Linux那些事儿

系列丛书

之

我是Block层

庞大的Block层,

讲述Linux内核2.6.22.1中Block IO layer的故事

[1] 原文为blog.csdn.net/fudan_abc 上的《linux 那些事儿之我是Block层》，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

很久以前,天还是蓝的,水也是绿的,庄稼是长在地里的,猪肉是可以放心吃的,耗子还是怕猫的,法庭是讲理的,上床是先结婚的,理发店是只管理发的,药是可以治病的,医生是救死扶伤的,拍电影是不需要陪导演睡觉的,照相是要穿衣服的,欠钱是要还的,孩子的爸爸是明确的,学校是不图挣钱的,白痴是不能当教授的,卖狗肉是不能挂羊头的,男就是男的女的就是女的.那时候 Block 层还是一部分附属于 drivers/目录下一部分附属于 fs/目录下的.

但后来一切都变了.2005 年秋天,Block 层搬出了 drivers/和 fs/目录,从 2.6.15 的内核开始,顶层目录下面有了一个叫做 block 的目录,内核目录结构变成了现在这个样子:

```
localhost-1:/usr/src/linux-2.6.22.1 # ls
COPYING  Documentation  MAINTAINERS  README  arch  crypto  fs  init
kernel  mm  scripts  sound  CREDITS  Kbuild  Makefile  REPORTING-BUGS  block
drivers  include  ipc  lib  net  security  usr
```

进入 block 目录,用旁光看一下:

```
localhost:/usr/src/linux-2.6.22.1/block # ls
Kconfig  Makefile  blktrace.c  deadline-iosched.c  genhd.c  ll_rw_blk.c
scsi_ioctl.c
Kconfig.iosched  as-iosched.c  cfq-iosched.c  elevator.c  ioctl.c  noop-iosched.c
```

用 wc 命令统计一下:

```
localhost:/usr/src/linux-2.6.22.1/block # wc -l *
     54 Kconfig
     73 Kconfig.iosched
     12 Makefile
   1485 as-iosched.c
    562 blktrace.c
   2254 cfq-iosched.c
    485 deadline-iosched.c
   1160 elevator.c
    831 genhd.c
    304 ioctl.c
   4117 ll_rw_blk.c
    118 noop-iosched.c
    654 scsi_ioctl.c
  12109 total
```

一万二千多行.还好我们不用每个文件都去看.

老规矩,先看一下 Makefile 和 Kconfig,

```
localhost:/usr/src/linux-2.6.22.1/block # cat Makefile
#
# Makefile for the kernel block layer
#
```

```
obj-$(CONFIG_BLOCK) := elevator.o ll_rw_blk.o ioctl.o genhd.o scsi_ioctl.o

obj-$(CONFIG_IOSCHED_NOOP)          += noop-iosched.o
obj-$(CONFIG_IOSCHED_AS)            += as-iosched.o
obj-$(CONFIG_IOSCHED_DEADLINE)      += deadline-iosched.o
obj-$(CONFIG_IOSCHED_CFQ)           += cfq-iosched.o

obj-$(CONFIG_BLK_DEV_IO_TRACE)      += blktrace.o
```

很显然,经常在地铁站里吆喝着说刘德华死了的那位卖报的哥们儿也知道,这里最重要的一个选项是 CONFIG_BLOCK,而剩下几个我们看一下 Kconfig 以及 Kconfig.iosched 就知道,是和 IO 调度算法有关的,并不一定每种算法都要清楚,看其中一种就凑合了.

那么整个 Block 子系统的入口在哪里呢?一路走来的兄弟相信不难找到,在 block/genhd.c 中有这么一行:

```
subsys_initcall(genhd_device_init);
```

所以很明显,genhd_device_init 将为我们掀开故事的大幕.

Block 子系统的初始化

于是我们从 genhd_device_init()开始看起.

```
static int __init genhd_device_init(void)
{
        int err;

        bdev_map = kobj_map_init(base_probe, &block_subsys_lock);
        blk_dev_init();
        err = subsystem_register(&block_subsys);
        if (err < 0)
                printk(KERN_WARNING "%s: subsystem_register error: %d\n",
                        __FUNCTION__, err);
        return err;
}
```

这个初始化函数看起来粉简单,然而,正如电影<<十分爱>>里面说的一样,有时候看到的不一定是真的,真的不一定看的到.早在我还没断奶的时候,我就听说了 Block 子系统是如何如何的复杂,赫赫有名的 ll_rw_blk.c 是如何如何的深奥,也许那时候,我是个天才,可是,后来经过二十多年的社会主义教育后,终于成功的被培育成了庸才!所以现在的我要想看懂这代码可真不是件容易的事儿.

首先关注来自 block/ll_rw_blk.c 中的 blk_dev_init().

```
int __init blk_dev_init(void)
{
        int i;

        kblockd_workqueue = create_workqueue("kblockd");
```

```
        if (!kblockd_workqueue)
                panic("Failed to create kblockd\n");

        request_cachep = kmem_cache_create("blkdev_requests",
                        sizeof(struct request), 0, SLAB_PANIC, NULL, NULL);

        requestq_cachep = kmem_cache_create("blkdev_queue",
                        sizeof(request_queue_t), 0, SLAB_PANIC, NULL, NULL);

        iocontext_cachep = kmem_cache_create("blkdev_ioc",
                        sizeof(struct io_context), 0, SLAB_PANIC, NULL, NULL);

        for_each_possible_cpu(i)
                INIT_LIST_HEAD(&per_cpu(blk_cpu_done, i));

        open_softirq(BLOCK_SOFTIRQ, blk_done_softirq, NULL);
        register_hotcpu_notifier(&blk_cpu_notifier);

        blk_max_low_pfn = max_low_pfn - 1;
        blk_max_pfn = max_pfn - 1;

        return 0;
}
```

这个函数虽然不长,但是如果你能轻轻松松看懂这个函数,那么你完全有资格在简历里面写上自己精通 Linux,当然就算你啥也不懂多写几个精通也很正常,我就这么干的,这就是江湖.

首先第一个函数,create_workqueue()干的什么事情你也许不是很清楚,但是你不要忘了每次你用 ps 命令看进程的时候你都能看到一个叫做 kblockd 的玩意儿.比如:

```
[root@localhost ~]# ps -el | grep kblockd
1 S     0    80     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/0
1 S     0    81     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/1
1 S     0    82     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/2
1 S     0    83     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/3
1 S     0    84     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/4
1 S     0    85     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/5
1 S     0    86     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/6
1 S     0    87     2  0  70  -5 -     0 worker ?        00:00:00 kblockd/7
```

以上这个 kblockd 之所以有 8 个,是因为我的机器里有 8 个处理器.

这里返回值赋给了 kblocked_workqueue:

```
64 static struct workqueue_struct *kblockd_workqueue;
```

接下来,三个 kmem_cache_create()咱们当然不再是初次见面了,不过这里我们不妨看一下效果.我推荐给你的方法是使用 cat /proc/slabinfo 看一下,不过这个命令显示出来的信息太多了点,不方便我贴出来,所以我改用另外一招,在 kdb 里使用 slab 命令.当然,目的是一样的,就是为了

展示出 slab 内存的分配.凡是用 kmem_cache_create 申请过内存的都在这里留下了案底.比如咱们这里的 blkdev_ioc,blkdev_requests,blkdev_queue.

```
[0]kdb> slab
name                 actobj   nobj   size ob/sl pg/sl actsl   nsl
isofs_inode_cache       0       0     608     6     1     0     0
ext2_inode_cache        0       0     720     5     1     0     0
ext2_xattr              0       0      88    44     1     0     0
dnotify_cache           2      92      40    92     1     1     1
dquot                   0       0     256    15     1     0     0
eventpoll_pwq           1      53      72    53     1     1     1
eventpoll_epi           1      20     192    20     1     1     1
inotify_event_cache     0       0      40    92     1     0     0
inotify_watch_cache     1      53      72    53     1     1     1
kioctx                  0       0     320    12     1     0     0
kiocb                   0       0     256    15     1     0     0
fasync_cache            0       0      24   144     1     0     0
shmem_inode_cache     446     500     752     5     1   100   100
posix_timers_cache      0       0     128    30     1     0     0
uid_cache               8      30     128    30     1     1     1
ip_mrt_cache            0       0     128    30     1     0     0
tcp_bind_bucket        11     112      32   112     1     1     1
inet_peer_cache         0       0     128    30     1     0     0
secpath_cache           0       0      64    59     1     0     0
xfrm_dst_cache          0       0     384    10     1     0     0
ip_dst_cache           89     160     384    10     1    16    16
arp_cache               3      15     256    15     1     1     1
RAW                     9      10     768     5     1     2     2
UDP                    10      20     768     5     1     4     4
tw_sock_TCP             0       0     192    20     1     0     0
request_sock_TCP        0       0     128    30     1     0     0
TCP                    11      15    1536     5     2     3     3
blkdev_ioc             36     335      56    67     1     5     5
blkdev_queue           26      35    1576     5     2     7     7
blkdev_requests        77     168     272    14     1    12    12
```

当然,我在这里做了很多删减,否则肯定得列出好几页来.虽说哥们儿总被人称作垃圾中的战斗机,人渣中的 VIP,可是毕竟脸皮没有赵丽华老师那么厚,就不贴那么多行了.

3717 行,for_each_possible_cpu,针对每个 cpu 的循环,很显然,我们走到今天,smp 的代码也不得不去接触一点了.虽然我们都不懂 smp,可是人生不能象做菜,把所有的料都准备好了才下锅,此时此刻,我们不得不去面对 smp.

再下来,open_softirq.这也是一个骨灰级的函数了.它的作用是开启使用软中断向量,咱们这里开启的是 BLOCK_SOFTIRQ.准确一点说 open_softirq 的作用是初始化 softirq.而真正激活 softirq 的函数是日后我们会见到的 raise_softirq()或者 raise_softirq_irqoff(),在真正处理 softirq 的时候,咱们这里传递进去的 blk_done_softirq()函数就会被执行.此乃后话,不表.

然后是,register_hotcpu_notifier().老实说,真的没有一个函数是省油的灯,我这个汗哪!不过,看

了这么多代码之后你会发现,眼前这个函数是最性感的一个函数,因为它实在太前卫了,它的存在为了支持 CPU 的热插拔.要让它的存在有意义,你必须在编译内核的时候打开编译开关 CONFIG_HOTPLUG_CPU,否则它只是一个空函数.不过我谨慎估计你不会做这么性感的选择吧,因为你既不是梁朝伟,也不是佟大为.

剩下两行,max_low_pfn 表示 Low Memory 中最大的物理页帧号,(Page frame number of the last page frame directly mapped by the kernel(low memory))确切的说是 Low memory 中最大的物理页帧号加上 1.max_pfn 表示整个物理内存的最后一个可用的页帧号(Page frame number of the last usable page frame),确切的说也应该是最后一个可用的页帧号加上 1.所以这个取名是不合理的,也正是因为如此,咱们这里当 block 层也要用这些概念的时候就事先减掉了 1.所谓的 Low Memory,对那些 32 位的机器中,在我的记忆中,大约也就是指的 896M 以下的部分.所以我们下面可以利用 kdb 来检查一下 max_low_pfn 这个变量的值.

```
[5]kdb> md max_low_pfn
c0809900 00038000 00000847 00100000 00000000     ....G...........
c0809910 00000000 00000000 00000000 00000000     ................
c0809920 00000000 00000000 00000000 00000000     ................
c0809930 00000000 00000000 00000000 00000000     ................
c0809940 0040029b 00000000 00000000 00000000     ..@.............
c0809950 00000000 00000000 00000000 00000000     ................
c0809960 00000000 00038000 00000000 00000180     ................
c0809970 00003135 030f6000 c06a2700 c06a2700     51...`...'j..'j.
```

首先,可以看到 max_low_pfn 的值是 0x38000,换成十进制就是 229376,乘以 Page Size,4k,得到 917504,除以 1024 从而把单位换成 M,得到 896,所以很显然,max_low_pfn 标志的是 896M 以上的那一个 page.而 blk_max_low_pfn 比它少一,正好可以名副其实.

结束了 blk_dev_init()我们再回到 genhd_device_init()中来,很显然,这里还有两个函数我们并没有讲,一个是 kobj_map_init(),一个是 subsystem_register().相比之下,其实后者更容易理解,注册一个子系统,即 Block 子系统.反观前者,其实是农夫山泉,有点难.

