test Unit Ready commands with 1 errors
```

这里 sg_turs 这个命令就是用来手工发送 Test Unit Ready 用的.不过要使用这个命令,你得安装 sg3_utils 系列软件包.

```
[root@localhost dev]# rpm -qa | grep sg3_utils
sg3_utils-devel-1.20-2.1
sg3_utils-1.20-2.1
sg3_utils-libs-1.20-2.1
```

我们看到在我的五块硬盘中,前四块都没有问题,但是第六块就报错了.所以在执行完命令之

后,我们用 the_result 记录下结果,并且在 1046 行调用 scsi_status_is_good()来判断结果.关于 scsi_status_is_good()以及和它相关的一些宏定义于 include/scsi/scsi.h 文件中:

```
125 /*
126  *  SCSI Architecture Model (SAM) Status codes. Taken from SAM-3 draft
127  *  T10/1561-D Revision 4 Draft dated 7th November 2002.
128  */
129 #define SAM_STAT_GOOD                   0x00
130 #define SAM_STAT_CHECK_CONDITION 0x02
131 #define SAM_STAT_CONDITION_MET    0x04
132 #define SAM_STAT_BUSY                   0x08
133 #define SAM_STAT_INTERMEDIATE     0x10
134 #define SAM_STAT_INTERMEDIATE_CONDITION_MET 0x14
135 #define SAM_STAT_RESERVATION_CONFLICT 0x18
136 #define SAM_STAT_COMMAND_TERMINATED 0x22        /* obsolete in SAM-3 */
137 #define SAM_STAT_TASK_SET_FULL    0x28
138 #define SAM_STAT_ACA_ACTIVE       0x30
139 #define SAM_STAT_TASK_ABORTED     0x40
140
141 /** scsi_status_is_good - check the status return.
142  *
143  * @status: the status passed up from the driver (including host and
144  *          driver components)
145  *
146  * This returns true for known good conditions that may be treated as
147  * command completed normally
148  */
149 static inline int scsi_status_is_good(int status)
150 {
151         /*
152          * FIXME: bit0 is listed as reserved in SCSI-2, but is
153          * significant in SCSI-3.  For now, we follow the SCSI-2
154          * behaviour and ignore reserved bits.
155          */
156         status &= 0xfe;
157         return ((status == SAM_STAT_GOOD) ||
158                 (status == SAM_STAT_INTERMEDIATE) ||
159                 (status == SAM_STAT_INTERMEDIATE_CONDITION_MET) ||
160                 /* FIXME: this is obsolete in SAM-3 */
161                 (status == SAM_STAT_COMMAND_TERMINATED));
162 }
```

上面的那些宏被称为状态码, scsi_execute_req()的返回值就是这些状态码中的一个.而其中可以被认为是 good 的状态就是 scsi_status_is_good 函数中列出来的这四种,当然理论上来说最理想的就是 SAM_STAT_GOOD,而另外这几种也勉强算是可以接受,将就将就的让它过去.

不过有一点必须明白的是,the_result 和状态码还是有区别的,毕竟状态码只有那么多,用 8 位来表示足矣,而 the_result 我们看到是 unsigned int,显然它不只是 8 位,于是我们就充分利用资源,因此就有了下面这些宏,

```
358 /*
359  *  Use these to separate status msg and our bytes
360  *
361  *  These are set by:
362  *
363  *          status byte = set from target device
364  *          msg_byte    = return status from host adapter itself.
365  *          host_byte   = set by low-level driver to indicate status.
366  *          driver_byte = set by mid-level.
367  */
368 #define status_byte(result) (((result) >> 1) & 0x7f)
369 #define msg_byte(result)    (((result) >> 8) & 0xff)
370 #define host_byte(result)   (((result) >> 16) & 0xff)
371 #define driver_byte(result) (((result) >> 24) & 0xff)
372 #define suggestion(result)  (driver_byte(result) & SUGGEST_MASK)
```

也就是说除了最低的那个 byte 是作为 status byte 用,剩下的 byte 我们也没浪费,它们都被用来承载信息,其中 driver_byte,即 bit23 到 bit31,这 8 位被用来承载 mid-level 设置的信息.而这里用它和 DRIVER_SENSE 相与,则判断的是是否有 sense data,我们当初在 usb-storage 故事中就说过,scsi 世界里的 sense data 就是错误信息.这里 1025 行至 1048 行的这个 do-while 循环就是如果不成功就最多重复三次,循环结束了之后,1050 行再次判断有没有 sense data,如果没有,则说明也许成功了.

Scsi 子系统最无耻的地方就在于错误判断的代码特别的多.而针对 sense data 的处理则是错误判断的一部分.

```
 8 /*
 9  * This is a slightly modified SCSI sense "descriptor" format header.
10  * The addition is to allow the 0x70 and 0x71 response codes. The idea
11  * is to place the salient data from either "fixed" or "descriptor" sense
12  * format into one structure to ease application processing.
13  *
14  * The original sense buffer should be kept around for those cases
15  * in which more information is required (e.g. the LBA of a MEDIUM ERROR).
16  */
17 struct scsi_sense_hdr {             /* See SPC-3 section 4.5 */
18         u8 response_code;           /* permit: 0x0, 0x70, 0x71, 0x72, 0x73 */
19         u8 sense_key;
20         u8 asc;
21         u8 ascq;
22         u8 byte4;
23         u8 byte5;
24         u8 byte6;
25         u8 additional_length;    /* always 0 for fixed sense format */
```

```
26 };
27
28 static inline int scsi_sense_valid(struct scsi_sense_hdr *sshdr)
29 {
30              if (!sshdr)
31                           return 0;
32
33              return (sshdr->response_code & 0x70) == 0x70;
34 }
```

这里定义的 struct scsi_sense_hdr 就是被用来描述一个 sense data.”hdr”就是 header 的意思,因为 sense data 可能长度比较长,但是其前 8 个 bytes 是最重要的,所以这部分被叫做 header,或者说头部,大多数情况下只要理睬头部就够了.

我们看函数 scsi_execute_req()中第六个参数是 struct scsi_sense_hdr *sshdr,换言之,如果命令执行出错了,那么 sense data 就会通过这个参数返回.所以咱们定义了 sshdr,然后咱们通过判断它和它的各个成员,来决定下一步.

而 sense data 中,最基本的一个元素叫做 response_code,它相当于为一个 sense data 定了性,即它属于哪一个类别,因为 sense data 毕竟有很多种.response code 总共就是 8 个 bits,目前使用的值只有 70h,71h,72h,73h,其它的像 00h 到 6Fh 以及 74h 到 7Eh 这些都是保留的,以备将来之用.所以这里判断的就是 response code 得是 0x70,0x71,0x72,0x73 才是 valid,否则就是 invalid.这就是 scsi_sense_valid()做的事情.

关于 sense data,事实上,坊间一直流传着一本叫做 SCSI Primary Commands(SPC)的秘籍,在这本秘籍的第四章,确切的说是 4.5 节,名字就叫做 Sense data,即这一节是专门介绍 Sense Data 的.Sense data 中最有意义的东西叫做 sense key 和 sense code.这两个概念基本上确定了你这个错误究竟是什么错误.

1048 行,我们判断 sshdr 的 sense_key 是不是等于 UNIT_ATTENTION,这个信息表示这个设备可能被重置了或者可移动的介质发生了变化,或者更通俗一点说,只要设备发生了一些变化,然后它希望引起主机控制器的关注,比如说设备原本是 on-line 的,突然变成了 off-line,或者反过来,设备从 off-line 回到了 on-line.在正式读写设备之前,如果有 UNIT_ATTENTION 条件,必须把它给清除掉.而这(清除 UNIT ATTENTION)也正是 Test Unit Ready 的工作之一.

而如果 sense key 等于 NOT_READY,则表明这个 logical unit 不能被访问.(NOT READY: Indicates that the logical unit is not accessible.)而如果 sense key 等于 NOT READY,而 asc 等于 04h,ascq 等于 03h,这表明”Logical Unit Not Ready,Manual Intervention required”.(详见 SPC-4,附录 D 部分)这说明需要人工干预.

当然大多数情况下,应该执行的是 1079 行这个 else if 所包含的代码.即磁盘确实应该是 NOT_READY,于是我们需要发送下一个命令,即 START STOP,在另一部江湖武功秘籍名为 SCSI Block Commands-2(SBC-2)的书中,5.17 节专门介绍了 START STOP UNIT 这个命令.这个命令简而言之,就相当于电源开关,SBC-2 中 Table 48 给出了这个命令的格式:

Table 48 — START STOP UNIT command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (1Bh)							
1	Reserved							IMMED
2	Reserved							
3								
4	POWER CONDITION				Reserved		LOEJ	START
5	CONTROL							

结合代码看,咱们把 cmd[4]设置为 1,实际上就等于是把这张图里的 START 位设置为 1.而在 SBC-2 中,这个 START 位的含义如下:

If the START bit is set to zero, then the logical unit shall transition to the stopped power condition, disable the idle condition timer if it is active (see SPC-3), and disable the standby condition timer if it is active (see SPC-3). If the START bit set to one, then the logical unit shall transition to the active power condition, enable the idle condition timer if it is active, and enable the standby condition timer if it is active.

很明显,这就是真正的电源开关.因此,1086 行再次调用 scsi_execute_req 以执行 START STOP UNIT 命令,就是真正的让硬盘转起来.或者用郭富城的话说,动起来!

于是我们就很清楚从 1022 行直到 1120 行这一百行代码的 do-while 循环的意思了.其理想情况的流程就是:

1. 软件说:磁盘磁盘我问你,你准备好了没有?
2. 磁盘说:没有!
3. 软件说:磁盘磁盘你听着,你快给我转起来!
4. 软件:睡眠 1000 毫秒之后重复第一步的问题.(但磁盘这次可能走第二步,也可能走第五步.)
5. 磁盘说:是的,我准备好了,我们时刻准备着.
6. 这时,1057 行 break 语句会被执行,从而循环结束.sd_spinup_disk()函数也就结束了它的使命.
7. 在第一次走到第四步的时候,会设置 spintime_expire 为 100 秒,即这个时间为软件忍耐极限,磁盘你只要在 100 秒之内给我动起来,我就既往不咎,倘若给你 100 秒你还敬酒不吃吃罚酒,那就没办法了,while 循环自然结束,1126 行这个 printk 语句执行,告诉上级说,not responding,换言之,这厮没救了,整个一扶不起的阿斗.

三座大山(一)

好不容易结束了 sd_spinup_disk(),马上我们就遇到了三座大山.它们是 sd_read_capacity(),sd_read_write_protect_flag(),sd_read_cache_type(),要继续往下看,我们不得不先推翻这三座大山.旧的三座大山已经在毛主席的英明领导下成功推翻了,但是今天我们的人民却身处新三座大山的压迫之下,眼前这三个函数堪比臭名昭著的房改医改教改.要知道整个 sd.c 这个文件也不过是 1900 行,可是光这三个函数就占了 360 行,你不服不行啊!

第一座大山,sd_read_capacity.