搜索整个内核代码你会惊讶的发现,整个内核代码中这个 kobj_map_init()函数竟然只被调用了两次.

```
localhost:/usr/src/linux-2.6.22.1 # grep -r kobj_map_init *
block/genhd.c:    bdev_map = kobj_map_init(base_probe, &block_subsys_lock);
drivers/base/map.c:struct kobj_map *kobj_map_init(kobj_probe_t *base_probe, struct mutex *lock)
fs/char_dev.c:    cdev_map = kobj_map_init(base_probe, &chrdevs_lock);
include/linux/kobj_map.h:struct kobj_map *kobj_map_init(kobj_probe_t *, struct mutex *);
```

可以看到,它被定义于 drivers/base/map.c,在 include/linux/kobj_map.h 中做了声明,而调用它的地方就是 block/genhd.c 和 fs/char_dev.c,前者正是我们这里遇到的这个.为了了解这个函数做了什么,我们需要先认识一些结构体,第一个要认识的就是 struct kobj_map,定义于 drivers/base/map.c:

```
struct kobj_map {
        struct probe {
                struct probe *next;
                dev_t dev;
                unsigned long range;
                struct module *owner;
```

```
                kobj_probe_t *get;
                int (*lock)(dev_t, void *);
                void *data;
        } *probes[255];
        struct mutex *lock;
};
```

咱们这里用到的 bdev_map 正是 struct kobj_map 结构体指针,就定义于 block/genhd.c:

```
static struct kobj_map *bdev_map;
```

而 kobj_map_init()的定义是这样子的:

```
struct kobj_map *kobj_map_init(kobj_probe_t *base_probe, struct mutex *lock)
{
        struct kobj_map *p = kmalloc(sizeof(struct kobj_map), GFP_KERNEL);
        struct probe *base = kzalloc(sizeof(*base), GFP_KERNEL);
        int i;

        if ((p == NULL) || (base == NULL)) {
                kfree(p);
                kfree(base);
                return NULL;
        }

        base->dev = 1;
        base->range = ~0;
        base->get = base_probe;
        for (i = 0; i < 255; i++)
                p->probes[i] = base;
        p->lock = lock;
        return p;
}
```

看得出,申请了一个 struct kobj_map 的指针 p,然后最后返回的也是 p,即最后把一切都献给了 bdev_kmap.而这里真正干的事情无非就是让 bdev_kmap->probes[]数组全都等于 base.换言之,它们的 get 指针全都等于了咱们这里传递进来的 base_probe 函数,这个函数也不很短,来自 block/genhd.c:

```
static struct kobject *base_probe(dev_t dev, int *part, void *data)
{
        if (request_module("block-major-%d-%d", MAJOR(dev), MINOR(dev)) > 0)
                /* Make old-style 2.4 aliases work */
                request_module("block-major-%d", MAJOR(dev));
        return NULL;
}
```

这个函数看起来怪怪的,定义了三个参数却只使用其中的一个,定义的返回值类型是 struct kobject*实际上却偏偏只返回 NULL.看这个函数不由得让我想起了在复旦的那段日子,当复旦大学将同性恋课程搬进课堂后,我看周围的人们都有一种深邃而扑朔迷离的眼神,每个人都怪怪的,校园里充满了同性恋的味道.但这个函数怪是怪,总还是有点逻辑,每一个学过钱能老

师那本<<C++程序设计教程>>的男人都会很快醒悟,这里八成是利用了 C++的基类派生类那种函数重载的理念.没错,你的直觉是正确的,日后我们会彻底明白的.

注册一个块设备驱动

看完了 block 子系统的初始化之后,我曾一度迷茫过,也曾辗转反侧,也曾苦恼万分,我完全不知道下一步该怎么走,几经思索,思索着我和中国的未来,徘徊过后,彷徨过后,终于决定,和 scsi disk 驱动同步进行往下走,因为 scsi disk 那边会调用许多 block 层这边提供的函数,于是我们就在这边来看看这些函数究竟是干什么的.

第一个函数当然就是 register_blkdev().

```
int register_blkdev(unsigned int major, const char *name)
{
        struct blk_major_name **n, *p;
        int index, ret = 0;

        mutex_lock(&block_subsys_lock);

        /* temporary */
        if (major == 0) {
                for (index = ARRAY_SIZE(major_names)-1; index > 0; index--) {
                        if (major_names[index] == NULL)
                                break;
                }

                if (index == 0) {
                        printk("register_blkdev: failed to get major for %s\n",
                               name);
                        ret = -EBUSY;
                        goto out;
                }
                major = index;
                ret = major;
        }

        p = kmalloc(sizeof(struct blk_major_name), GFP_KERNEL);
        if (p == NULL) {
                ret = -ENOMEM;
                goto out;
        }

        p->major = major;
        strlcpy(p->name, name, sizeof(p->name));
```

```
87          p->next = NULL;
88          index = major_to_index(major);
89
90          for (n = &major_names[index]; *n; n = &(*n)->next) {
91                  if ((*n)->major == major)
92                          break;
93          }
94          if (!*n)
95                  *n = p;
96          else
97                  ret = -EBUSY;
98
99          if (ret < 0) {
100                 printk("register_blkdev: cannot get major %d for %s\n",
101                        major, name);
102                 kfree(p);
103         }
104 out:
105         mutex_unlock(&block_subsys_lock);
106         return ret;
107 }
```

从 sd 那边调用这个函数来看,咱们是指定了主设备号了的.换言之,这里的 major 是非零值,而 struct blk_major_name 的定义也在 block/genhd.c 中:

```
27 static struct blk_major_name {
28          struct blk_major_name *next;
29          int major;
30          char name[16];
31 } *major_names[BLKDEV_MAJOR_HASH_SIZE];
```

注意这里顺便定义了一个数组 major_names,咱们这里也用到了.

这其中 BLKDEV_MAJOR_HASH_SIZE 定义于 include/linux/fs.h:

```
1575 #define BLKDEV_MAJOR_HASH_SIZE    255
```

即数组 major_names[]有 255 个元素,换言之,咱们定义了 255 个指针.

而 88 行这个内联函数同样来自 block/genhd.c:

```
33 /* index in the above - for now: assume no multimajor ranges */
34 static inline int major_to_index(int major)
35 {
36          return major % BLKDEV_MAJOR_HASH_SIZE;
37 }
```

比如咱们传递的 major 是 8,那么 major_to_index 就是 8.

不难理解,register_blkdev()这个函数做的事情就是,为这 255 个指针找到归属.即先在 79 行调用 kmalloc 申请一个 struct blk_major_name 结构体并且让 p 指向它,接下来为 p 赋值,而 n 将指向 major_names[index],比如 index 就是 8,那么 n 就指向 major_names[8],一开始它肯定为空,所以直接执行 94 行并进而 95 行,于是就把赋好值的 p 的那个结构体赋给了 major_names[8],因此,major_names[8]就既有 major 也有 name 了,name 就是”sd”.

那么此时此刻的效果是什么?告诉你,不是在/dev/目录下面有 sda,sdb 之类的文件,而是通过/proc/devices 能够看到这个块设备驱动注册了.

```
localhost:/usr/src/linux-2.6.22.1 # 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
 21 sg
 29 fb
128 ptm
136 pts
162 raw
180 usb
189 usb_device
254 megaraid_sas_ioctl

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

驱动不过一出戏,内存申请为哪般?

下一个函数,alloc_disk().在 sd.c 中咱们传递进来的参数是 16.

```
struct gendisk *alloc_disk(int minors)
{
        return alloc_disk_node(minors, -1);
}

struct gendisk *alloc_disk_node(int minors, int node_id)
{
        struct gendisk *disk;

        disk = kmalloc_node(sizeof(struct gendisk), GFP_KERNEL, node_id);
        if (disk) {
                memset(disk, 0, sizeof(struct gendisk));
                if (!init_disk_stats(disk)) {
                        kfree(disk);
                        return NULL;
                }
                if (minors > 1) {
                        int size = (minors - 1) * sizeof(struct hd_struct *);
                                disk->part = kmalloc_node(size, GFP_KERNEL, node_id);
                        if (!disk->part) {
                                kfree(disk);
                                return NULL;
                        }
                        memset(disk->part, 0, size);
                }
                disk->minors = minors;
                kobj_set_kset_s(disk,block_subsys);
                kobject_init(&disk->kobj);
                rand_initialize_disk(disk);
                INIT_WORK(&disk->async_notify,
                        media_change_notify_thread);
        }
        return disk;
```

```
753 }
```

因此我们做的事情就是申请了一个 struct gendisk 结构体.毫无疑问,这个结构体是我们这个故事中最重要的结构体之一,来自 include/linux/genhd.h:

```
struct gendisk {
        int major;                      /* major number of driver */
        int first_minor;
        int minors;                     /* maximum number of minors, =1 for
                                         * disks that can't be partitioned. */
        char disk_name[32];             /* name of major driver */
        struct hd_struct **part;        /* [indexed by minor] */
        int part_uevent_suppress;
        struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        sector_t capacity;

        int flags;
        struct device *driverfs_dev;
        struct kobject kobj;
        struct kobject *holder_dir;
        struct kobject *slave_dir;

        struct timer_rand_state *random;
        int policy;

        atomic_t sync_io;               /* RAID */
        unsigned long stamp;
        int in_flight;
#ifdef  CONFIG_SMP
        struct disk_stats *dkstats;
#else
        struct disk_stats dkstats;
#endif
        struct work_struct async_notify;
};
```

因为 minors 我们给的是 16,所以 736 行的 if 语句肯定是满足的.于是 size 等于 15 个 sizeof(struct hd_struct *),而 part 我们看到是 struct hd_struct 的二级指针,这里我们看到 kmalloc_node(),这个函数中的 node/node_id 这些概念指的是 NUMA 技术中的节点,对于咱们这些根本就不会接触 NUMA 的人来说 kmalloc_node()就等于 kmalloc(),因此这里做的就是申请内存并且初始化为 0.要说明的一点是,part 就是 partition 的意思,日后它将扮演我们常说的分区的角色.

然后,disk->minors 设置为了 16.

746 行,kobj_set_kset_s(),block_subsys 是我们前面注册的子系统,从数据结构来说,它的定义如下,来自 block/genhd.c:

```
struct kset block_subsys;
```

其实也就是一个 struct kset.而这里的 kobj_set_kset_s 的作用就是让 disk 对应 kobject 的 kset 等于 block_subsys.也就是说让 kobject 找到它的 kset.(如果你还记得当初我们在我是 Sysfs 中分析的 kobject 和 kset 的那套理论的话,你不会不明白这里的意图.)而 kobject_init()初始化一个 kobject,这个函数通常就是出现在设置了 kobject 的 kset 之后.

网友”暗恋未遂”打断了我,他说这行代码并不是定义一个结构体.它更像是一个声明,而不像是定义.我仔细一看,似乎真的是的,这里的确是声明,而定义并不在这里,Linux 内核代码的确是虚虚实实真真假假,一不小心就会看走眼,写代码的哥们儿果然是深谙兵不厌诈的道理.但愿他们只是借此表达他们对现实社会的不满吧,毕竟在这年头,只有假货是真的,别的都是假的.那么定义在哪里呢?同一个文件中:

```
decl_subsys(block, &ktype_block, &block_uevent_ops);
```

这个 decl_subsys 来自 include/linux/kobject.h:

```
#define decl_subsys(_name,_type,_uevent_ops) \
struct kset _name##_subsys = { \
        .kobj = { .name = __stringify(_name) }, \
        .ktype = _type, \
        .uevent_ops =_uevent_ops, \
}
```

结合这个宏的定义,我们知道,我们等效于做了下面这么一件事情:

```
struct kset block_subsys = { \
        .kobj = { .name = __stringify(block) }, \
        .ktype = &ktype_block, \
        .uevent_ops = &block_uevent_ops, \
}
```

正是因为有了这么一个定义,正是因为这里我们把”block”给了 block_subsys 的 kobj 的 name 成员,所以当我们在 block 子系统初始化的时候调用 subsystem_register(&block_subsys)之后,我们才会在/sys/目录下面看到”block”子目录.

```
localhost:~ # ls /sys/
block   bus   class   devices   firmware   fs   kernel   module   power
```

749 行,初始化一个工作队列.到时候用到了再来看.

至此,alloc_disk_node 就将返回,从而 alloc_disk 也就返回了.

浓缩就是精华?(一)

人,生在床上,死在床上;欲生欲死,还是在床上.这句话非常有道理.有人说它有点俗,但,我并不这么认为.我因为经常坐在床上一边看 A 片一边看代码,所以对这句话体会颇深,事实上它形象的描述了我坐在床上看代码时复杂的心情,说欲生欲死,一点也不夸张,尤其是当我看到 add_disk()这个无比变态的函数的时候.我不禁感慨,上帝欲使人灭亡,必先使其疯狂;上帝欲使人疯狂,必先使其看 Linux 内核代码.

```
/**
 * add_disk - add partitioning information to kernel list
 * @disk: per-device partitioning information
 *
```

```
 * This function registers the partitioning information in @disk
 * with the kernel.
 */
void add_disk(struct gendisk *disk)
{
        disk->flags |= GENHD_FL_UP;
        blk_register_region(MKDEV(disk->major, disk->first_minor),
                        disk->minors, NULL, exact_match, exact_lock, disk);
        register_disk(disk);
        blk_register_queue(disk);
}
```

老实说当我一开始看到这个函数只有四行代码的时候,我几乎喜极而泣.但很快我就发现自己的想法 Too Simple, Sometimes Naive 了.这个函数虽然只有四行代码,可是超级复杂,旗下三个函数,一个比一个拽,我渐渐困惑,写代码的哥们儿有必要写这种浓缩版的函数么?要黑赵丽华老师也不至于这么表现吧?

头一个,blk_register_region,来自 block/genhd.c:

```
/*
 * Register device numbers dev..(dev+range-1)
 * range must be nonzero
 * The hash chain is sorted on range, so that subranges can override.
 */
void blk_register_region(dev_t dev, unsigned long range, struct module *module,
                        struct kobject *(*probe)(dev_t, int *, void *),
                        int (*lock)(dev_t, void *), void *data)
{
        kobj_map(bdev_map, dev, range, module, probe, lock, data);
}
```

这里 kobj_map()其实是远方的来客,它来自 drivers/base/map.c:

```
int kobj_map(struct kobj_map *domain, dev_t dev, unsigned long range,
             struct module *module, kobj_probe_t *probe,
             int (*lock)(dev_t, void *), void *data)
{
        unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1;
        unsigned index = MAJOR(dev);
        unsigned i;
        struct probe *p;

        if (n > 255)
                n = 255;

        p = kmalloc(sizeof(struct probe) * n, GFP_KERNEL);

        if (p == NULL)
                return -ENOMEM;
```

```

        for (i = 0; i < n; i++, p++) {
                p->owner = module;
                p->get = probe;
                p->lock = lock;
                p->dev = dev;
                p->range = range;
                p->data = data;
        }
        mutex_lock(domain->lock);
        for (i = 0, p -= n; i < n; i++, p++, index++) {
                struct probe **s = &domain->probes[index % 255];
                while (*s && (*s)->range < range)
                        s = &(*s)->next;
                p->next = *s;
                *s = p;
        }
        mutex_unlock(domain->lock);
        return 0;
}
```

结合我们的 sd_probe 函数来看,我们在 sd_probe()中说了,first_minor 无非就是 0,16,32,48 这样一系列的数,而 minors 总是 16,换言之按照这里我们的上下文 range 就是 16,这种情况下 n 只能是 1.

Domain 就是 bdev_map,于是我们即便不看代码也能猜到,这个函数的主要目的就是为 bdev_map 的 probes 这个指针数组赋值,假设我们的 major 是 8,那么这里就是为 probes[8]赋值.对比形参实参可以看到,我们为 get 指针赋的是 exact_match().这个函数同样来自于 block/genhd.c:

```
static struct kobject *exact_match(dev_t dev, int *part, void *data)
{
        struct gendisk *p = data;
        return &p->kobj;
}
```

即,比如说我们的 index 或者说 major number 是 8 的话,那么这之后,bdev_map->probes[8]所对应的 get 指针就指向了 exact_match.

同时,data 指针赋上了 disk,即 struct gendisk 指针 disk.

老实说,现在我们完全看不出这么做的意义,或者说 blk_register_region 这个函数究竟有什么价值现在完全体现不出来.但是其实这是 Linux 中实现的一种管理设备号的机制,这里利用了传说中的哈希表来管理设备号,哈希表的优点大家知道,便于查找,而我们的目的是为了通过给定的一个设备号就能迅速得到它所对应的 kobject 指针,对于块设备来说,得到 kobject 是为了得到其对应的 gendisk.

那么什么时候会需要这样做呢?Ok,比如你执行 fdisk –l /dev/sda,从而 open 系统调用或者说函数 sys_open 会被执行,如果你一路跟踪,你会发现到后来会有一个叫做 get_gendisk()的函数被调用.这个函数实际上也是我们这边定义的,来自 block/genhd.c:

```
/**
```

```
* get_gendisk - get partitioning information for a given device
* @dev: device to get partitioning information for
*
* This function gets the structure containing partitioning
* information for the given device @dev.
*/
struct gendisk *get_gendisk(dev_t dev, int *part)
{
        struct kobject *kobj = kobj_lookup(bdev_map, dev, part);
        return kobj ? to_disk(kobj) : NULL;
}
```

于是我们来看 kobj_lookup().来自 drivers/base/map.c:

```
struct kobject *kobj_lookup(struct kobj_map *domain, dev_t dev, int *index)
{
        struct kobject *kobj;
        struct probe *p;
        unsigned long best = ~0UL;

retry:
        mutex_lock(domain->lock);
        for (p = domain->probes[MAJOR(dev) % 255]; p; p = p->next) {
                struct kobject *(*probe)(dev_t, int *, void *);
                struct module *owner;
                void *data;

                if (p->dev > dev || p->dev + p->range - 1 < dev)
                        continue;
                if (p->range - 1 >= best)
                        break;
                if (!try_module_get(p->owner))
                        continue;
                owner = p->owner;
                data = p->data;
                probe = p->get;
                best = p->range - 1;
                *index = dev - p->dev;
                if (p->lock && p->lock(dev, data) < 0) {
                        module_put(owner);
                        continue;
                }
                mutex_unlock(domain->lock);
                kobj = probe(dev, index, data);
                /* Currently ->owner protects _only_ ->probe() itself. */
                module_put(owner);
```

```
                if (kobj)
                        return kobj;
                goto retry;
        }
        mutex_unlock(domain->lock);
        return NULL;
}
```

现在我们隐隐约约的感觉到,kobj_map_init()和 kobj_map()以及 kobj_lookup()是一个系列的,它们都是为 Linux 设备号管理服务的,就好比舒淇,李丽珍,钟丽缇是一个系列的,她们都是为三级片市场服务的.首先,kobj_map_init 提供的是一次性服务,它的使命是建立了 bdev_map 这个 struct kobj_map.然后 kobj_map 是每次在 blk_register_region 中被调用的,然而,在这个五彩缤纷的世界中,调用 blk_register_region()的地方可真不少,随便一搜索就是一大把,而我们这个在 add_disk 中调用只是其中之一,其它的比如 RAID 驱动那边,软驱驱动那边,都会有调用这个 blk_register_region 的需求,而 kobj_lookup()发生在什么情况下呢?它提供的其实是售后服务.当块设备驱动完成了初始化工作,当它在内核中站稳了脚跟,会有一个设备文件和它相对应,这个文件会出现在/dev 目录下.在不久的将来,当 open 系统调用试图打开块设备文件的时候就会调用它,更准确地说,sys_open 经由 filp_open 然后是 dentry_open(),最终会找到 blkdev_open,blkdev_open 会调用 do_open,do_open()会调用 get_gendisk(),要想明白这个理儿,得先看一下 dev_t 这个结构.dev_t 实际上就是 u32,也即就是 32 个 bits.前面咱们看到的 MKDEV,MAJOR,都来自 include/linux/kdev_t.h:

```
#define MINORBITS       20
#define MINORMASK       ((1U << MINORBITS) - 1)

#define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
#define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

通过这几个宏,我们不难看出 dev_t 的意义了,32 个 bits,其中高 12 位被用来记录设备的主设备号,低 20 位用来记录设备的次设备号.而 MKDEV 就是建立一个设备号.ma 代表主设备号,mi 代表次设备号,ma 左移 20 位再和 mi 相或,反过来,MAJOR 就是从 dev 中取主设备号,MINOR 就是从 dev 中取次设备号.不多说了,杭州西湖畔拉皮条的都知道怎么回事了.

当一个设备闯入 Linux 的内心时,首先它会有一个居住证号,这就是 dev_t,很显然,每个人的居住证号不一样,它是唯一的.(为什么不说是身份证号?因为居住证意味着当设备离开 Linux 系统的时候就可以销毁,所以它更能体现设备的流动性.)建立一个设备文件的时候,其设备号是确定的,而我们每次建立一个文件都会建立一个结构体变量,它就是 struct inode,而 struct inode 拥有成员 dev_t i_dev,所以日后我们从 struct inode 就可以得到其设备号 dev_t,而这里 kobj_map 这一系列函数使得我们可以从 dev_t 找到对应的 kobject,然后进一步作为磁盘驱动,我们不可避免的需要访问磁盘对应的 gendisk 结构体指针,而 get_gendisk()就是在这时候应运而生并粉墨登场的.咱们看到 get_gendisk()的两个参数,dev_t dev 和 int *part,前者就是设备号,而后者传递的是一个指针,这表示什么呢?这表示,

1. 如果这个设备号对应的是一个分区,那么 part 变量就用来保存分区的编号.
2. 如果这个设备号对应的是整个设备而不是某个分区,那么 part 就只要设置成 0 就 ok 了.

那么得到 gendisk 的目的又是什么呢?我们注意到 struct gendisk 有一个成员,struct block_device_operations *fops,而这个指针才是用来真正执行操作的,每一个块设备驱动都准备了这么一个结构体,比如咱们在 sd 中定义的那个:

```
static struct block_device_operations sd_fops = {
        .owner                  = THIS_MODULE,
        .open                   = sd_open,
        .release                = sd_release,
        .ioctl                  = sd_ioctl,
        .getgeo                 = sd_getgeo,
#ifdef CONFIG_COMPAT
        .compat_ioctl           = sd_compat_ioctl,
#endif
        .media_changed          = sd_media_changed,
        .revalidate_disk        = sd_revalidate_disk,
};
```

正是因为有这种种暧昧关系,我们才能一步一步从 sys_open 最终走到 sd_open,也才能从用户层一步一步走到块设备驱动层,如同董卿姐姐能够从上海一步步走向央视.

浓缩就是精华?(二)

第二个,register_disk,来头不小,它来自遥远的 fs/partitions/check.c:

```
/* Not exported, helper to add_disk(). */
void register_disk(struct gendisk *disk)
{
        struct block_device *bdev;
        char *s;
        int i;
        struct hd_struct *p;
        int err;

        strlcpy(disk->kobj.name,disk->disk_name,KOBJ_NAME_LEN);
        /* ewww... some of these buggers have / in name... */
        s = strchr(disk->kobj.name, '/');
        if (s)
                *s = '!';
        if ((err = kobject_add(&disk->kobj)))
                return;
        err = disk_sysfs_symlinks(disk);
        if (err) {
                kobject_del(&disk->kobj);
                return;
        }
        disk_sysfs_add_subdirs(disk);

        /* No minors to use for partitions */
```

```
497             if (disk->minors == 1)
498                     goto exit;
499
500             /* No such device (e.g., media were just removed) */
501             if (!get_capacity(disk))
502                     goto exit;
503
504             bdev = bdget_disk(disk, 0);
505             if (!bdev)
506                     goto exit;
507
508             /* scan partition table, but suppress uevents */
509             bdev->bd_invalidated = 1;
510             disk->part_uevent_suppress = 1;
511             err = blkdev_get(bdev, FMODE_READ, 0);
512             disk->part_uevent_suppress = 0;
513             if (err < 0)
514                     goto exit;
515             blkdev_put(bdev);
516
517 exit:
518             /* announce disk after possible partitions are already created */
519             kobject_uevent(&disk->kobj, KOBJ_ADD);
520
521             /* announce possible partitions */
522             for (i = 1; i < disk->minors; i++) {
523                     p = disk->part[i-1];
524                     if (!p || !p->nr_sects)
525                             continue;
526                     kobject_uevent(&p->kobj, KOBJ_ADD);
527             }
528 }
```

如果你不懂 Linux 2.6 的统一设备模型,那你要看懂这段代码估计够呛.但好在我们在<<我是Sysfs>>中对 kobject 方面的东西做了介绍.所以这里我们不会深入到 kobject 相关的函数内部中去,也不会深入到 sysfs 提供的函数内部,点到为止.

首先 487 行这个 kobject_add 的作用是很直观的,在 Sysfs 中为这块磁盘建一个子目录.就比如下面这些目录中的那个 sdf,就是为我的 U 盘而建立的,我要是把这个调用 kobject_add 函数这行注释掉,保证你就看不到这个 sdf 目录.

```
[root@lfg2 ~]# ls /sys/block/
md0    ram1    ram11    ram13    ram15    ram3    ram5    ram7    ram9    sdb    sdd    sdf  ram0
ram10   ram12   ram14   ram2    ram4   ram6   ram8   sda    sdc   sde   sdg
```

这时候网友”塞翁失身”提出两个问题:

第一,为什么 kobject_add 这么一调用,生成的这个子目录的名字就叫做”sdf”,而不叫做别的?君还记得在 sd_probe 中我们做过一件事情么,当时我们可是精心计算过 disk_name 的,而

disk_name 正是 struct gendisk 的一个成员,这里我们看到 482 行我们把 disk_name 给了 kobj.name,这就是为什么我们调用 kobject_add 添加一个 kobject 的时候,它的名字就是我们当时的 disk_name.

第二,为什么生成的这个子目录是在/sys/block 目录下面,而不是在别的位置?还记得在 alloc_disk_node 中我们申请 struct gendisk 的情景么?那句 kobj_set_kset_s(disk,block_subsys) 做的就是让 disk 对应的 kobject 从属于 block_subsys 对应的 kobject 下面.这就是为什么我们现在添加这个 kobject 的时候,它很自然的就会在/sys/block 子目录下面建立文件.

继续走, disk_sysfs_symlinks 来自 fs/partitions/check.c,这个函数虽然不短,但是比较浅显易懂.

```
static int disk_sysfs_symlinks(struct gendisk *disk)
{
        struct device *target = get_device(disk->driverfs_dev);
        int err;
        char *disk_name = NULL;

        if (target) {
                disk_name = make_block_name(disk);
                if (!disk_name) {
                        err = -ENOMEM;
                        goto err_out;
                }

                err = sysfs_create_link(&disk->kobj, &target->kobj, "device");
                if (err)
                        goto err_out_disk_name;

                err = sysfs_create_link(&target->kobj, &disk->kobj, disk_name);
                if (err)
                        goto err_out_dev_link;
        }

        err = sysfs_create_link(&disk->kobj, &block_subsys.kobj,
                                "subsystem");
        if (err)
                goto err_out_disk_name_lnk;

        kfree(disk_name);

        return 0;

err_out_disk_name_lnk:
        if (target) {
                sysfs_remove_link(&target->kobj, disk_name);
err_out_dev_link:
                sysfs_remove_link(&disk->kobj, "device");
```

```
465 err_out_disk_name:
466                     kfree(disk_name);
467 err_out:
468                     put_device(target);
469           }
470           return err;
471 }
```

我们用实际效果来解读这个函数.首先我们看正常工作的 U 盘会在/sys/block/sdf 下面有哪些内容:

[root@localhost ~]# ls /sys/block/sdf/

capability dev device holders queue range removable size slaves stat subsystem uevent

442 行的 sysfs_create_link 这么一行创建的就是这里这个 device 这个软链接文件.我们来看它链接到哪里去了?