```
1130 /*
```

```
1131 * read disk capacity
1132 */
1133 static void
1134 sd_read_capacity(struct scsi_disk *sdkp, unsigned char *buffer)
1135 {
1136         unsigned char cmd[16];
1137         int the_result, retries;
1138         int sector_size = 0;
1139         int longrc = 0;
1140         struct scsi_sense_hdr sshdr;
1141         int sense_valid = 0;
1142         struct scsi_device *sdp = sdkp->device;
1143
1144 repeat:
1145         retries = 3;
1146         do {
1147                 if (longrc) {
1148                         memset((void *) cmd, 0, 16);
1149                         cmd[0] = SERVICE_ACTION_IN;
1150                         cmd[1] = SAI_READ_CAPACITY_16;
1151                         cmd[13] = 12;
1152                         memset((void *) buffer, 0, 12);
1153                 } else {
1154                         cmd[0] = READ_CAPACITY;
1155                         memset((void *) &cmd[1], 0, 9);
1156                         memset((void *) buffer, 0, 8);
1157                 }
1158
1159                 the_result = scsi_execute_req(sdp, cmd, DMA_FROM_DEVICE,
1160                                               buffer, longrc ? 12 : 8, &sshdr,
1161                                               SD_TIMEOUT, SD_MAX_RETRIES);
1162
1163                 if (media_not_present(sdkp, &sshdr))
1164                         return;
1165
1166                 if (the_result)
1167                         sense_valid = scsi_sense_valid(&sshdr);
1168                 retries--;
1169
1170         } while (the_result && retries);
1171
1172         if (the_result && !longrc) {
1173                 sd_printk(KERN_NOTICE, sdkp, "READ CAPACITY failed\n");
1174                 sd_print_result(sdkp, the_result);
```

```
1175            if (driver_byte(the_result) & DRIVER_SENSE)
1176                    sd_print_sense_hdr(sdkp, &sshdr);
1177            else
1178                    sd_printk(KERN_NOTICE, sdkp, "Sense not available.\n");
1179
1180            /* Set dirty bit for removable devices if not ready -
1181             * sometimes drives will not report this properly. */
1182            if (sdp->removable &&
1183               sense_valid && sshdr.sense_key == NOT_READY)
1184                    sdp->changed = 1;
1185
1186            /* Either no media are present but the drive didn't tell us,
1187              or they are present but the read capacity command fails */
1188            /* sdkp->media_present = 0; -- not always correct */
1189            sdkp->capacity = 0; /* unknown mapped to zero - as usual */
1190
1191            return;
1192    } else if (the_result && longrc) {
1193            /* READ CAPACITY(16) has been failed */
1194            sd_printk(KERN_NOTICE, sdkp, "READ CAPACITY(16) failed\n");
1195            sd_print_result(sdkp, the_result);
1196            sd_printk(KERN_NOTICE, sdkp, "Use 0xffffffff as device size\n");
1197
1198            sdkp->capacity = 1 + (sector_t) 0xffffffff;
1199            goto got_data;
1200    }
1201
1202    if (!longrc) {
1203            sector_size = (buffer[4] << 24) |
1204                    (buffer[5] << 16) | (buffer[6] << 8) | buffer[7];
1205            if (buffer[0] == 0xff && buffer[1] == 0xff &&
1206               buffer[2] == 0xff && buffer[3] == 0xff) {
1207                    if(sizeof(sdkp->capacity) > 4) {
1208                            sd_printk(KERN_NOTICE, sdkp, "Very big device. "
1209                                    "Trying to use READ CAPACITY(16).\n");
1210                            longrc = 1;
1211                            goto repeat;
1212                    }
1213                    sd_printk(KERN_ERR, sdkp, "Too big for this kernel. Use "
1214                             "a kernel compiled with support for large "
1215                             "block devices.\n");
1216                    sdkp->capacity = 0;
1217                    goto got_data;
1218            }
```

```
1219            sdkp->capacity = 1 + (((sector_t)buffer[0] << 24) |
1220                    (buffer[1] << 16) |
1221                    (buffer[2] << 8) |
1222                    buffer[3]);
1223        } else {
1224            sdkp->capacity = 1 + (((u64)buffer[0] << 56) |
1225                    ((u64)buffer[1] << 48) |
1226                    ((u64)buffer[2] << 40) |
1227                    ((u64)buffer[3] << 32) |
1228                    ((sector_t)buffer[4] << 24) |
1229                    ((sector_t)buffer[5] << 16) |
1230                    ((sector_t)buffer[6] << 8) |
1231                    (sector_t)buffer[7]);
1232
1233            sector_size = (buffer[8] << 24) |
1234                    (buffer[9] << 16) | (buffer[10] << 8) | buffer[11];
1235        }
1236
1237        /* Some devices return the total number of sectors, not the
1238         * highest sector number. Make the necessary adjustment. */
1239        if (sdp->fix_capacity) {
1240            --sdkp->capacity;
1241
1242        /* Some devices have version which report the correct sizes
1243         * and others which do not. We guess size according to a heuristic
1244         * and err on the side of lowering the capacity. */
1245        } else {
1246            if (sdp->guess_capacity)
1247                    if (sdkp->capacity & 0x01) /* odd sizes are odd */
1248                            --sdkp->capacity;
1249        }
1250
1251 got_data:
1252        if (sector_size == 0) {
1253            sector_size = 512;
1254            sd_printk(KERN_NOTICE, sdkp, "Sector size 0 reported, "
1255                      "assuming 512.\n");
1256        }
1257
1258        if (sector_size != 512 &&
1259            sector_size != 1024 &&
1260            sector_size != 2048 &&
1261            sector_size != 4096 &&
1262            sector_size != 256) {
```

```
1263            sd_printk(KERN_NOTICE, sdkp, "Unsupported sector size %d.\n",
1264                    sector_size);
1265            /*
1266             * The user might want to re-format the drive with
1267             * a supported sectorsize. Once this happens, it
1268             * would be relatively trivial to set the thing up.
1269             * For this reason, we leave the thing in the table.
1270             */
1271            sdkp->capacity = 0;
1272            /*
1273             * set a bogus sector size so the normal read/write
1274             * logic in the block layer will eventually refuse any
1275             * request on this device without tripping over power
1276             * of two sector size assumptions
1277             */
1278            sector_size = 512;
1279        }
1280        {
1281            /*
1282             * The msdos fs needs to know the hardware sector size
1283             * So I have created this table. See ll_rw_blk.c
1284             * Jacques Gelinas (Jacques@solucorp.qc.ca)
1285             */
1286            int hard_sector = sector_size;
1287            sector_t sz = (sdkp->capacity/2) * (hard_sector/256);
1288            request_queue_t *queue = sdp->request_queue;
1289            sector_t mb = sz;
1290
1291            blk_queue_hardsect_size(queue, hard_sector);
1292            /* avoid 64-bit division on 32-bit platforms */
1293            sector_div(sz, 625);
1294            mb -= sz - 974;
1295            sector_div(mb, 1950);
1296
1297            sd_printk(KERN_NOTICE, sdkp,
1298                    "%llu %d-byte hardware sectors (%llu MB)\n",
1299                    (unsigned long long)sdkp->capacity,
1300                    hard_sector, (unsigned long long)mb);
1301        }
1302
1303        /* Rescale capacity to 512-byte units */
1304        if (sector_size == 4096)
1305            sdkp->capacity <<= 3;
1306        else if (sector_size == 2048)
```

```
1307            sdkp->capacity <<= 2;
1308        else if (sector_size == 1024)
1309            sdkp->capacity <<= 1;
1310        else if (sector_size == 256)
1311            sdkp->capacity >>= 1;
1312
1313        sdkp->device->sector_size = sector_size;
1314 }
```

洋洋洒洒200余行.简而言之,这个函数用一句话来表达就是知道这个磁盘的容量,或者专业一点说,发送 READ CAPACITY 命令.而熟悉 SCSI 命令集的兄弟们应该知道,很多 SCSI 命令都有至少两种版本,不同版本的命令格式会不一样,当然返回的信息量也不尽相同,比如 READ CAPACITY 命令就有 10 个字节的和 16 个字节的两个版本.在 SBC-2 的 5.10 节和 5.11 节分别介绍的是 READ CAPACITY(10) command 和 READ CAPACITY(16) command.后者比前者多一个保护信息.但是在我们读之前我们并不知道该用哪个命令,所以这里的基本思路就是先用短命令,如果失败了就试一下长命令,这就是 1211 行 goto repeat 的目的.在 goto repeat 之前 1210 行设置了 longrc 为 1.我们这里先给出来自 SBC-2 中对 READ CAPACITY 命令的格式定义:

Table 34 — READ CAPACITY (10) command

Byte\Bit	7	6	5	4	3	2	1	0
0	OPERATION CODE (25h)							
1	Reserved							Obsolete
2	(MSB)							
5	LOGICAL BLOCK ADDRESS							(LSB)
6	Reserved							
7								
8	Reserved							PMI
9	CONTROL							

我们可以用一个实例来描述这个命令,sg_readcap 可以手工发送 READ CAPACITY 命令.下面是针对我的一个号称 128M 的 U 盘发送这个命令的结果.

```
[root@localhost ~]# sg_readcap /dev/sdc
Read Capacity results:
   Last logical block address=257535 (0x3edff), Number of blocks=257536
   Logical block length=512 bytes
Hence:
   Device size: 131858432 bytes, 125.8 MiB, 0.13 GB
```

与此同时,我们结合代码来看,这个函数实际上比较麻烦的地方在于对 buffer 数组的判断.实际上 buffer 数组装载了 READ CAPACITY 命令的返回信息.而我们从 1203 行开始判断,首先我们知道这个 buffer 是我们在 sd_revalidate_disk()中申请的.其大小为 SD_BUF_SIZE,即 512 个字节.那么这个 buffer 的数据究竟是什么模样呢?SBC-2 中 Table-35 对 READ CAPACITY(10) 的返回数据给出了如图的格式,

Table 35 — READ CAPACITY (10) parameter data

Byte\Bit	7	6	5	4	3	2	1	0
0	(MSB)							
3				RETURNED LOGICAL BLOCK ADDRESS				(LSB)
4	(MSB)							
7				BLOCK LENGTH IN BYTES				(LSB)

这里 byte4,byte5,byte6,byte7 共同描述了 Block 的大小.即所谓的扇区大小,或者说代码中的 sector_size,大多数情况下我们看到的都是 512bytes.这里我的这个 U 盘当然也属于这种情况. RETURNED LOGICAL BLOCK ADDRESS 就是告诉你这个设备有多少个 Block,或者通俗点说,有多少个扇区.当然,更准确地说,如果你这个磁盘有 N 个 Block,那么这里返回的是最后一个 Block 的编号,因为编号是从 0 开始,所以最后一个 Block 的编号就是 N-1.所以这里返回的是 N-1.而 SBC-2 规定,倘若 byte0,byte1,byte2,byte3 如果全为 FF,那么说明 READ CAPACITY(10)不足以读取这块磁盘的容量.这有点类似于传说中的缓冲区溢出.这种情况下再判断一下,如果 sizeof(sdkp->capacity)确实大于 4,那么这里溢出了我们 goto repeat,改而发送 READ CAPACITY(16).实际上,因为 capacity 是 sector_t 类型的,而在 include/linux/types.h 中,sector_t 是这么定义的,

```
140 #ifdef CONFIG_LBD
141 typedef u64 sector_t;
142 #else
143 typedef unsigned long sector_t;
144 #endif
```

所以,sector_t 的 size 有可能是是大于 4 的,也有可能是等于 4 的.如果等于 4 那就没办法了.只能设置 capacity 为 0.我们没有办法记录下究竟有多少个扇区,那么我们大不了就不记录.(同时我们下面也可以看到几处我们设置了 capacity 为 0,其目的都是一样,只做力所能及的事情,而不是强人所难,毕竟强扭的瓜不甜.)

当然如果没有溢出,那么就执行 1219 行,设置 sdkp 的 capacity,刚才说了,它和 byte0,byte1,byte2,byte3 的共同作用的区别就是 N 和 N-1 的关系,所以这里我们看到需要加上 1.因此 sdkp->capacity 记录的就是磁盘有多少个扇区.

而 1223 行这个 else 这一段,就是针对长命令的 buffer 进行处理的,因为 SBC-2 规定了,长命令的返回结果是下面这幅图这样的:

Table 37 — READ CAPACITY (16) parameter data

Byte\Bit	7	6	5	4	3	2	1	0
0	(MSB)							
7				RETURNED LOGICAL BLOCK ADDRESS				(LSB)
8	(MSB)							
11				BLOCK LENGTH IN BYTES				(LSB)
12	Reserved						RTO_EN	PROT_EN
13				Reserved				
31								

可以看出,这次 byte0,byte1,…,byte7 这 8 个 byte 共同作用来表示了 Block 数.而 byte8,byte9,byte10,byte11 共同作用表示了 block 的大小,或者说扇区大小.

1239 行说的也就是 N 和 N-1 的那件事,有些设备不按常理出牌,它汇报的时候已经把那个 1

给包括进来了,所以这里咱们只能再减一,凡是有这种特殊需求的设备会设置 fix_capacity.
1245 行又是针对另外一些不按常理出牌的设备的应对措施.这个咱就飘过了.毕竟连磁盘的大小都要别人去猜这厂家也太无耻了.
1252 行,对于那些内向的设备,我们只能假设它们是遵守游戏规则的,我们假设它们的扇区大小是大众化的 512.
另一方面,1258 行这一段,众所周知,扇区大小总是 512,1024,2048,4096,最次的也是 256.除此之外的设备基本上就可以去参加设备残奥会了,没必要拿出来丢人现眼.
1280 行至 1301 行的目的在注释里说得很清楚,咱们可以飘过不理.只是需要注意 1291 行调用了 blk_queue_hardsect_size(),这个函数非常的短,就是一句话,即把一个 struct request_queue_t 指针的成员 hardsect_size 的值设置为这里的参数 hard_sector.还是那句话,基本上也就是设置成 512,毕竟这是绝对主流.如果你的设备非要显示一下 80 后的与众不同的个性,那我也没办法.只是庄子曾经曰过:”莫装吊,装吊遭狗咬!”
1304 行开始的这一段 if-else if,就是针对 sector_size 调整一下 capacity,因为 capacity 应该用来记录有多少个扇区,而我们希望在代码中统一使用 512 字节的扇区,(这也是 Linux 中的一贯规矩)所以这里需要按比例调整一下.即原本读出来是说有 100 个扇区,但是每个扇区比如是 4096 个字节,那么如果我们要以从软件角度来说以 512 字节进行访问,那么我就可以记录说这个磁盘有 800 个扇区.
最后,1313 行,把 sector_size 也记录在 sdkp 的成员 struct scsi_device 指针 device 的 sector_size 内.

三座大山(二)

第二座大山,sd_read_write_protect_flag.

```
1327 /*
1328  * read write protect setting, if possible - called only in sd_revalidate_disk()
1329  * called with buffer of length SD_BUF_SIZE
1330  */
1331 static void
1332 sd_read_write_protect_flag(struct scsi_disk *sdkp, unsigned char *buffer)
1333 {
1334         int res;
1335         struct scsi_device *sdp = sdkp->device;
1336         struct scsi_mode_data data;
1337
1338         set_disk_ro(sdkp->disk, 0);
1339         if (sdp->skip_ms_page_3f) {
1340                 sd_printk(KERN_NOTICE, sdkp, "Assuming Write Enabled\n");
1341                 return;
1342         }
1343
1344         if (sdp->use_192_bytes_for_3f) {
1345                 res = sd_do_mode_sense(sdp, 0, 0x3F, buffer, 192, &data, NULL);
```

```
1346            } else {
1347                    /*
1348                     * First attempt: ask for all pages (0x3F), but only 4 bytes.
1349                     * We have to start carefully: some devices hang if we ask
1350                     * for more than is available.
1351                     */
1352                    res = sd_do_mode_sense(sdp, 0, 0x3F, buffer, 4, &data, NULL);
1353
1354                    /*
1355                     * Second attempt: ask for page 0 When only page 0 is
1356                     * implemented, a request for page 3F may return Sense Key
1357                     * 5: Illegal Request, Sense Code 24: Invalid field in
1358                     * CDB.
1359                     */
1360                    if (!scsi_status_is_good(res))
1361                            res = sd_do_mode_sense(sdp, 0, 0, buffer, 4, &data,
NULL);
1362
1363                    /*
1364                     * Third attempt: ask 255 bytes, as we did earlier.
1365                     */
1366                    if (!scsi_status_is_good(res))
1367                            res = sd_do_mode_sense(sdp, 0, 0x3F, buffer, 255,
1368                                                   &data, NULL);
1369            }
1370
1371            if (!scsi_status_is_good(res)) {
1372                    sd_printk(KERN_WARNING, sdkp,
1373                              "Test WP failed, assume Write Enabled\n");
1374            } else {
1375                    sdkp->write_prot = ((data.device_specific & 0x80) != 0);
1376                    set_disk_ro(sdkp->disk, sdkp->write_prot);
1377                    sd_printk(KERN_NOTICE, sdkp, "Write Protect is %s\n",
1378                              sdkp->write_prot ? "on" : "off");
1379                    sd_printk(KERN_DEBUG, sdkp,
1380                              "Mode Sense: %02x %02x %02x %02x\n",
1381                              buffer[0], buffer[1], buffer[2], buffer[3]);
1382            }
1383 }
```

这个函数看似很长,其实有意义的就是一行,那就是 1376 行,调用 set_disk_ro()从而确定本磁盘是否是写保护的.

1338 行,set_disk_ro 就是设置磁盘只读,为 0 就是可读可写,为 1 才是设置为只读.但是咱们这只是软件意义上的作个记录而已,硬件上还得听磁盘自己的.所以我们通过下面一大段代码最终得到这一信息,最终在 1376 行再次设置.

那么如何得知写保护是否设置了呢?发送命令给设备,这个命令就是 MODE SENSE.MODE SENSE 这个命令的目的在于获得设备内部很多潜在的信息,这其中包括设备是否设置了写保护,当然还有更多 SCSI 特有的信息.只不过我们此时此刻只关注写保护设了没有.这些特性就像设备的天性一样,在它出生的时候就设置好了,当然有些天性也是可以改变的,就比如范冰冰,可能她生下来的时候长相平平,但是经过整容,变成了美女.又比如何丽秀,原本是男人,后来却变成了女人.而对于 SCSI 设备来说,很多特性可以改变,但是有些特性就不可以改变了,比如 medium type,即它属于哪种类型的设备,对于 SCSI Block 设备,其内部保存 MEDIUM TYPE 的这个 byte 一定是 00h.

在咱们的驱动中为了发送这个命令,还作了两次包装,先调用 sd_do_mode_sense().