[root@localhost ~]# ls -l /sys/block/sdf/device

lrwxrwxrwx 1 root root 0 Dec 13 07:09 /sys/block/sdf/device -> ../../devices/pci0000:00/0000:00:1d.7/usb4/4-4/4-4:1.0/host24/target24:0:0/24:0:0:0

而 446 行这个 sysfs_create_link 则从那边又建立一个反链接,又给链接回来了.

[root@localhost~]# ls /sys/devices/pci0000\:00/0000\:00\:1d.7/usb4/4-4/4-4\:1.0/host24/target24\:0\:0/24\:0\:0\:0/

block:sdf driver ioerr_cnt model rescan scsi_generic:sg7 timeout bus generic iorequest_cnt power rev scsi_level type delete iocounterbits max_sectors queue_depth scsi_device:24:0:0:0 state uevent device_blocked iodone_cnt modalias queue_type scsi_disk:24:0:0:0 subsystem vendor

很明显,就是这个 block:sdf.

[root@localhost~]# ls -l /sys/devices/pci0000\:00/0000\:00\:1d.7/usb4/4-4/4-4\:1.0/host24/target24\:0\:0/24\:0\:0\:0/block\:sdf

lrwxrwxrwx 1 root root 0 Dec 13 21:16 /sys/devices/pci0000:00/0000:00:1d.7/usb4/4-4/4-4:1.0/host24/target24:0:0/24:0:0:0/block:sdf -> ../../../../../../../../block/sdf

于是这就等于你中有我我中有你,你那边有一个文件链接到了我这边,我这边有一个文件链接到了你那边.

然后 451 行再次调用 sysfs_create_link.这次很显然,生成的是/sys/block/sdf/subsystem 这个软链接文件.

[root@localhost ~]# ls -l /sys/block/sdf/subsystem

lrwxrwxrwx 1 root root 0 Dec 13 07:09 /sys/block/sdf/subsystem -> ../../block

三个链接文件建立好之后,disk_sysfs_symlinks 也就结束了它的使命.接下来一个函数是 disk_sysfs_add_subdirs.同样来自 fs/partitions/check.c:

```
342 static inline void disk_sysfs_add_subdirs(struct gendisk *disk)
343 {
344           struct kobject *k;
345
```

```
        k = kobject_get(&disk->kobj);
        disk->holder_dir = kobject_add_dir(k, "holders");
        disk->slave_dir = kobject_add_dir(k, "slaves");
        kobject_put(k);
}
```

这个函数的意图太明显了,相信虹口足球场外倒卖演唱会门票的黄牛党们都能看懂,无非就是建立 holders 和 slaves 两个子目录.

504 行,bdget_disk,这是一个内联函数,<<Thinking in C++>>告诉我们内联函数最好定义在头文件中,所以这个函数来自 include/linux/genhd.h:

```
static inline struct block_device *bdget_disk(struct gendisk *disk, int index)
{
        return bdget(MKDEV(disk->major, disk->first_minor) + index);
}
```

又是一次声东击西的调用.bdget 来自 fs/block_dev.c:

```
struct block_device *bdget(dev_t dev)
{
        struct block_device *bdev;
        struct inode *inode;

        inode = iget5_locked(bd_mnt->mnt_sb, hash(dev),
                        bdev_test, bdev_set, &dev);

        if (!inode)
                return NULL;

        bdev = &BDEV_I(inode)->bdev;

        if (inode->i_state & I_NEW) {
                bdev->bd_contains = NULL;
                bdev->bd_inode = inode;
                bdev->bd_block_size = (1 << inode->i_blkbits);
                bdev->bd_part_count = 0;
                bdev->bd_invalidated = 0;
                inode->i_mode = S_IFBLK;
                inode->i_rdev = dev;
                inode->i_bdev = bdev;
                inode->i_data.a_ops = &def_blk_aops;
                mapping_set_gfp_mask(&inode->i_data, GFP_USER);
                inode->i_data.backing_dev_info = &default_backing_dev_info;
                spin_lock(&bdev_lock);
                list_add(&bdev->bd_list, &all_bdevs);
                spin_unlock(&bdev_lock);
                unlock_new_inode(inode);
        }
```

```
        return bdev;
}
```

真是祸不单行今日行啊,一下子跳出来两个变态的结构体来.struct block_device 和 struct inode. 在 include/linux/fs.h 中定义了这么一个结构体:

```
struct block_device {
        dev_t                   bd_dev;  /* not a kdev_t - it's a search key */
        struct inode *          bd_inode;       /* will die */
        int                     bd_openers;
        struct mutex            bd_mutex;       /* open/close mutex */
        struct semaphore        bd_mount_sem;
        struct list_head        bd_inodes;
        void *                  bd_holder;
        int                     bd_holders;
#ifdef CONFIG_SYSFS
        struct list_head        bd_holder_list;
#endif
        struct block_device *   bd_contains;
        unsigned                bd_block_size;
        struct hd_struct *      bd_part;
        /* number of times partitions within this device have been opened. */
        unsigned                bd_part_count;
        int                     bd_invalidated;
        struct gendisk *        bd_disk;
        struct list_head        bd_list;
        struct backing_dev_info *bd_inode_backing_dev_info;
        /*
         * Private data.  You must have bd_claim'ed the block_device
         * to use this.  NOTE:  bd_claim allows an owner to claim
         * the same device multiple times, the owner must take special
         * care to not mess up bd_private for that case.
         */
        unsigned long           bd_private;
};
```

很明显,Linux 中每一个 Block 设备都由这么一个结构体变量表示,这玩意儿因此被称作块设备描述符.inode 咱们不具体讲,但是这里挺逗的一个结构体是 struct bdev_inode,

```
struct bdev_inode {
        struct block_device bdev;
        struct inode vfs_inode;
};
```

把两个变态的结构体组合起来就变成了第三个变态的结构体.

但是网名为”避孕套一直用雕牌”的哥们儿问我,bdev_inode 好像没出现过,讲它干嘛?我想说看问题要看本质,不要被表面迷惑,这个世界上很多事情都不像表面上看起来那样.不信你看 BDEV_I,这个内联函数来自 fs/block_dev.c:

```
static inline struct bdev_inode *BDEV_I(struct inode *inode)
```

```
{
        return container_of(inode, struct bdev_inode, vfs_inode);
}
```

很显然,从 inode 得到相应的 bdev_inode.于是 565 行这个&BDEV_I(inode)->bdev 表示的就是 inode 对应的 bdev_inode 的成员 struct block_device bdev.

但是结构体变量这东西不像公共汽车,只需等待就会自动来到你的面前,而需要你去申请才会有.iget5_locked 就是干这件事情的,这个函数来自 fs/inode.c,我们显然不会去深入看它,只能告诉你,这个函数这么一执行,我们就既有 inode 又有 block_device 了.而且对于第一次申请的 inode,其 i_state 成员是设置了 I_NEW 这个 flag 的,所以 bdget()函数中,567 行这一段 if 语句是要被执行的.这一段 if 语句的作用就是初始化 inode 结构体指针 inode 以及 block_device 结构体指针 bdev.而函数最终返回的也正是 bdev.需要强调一下,bdev 正是从这一刻开始正式出现在我们的故事中的.

回到 register_disk()中,继续往下.下一个重量级的函数是 blkdev_get,来自 fs/block_dev.c:

```
static int __blkdev_get(struct block_device *bdev, mode_t mode, unsigned flags,
                        int for_part)
{
        /*
         * This crockload is due to bad choice of ->open() type.
         * It will go away.
         * For now, block device ->open() routine must _not_
         * examine anything in 'inode' argument except ->i_rdev.
         */
        struct file fake_file = {};
        struct dentry fake_dentry = {};
        fake_file.f_mode = mode;
        fake_file.f_flags = flags;
        fake_file.f_path.dentry = &fake_dentry;
        fake_dentry.d_inode = bdev->bd_inode;

        return do_open(bdev, &fake_file, for_part);
}

int blkdev_get(struct block_device *bdev, mode_t mode, unsigned flags)
{
        return __blkdev_get(bdev, mode, flags, 0);
}
```

看到 blkdev_get 调用的是__blkdev_get,所以我们两个函数一块贴出来了.

很显然,真正需要看的却是 do_open,来自同一个文件.

```
/*
 * bd_mutex locking:
 *
 *  mutex_lock(part->bd_mutex)
 *    mutex_lock_nested(whole->bd_mutex, 1)
 */
```

```

static int do_open(struct block_device *bdev, struct file *file, int for_part)
{
	struct module *owner = NULL;
	struct gendisk *disk;
	int ret = -ENXIO;
	int part;

	file->f_mapping = bdev->bd_inode->i_mapping;
	lock_kernel();
	disk = get_gendisk(bdev->bd_dev, &part);
	if (!disk) {
		unlock_kernel();
		bdput(bdev);
		return ret;
	}
	owner = disk->fops->owner;

	mutex_lock_nested(&bdev->bd_mutex, for_part);
	if (!bdev->bd_openers) {
		bdev->bd_disk = disk;
		bdev->bd_contains = bdev;
		if (!part) {
			struct backing_dev_info *bdi;
			if (disk->fops->open) {
				ret = disk->fops->open(bdev->bd_inode, file);
				if (ret)
					goto out_first;
			}
			if (!bdev->bd_openers) {
bd_set_size(bdev,(loff_t)get_capacity(disk)<<9);
				bdi = blk_get_backing_dev_info(bdev);
				if (bdi == NULL)
					bdi = &default_backing_dev_info;
				bdev->bd_inode->i_data.backing_dev_info = bdi;
			}
			if (bdev->bd_invalidated)
				rescan_partitions(disk, bdev);
		} else {
			struct hd_struct *p;
			struct block_device *whole;
			whole = bdget_disk(disk, 0);
```

```
			ret = -ENOMEM;
			if (!whole)
				goto out_first;
			BUG_ON(for_part);
			ret = __blkdev_get(whole, file->f_mode, file->f_flags, 1);
			if (ret)
				goto out_first;
			bdev->bd_contains = whole;
			p = disk->part[part - 1];
			bdev->bd_inode->i_data.backing_dev_info =
			   whole->bd_inode->i_data.backing_dev_info;
			if (!(disk->flags & GENHD_FL_UP) || !p || !p->nr_sects) {
				ret = -ENXIO;
				goto out_first;
			}
			kobject_get(&p->kobj);
			bdev->bd_part = p;
			bd_set_size(bdev, (loff_t) p->nr_sects << 9);
		}
	} else {
		put_disk(disk);
		module_put(owner);
		if (bdev->bd_contains == bdev) {
			if (bdev->bd_disk->fops->open) {
				ret = bdev->bd_disk->fops->open(bdev->bd_inode, file);
				if (ret)
					goto out;
			}
			if (bdev->bd_invalidated)
				rescan_partitions(bdev->bd_disk, bdev);
		}
	}
	bdev->bd_openers++;
	if (for_part)
		bdev->bd_part_count++;
	mutex_unlock(&bdev->bd_mutex);
	unlock_kernel();
	return 0;

out_first:
	bdev->bd_disk = NULL;
	bdev->bd_inode->i_data.backing_dev_info = &default_backing_dev_info;
```

```
1193         if (bdev != bdev->bd_contains)
1194                 __blkdev_put(bdev->bd_contains, 1);
1195         bdev->bd_contains = NULL;
1196         put_disk(disk);
1197         module_put(owner);
1198 out:
1199         mutex_unlock(&bdev->bd_mutex);
1200         unlock_kernel();
1201         if (ret)
1202                 bdput(bdev);
1203         return ret;
1204 }
```

天哪.内核函数没有最变态,只有更变态.

一开始的时候,bd_openers 是被初始化为了 0,所以 1128 这个 if 语句是要被执行的.bd_openers 为 0 表示一个文件还没有被打开过.

一开始我们还没有涉及到分区的信息,所以一开始我们只有 sda 这个概念,而没有 sda1,sda2,sda3…这些概念.这时候我们调用 get_gendisk 得到的 part 一定是 0.所以 1131 行的 if 语句也会执行.而 disk->fops->open 很明显,就是 sd_open.(因为我们在 sd_probe 中曾经设置了 gd->fops 等于&sd_fops.)

但此时此刻我们执行 sd_open 实际上是不做什么正经事儿的.顶多就是测试一下看看 sd_open 能不能执行,如果能执行,那么就返回 0.如果根本就不能执行,那就赶紧汇报错误.

接下来还有几个函数,主要做一些赋值,暂时先飘过.等到适当的时候需要看了再回来看.