```
1316 /* called with buffer of length 512 */
1317 static inline int
1318 sd_do_mode_sense(struct scsi_device *sdp, int dbd, int modepage,
1319                  unsigned char *buffer, int len, struct scsi_mode_data *data,
1320                  struct scsi_sense_hdr *sshdr)
1321 {
1322         return scsi_mode_sense(sdp, dbd, modepage, buffer, len,
1323                                SD_TIMEOUT, SD_MAX_RETRIES, data,
1324                                sshdr);
1325 }
```

而 sd_do_mode_sense 调用来自 scsi 核心层统一提供的 scsi_mode_sense(),关于后者我们就不详细介绍了,总之执行之后,结果就是保存在了 data 中,而 data 是 struct scsi_mode_data 结构体变量.

```
16 struct scsi_mode_data {
17         __u32   length;
18         __u16   block_descriptor_length;
19         __u8    medium_type;
20         __u8    device_specific;
21         __u8    header_length;
22         __u8    longlba:1;
23 };
```

这里每一个成员都在 scsi 协议中能够找到对应物.就比如刚才说得 medium_type,对于 SCSI 磁盘,它一定是 00h.这是没得商量的.

我们最终是在 1375 行作的判断,看 1375 行,为啥判断 device_specific 和 0x80 相与呢?SBC-2 中有一幅图描述了这个 Device Specific 的玩意儿.

Table 97 — DEVICE-SPECIFIC PARAMETER field for direct-access block devices

Bit	7	6	5	4	3	2	1	0
	WP	Reserved		DPOFUA	Reserved			

这里 bit7 叫做 WP,即 Write Protect,写保护位.N 年前当咱们刚开始用软盘的时候就听说了写保护,所以对这个概念我们并不陌生.如果这一位为 1 就说明设置了写保护,反之则是没有设置.

如果没有设置写保护,那么在日志文件里我们就能看到类似下面这行的一句话:

Dec 6 08:47:05 localhost kernel: sdb: Write Protect is off

因此, sd_read_write_protect_flag 这一个函数的流程就是:

1. 软件问:磁盘磁盘你设置了写保护吗?
2. 如果磁盘说:是的我设置了.
3. 软件打印: sdb: Write Protect is on
4. 如果磁盘说:不,我没有设置.
5. 软件打印: sdb: Write Protect is off

最后说一下,1344 行,判断有没有设置 use_192_bytes_for_3f,这是因为实践表明,很多磁盘只能接受 MODE SENSE 在 page=0x3f 时传输长度为 192bytes,所以咱们在定义 struct scsi_device 的时候为这些设备准备了这么一个 flag,在 scsi 总线扫描设备初始化的时候就可以设置这么一个 flag.相应的我们发送命令的时候就设置好 192.

另一个 1339 行,skip_ms_page_3f,这也是一个类似的 flag,MODE SENSE 命令有一个参数 page,同样是实践表明,某些愚蠢的设备在 page=0x3f 的时候会出错.所以写代码的做出让步,又准备了一个 flag.

如果你还不是很明白这个 page 是啥意思,那么让我们来看一下 SPC-4 中 MODE SENSE 命令的格式是如何的.

首先是 6 字节的.

Table 97 — MODE SENSE(6) command

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (1Ah)							
1	Reserved				DBD	Reserved		
2	PC		PAGE CODE					
3	SUBPAGE CODE							
4	ALLOCATION LENGTH							
5	CONTROL							

然后是 10 字节的.

Table 100 — MODE SENSE(10) command

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE (5Ah)							
1	Reserved			LLBAA	DBD	Reserved		
2	PC		PAGE CODE					
3	SUBPAGE CODE							
4 6	Reserved							
7	(MSB)							
8	ALLOCATION LENGTH							(LSB)
9	CONTROL							

这其中,PAGE CODE 就是我们上面说的 page,很明显,它一共占 6 个 bits.因此它的取值范围就是 00h 到 3Fh(即 11 1111).而我们上面说到 3fh,就是说当你发送 MODE SENSE 命令的时候,设置 PAGE CODE 为 3fh 的时候,因为 3fh 是最后一个 page,很多设备都会有一些莫名其妙的错误,搞得我们很没面子,于是我们需要设置种种 flag 来处理这些情况.

当然,你可能还想知道为什么需要 PAGE 这么一个概念.Ok,其实这样来的,众所周知,开源社区有很多寂寞男,但是很少有女人,毕竟亚里士多德曾经说过:”女人做程序,既毁了女人,也毁了程序.”而 SCSI 设计者作为同样是 IT 工作者,他们对开源社区的兄弟们也很同情,所以他们在

设计 SCSI 的时候一直希望把对开源社区兄弟们美好的祝愿寄托在设备中,他们想,开源社区缺女人,而我们经常说,女人就象一本书,(当然了,胖女人就象一本辞海.除了必要时,没有愿意去翻她.),于是他们在设备内部保存了一本书,这本书就是设备的<<我的自白书>>,或者用更加时尚的话说,这本书就是设备的性感写真集,而你要阅读这本书,你就必须发送 MODE SENSE 命令,但是就像你读别的书一样,你必须一页一页的读,所以你需要给定一个 PAGE CODE,或者说页码,同时我们看到 Byte3 叫做 SUBPAGE CODE,这就是子页号码,你索性就理解为一页中某一个段落好了,即设备允许你一页一页的读,也允许你一段一段的读.很显然,由于 SUBPAGE CODE 是 8 个 bits,因此其最大值就是 255.即一个 page 可以有最多 255 个 subpage.

三座大山(三)

接下来,第三座大山是 sd_read_cache_type.

```
1385 /*
1386  * sd_read_cache_type - called only from sd_revalidate_disk()
1387  * called with buffer of length SD_BUF_SIZE
1388  */
1389 static void
1390 sd_read_cache_type(struct scsi_disk *sdkp, unsigned char *buffer)
1391 {
1392         int len = 0, res;
1393         struct scsi_device *sdp = sdkp->device;
1394
1395         int dbd;
1396         int modepage;
1397         struct scsi_mode_data data;
1398         struct scsi_sense_hdr sshdr;
1399
1400         if (sdp->skip_ms_page_8)
1401                 goto defaults;
1402
1403         if (sdp->type == TYPE_RBC) {
1404                 modepage = 6;
1405                 dbd = 8;
1406         } else {
1407                 modepage = 8;
1408                 dbd = 0;
1409         }
1410
1411         /* cautiously ask */
1412         res = sd_do_mode_sense(sdp, dbd, modepage, buffer, 4, &data, &sshdr);
1413
```

```
1414            if (!scsi_status_is_good(res))
1415                    goto bad_sense;
1416
1417            if (!data.header_length) {
1418                    modepage = 6;
1419                    sd_printk(KERN_ERR, sdkp, "Missing header in MODE_SENSE response\n");
1420            }
1421
1422            /* that went OK, now ask for the proper length */
1423            len = data.length;
1424
1425            /*
1426              * We're only interested in the first three bytes, actually.
1427              * But the data cache page is defined for the first 20.
1428              */
1429            if (len < 3)
1430                    goto bad_sense;
1431            if (len > 20)
1432                    len = 20;
1433
1434            /* Take headers and block descriptors into account */
1435            len += data.header_length + data.block_descriptor_length;
1436            if (len > SD_BUF_SIZE)
1437                    goto bad_sense;
1438
1439            /* Get the data */
1440            res = sd_do_mode_sense(sdp, dbd, modepage, buffer, len, &data, &sshdr);
1441
1442            if (scsi_status_is_good(res)) {
1443                    int offset = data.header_length + data.block_descriptor_length;
1444
1445                    if (offset >= SD_BUF_SIZE - 2) {
1446                            sd_printk(KERN_ERR, sdkp, "Malformed MODE SENSE response\n");
1447                            goto defaults;
1448                    }
1449
1450                    if ((buffer[offset] & 0x3f) != modepage) {
1451                            sd_printk(KERN_ERR, sdkp, "Got wrong page\n");
1452                            goto defaults;
1453                    }
1454
1455                    if (modepage == 8) {
```