而 1146 行这个 rescan_partitions()显然是我们要看的,首先我们在调用 blkdev_get 之前把 bd_invalidated 设置为了 1,所以这个函数这次一定会被执行.从这一刻开始分区信息闯入了我们的生活.这个函数来自 fs/partitions/check.c:

```
int rescan_partitions(struct gendisk *disk, struct block_device *bdev)
{
        struct parsed_partitions *state;
        int p, res;

        if (bdev->bd_part_count)
                return -EBUSY;
        res = invalidate_partition(disk, 0);
        if (res)
                return res;
        bdev->bd_invalidated = 0;
        for (p = 1; p < disk->minors; p++)
                delete_partition(disk, p);
        if (disk->fops->revalidate_disk)
                disk->fops->revalidate_disk(disk);
        if (!get_capacity(disk) || !(state = check_partition(disk, bdev)))
                return 0;
        if (IS_ERR(state))      /* I/O error reading the partition table */
                return -EIO;
```

```
        for (p = 1; p < state->limit; p++) {
                sector_t size = state->parts[p].size;
                sector_t from = state->parts[p].from;
                if (!size)
                        continue;
                if (from + size > get_capacity(disk)) {
                        printk(" %s: p%d exceeds device capacity\n",
                                disk->disk_name, p);
                }
                add_partition(disk, p, from, size, state->parts[p].flags);
#ifdef CONFIG_BLK_DEV_MD
                if (state->parts[p].flags & ADDPART_FLAG_RAID)
                        md_autodetect_dev(bdev->bd_dev+p);
#endif
        }
        kfree(state);
        return 0;
}
```

其实就算我们一行代码都不看也知道这个函数在干嘛,正如我们说的,这个函数执行过后,关于分区的信息我们就算都有了.关于分区,我们是用 struct hd_struct 这么个结构体来表示的,而 struct hd_struct 也正是 struct gendisk 的成员,并且是个二级指针.一开始这个指针并无所指,或者说一开始我们并没有为 struct hd_struct 申请空间,所以我即使不贴出下面这个 delete_partition 函数的代码你也应该知道,此时此刻,它什么也不会干.

```
void delete_partition(struct gendisk *disk, int part)
{
        struct hd_struct *p = disk->part[part-1];
        if (!p)
                return;
        if (!p->nr_sects)
                return;
        disk->part[part-1] = NULL;
        p->start_sect = 0;
        p->nr_sects = 0;
        p->ios[0] = p->ios[1] = 0;
        p->sectors[0] = p->sectors[1] = 0;
        sysfs_remove_link(&p->kobj, "subsystem");
        kobject_unregister(p->holder_dir);
        kobject_uevent(&p->kobj, KOBJ_REMOVE);
        kobject_del(&p->kobj);
        kobject_put(&p->kobj);
}
```

而 revalidate_disk 指针指向的就是 sd_revalidate_disk,这个函数我们在讲述 sd 的时候对它做足了文章.在 sd_probe 调用 add_disk 之前,就已经执行过这个函数,这里只不过是再执行一次罢了.

接着,get_capacity().没有比这个函数更简单的函数了.来自 include/linux/genhd.h:

```
static inline sector_t get_capacity(struct gendisk *disk)
{
        return disk->capacity;
}
```

而 check_partition 就稍微复杂一些了,来自 fs/partitions/check.c:

```
static struct parsed_partitions *
check_partition(struct gendisk *hd, struct block_device *bdev)
{
        struct parsed_partitions *state;
        int i, res, err;

        state = kmalloc(sizeof(struct parsed_partitions), GFP_KERNEL);
        if (!state)
                return NULL;

        disk_name(hd, 0, state->name);
        printk(KERN_INFO " %s:", state->name);
        if (isdigit(state->name[strlen(state->name)-1]))
                sprintf(state->name, "p");

        state->limit = hd->minors;
        i = res = err = 0;
        while (!res && check_part[i]) {
                memset(&state->parts, 0, sizeof(state->parts));
                res = check_part[i++](state, bdev);
                if (res < 0) {
                        /* We have hit an I/O error which we don't report now.
                        * But record it, and let the others do their job.
                        */
                        err = res;
                        res = 0;
                }

        }
        if (res > 0)
                return state;
        if (err)
        /* The partition is unrecognized. So report I/O errors if there were any */
                res = err;
        if (!res)
                printk(" unknown partition table\n");
        else if (warn_no_part)
                printk(" unable to read partition table\n");
```

```
194         kfree(state);
195         return ERR_PTR(res);
196 }
```

首先,struct parsed_partitions 结构体定义于 fs/partitions/check.h 这么一个头文件中:

```
enum { MAX_PART = 256 };

struct parsed_partitions {
        char name[BDEVNAME_SIZE];
        struct {
                sector_t from;
                sector_t size;
                int flags;
        } parts[MAX_PART];
        int next;
        int limit;
};
```

这个结构体是我们用来记录分区信息的.

而 173 行这个 check_part 是何许人物?在 fs/partitions/check.c 中找到了它:

```
int warn_no_part = 1; /*This is ugly: should make genhd removable media aware*/

static int (*check_part[])(struct parsed_partitions *, struct block_device *) = {
        /*
         * Probe partition formats with tables at disk address 0
         * that also have an ADFS boot block at 0xdc0.
         */
#ifdef CONFIG_ACORN_PARTITION_ICS
        adfspart_check_ICS,
#endif
#ifdef CONFIG_ACORN_PARTITION_POWERTEC
        adfspart_check_POWERTEC,
#endif
#ifdef CONFIG_ACORN_PARTITION_EESOX
        adfspart_check_EESOX,
#endif

        /*
         * Now move on to formats that only have partition info at
         * disk address 0xdc0.   Since these may also have stale
         * PC/BIOS partition tables, they need to come before
         * the msdos entry.
         */
#ifdef CONFIG_ACORN_PARTITION_CUMANA
        adfspart_check_CUMANA,
#endif
```

```
#ifdef CONFIG_ACORN_PARTITION_ADFS
        adfspart_check_ADFS,
#endif

#ifdef CONFIG_EFI_PARTITION
        efi_partition,          /* this must come before msdos */
#endif
#ifdef CONFIG_SGI_PARTITION
        sgi_partition,
#endif
#ifdef CONFIG_LDM_PARTITION
        ldm_partition,          /* this must come before msdos */
#endif
#ifdef CONFIG_MSDOS_PARTITION
        msdos_partition,
#endif
#ifdef CONFIG_OSF_PARTITION
        osf_partition,
#endif
#ifdef CONFIG_SUN_PARTITION
        sun_partition,
#endif
#ifdef CONFIG_AMIGA_PARTITION
        amiga_partition,
#endif
#ifdef CONFIG_ATARI_PARTITION
        atari_partition,
#endif
#ifdef CONFIG_MAC_PARTITION
        mac_partition,
#endif
#ifdef CONFIG_ULTRIX_PARTITION
        ultrix_partition,
#endif
#ifdef CONFIG_IBM_PARTITION
        ibm_partition,
#endif
#ifdef CONFIG_KARMA_PARTITION
        karma_partition,
#endif
#ifdef CONFIG_SYSV68_PARTITION
        sysv68_partition,
#endif
        NULL
```

```
};
```

好家伙,一下子定义了这么多函数,要是每个都要看那我他妈还要不要活了.也亏了哥们儿是曾经的复旦大学优秀团员,要不然还不被吓死去了.

不过情况总还没有那么遭,我们不用像某些媒体一样每次都把夸大事实,以至于每年的洪水或干旱都被认定是百年一遇,搞得我们不禁怀疑自己到底活过了几个百年?眼下的情况其实很好对付,除非你就是专门研究分区表格式的,否则这一堆函数你一个也不用看.如果你真是研究分区表格式的,那么 fs/partitions 目录下面的文件你就得仔细看看了,各种格式的都有,你就捡自己需要的看吧.

```
localhost:/usr/src/linux-2.6.22.1 # ls fs/partitions/
Kconfig    acorn.h   atari.c   check.h   ibm.c     karma.h   mac.c     msdos.h   sgi.c     sun.h
ultrix.c  Makefile   amiga.c   atari.h   efi.c     ibm.h     ldm.c     mac.h     osf.c     sgi.h
sysv68.c   ultrix.h  acorn.c   amiga.h   check.c   efi.h     karma.c   ldm.h     msdos.c   osf.h
sun.c     sysv68.h
```

基本上我想说的是,以上那么多个函数其目的就是一个,为了找到分区信息.而且最终分区信息总是会被记录在那个 struct parsed_partitions 结构体的指针.而接下来我们就会用到其中的信息,这其中像 size 啊,from 啊,这些变量的意思不言自明.

然后我们就来到了 add_partition,仍然是来自 fs/partitions/check.c:

```
void add_partition(struct gendisk *disk, int part, sector_t start, sector_t len, int flags)
{
        struct hd_struct *p;

        p = kmalloc(sizeof(*p), GFP_KERNEL);
        if (!p)
                return;

        memset(p, 0, sizeof(*p));
        p->start_sect = start;
        p->nr_sects = len;
        p->partno = part;
        p->policy = disk->policy;

        if (isdigit(disk->kobj.name[strlen(disk->kobj.name)-1]))
snprintf(p->kobj.name,KOBJ_NAME_LEN,"%sp%d",disk->kobj.name,part);
        else
snprintf(p->kobj.name,KOBJ_NAME_LEN,"%s%d",disk->kobj.name,part);
        p->kobj.parent = &disk->kobj;
        p->kobj.ktype = &ktype_part;
        kobject_init(&p->kobj);
        kobject_add(&p->kobj);
        if (!disk->part_uevent_suppress)
                kobject_uevent(&p->kobj, KOBJ_ADD);
        sysfs_create_link(&p->kobj, &block_subsys.kobj, "subsystem");
```

```
396         if (flags & ADDPART_FLAG_WHOLEDISK) {
397                 static struct attribute addpartattr = {
398                         .name = "whole_disk",
399                         .mode = S_IRUSR | S_IRGRP | S_IROTH,
400                         .owner = THIS_MODULE,
401                 };
402
403                 sysfs_create_file(&p->kobj, &addpartattr);
404         }
405         partition_sysfs_add_subdir(p);
406         disk->part[part-1] = p;
407 }
```

有了之前的经验,现在再看这些 kobject 相关的,sysfs 相关的函数就容易多了.

389 行这个 p->kobj.parent = &disk->kobj 保证了我们接下来生成的东西在刚才的目录之下,即 sda1,sda2,…在 sda 目录下.

```
[root@localhost tedkdb]# ls /sys/block/sda/
capability  device  queue  removable  sda10  sda12  sda14  sda2  sda5  sda7  sda9
slaves  subsystem  dev  holders  range  sda1  sda11  sda13  sda15  sda3
sda6  sda8  size  stat  uevent
```

而 395 行 sysfs_create_link 的效果也很显然,

```
[root@localhost tedkdb]# ls -l /sys/block/sda/sda1/subsystem
lrwxrwxrwx 1 root root 0 Dec 13 03:15 /sys/block/sda/sda1/subsystem -> ../../../block
```

而 partition_sysfs_add_subdir 也没什么好说的,来自 fs/partitions/check.c:

```
static inline void partition_sysfs_add_subdir(struct hd_struct *p)
{
        struct kobject *k;

        k = kobject_get(&p->kobj);
        p->holder_dir = kobject_add_dir(k, "holders");
        kobject_put(k);
}
```

添加了 holders 子目录.

```
[root@localhost tedkdb]# ls /sys/block/sda/sda1/
dev  holders  size  start  stat  subsystem  uevent
```

最后,让我们记住这个函数做过的一件事情,对 p 的各个成员进行了赋值,而在函数的结尾处把 disk->part[part-1]指向了 p.也就是说,从此以后,struct hd_struct 这个指针数组里就应该有内容了,而不再是空的.

到这里,rescan_partitions()宣告结束,回到 do_open()中.1183 行,让 bd_openers 这个引用计数增加 1,如果 for_part 有值,那么就让它对应的引用计数也加 1.然后 do_open 也就华丽丽的结束了,像多米诺骨牌一样,__blkdev_get 和 blkdev_get 相继返回.blkdev_put 和 blkdev_get 做的事情基本相反,我们就不看了,只是需要注意,它把刚才增加上去的这两个引用计数给减了回去.

最后,register_disk()中调用的最后一个函数就是 kobject_uevent(),这个函数就是通知用户空间的进程 udevd,告诉它有事件发生了,如果你使用的发行版正确配置了 udev 的配置文件(详见/etc/udev/目录下),那么其效果就是让/dev 目录下面有了相应的设备文件.比如:

```
[root@localhost tedkdb]# ls /dev/sda*
/dev/sda    /dev/sda10   /dev/sda12   /dev/sda14   /dev/sda2   /dev/sda5   /dev/sda7   /dev/sda9
/dev/sda1   /dev/sda11   /dev/sda13   /dev/sda15   /dev/sda3   /dev/sda6   /dev/sda8
```

至于为什么,你可以去阅读关于 udev 的知识,这是用户空间的程序,咱们就不多说了.

浓缩就是精华?(三)

第三个,blk_register_queue().

```
4079 int blk_register_queue(struct gendisk *disk)
4080 {
4081         int ret;
4082
4083         request_queue_t *q = disk->queue;
4084
4085         if (!q || !q->request_fn)
4086                 return -ENXIO;
4087
4088         q->kobj.parent = kobject_get(&disk->kobj);
4089
4090         ret = kobject_add(&q->kobj);
4091         if (ret < 0)
4092                 return ret;
4093
4094         kobject_uevent(&q->kobj, KOBJ_ADD);
4095
4096         ret = elv_register_queue(q);
4097         if (ret) {
4098                 kobject_uevent(&q->kobj, KOBJ_REMOVE);
4099                 kobject_del(&q->kobj);
4100                 return ret;
4101         }
4102
4103         return 0;
4104 }
```

首先,4090 行这个 kobject_add 很好解释,在/sys/block/sda/目录下面又多一个子目录而已,但问题是,这个 q 究竟是什么?这里我们把 disk->queue 赋给了它,而 disk->queue 又是什么呢?回过头去看 sd_probe(),当时我们有这么一句,

```
1662         gd->queue = sdkp->device->request_queue;
```

而 sdkp 是 struct scsi_disk 结构体指针,其 device 成员是 struct scsi_device 指针,那么这个 request_queue 呢?是 struct request_queue 结构体指针,表示的是一个请求队列.但它是从哪儿来的呢?一路走来的兄弟们可能会猜到,事实上 scsi 设备驱动和 usb 设备驱动有一点是相同的,在它们的 probe 函数被调用之前,核心层实际上已经为它们做了许多工作了.比如 usb 那边就

是为 usb 设备申请 usb_device 结构体变量,而这边也是如此,申请了 scsi_device 结构体变量,为它的一些成员赋好了值,这其中就包括了这个请求队列.

准确地说,在 scsi 总线扫描的时候,每当探测到一个设备,就会调用 scsi_alloc_sdev()函数,这个函数我们无意多说,但是可以告诉你的是,它会调用一个叫做 scsi_alloc_queue()的函数.而这个函数涉及到很多 block 层提供的函数,所以我们不得不从这里开始看起,来自 drivers/scsi/scsi_lib.c:

```
struct request_queue *__scsi_alloc_queue(struct Scsi_Host *shost,
                                         request_fn_proc *request_fn)
{
        struct request_queue *q;

        q = blk_init_queue(request_fn, NULL);
        if (!q)
                return NULL;

        blk_queue_max_hw_segments(q, shost->sg_tablesize);
        blk_queue_max_phys_segments(q, SCSI_MAX_PHYS_SEGMENTS);
        blk_queue_max_sectors(q, shost->max_sectors);
        blk_queue_bounce_limit(q, scsi_calculate_bounce_limit(shost));
        blk_queue_segment_boundary(q, shost->dma_boundary);

        if (!shost->use_clustering)
                clear_bit(QUEUE_FLAG_CLUSTER, &q->queue_flags);
        return q;
}
EXPORT_SYMBOL(__scsi_alloc_queue);

struct request_queue *scsi_alloc_queue(struct scsi_device *sdev)
{
        struct request_queue *q;

        q = __scsi_alloc_queue(sdev->host, scsi_request_fn);
        if (!q)
                return NULL;

        blk_queue_prep_rq(q, scsi_prep_fn);
        blk_queue_issue_flush_fn(q, scsi_issue_flush_fn);
        blk_queue_softirq_done(q, scsi_softirq_done);
        return q;
}
```

这两个函数因为调用关系所以一并贴了出来.

我们首先要看的很自然就是 blk_init_queue(),它来自 block/ll_rw_blk.c:

```
/**
 * blk_init_queue  - prepare a request queue for use with a block device
```

```
 * @rfn:    The function to be called to process requests that have been
 *             placed on the queue.
 * @lock: Request queue spin lock
 *
 * Description:
 *       If a block device wishes to use the standard request handling procedures,
 *       which sorts requests and coalesces adjacent requests, then it must
 *       call blk_init_queue().    The function @rfn will be called when there
 *       are requests on the queue that need to be processed.    If the device
 *       supports plugging, then @rfn may not be called immediately when requests
 *       are available on the queue, but may be called at some time later instead.
 *       Plugged queues are generally unplugged when a buffer belonging to one
 *       of the requests on the queue is needed, or due to memory pressure.
 *
 *       @rfn is not required, or even expected, to remove all requests off the
 *       queue, but only as many as it can handle at a time.    If it does leave
 *       requests on the queue, it is responsible for arranging that the requests
 *       get dealt with eventually.
 *
 *       The queue spin lock must be held while manipulating the requests on the
 *       request queue; this lock will be taken also from interrupt context, so irq
 *       disabling is needed for it.
 *
 *       Function returns a pointer to the initialized request queue, or NULL if
 *       it didn't succeed.
 *
 * Note:
 *       blk_init_queue() must be paired with a blk_cleanup_queue() call
 *       when the block device is deactivated (such as at module unload).
 **/

request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
{
        return blk_init_queue_node(rfn, lock, -1);
}
EXPORT_SYMBOL(blk_init_queue);

request_queue_t *
blk_init_queue_node(request_fn_proc *rfn, spinlock_t *lock, int node_id)
{
        request_queue_t *q = blk_alloc_queue_node(GFP_KERNEL, node_id);

        if (!q)
                return NULL;
```

```

	q->node = node_id;
	if (blk_init_free_list(q)) {
		kmem_cache_free(requestq_cachep, q);
		return NULL;
	}

	/*
	 * if caller didn't supply a lock, they get per-queue locking with
	 * our embedded lock
	 */
	if (!lock) {
		spin_lock_init(&q->__queue_lock);
		lock = &q->__queue_lock;
	}

	q->request_fn		= rfn;
	q->prep_rq_fn		= NULL;
	q->unplug_fn		= generic_unplug_device;
	q->queue_flags		= (1 << QUEUE_FLAG_CLUSTER);
	q->queue_lock		= lock;

	blk_queue_segment_boundary(q, 0xffffffff);

	blk_queue_make_request(q, __make_request);
	blk_queue_max_segment_size(q, MAX_SEGMENT_SIZE);

	blk_queue_max_hw_segments(q, MAX_HW_SEGMENTS);
	blk_queue_max_phys_segments(q, MAX_PHYS_SEGMENTS);

	q->sg_reserved_size = INT_MAX;

	/*
	 * all done
	 */
	if (!elevator_init(q, NULL)) {
		blk_queue_congestion_threshold(q);
		return q;
	}

	blk_put_queue(q);
	return NULL;
}
```

别看这些函数都很可怕,真正我们目前需要关注的其实只是其中的某几个而已.它们这个

blk_alloc_queue_node 和 elevator_init().前者来自 block/ll_rw_blk.c,后者则来自 block/elevator.c:

```
1836 request_queue_t *blk_alloc_queue_node(gfp_t gfp_mask, int node_id)
1837 {
1838         request_queue_t *q;
1839
1840         q = kmem_cache_alloc_node(requestq_cachep, gfp_mask, node_id);
1841         if (!q)
1842                 return NULL;
1843
1844         memset(q, 0, sizeof(*q));
1845         init_timer(&q->unplug_timer);
1846
1847         snprintf(q->kobj.name, KOBJ_NAME_LEN, "%s", "queue");
1848         q->kobj.ktype = &queue_ktype;
1849         kobject_init(&q->kobj);
1850
1851         q->backing_dev_info.unplug_io_fn = blk_backing_dev_unplug;
1852         q->backing_dev_info.unplug_io_data = q;
1853
1854         mutex_init(&q->sysfs_lock);
1855
1856         return q;
1857 }
```

还记得本故事最早时期讲的那个 blk_dev_init 吧,当时我们调用 kmem_cache_create()申请了一个内存池 request_cachep,现在就该用它了.从这个池子里申请了一个 struct request_queue_t 结构体的空间,给了指针 q,然后 1844 行初始化为 0.而 1847 行让 q 的 kobj.name 等于"queue",这就是为什么今后我们在/sys/block/sda/目录下面能看到一个叫做"queue"的目录.

```
[root@localhost ~]# ls /sys/block/sda/
capability   device   queue   removable   sda10   sda12   sda14   sda2   sda5   sda7   sda9
slaves   subsystem  dev         holders   range   sda1        sda11   sda13   sda15   sda3
sda6   sda8   size   stat     uevent
```

而这个 queue 目录下面的内容是什么呢?

```
[root@localhost ~]# ls /sys/block/sda/queue/
iosched   max_hw_sectors_kb   max_sectors_kb   nr_requests   read_ahead_kb   scheduler
```

这几个文件从哪来的?注意 1848 行那个 queue_ktype.

```
4073 static struct kobj_type queue_ktype = {
4074         .sysfs_ops      = &queue_sysfs_ops,
4075         .default_attrs  = default_attrs,
4076         .release        = blk_release_queue,
4077 };
```

如果你真懂设备模型,那么你一定会去查看这个 default_attrs 是什么,

```
3988 static struct queue_sysfs_entry queue_requests_entry = {
3989         .attr = {.name = "nr_requests", .mode = S_IRUGO | S_IWUSR },
3990         .show = queue_requests_show,
```

```
        .store = queue_requests_store,
};

static struct queue_sysfs_entry queue_ra_entry = {
        .attr = {.name = "read_ahead_kb", .mode = S_IRUGO | S_IWUSR },
        .show = queue_ra_show,
        .store = queue_ra_store,
};

static struct queue_sysfs_entry queue_max_sectors_entry = {
        .attr = {.name = "max_sectors_kb", .mode = S_IRUGO | S_IWUSR },
        .show = queue_max_sectors_show,
        .store = queue_max_sectors_store,
};

static struct queue_sysfs_entry queue_max_hw_sectors_entry = {
        .attr = {.name = "max_hw_sectors_kb", .mode = S_IRUGO },
        .show = queue_max_hw_sectors_show,
};

static struct queue_sysfs_entry queue_iosched_entry = {
        .attr = {.name = "scheduler", .mode = S_IRUGO | S_IWUSR },
        .show = elv_iosched_show,
        .store = elv_iosched_store,
};

static struct attribute *default_attrs[] = {
        &queue_requests_entry.attr,
        &queue_ra_entry.attr,
        &queue_max_hw_sectors_entry.attr,
        &queue_max_sectors_entry.attr,
        &queue_iosched_entry.attr,
        NULL,
};
```

看到了吗?是一个指针数组,按照设备模型的理论来说,这些就是定义了一些属性,kobject 的属性,看到这些属性的 name 是不是和刚才那个 queue 目录下面的文件名字是一样的?没错,queue 目录下面每个文件就是和这里这些属性一一对应的.不过有一个东西例外,它就是 iosched,这不是一个文件,这是一个目录.

```
[root@localhost ~]# ls -l /sys/block/sdf/queue/
total 0
drwxr-xr-x 2 root root      0 Dec 14 02:46 iosched
-r--r--r-- 1 root root 4096 Dec 14 06:21 max_hw_sectors_kb
-rw-r--r-- 1 root root 4096 Dec 14 06:21 max_sectors_kb
-rw-r--r-- 1 root root 4096 Dec 14 06:21 nr_requests
```

```
-rw-r--r-- 1 root root 4096 Dec 14 06:21 read_ahead_kb
-rw-r--r-- 1 root root 4096 Dec 14 06:21 scheduler
[root@localhost ~]# ls /sys/block/sdf/queue/iosched/
back_seek_max        fifo_expire_async    quantum         slice_async_rq    slice_sync
back_seek_penalty    fifo_expire_sync     slice_async     slice_idle
```

关于这个目录,我们来看另一个函数,elevator_init(),来自 block/elevator.c:

```
int elevator_init(request_queue_t *q, char *name)
{
        struct elevator_type *e = NULL;
        struct elevator_queue *eq;
        int ret = 0;
        void *data;

        INIT_LIST_HEAD(&q->queue_head);
        q->last_merge = NULL;
        q->end_sector = 0;
        q->boundary_rq = NULL;

        if (name && !(e = elevator_get(name)))
                return -EINVAL;

        if (!e && *chosen_elevator && !(e = elevator_get(chosen_elevator)))
                printk("I/O scheduler %s not found\n", chosen_elevator);

        if (!e && !(e = elevator_get(CONFIG_DEFAULT_IOSCHED))) {
                printk("Default I/O scheduler not found, using no-op\n");
                e = elevator_get("noop");
        }

        eq = elevator_alloc(q, e);
        if (!eq)
                return -ENOMEM;

        data = elevator_init_queue(q, eq);
        if (!data) {
                kobject_put(&eq->kobj);
                return -ENOMEM;
        }

        elevator_attach(q, eq, data);
        return ret;
}
```

重点关注 elevator_alloc().

```
static elevator_t *elevator_alloc(request_queue_t *q, struct elevator_type *e)
```

```
180 {
181         elevator_t *eq;
182         int i;
183
184         eq = kmalloc_node(sizeof(elevator_t), GFP_KERNEL, q->node);
185         if (unlikely(!eq))
186                 goto err;
187
188         memset(eq, 0, sizeof(*eq));
189         eq->ops = &e->ops;
190         eq->elevator_type = e;
191         kobject_init(&eq->kobj);
192         snprintf(eq->kobj.name, KOBJ_NAME_LEN, "%s", "iosched");
193         eq->kobj.ktype = &elv_ktype;
194         mutex_init(&eq->sysfs_lock);
195
196         eq->hash = kmalloc_node(sizeof(struct hlist_head) * ELV_HASH_ENTRIES,
197                                         GFP_KERNEL, q->node);
198         if (!eq->hash)
199                 goto err;
200
201         for (i = 0; i < ELV_HASH_ENTRIES; i++)
202                 INIT_HLIST_HEAD(&eq->hash[i]);
203
204         return eq;
205 err:
206         kfree(eq);
207         elevator_put(e);
208         return NULL;
209 }
```

无非就是申请一个 struct elevator_t 结构体变量的空间并且初始化为 0.