```
1456                            sdkp->WCE = ((buffer[offset + 2] & 0x04) != 0);
1457                            sdkp->RCD = ((buffer[offset + 2] & 0x01) != 0);
1458                    } else {
1459                            sdkp->WCE = ((buffer[offset + 2] & 0x01) == 0);
1460                            sdkp->RCD = 0;
1461                    }
1462
1463                    sdkp->DPOFUA = (data.device_specific & 0x10) != 0;
1464                    if (sdkp->DPOFUA && !sdkp->device->use_10_for_rw) {
1465                            sd_printk(KERN_NOTICE, sdkp,
1466                                      "Uses READ/WRITE(6), disabling FUA\n");
1467                            sdkp->DPOFUA = 0;
1468                    }
1469
1470                    sd_printk(KERN_NOTICE, sdkp,
1471                            "Write cache: %s, read cache: %s, %s\n",
1472                            sdkp->WCE ? "enabled" : "disabled",
1473                            sdkp->RCD ? "disabled" : "enabled",
1474                            sdkp->DPOFUA ? "supports DPO and FUA"
1475                            : "doesn't support DPO or FUA");
1476
1477                    return;
1478            }
1479
1480 bad_sense:
1481            if (scsi_sense_valid(&sshdr) &&
1482                sshdr.sense_key == ILLEGAL_REQUEST &&
1483                sshdr.asc == 0x24 && sshdr.ascq == 0x0)
1484                    /* Invalid field in CDB */
1485                    sd_printk(KERN_NOTICE, sdkp, "Cache data unavailable\n");
1486            else
1487                    sd_printk(KERN_ERR, sdkp, "Asking for cache data failed\n");
1488
1489 defaults:
1490            sd_printk(KERN_ERR, sdkp, "Assuming drive cache: write through\n");
1491            sdkp->WCE = 0;
1492            sdkp->RCD = 0;
1493            sdkp->DPOFUA = 0;
1494 }
```

很显然,这个函数最主要的工作还是调用 sd_do_mode_sense,即还是发送 MODE SENSE 命令.我们前面说过,SCSI 设备写真集最多就是 64 页(64=0x3f+1).而这里我们给 modepage 赋值为 8,或者对于 RBC,赋值为 6.这是为什么呢?首先我们必须明确,我们眼下的目的是读取设备写真集中关于 Cache 的信息,事实上每个 SCSI 磁盘,或者更有专业精神的说法,每一个 Direct-access block device,都可以实现 caches,通过使用 cache 可以提高设备的性能,比如可以

减少访问时间,比如可以增加数据吞吐量.而在 SBC-2 中,为 SCSI 磁盘定义了一个 Mode Page 专门用来描述和 cache 相关的信息.我们可以从下面这张表中看到,

Table 98 — Mode page codes for direct-access block devices

Mode page code	Description	Reference
00h	Vendor-specific (does not require page format)	
00h-3Eh/FFh	Return all subpages [a]	SPC-3
01h	Read-Write Error Recovery mode page	6.3.4
02h	Disconnect-Reconnect mode page	SPC-3
03h	Obsolete (Format Device mode page)	
04h	Obsolete (Rigid Disk Geometry mode page)	
05h	Obsolete (Flexible Disk mode page)	
06h	Reserved	
07h	Verify Error Recovery mode page	6.3.5
08h	Caching mode page	6.3.3
09h	Obsolete	
0Ah/00h	Control mode page	SPC-3
0Ah/01h	Control Extension mode page	SPC-3
0Ah/02h - 3Eh	Reserved	
0Bh	Obsolete (Medium Types Supported mode page)	
0Ch	Obsolete (Notch And Partition mode page)	
0Dh	Obsolete	
0Eh - 0Fh	Reserved	
10h	XOR Control mode page	6.3.6
11h - 13h	Reserved	
14h	Enclosure Services Management mode page [b]	SES-2
15h - 17h	Reserved	
18h	Protocol-Specific LUN mode page	SPC-3
19h	Protocol-Specific Port mode page	SPC-3
1Ah	Power Condition mode page	SPC-3
1Bh	Reserved	
1Ch	Informational Exceptions Control mode page	SPC-3
1Dh - 1Fh	Reserved	
20h - 3Eh	Vendor-specific (does not require page format)	
3Fh/00h	Return all mode pages [a]	SPC-3
3Fh/01h - 3Eh	Reserved	
3Fh/FFh	Return all mode pages and subpages [a]	SPC-3

[a] Valid only for the MODE SENSE command
[b] Valid only if the ENCSERV bit is set to one in the standard INQUIRY data (see SPC-3)

08h 这个 Page,被叫做 caching mode page,这一个 Page 就是我们需要的.这也就是为什么我们赋值 modepage 为 8.而对于遵循 RBC 协议的设备这个值会是 6,这个我们不去理睬.
下面我们需要理解两个东西.一个是这个 Caching Mode page 究竟长什么样.另一个是这里我

们看到的 1443 行定义的 offset 到底表示什么意思?
先看第二个问题.SPC-4 中的 Table238 定义了 MODE SENSE 命令的返回值的格式:

Table 238 — Mode parameter list

Bit / Byte	7	6	5	4	3	2	1	0
	Mode parameter header							
	Block descriptor(s)							
	Mode page(s) or vendor specific (e.g., page code set to zero)							

可以看到这个命令返回值一共有三部分,即 Mode Parameter Header,Block Descriptor,Mode Page(s).而 Mode Page 出现在第三部分.比如我们这里点名要 Mode Page 8,那么它就出现在这里的第三部分.首先我们所有的返回值都保存在 buffer[]数组中,如果我们要访问 Mode Pages 这一部分,我们就必须知道前面两个部分的长度.假设前面两个部分的长度为 offset,那么我们要访问第三部分就可以使用 buffer[offset],这样我们就知道这个 offset 的含义了.那么前两部分究竟有多长呢?换言之这个 offset 究竟是多少?
我们先看第一部分是如何定义的,对于 6 字节的 MODE SENSE,

Table 239 — Mode parameter header(6)

Bit / Byte	7	6	5	4	3	2	1	0
0	MODE DATA LENGTH							
1	MEDIUM TYPE							
2	DEVICE-SPECIFIC PARAMETER							
3	BLOCK DESCRIPTOR LENGTH							

而对于 10 字节的 MODE SENSE 命令,这部分稍微复杂些.

Table 240 — Mode parameter header(10)

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1	MODE DATA LENGTH							(LSB)
2	MEDIUM TYPE							
3	DEVICE-SPECIFIC PARAMETER							
4	Reserved							LONGLBA
5	Reserved							
6	(MSB)							
7	BLOCK DESCRIPTOR LENGTH							(LSB)

如果你深入 scsi_mode_sense()函数,你会发现,其实 data.header_length 恰恰就是这个 Mode Parameter Header 的长度,而 data.block_descriptor_length 恰恰就是第二部分的长度,即 Block Descriptor 的长度.这就是为什么我们会在 1443 行令 offset 等于这俩之和.
于是我们用 buffer[offset]就定位到了 Mode Page 这一部分,但是 Mode Page 具体长什么样呢?或者更直接一点,Caching Mode Page 长什么样?让 SBC-2 的 Table 101 来告诉你.

Table 101 — Caching mode page

Byte\Bit	7	6	5	4	3	2	1	0
0	PS	Reserved	PAGE CODE (08h)					
1	PAGE LENGTH (12h)							
2	IC	ABPF	CAP	DISC	SIZE	WCE	MF	RCD
3	DEMAND READ RETENTION PRIORITY				WRITE RETENTION PRIORITY			
4	(MSB)	DISABLE PRE-FETCH TRANSFER LENGTH						
5								(LSB)
6	(MSB)	MINIMUM PRE-FETCH						
7								(LSB)
8	(MSB)	MAXIMUM PRE-FETCH						
9								(LSB)
10	(MSB)	MAXIMUM PRE-FETCH CEILING						
11								(LSB)
12	FSW	LBCSS	DRA	Vendor specific		Reserved		NV_DIS
13	NUMBER OF CACHE SEGMENTS							
14	(MSB)	CACHE SEGMENT SIZE						
15								(LSB)
16	Reserved							
17	Obsolete							
19								

我们看到这个 Page 一共有 19 个 bytes,而我们知道 buffer[offset]就应该对应它的 Byte0.而这里我们看到 Byte0 的 bit0 到 bit5 代表的就是 PAGE CODE,即对于 caching mode page 来说它应该是 08h,这就是为什么我们在 1450 行要取 buffer[offset]的低 6 位来判断它是否是我们期待的那个 08h.如果不是,就说明错了.事实上,任何一个 Mode Page 的这 6 位表示的都是 Page Code.这个位置就相当于我们在校期间的学号.分辨一个人是不是你要找的人,你可以通过学号去判别.

而接下来,再次根据 modepage 是 8 还是 6 来做不同的赋值,我们还是只看主流的情况,即考虑 modepage 为 8 的情况.buffer[offset+2]就是这里的 Byte2.很明显对照这张图来看,我们要的是这里的 WCE 和 RCD 这两个 bits,看看它们是 1 还是 0.那么这两位的含义是什么呢?SBC-2 中对这两位是这样描述的.

A writeback cache enable (WCE) bit set to zero specifies that the device server shall return GOOD status for a WRITE command only after successfully writing all of the data to the medium. A WCE bit set to one specifies that the device server may return GOOD status for a WRITE command after successfully receiving the data and prior to having successfully written it to the medium.

A read cache disable (RCD) bit set to zero specifies that the device server may return data requested by a READ command by accessing either the cache or medium. A RCD bit set to one specifies that the device server shall transfer all of the data requested by a READ command from the medium (i.e., data shall not be transferred from the cache).

很显然以上这些词汇基本上都是我们九年制义务教育中学过的.唯一一个例外也许就是 cache,记下这两个 bit 的值对我们之后的工作有用,所以我们费尽周折处心积虑不择手段翻山

越岭跋山涉水就是要得到这两个 bit 的值.WCE 为 1 说明我们在写操作的时候可以启用 cache,即只要写入数据到了 cache 中就先返回成功,而不用等到真正写到介质中以后再返回.RCD 为 1 则说明我们读数据的时候必须从介质中读,而不是从 cache 中读.

最后,一个叫做 DPOFUA 的 bit 也是需要我们牢记心中的.看到这个东西来自 data.device_specific,这个东西就是前面那幅 Mode Parameter Header 中的 DEVICE SPECIFIC PARAMETER,对于遵循 SBC-2 的设备,这一项的格式也是专门有定义的:

Table 97 — DEVICE-SPECIFIC PARAMETER field for direct-access block devices

Bit	7	6	5	4	3	2	1	0
	WP	Reserved		DPOFUA	Reserved			

其实这幅图我们似曾相识.之前我们就是通过这个 WP 位的了解设备是否设置了写保护的.而这里 bit4 叫做 DPOFUA,这一个 Bit 如果为 1.说明设备支持 DPO 和 FUA bits,如果为 0,说明设备并不支持 DPO 和 FUA bits.DPO 是 disable page out 的缩写,FUA 是 force unit access 的缩写.

我知道,一味的复制粘贴是一件很无耻的事情,但是你也不用对我要求太高,因为现在本来就是一个道德沦丧的社会.在这个社会里,偷一个人的主意是剽窃,偷很多人的主意就是研究,所以我只好时而剽窃,时而研究.

When the cache becomes full of logical blocks, new logical blocks may replace those currently in the cache. The disable page out (DPO) bit in the CDB of commands performing write, read, or verify operations allows the application client to influence the replacement of logical blocks in the cache. For write operations, setting the DPO bit to one specifies that the device server should not replace existing logical blocks in the cache with the new logical blocks being written. For read and verify operations, setting the DPO bit to one specifies that the device server should not replace logical blocks in the cache with the logical blocks that are being read.

Application clients may use the force unit access (FUA) bit in the CDB of commands performing write or read operations to specify that the device server shall access the medium. For a write operation, setting the FUA bit to one causes the device server to complete the data write to the medium before completing the command. For a read operation, setting the FUA bit to one causes the device server to retrieve the logical blocks from the medium rather than from the cache.

When the DPO and FUA bits are both set to one, write and read operations effectively bypass the cache.

以上引文来自 SBC-2 的 4.10,专门介绍 Cache 的一节.如果你的英文和我一样优秀,没有怎么作弊就通过了四六级,那么我建议你认真阅读一下以上的文字,多学习英文有好处,至少等你往某师范学院学报投稿<<为人民服务与党的先进性>>的时候,不会像某作者一样把英文标题取为”Behave the people’s advanced sex of the service and party”,等你向某省经济管理干部学院学报发表<<开拓进取真抓实干不断开创西部大开发的新局面>>的时候,不会像某人把真抓实干翻译成 Really Grasp Solid Fuck.

而如果你不想认真看,那么一句话总结,上面这段话说的就是如果你的 DPO 和 FUA bits 被设置成了 1,那么你的读写操作都不会使用 cache.因为 DPOFUA 是这里的 bit4,所以我们看到 1463 行,data.device_specific 就是和 0x10 相与,广州飞车党的兄弟们都知道,这样得到的就是 bit4.

不过,和 MODE SENSE 命令一样,READ/WRITE 命令也有 6 字节和 10 字节的,对于 READ/WRITE 操作,默认情况下咱们会先尝试使用 10 字节的.但是咱们也允许你违反游戏规

则.struct scsi_device 结构体中 unsigned use_10_for_rw,就是你可以设置的.默认情况下,咱们会在设备初始化的时候,确切的说,在 scsi 总线扫描的时候,scsi_add_lun 函数中会把这个 flag 设置为 1.但如果你偏要特立独行,那也随便你.但是实际上 6 字节的 READ/Write 命令中没有定义 FUA,DPO,所以这里我们需要设置 DPOFUA 为 0.
最后,带着 sdkp->WCE,sdkp->RCD,sdkp->DPOFUA,sd_read_cache_type()函数满载而归.
翻过了三座大山,我们回到了 sd_revalidate_disk.

从应用层走来的 ioctl

2007 年过去了,这一年里明星们一如既往,大牌们继续做领军人物,而希望上位的小辈也使尽手段.该恋爱的恋爱,该炒作的炒作,该整容的整容.功成名就的就做慈善,有待提高的就造绯闻.而我该做的,是继续写我的 blog,继续说 Linux 那些鸟事儿,继续说那些无聊的函数,在说完了 sd_probe 之后,我们要接触一些新的函数了,首先推出的是 ioctl,具体到 sd 模块中就是 sd_ioctl.当我们向 scsi 磁盘发送命令的时候,这个函数多半会被调用.现在让我们用 kdb 来演示一下:
先在 kdb 中设置断点,sd_ioctl,