而真正引发我们兴趣的是 192 行,很显然,就是因为这里把 eq 的 kobj 的 name 设置为”iosched”,才会让我们在 queue 目录下看到那个”iosched”子目录.

而这个子目录下那些乱七八糟的文件又来自哪里呢?正是下面这个 elv_register_queue()函数,这个我们在 blk_register_queue()中调用的函数.

```
931 int elv_register_queue(struct request_queue *q)
932 {
933         elevator_t *e = q->elevator;
934         int error;
935
936         e->kobj.parent = &q->kobj;
937
938         error = kobject_add(&e->kobj);
939         if (!error) {
```

```
940                 struct elv_fs_entry *attr = e->elevator_type->elevator_attrs;
941                 if (attr) {
942                         while (attr->attr.name) {
943                                 if (sysfs_create_file(&e->kobj, &attr->attr))
944                                         break;
945                                 attr++;
946                         }
947                 }
948                 kobject_uevent(&e->kobj, KOBJ_ADD);
949         }
950         return error;
951 }
```

936 行保证了,iosched 是出现在 queue 目录下而不是出现在别的地方,而 942 行这个 while 循环则是创建 iosched 目录下面那么多文件的.我们先来看这个 attr 到底是什么,这里它指向了 e->elevator_type->elevator_attrs,而在刚才那个 elevator_alloc()函数中,190 行,我们看到了 eq->elevator_type 被赋上了 e,回溯至 elevator_init(),我们来看 e 究竟是什么.

首先,当我们在 blk_init_queue_node()中调用 elevator_init 的时候,传递的第二个参数是 NULL,即 name 指针是 NULL.

那么很明显,235 行和 238 行这两个 if 语句对于 e 的取值至关重要.而到了现在,传说中的电梯算法也不得不介绍了.

话说,在 Linux 中如果你要读写一些磁盘数据,你需要创建一个 block device request.这个 request 基本上描述了请求的扇区以及操作的类型.(即,你是要读还是要写)而对于一个设备来说,请求多了自然就应该使用某种数据结构来存储它们,很显然我们会使用队列,于是,Linux 中为每个块设备准备了一个请求队列,即所谓的 request queue.每接收到一个请求,就把它插入到 request queue 这个队列中去.

那么这里有一个问题,比如说队列里有好几十个请求,那么谁先执行谁后执行呢?是不是谁先提交就先执行谁?不是.这里需要调度,否则磁盘的性能就会很糟糕.

比如说英超联赛,拿我家切尔西来举例,一个赛季 38 场英超联赛,如果说赛程是一场主场一场客场一场主场一场客场一场主场一场客场…,那么这样的赛程一定是很糟糕的,因为球员要不停的奔波,每踢一场比赛就得进行一次车旅劳顿,球员纷纷疲于奔命,状态根本无法保证,那么比这个好点的赛程是什么?比如,连续几个主场,连续几个客场,那么至少在连续的这几个主场作战的期间球员们不用把体力消耗在旅途中,而在连续的几个客场中,怎么安排又有区别了,假设有这样四个连续的客场,对手分别是曼联,曼城,利物浦,埃弗顿,那么理想的赛程是,踢曼联和踢曼城这两场相邻,踢利物浦和踢埃弗顿这两场相邻,这样旅途耗费时间最少,那么最恶劣的赛程是什么呢?先去曼彻斯特踢曼联,然后去利物浦踢利物浦,然后又折回曼彻斯特踢曼城,再然后又杀回利物浦去战埃弗顿,很显然这样的赛程是最艰苦的,这就是所谓的魔鬼赛程.所以赛程的好坏很有可能影响一支球队的战绩.

而磁盘调度也是如此.磁头的移来移去是很费时间的,如果我这一次要读的扇区在”曼彻斯特”,下一次要读的扇区又在”利物浦”,下下次又回到”曼彻斯特”,然后又去到”利物浦”,这样显然会影响磁盘的性能.所以如果我们能够改变这种顺序,能够让前后两次访问的扇区尽量在相邻的位置,那么毫无疑问将提高磁盘的性能.而完成这项工作的叫做 IO 调度器.(The I/O Scheduler)

IO 调度器的总体目标是希望让磁头能够总是往一个方向移动,移动到底了再往反方向走,这恰恰就是现实生活中的电梯模型,所以 IO 调度器也被叫做电梯.(elevator)而相应的算法也就被叫做电梯算法.而 Linux 中 IO 调度的电梯算法有好几种,一个叫做 as(Anticipatory),一个叫做

cfq(Complete Fairness Queueing),一个叫做 deadline,还有一个叫做 noop(No Operation).具体使用哪种算法我们可以在启动的时候通过内核参数 elevator 来指定.比如在我的 grub 配置文件中就这样设置过:

```
###Don't change this comment - YaST2 identifier: Original name: linux###
title Linux
    kernel (hd0,0)/vmlinuz root=/dev/sda3 selinux=0 resume=/dev/sda2 splash=silent elevator=cfq showopts console=ttyS0,9600 console=tty0
    initrd (hd0,0)/initrd
```

让 elevator=cfq,因此 cfq 算法将是我们的 IO 调度器所采用的算法.而另一方面我们也可以单独的为某个设备指定它所采用的 IO 调度算法,这就通过修改在/sys/block/sda/queue/目录下面的 scheduler 文件.比如我们可以先看一下我的这块硬盘:

```
[root@localhost ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

可以看到我们这里采用的是 cfq.

Ok,现在还不是细说这几种算法的时刻,我们接着刚才的话题,还看 elevator_init().

首先 chosen_elevator 是定义于 block/elevator.c 中的一个字符串.

```
static char chosen_elevator[16];
```

这个字符串就是用来记录启动参数 elevator 的.如果没有设置,那就没有值.

而 CONFIG_DEFAULT_IOSCHED 是一个编译选项.它就是一字符串,在编译内核的时候设置的,比如我的是 cfq.

```
CONFIG_DEFAULT_IOSCHED="cfq"
```

你当然也可以选择其它三个,看个人喜好了,喜欢哪个就选择哪个.我的建议是,喜欢的就要拥有她,不要害怕结果.总之这个字符串会传递给 elevator_get 这个来自 block/elevator.c 的函数:

```
static struct elevator_type *elevator_get(const char *name)
{
        struct elevator_type *e;

        spin_lock(&elv_list_lock);

        e = elevator_find(name);
        if (e && !try_module_get(e->elevator_owner))
                e = NULL;

        spin_unlock(&elv_list_lock);

        return e;
}
```

这里 elevator_find()也来自同一个文件.

```
static struct elevator_type *elevator_find(const char *name)
{
        struct elevator_type *e;
        struct list_head *entry;

        list_for_each(entry, &elv_list) {
```

```

                e = list_entry(entry, struct elevator_type, list);

                if (!strcmp(e->elevator_name, name))
                        return e;
        }

        return NULL;
}
```

&elv_list 是什么?首先,复旦南区后门卖炒饭的那儿对夫妻都知道 elv_list 一定是一个链表.但是这张链表具体是什么内容呢?事实上,甭管是这四种算法中的哪一种,在正式登台演出之前,都需要做一些初始化,初始化过程中最本质的一项工作就是调用 elv_register()函数来注册自己.而这个注册主要就是往 elv_list 这张链表里登记.

```
int elv_register(struct elevator_type *e)
{
        char *def = "";

        spin_lock(&elv_list_lock);
        BUG_ON(elevator_find(e->elevator_name));
        list_add_tail(&e->list, &elv_list);
        spin_unlock(&elv_list_lock);

        if (!strcmp(e->elevator_name, chosen_elevator) ||
                        (!*chosen_elevator &&
                         !strcmp(e->elevator_name, CONFIG_DEFAULT_IOSCHED)))
                                def = " (default)";

        printk(KERN_INFO "io scheduler %s registered%s\n", e->elevator_name, def);
        return 0;
}
```

看到 list_add_tail 那行了吗.那么这个 elevator_type 结构体又代表了什么呢?正如其名,它代表着一种电梯算法的类型,比如对于 cfq,在 cfq-iosched.c 文件中,就定义了这么一个结构体变量 iosched_cfq.

```
static struct elevator_type iosched_cfq = {
        .ops = {
                .elevator_merge_fn =            cfq_merge,
                .elevator_merged_fn =           cfq_merged_request,
                .elevator_merge_req_fn =        cfq_merged_requests,
                .elevator_allow_merge_fn =      cfq_allow_merge,
                .elevator_dispatch_fn =         cfq_dispatch_requests,
                .elevator_add_req_fn =          cfq_insert_request,
                .elevator_activate_req_fn =     cfq_activate_request,
```

```
                        .elevator_deactivate_req_fn =    cfq_deactivate_request,
                        .elevator_queue_empty_fn =       cfq_queue_empty,
                        .elevator_completed_req_fn =     cfq_completed_request,
                        .elevator_former_req_fn =        elv_rb_former_request,
                        .elevator_latter_req_fn =        elv_rb_latter_request,
                        .elevator_set_req_fn =           cfq_set_request,
                        .elevator_put_req_fn =           cfq_put_request,
                        .elevator_may_queue_fn =         cfq_may_queue,
                        .elevator_init_fn =              cfq_init_queue,
                        .elevator_exit_fn =              cfq_exit_queue,
                        .trim =                          cfq_free_io_context,
                },
                .elevator_attrs =       cfq_attrs,
                .elevator_name =        "cfq",
                .elevator_owner =       THIS_MODULE,
        };
```

同样,我们可以找到,对于 noop,也有类似的变量.

```
        static struct elevator_type elevator_noop = {
                .ops = {
                        .elevator_merge_req_fn          = noop_merged_requests,
                        .elevator_dispatch_fn           = noop_dispatch,
                        .elevator_add_req_fn            = noop_add_request,
                        .elevator_queue_empty_fn        = noop_queue_empty,
                        .elevator_former_req_fn         = noop_former_request,
                        .elevator_latter_req_fn         = noop_latter_request,
                        .elevator_init_fn               = noop_init_queue,
                        .elevator_exit_fn               = noop_exit_queue,
                },
                .elevator_name = "noop",
                .elevator_owner = THIS_MODULE,
        };
```

所以,我们就知道这个 e 到底是要得到什么了,如果你什么都没设置,那么它只能选择最差的那个,noop.于是到现在我们终于明白 elv_register_queue()中那个 e->elevator_type 是啥了.而我们要的是 e->elevator_type->elevator_attrs.对于 cfq,很显然,它就是 cfq_attrs.在 block/cfq-iosched.c 中:

```
        static struct elv_fs_entry cfq_attrs[] = {
                CFQ_ATTR(quantum),
                CFQ_ATTR(fifo_expire_sync),
                CFQ_ATTR(fifo_expire_async),
                CFQ_ATTR(back_seek_max),
                CFQ_ATTR(back_seek_penalty),
                CFQ_ATTR(slice_sync),
                CFQ_ATTR(slice_async),
                CFQ_ATTR(slice_async_rq),
```

```
        CFQ_ATTR(slice_idle),
        __ATTR_NULL
};
```

所以,那个 while 循环的 sysfs_create_file 的功绩就是以上面这个数组的元素的名字建立一堆的文件.而这正是我们在/sys/block/sdf/queue/iosched/目录下面看到的那些文件.

至此,elv_register_queue 就算是结束了,从而 blk_register_queue()也就结束了,而 add_disk 这个不朽的函数终于大功告成.这一刻开始,整个块设备工作的大舞台就已经搭好了.对于 sd 那边来说,sd_probe 就是在结束 add_disk 之后结束的.

看完之后,我深深的吸了一口气,我不得不承认,add_disk 这个函数,这个只有四行代码的函数,很好,很强大.写代码毕竟不是写琼瑶剧本,不可能像<<一帘幽梦>>里的一句"我爱你",需要用四十几集来诠释,那才叫一个深刻呢!

scsi 命令的前世今生(一)

现在我们块设备也有了,队列也有了,要提交请求也就可以开始提交了.那就让我们来研究一下如何提交请求如何处理请求吧.不过哥们儿有言在先,出错处理的那些乱七八糟的代码咱们就不理睬了.

仍然以 scsi 磁盘举例,最初 scsi 这边发送的是 scsi 命令,可是从 block 走就得变成 request,然而走到 usb-storage 那边又得变回 scsi 命令.换言之,这整个过程 scsi 命令要变两次身.

首先让我们从 sd 那边很常用的一个函数开始,我们来看 scsi 命令是如何在光天化日之下被偷梁换柱的变成了 request,这个函数就是 scsi_execute_req().来自 drivers/scsi/scsi_lib.c:

```
int scsi_execute_req(struct scsi_device *sdev, const unsigned char *cmd,
                     int data_direction, void *buffer, unsigned bufflen,
                     struct scsi_sense_hdr *sshdr, int timeout, int retries)
{
        char *sense = NULL;
        int result;

        if (sshdr) {
                sense = kzalloc(SCSI_SENSE_BUFFERSIZE, GFP_NOIO);
                if (!sense)
                        return DRIVER_ERROR << 24;
        }
        result = scsi_execute(sdev, cmd, data_direction, buffer, bufflen,
                                  sense, timeout, retries, 0);
        if (sshdr)
                scsi_normalize_sense(sense, SCSI_SENSE_BUFFERSIZE, sshdr);

        kfree(sense);
        return result;
}
```

这里面最需要关注的就是一个函数,scsi_execute(),来自同一个文件.