```
Entering kdb (current=0xffff81022adcf140, pid 4074) on processor 6 due to KDB_ENTER()
[6]kdb> bp sd_ioctl
Instruction(i) BP #0 at 0xffffffff880b1de2 ([sd_mod]sd_ioctl)
    is enabled globally adjust 1
[6]kdb> go
```

然后执行一个 scsi 命令,比如 INQUIRY 命令.

```
[root@lfg2 tedkdb]# sg_inq /dev/sdg
```

不用说,电光火石之间 kdb 提示符就打印了出来,

```
Entering kdb (current=0xffff81022cdae760, pid 4095) on processor 5 due to Breakpoint @ 0xffffffff880b1de2
[5]kdb> bt
Stack traceback for pid 4095
0xffff81022cdae760        4095        4044  1     5     R  0xffff81022cdaea40 *sg_inq
rsp                       rip                 Function (args)
 ======================== <debug>
0xffff81022f619fd8 0xffffffff880b1de2 [sd_mod]sd_ioctl
 ======================== <normal>
0xffff81022d7f7d40 0xffffffff803101a0 blkdev_driver_ioctl+0x63
0xffff81022d7f7d80 0xffffffff80310810 blkdev_ioctl+0x65b
0xffff81022d7f7da0 0xffffffff80278825 __handle_mm_fault+0x6d3
0xffff81022d7f7db0 0xffffffff8029a1ea may_open+0x65
0xffff81022d7f7e10 0xffffffff80322a48 __up_read+0x7a
0xffff81022d7f7e40 0xffffffff80249b4f up_read+0x9
0xffff81022d7f7e50 0xffffffff80482d6d do_page_fault+0x48a
0xffff81022d7f7e60 0xffffffff8027acff __vma_link+0x52
0xffff81022d7f7ea0 0xffffffff802b58d5 block_ioctl+0x1b
0xffff81022d7f7eb0 0xffffffff8029da98 do_ioctl+0x2c
```

```
0xffff81022d7f7ee0 0xffffffff8029dd6d vfs_ioctl+0x247
0xffff81022d7f7f30 0xffffffff8029dde9 sys_ioctl+0x5f
0xffff81022d7f7f80 0xffffffff80209efc tracesys+0xdc
[5]kdb>
```

实际上走的路径是一个系统调用 ioctl,或者说从 sys_ioctl 进来的,最终走到了 sd_ioctl.
关于 sd_ioctl,来自 drivers/scsi/sd.c:

```
641 /**
642  *      sd_ioctl - process an ioctl
643  *      @inode: only i_rdev/i_bdev members may be used
644  *      @filp: only f_mode and f_flags may be used
645  *      @cmd: ioctl command number
646  *      @arg: this is third argument given to ioctl(2) system call.
647  *      Often contains a pointer.
648  *
649  *      Returns 0 if successful (some ioctls return postive numbers on
650  *      success as well). Returns a negated errno value in case of error.
651  *
652  *      Note: most ioctls are forward onto the block subsystem or further
653  *      down in the scsi subsytem.
654  **/
655 static int sd_ioctl(struct inode * inode, struct file * filp,
656                     unsigned int cmd, unsigned long arg)
657 {
658         struct block_device *bdev = inode->i_bdev;
659         struct gendisk *disk = bdev->bd_disk;
660         struct scsi_device *sdp = scsi_disk(disk)->device;
661         void __user *p = (void __user *)arg;
662         int error;
663
664         SCSI_LOG_IOCTL(1, printk("sd_ioctl: disk=%s, cmd=0x%x\n",
665                                  disk->disk_name, cmd));
666
667         /*
668          * If we are in the middle of error recovery, don't let anyone
669          * else try and use this device.    Also, if error recovery fails, it
670          * may try and take the device offline, in which case all further
671          * access to the device is prohibited.
672          */
673         error = scsi_nonblockable_ioctl(sdp, cmd, p, filp);
674         if (!scsi_block_when_processing_errors(sdp) || !error)
675                 return error;
676
677         /*
678          * Send SCSI addressing ioctls directly to mid level, send other
```

```
679              * ioctls to block level and then onto mid level if they can't be
680          * resolved.
681              */
682             switch (cmd) {
683                     case SCSI_IOCTL_GET_IDLUN:
684                     case SCSI_IOCTL_GET_BUS_NUMBER:
685                             return scsi_ioctl(sdp, cmd, p);
686                     default:
687                             error = scsi_cmd_ioctl(filp, disk, cmd, p);
688                             if (error != -ENOTTY)
689                                     return error;
690             }
691             return scsi_ioctl(sdp, cmd, p);
692 }
```

继续设置断点,scsi_cmd_ioctl 和 scsi_ioctl,会发现,调用的是 scsi_cmd_ioctl.

```
Instruction(i) breakpoint #2 at 0xffffffff802ee554 (adjusted)
0xffffffff802ee554 scsi_cmd_ioctl:              int3

Entering kdb (current=0xffff81022e1a01c0, pid 3583) due to Breakpoint @ 0xffffffff802ee554
kdb> bt
Stack traceback for pid 3583
0xffff81022e1a01c0       3583       3425   1      0     R    0xffff81022e1a0490 *sg_inq
rsp                      rip                Function (args)
 ======================== <debug>
0xffffffff805dafa0 0xffffffff88080dcc [sd_mod]sd_ioctl
0xffffffff805dafd8 0xffffffff802ee554 scsi_cmd_ioctl
 ======================== <normal>
0xffff8102112d7d70 0xffffffff88080e68 [sd_mod]sd_ioctl+0x9c
0xffff8102112d7db0 0xffffffff802ec9b2 blkdev_driver_ioctl+0x3a
0xffff8102112d7dc0 0xffffffff802ed00d blkdev_ioctl+0x627
0xffff8102112d7e40 0xffffffff80239e1b up_read+0x9
0xffff8102112d7e50 0xffffffff80435872 do_page_fault+0x48a
0xffff8102112d7ea0 0xffffffff802641f6 vma_link+0x3a
0xffff8102112d7ee0 0xffffffff80295155 block_ioctl+0x1b
0xffff8102112d7ef0 0xffffffff8027ec3b do_ioctl+0x1b
0xffff8102112d7f00 0xffffffff8027ee7e vfs_ioctl+0x20e
0xffff8102112d7f30 0xffffffff8027eeef sys_ioctl+0x5f
0xffff8102112d7f80 0xffffffff80209c8c tracesys+0xdc
```

其中的那来自 block/scsi_ioctl.c

```
520 int scsi_cmd_ioctl(struct file *file, struct gendisk *bd_disk, unsigned int cmd, void
__user *arg)
521 {
522             request_queue_t *q;
523             int err;
```

```
524
525             q = bd_disk->queue;
526             if (!q)
527                     return -ENXIO;
528
529             if (blk_get_queue(q))
530                     return -ENXIO;
531
532             switch (cmd) {
533                     /*
534                      * new sgv3 interface
535                      */
536                     case SG_GET_VERSION_NUM:
537                             err = sg_get_version(arg);
538                             break;
539                     case SCSI_IOCTL_GET_IDLUN:
540                             err = scsi_get_idlun(q, arg);
541                             break;
542                     case SCSI_IOCTL_GET_BUS_NUMBER:
543                             err = scsi_get_bus(q, arg);
544                             break;
545                     case SG_SET_TIMEOUT:
546                             err = sg_set_timeout(q, arg);
547                             break;
548                     case SG_GET_TIMEOUT:
549                             err = sg_get_timeout(q);
550                             break;
551                     case SG_GET_RESERVED_SIZE:
552                             err = sg_get_reserved_size(q, arg);
553                             break;
554                 case SG_SET_RESERVED_SIZE:
555                             err = sg_set_reserved_size(q, arg);
556                             break;
557                     case SG_EMULATED_HOST:
558                             err = sg_emulated_host(q, arg);
559                             break;
560                     case SG_IO: {
561                             struct sg_io_hdr hdr;
562
563                             err = -EFAULT;
564                             if (copy_from_user(&hdr, arg, sizeof(hdr)))
565                                     break;
566                             err = sg_io(file, q, bd_disk, &hdr);
567                             if (err == -EFAULT)
```

```
568                                     break;
569
570                             if (copy_to_user(arg, &hdr, sizeof(hdr)))
571                                     err = -EFAULT;
572                             break;
573                     }
574                     case CDROM_SEND_PACKET: {
575                             struct cdrom_generic_command cgc;
576                             struct sg_io_hdr hdr;
577
578                             err = -EFAULT;
579                             if (copy_from_user(&cgc, arg, sizeof(cgc)))
580                                     break;
581                             cgc.timeout = clock_t_to_jiffies(cgc.timeout);
582                             memset(&hdr, 0, sizeof(hdr));
583                             hdr.interface_id = 'S';
584                             hdr.cmd_len = sizeof(cgc.cmd);
585                             hdr.dxfer_len = cgc.buflen;
586                             err = 0;
587                             switch (cgc.data_direction) {
588                                     case CGC_DATA_UNKNOWN:
589                                                     hdr.dxfer_direction =
SG_DXFER_UNKNOWN;
590                                             break;
591                                     case CGC_DATA_WRITE:
592                                                     hdr.dxfer_direction =
SG_DXFER_TO_DEV;
593                                             break;
594                                     case CGC_DATA_READ:
595                                                     hdr.dxfer_direction =
SG_DXFER_FROM_DEV;
596                                             break;
597                                     case CGC_DATA_NONE:
598                                                     hdr.dxfer_direction =
SG_DXFER_NONE;
599                                             break;
600                                     default:
601                                             err = -EINVAL;
602                             }
603                             if (err)
604                                     break;
605
606                             hdr.dxferp = cgc.buffer;
607                             hdr.sbp = cgc.sense;
```

```
608                             if (hdr.sbp)
609                                     hdr.mx_sb_len = sizeof(struct request_sense);
610                             hdr.timeout = cgc.timeout;
611                              hdr.cmdp = ((struct cdrom_generic_command __user*)
arg)->cmd;
612                             hdr.cmd_len = sizeof(cgc.cmd);
613
614                             err = sg_io(file, q, bd_disk, &hdr);
615                             if (err == -EFAULT)
616                                     break;
617
618                             if (hdr.status)
619                                     err = -EIO;
620
621                             cgc.stat = err;
622                             cgc.buflen = hdr.resid;
623                             if (copy_to_user(arg, &cgc, sizeof(cgc)))
624                                     err = -EFAULT;
625
626                             break;
627                     }
628
629                     /*
630                      * old junk scsi send command ioctl
631                      */
632                     case SCSI_IOCTL_SEND_COMMAND:
633                             printk(KERN_WARNING "program %s is using a
deprecated SCSI ioctl, please convert it to SG_IO\              n", current->comm);
634                             err = -EINVAL;
635                             if (!arg)
636                                     break;
637
638                             err = sg_scsi_ioctl(file, q, bd_disk, arg);
639                             break;
640                     case CDROMCLOSETRAY:
641                             err = blk_send_start_stop(q, bd_disk, 0x03);
642                             break;
643                     case CDROMEJECT:
644                             err = blk_send_start_stop(q, bd_disk, 0x02);
645                             break;
646                     default:
647                             err = -ENOTTY;
648             }
649
```

```
650             blk_put_queue(q);
651             return err;
652 }
```

通过 kdb 的跟踪,你会发现传递进来的是 cmd 是 SG_IO.换言之,switch-case 这一大段最终会定格在 560 行这个 SG_IO 里.那么这里涉及到一个结构体就很重要了,来自 include/scsi/sg.h 中的 sg_io_hdr.

```
83 typedef struct sg_io_hdr
84 {
85      int interface_id;               /* [i] 'S' for SCSI generic (required) */
86      int dxfer_direction;            /* [i] data transfer direction    */
87      unsigned char cmd_len;          /* [i] SCSI command length ( <= 16 bytes) */
88      unsigned char mx_sb_len;        /* [i] max length to write to sbp */
89      unsigned short iovec_count; /* [i] 0 implies no scatter gather */
90      unsigned int dxfer_len;         /* [i] byte count of data transfer */
91      void __user *dxferp;            /* [i], [*io] points to data transfer memory
92                                                      or scatter gather list */
93      unsigned char __user *cmdp; /* [i], [*i] points to command to perform */
94      void __user *sbp;               /* [i], [*o] points to sense_buffer memory */
95      unsigned int timeout;           /* [i] MAX_UINT->no timeout (unit: millisec) */
96      unsigned int flags;             /* [i] 0 -> default, see SG_FLAG... */
97      int pack_id;                    /* [i->o] unused internally (normally) */
98      void __user * usr_ptr;          /* [i->o] unused internally */
99      unsigned char status;           /* [o] scsi status */
100     unsigned char masked_status;/* [o] shifted, masked scsi status */
101     unsigned char msg_status;       /* [o] messaging level data (optional) */
102     unsigned char sb_len_wr;        /* [o] byte count actually written to sbp */
103     unsigned short host_status; /* [o] errors from host adapter */
104     unsigned short driver_status;/* [o] errors from software driver */
105     int resid;                      /* [o] dxfer_len - actual_transferred */
106     unsigned int duration;          /* [o] time taken by cmd (unit: millisec) */
107     unsigned int info;              /* [o] auxiliary information */
108 } sg_io_hdr_t;    /* 64 bytes long (on i386) */
```

这其中,cmdp 指针指向的不是别人,正是命令本身也.

有过 Linux 下编程经验的人一定不会对 copy_from_user 和 copy_to_user 这两个骨灰级的函数陌生吧,内核空间和用户空间传递数据的两个经典函数,更远咱就不说了,当年 2.4 内核那会儿课堂上老师让写一个简单的字符驱动的时候就用上这两个函数了.而这里具体来说,先是把 arg 里的玩意儿 copy 到 hdr 中,然后调用 sg_io 完成实质性的工作,然后再把 hdr 里的内容 copy 回到 arg 中.

进一步跟踪 sg_io,来自 block/scsi_ioctl.c:

```
225 static int sg_io(struct file *file, request_queue_t *q,
226                         struct gendisk *bd_disk, struct sg_io_hdr *hdr)
227 {
228             unsigned long start_time, timeout;
229             int writing = 0, ret = 0;
```

```
230         struct request *rq;
231         char sense[SCSI_SENSE_BUFFERSIZE];
232         unsigned char cmd[BLK_MAX_CDB];
233         struct bio *bio;
234
235         if (hdr->interface_id != 'S')
236                 return -EINVAL;
237         if (hdr->cmd_len > BLK_MAX_CDB)
238                 return -EINVAL;
239         if (copy_from_user(cmd, hdr->cmdp, hdr->cmd_len))
240                 return -EFAULT;
241         if (verify_command(file, cmd))
242                 return -EPERM;
243
244         if (hdr->dxfer_len > (q->max_hw_sectors << 9))
245                 return -EIO;
246
247         if (hdr->dxfer_len)
248                 switch (hdr->dxfer_direction) {
249                 default:
250                         return -EINVAL;
251                 case SG_DXFER_TO_DEV:
252                         writing = 1;
253                         break;
254                 case SG_DXFER_TO_FROM_DEV:
255                 case SG_DXFER_FROM_DEV:
256                         break;
257                 }
258
259         rq = blk_get_request(q, writing ? WRITE : READ, GFP_KERNEL);
260         if (!rq)
261                 return -ENOMEM;
262
263         /*
264          * fill in request structure
265          */
266         rq->cmd_len = hdr->cmd_len;
267         memset(rq->cmd, 0, BLK_MAX_CDB); /* ATAPI hates garbage after CDB */
268         memcpy(rq->cmd, cmd, hdr->cmd_len);
269
270         memset(sense, 0, sizeof(sense));
271         rq->sense = sense;
272         rq->sense_len = 0;
273
```

```
274             rq->cmd_type = REQ_TYPE_BLOCK_PC;
275
276             timeout = msecs_to_jiffies(hdr->timeout);
277             rq->timeout = (timeout < INT_MAX) ? timeout : INT_MAX;
278             if (!rq->timeout)
279                     rq->timeout = q->sg_timeout;
280             if (!rq->timeout)
281                     rq->timeout = BLK_DEFAULT_TIMEOUT;
282
283             if (hdr->iovec_count) {
284                     const int size = sizeof(struct sg_iovec) * hdr->iovec_count;
285                     struct sg_iovec *iov;
286
287                     iov = kmalloc(size, GFP_KERNEL);
288                     if (!iov) {
289                             ret = -ENOMEM;
290                             goto out;
291                     }
292
293                     if (copy_from_user(iov, hdr->dxferp, size)) {
294                             kfree(iov);
295                             ret = -EFAULT;
296                             goto out;
297                     }
298
299                     ret = blk_rq_map_user_iov(q, rq, iov, hdr->iovec_count,
300                                               hdr->dxfer_len);
301                     kfree(iov);
302             } else if (hdr->dxfer_len)
303                     ret = blk_rq_map_user(q, rq, hdr->dxferp, hdr->dxfer_len);
304
305             if (ret)
306                     goto out;
307
308             bio = rq->bio;
309             rq->retries = 0;
310
311             start_time = jiffies;
312
313             /* ignore return value. All information is passed back to caller
314               * (if he doesn't check that is his problem).
315               * N.B. a non-zero SCSI status is _not_ necessarily an error.
316               */
317             blk_execute_rq(q, bd_disk, rq, 0);
```

```
318
319             /* write to all output members */
320             hdr->status = 0xff & rq->errors;
321             hdr->masked_status = status_byte(rq->errors);
322             hdr->msg_status = msg_byte(rq->errors);
323             hdr->host_status = host_byte(rq->errors);
324             hdr->driver_status = driver_byte(rq->errors);
325             hdr->info = 0;
326             if (hdr->masked_status || hdr->host_status || hdr->driver_status)
327                     hdr->info |= SG_INFO_CHECK;
328             hdr->resid = rq->data_len;
329             hdr->duration = ((jiffies - start_time) * 1000) / HZ;
330             hdr->sb_len_wr = 0;
331
332             if (rq->sense_len && hdr->sbp) {
333                     int len = min((unsigned int) hdr->mx_sb_len, rq->sense_len);
334
335                     if (!copy_to_user(hdr->sbp, rq->sense, len))
336                             hdr->sb_len_wr = len;
337             }
338
339             if (blk_rq_unmap_user(bio))
340                     ret = -EFAULT;
341
342             /* may not have succeeded, but output values written to control
343              * structure (struct sg_io_hdr).    */
344 out:
345             blk_put_request(rq);
346             return ret;
347 }
```

不难看出这里 hdr 实际上扮演了两个角色,一个是输入,一个是输出.而我们为了得到信息所采取的手段依然是 blk_execute_rq,即仍然是以提交 request 的方式,在 blk_execute_rq 之后,实际上信息已经保存到了 rq 的各个成员里边,而这之后的代码就是把信息从 rq 的成员中转移到 hdr 的成员中.这种情况就好比,我去翠宫饭店游泳,我花 900 块钱办了一张游泳卡,半年有效期的那种,总共可以游 30 次,于是每次我去游泳就得凭这张卡进入,等我出来的时候,工作人员会把卡还给我,而卡上有 30 个格子,每有一次工作人员会划掉一格.于是 hdr 这个变量的作用就相当于我这张卡,进来出去都要和它打交道,并且进来出去时的状态是不一样的.

那么基本上这个函数在做什么我们是很清楚了的,我们再来关注一些细节.首先结合 struct sg_io_hdr 这个伟大的结构体来看代码.

interface_id,必须得是大"S".表示 Scsi Generic.从历史渊源来说表征当年那个叫做 sg 的模块.而与之对应的是另一个叫做 pg 的模块(parallel port generic driver,并行端口通用驱动),也会有 interface_id 这么一个变量,它的这个变量则被设置为"P".

dxfer_direction,这个就表示数据传输方向.比如对于写命令这个变量可以取值为 SG_DXFER_TO_DEV,对于读命令这个变量可以取值为 SG_DXFER_FROM_DEV.

cmd_len 就不用说了,表征该 scsi 命令的长度.它必须小于等于 16.因为 scsi 命令最多就是 16 个字节.这就是为什么 237 行判断是否大于 16.(BLK_MAX_CDB 被定义为 16.)
dxfer_len 就是数据传输阶段到底会传输多少字节.max_hw_sectors 表示单个 request 最大能传输多少个 sectors,这个是硬件限制.一个 sector 是 512 个字节,所以 244 行要左移 9 位,即乘上 512.
259 行,blk_get_request 基本上可以理解为申请一个 struct request 结构体.
然后 268 行,把 cmd 复制给 rq->cmd.而 cmd 只是刚才 239 行从 hdr->cmdp 中 copy 过来的.
需要注意的是 274 行这里设置了 rq->cmd_type 是 REQ_TYPE_BLOCK_PC,block 那边会按不同的命令类型进行不同的处理.
283 行,iovec_count,它和 dxferp 是分不开的.如果 iovec_count 为 0,dxferp 表示用户空间内存地址,如果 iovec_count 大于 0,那么 dxferp 实际指向了一个 scatter-gather 数组,这个数组中的每一个成员是一个 sg_iovec_t 结构体变量.struct sg_iovec 定义于 include/scsi/sg.h:

```
76 typedef struct sg_iovec /* same structure as used by readv() Linux system */
77 {                                 /* call. It defines one scatter-gather element. */
78         void __user *iov_base;          /* Starting address    */
79         size_t iov_len;                 /* Length in bytes    */
80 } sg_iovec_t;
```

而 blk_rq_map_user 和 blk_rq_map_user_iov 的作用是一样的,建立用户数据和 request 之间的映射.如今几乎每个人都听说过 Linux 中所谓的零拷贝特性,而传说中神奇的零拷贝这一刻离我们竟是如此之近.调用这两个函数的目的就是为了使用零拷贝技术.零拷贝的作用自不必说,改善性能,提高效率,这与十七大倡导的科学发展观是吻合的.然而,这其中涉及的这潭水未免太深了,我们就不去掺合了.
在执行完 blk_execute_rq 之后就简单了.
resid 表示还剩下多少字节没有传输.
duration 表示从 scsi 命令被发送到完成之间的时间,单位是毫秒.
sbp 指向用于写 Scsi sense buffer 的 user memory.
sb_len_wr 表示实际写了多少个 bytes 到 sbp 指向的 user memory 中.命令如果成功了基本上就不会写 sense buffer,这样的话 sb_len_wr 就是 0.
mx_sb_len 则表征能往 sbp 中写的最大的 size.人民大学东门外办假学生证的哥们儿都知道,sb_len_wr<=mx_sb_len.
那么最后,我们实际上就可以大致了解了从应用层来讲是如何给 scsi 设备发送命令的.sg_inq 实际上触发的是 ioctl 的系统调用,经过几次辗转反侧,最终 sd_ioctl 会被调用.而 sd_ioctl 会调用 scsi 核心层提供的函数,sg_io,最终走的路线依然是 blk_execute_rq,而关于这个函数最终如何与 usb-storage 牵上手的,我们在 block 层那边对 scsi 命令进行分析时已经详细的介绍过了.