```
/**
 * scsi_execute - insert request and wait for the result
 * @sdev:          scsi device
 * @cmd:           scsi command
 * @data_direction: data direction
 * @buffer:        data buffer
 * @bufflen:       len of buffer
 * @sense:         optional sense buffer
 * @timeout:       request timeout in seconds
 * @retries:       number of times to retry request
 * @flags:         or into request flags;
 *
 * returns the req->errors value which is the scsi_cmnd result
 * field.
 **/
int scsi_execute(struct scsi_device *sdev, const unsigned char *cmd,
                 int data_direction, void *buffer, unsigned bufflen,
                 unsigned char *sense, int timeout, int retries, int flags)
{
        struct request *req;
        int write = (data_direction == DMA_TO_DEVICE);
        int ret = DRIVER_ERROR << 24;

        req = blk_get_request(sdev->request_queue, write, __GFP_WAIT);

        if (bufflen &&  blk_rq_map_kern(sdev->request_queue, req,
                                        buffer, bufflen, __GFP_WAIT))
                goto out;

        req->cmd_len = COMMAND_SIZE(cmd[0]);
        memcpy(req->cmd, cmd, req->cmd_len);
        req->sense = sense;
        req->sense_len = 0;
        req->retries = retries;
        req->timeout = timeout;
        req->cmd_type = REQ_TYPE_BLOCK_PC;
        req->cmd_flags |= flags | REQ_QUIET | REQ_PREEMPT;

        /*
         * head injection *required* here otherwise quiesce won't work
         */
        blk_execute_rq(req->q, NULL, req, 1);

        ret = req->errors;
```

```
out:
        blk_put_request(req);

        return ret;
}
```

首先被调用的是 blk_get_request.来自 block/ll_rw_blk.c:

```
2215 struct request *blk_get_request(request_queue_t *q, int rw, gfp_t gfp_mask)
2216 {
2217         struct request *rq;
2218
2219         BUG_ON(rw != READ && rw != WRITE);
2220
2221         spin_lock_irq(q->queue_lock);
2222         if (gfp_mask & __GFP_WAIT) {
2223                 rq = get_request_wait(q, rw, NULL);
2224         } else {
2225                 rq = get_request(q, rw, NULL, gfp_mask);
2226                 if (!rq)
2227                         spin_unlock_irq(q->queue_lock);
2228         }
2229         /* q->queue_lock is unlocked at this point */
2230
2231         return rq;
2232 }
```

注意到我们调用这个函数的时候,第二个参数确实是__GFP_WAIT.所以 2223 行会被执行.get_request_wait()来自同一个文件:

```
static struct request *get_request_wait(request_queue_t *q, int rw_flags,
                                        struct bio *bio)
{
        const int rw = rw_flags & 0x01;
        struct request *rq;

        rq = get_request(q, rw_flags, bio, GFP_NOIO);
        while (!rq) {
                DEFINE_WAIT(wait);
                struct request_list *rl = &q->rq;

                prepare_to_wait_exclusive(&rl->wait[rw], &wait,
                                TASK_UNINTERRUPTIBLE);

                rq = get_request(q, rw_flags, bio, GFP_NOIO);

                if (!rq) {
                        struct io_context *ioc;
```

```

                        blk_add_trace_generic(q, bio, rw, BLK_TA_SLEEPRQ);

                        __generic_unplug_device(q);
                        spin_unlock_irq(q->queue_lock);
                        io_schedule();

                        /*
                         * After sleeping, we become a "batching" process and
                         * will be able to allocate at least one request, and
                         * up to a big batch of them for a small period time.
                         * See ioc_batching, ioc_set_batching
                         */
                        ioc = current_io_context(GFP_NOIO, q->node);
                        ioc_set_batching(q, ioc);

                        spin_lock_irq(q->queue_lock);
                }
                finish_wait(&rl->wait[rw], &wait);
        }

        return rq;
}
```

而真正被调用的又是 get_request(),仍然是来自同一个文件.

```
/*
 * Get a free request, queue_lock must be held.
 * Returns NULL on failure, with queue_lock held.
 * Returns !NULL on success, with queue_lock *not held*.
 */
static struct request *get_request(request_queue_t *q, int rw_flags,
                                   struct bio *bio, gfp_t gfp_mask)
{
        struct request *rq = NULL;
        struct request_list *rl = &q->rq;
        struct io_context *ioc = NULL;
        const int rw = rw_flags & 0x01;
        int may_queue, priv;

        may_queue = elv_may_queue(q, rw_flags);
        if (may_queue == ELV_MQUEUE_NO)
                goto rq_starved;

        if (rl->count[rw]+1 >= queue_congestion_on_threshold(q)) {
                if (rl->count[rw]+1 >= q->nr_requests) {
```

```
				ioc = current_io_context(GFP_ATOMIC, q->node);
				/*
				 * The queue will fill after this allocation, so set
				 * it as full, and mark this process as "batching".
				 * This process will be allowed to complete a batch of
				 * requests, others will be blocked.
				 */
				if (!blk_queue_full(q, rw)) {
					ioc_set_batching(q, ioc);
					blk_set_queue_full(q, rw);
				} else {
					if (may_queue != ELV_MQUEUE_MUST
							&& !ioc_batching(q, ioc)) {
						/*
						 * The queue is full and the allocating
						 * process is not a "batcher", and not
						 * exempted by the IO scheduler
						 */
						goto out;
					}
				}
			}
			blk_set_queue_congested(q, rw);
		}

		/*
		 * Only allow batching queuers to allocate up to 50% over the defined
		 * limit of requests, otherwise we could have thousands of requests
		 * allocated with any setting of ->nr_requests
		 */
		if (rl->count[rw] >= (3 * q->nr_requests / 2))
			goto out;

		rl->count[rw]++;
		rl->starved[rw] = 0;

		priv = !test_bit(QUEUE_FLAG_ELVSWITCH, &q->queue_flags);
		if (priv)
			rl->elvpriv++;

		spin_unlock_irq(q->queue_lock);

		rq = blk_alloc_request(q, rw_flags, priv, gfp_mask);
		if (unlikely(!rq)) {
```

```
			/*
			 * Allocation failed presumably due to memory. Undo anything
			 * we might have messed up.
			 *
			 * Allocating task should really be put onto the front of the
			 * wait queue, but this is pretty rare.
			 */
			spin_lock_irq(q->queue_lock);
			freed_request(q, rw, priv);

			/*
			 * in the very unlikely event that allocation failed and no
			 * requests for this direction was pending, mark us starved
			 * so that freeing of a request in the other direction will
			 * notice us. another possible fix would be to split the
			 * rq mempool into READ and WRITE
			 */
rq_starved:
			if (unlikely(rl->count[rw] == 0))
				rl->starved[rw] = 1;

			goto out;
		}

		/*
		 * ioc may be NULL here, and ioc_batching will be false. That's
		 * OK, if the queue is under the request limit then requests need
		 * not count toward the nr_batch_requests limit. There will always
		 * be some limit enforced by BLK_BATCH_TIME.
		 */
		if (ioc_batching(q, ioc))
			ioc->nr_batch_requests--;

		rq_init(q, rq);

		blk_add_trace_generic(q, bio, rw, BLK_TA_GETRQ);
out:
		return rq;
}
```

这个 elv_may_queue 来自 block/elevator.c:

```
int elv_may_queue(request_queue_t *q, int rw)
{
		elevator_t *e = q->elevator;

```

```
        if (e->ops->elevator_may_queue_fn)
                return e->ops->elevator_may_queue_fn(q, rw);

        return ELV_MQUEUE_MAY;
}
```

属于我们的那个elevator_t结构体变量是当初我们在elevator_init()中调用elevator_alloc()申请的.它的 ops 显然是和具体我们采用了哪种电梯有关系的.这里我们为了简便起见,做一个最不要脸的选择,选择”noop”,这种最简单最原始的机制.再一次贴出它的 elevator_type.

```
static struct elevator_type elevator_noop = {
        .ops = {
                .elevator_merge_req_fn          = noop_merged_requests,
                .elevator_dispatch_fn           = noop_dispatch,
                .elevator_add_req_fn            = noop_add_request,
                .elevator_queue_empty_fn        = noop_queue_empty,
                .elevator_former_req_fn         = noop_former_request,
                .elevator_latter_req_fn         = noop_latter_request,
                .elevator_init_fn               = noop_init_queue,
                .elevator_exit_fn               = noop_exit_queue,
        },
        .elevator_name = "noop",
        .elevator_owner = THIS_MODULE,
};
```

是不是觉得很开心. 对于我们选择的这种 noop 的电梯,elevator_may_queue_fn 根本就没有定义哎.虽然我们这样做很无耻,但是谁叫我们不幸生在现在的中国呢?只要我们够作践,够胆大,够无耻,够疯狂,所谓的道德底线不是”大底”,重心可以下移,完全有向下突破的机会.

带着一个返回值 ELV_MQUEUE_MAY,我们返回到 get_request()中来.rl 又是什么呢?2072 行我们让它指向了 q->rq.在这样一个危急关头,我不得不搬出一个复杂的结构体了,它就是 request_queue,或者叫 request_queue_t,定义于 include/linux/blkdev.h:

```
struct request_queue;
typedef struct request_queue request_queue_t;
struct request_queue
{
        /*
         * Together with queue_head for cacheline sharing
         */
        struct list_head        queue_head;
        struct request          *last_merge;
        elevator_t              *elevator;

        /*
         * the queue request freelist, one for reads and one for writes
         */
        struct request_list     rq;

```

```
        request_fn_proc              *request_fn;
        make_request_fn              *make_request_fn;
        prep_rq_fn                   *prep_rq_fn;
        unplug_fn                    *unplug_fn;
        merge_bvec_fn                *merge_bvec_fn;
        issue_flush_fn               *issue_flush_fn;
        prepare_flush_fn             *prepare_flush_fn;
        softirq_done_fn              *softirq_done_fn;

        /*
         * Dispatch queue sorting
         */
        sector_t                     end_sector;
        struct request               *boundary_rq;

        /*
         * Auto-unplugging state
         */
        struct timer_list            unplug_timer;
        int                          unplug_thresh;    /* After this many requests */
        unsigned long                unplug_delay;     /* After this many jiffies */
        struct work_struct           unplug_work;

        struct backing_dev_info backing_dev_info;

        /*
         * The queue owner gets to use this for whatever they like.
         * ll_rw_blk doesn't touch it.
         */
        void                         *queuedata;

        /*
         * queue needs bounce pages for pages above this limit
         */
        unsigned long                bounce_pfn;
        gfp_t                        bounce_gfp;

        /*
         * various queue flags, see QUEUE_* below
         */
        unsigned long                queue_flags;

        /*
         * protects queue structures from reentrancy. ->__queue_lock should
```

```
         * _never_ be used directly, it is queue private. always use
         * ->queue_lock.
         */
        spinlock_t              __queue_lock;
        spinlock_t              *queue_lock;

        /*
         * queue kobject
         */
        struct kobject kobj;

        /*
         * queue settings
         */
        unsigned long           nr_requests;    /* Max # of requests */
        unsigned int            nr_congestion_on;
        unsigned int            nr_congestion_off;
        unsigned int            nr_batching;

        unsigned int            max_sectors;
        unsigned int            max_hw_sectors;
        unsigned short          max_phys_segments;
        unsigned short          max_hw_segments;
        unsigned short          hardsect_size;
        unsigned int            max_segment_size;

        unsigned long           seg_boundary_mask;
        unsigned int            dma_alignment;

        struct blk_queue_tag    *queue_tags;

        unsigned int            nr_sorted;
        unsigned int            in_flight;

        /*
         * sg stuff
         */
        unsigned int            sg_timeout;
        unsigned int            sg_reserved_size;
        int                     node;
#ifdef CONFIG_BLK_DEV_IO_TRACE
        struct blk_trace        *blk_trace;
#endif
        /*
```

```
462          * reserved for flush operations
463          */
464         unsigned int            ordered, next_ordered, ordseq;
465         int                     orderr, ordcolor;
466         struct request          pre_flush_rq, bar_rq, post_flush_rq;
467         struct request          *orig_bar_rq;
468         unsigned int            bi_size;
469
470         struct mutex            sysfs_lock;
471 };
```

这里我们看到了 rq 其实是 struct request_list 结构体变量.这个结构体定义于同一个文件.

```
131 struct request_list {
132         int count[2];
133         int starved[2];
134         int elvpriv;
135         mempool_t *rq_pool;
136         wait_queue_head_t wait[2];
137 };
```

不过这些我们现在都不想看,我们想看的只有其中的几个函数,第一个是 2125 行 blk_alloc_request().来自 ll_rw_blk.c:

```
1970 static struct request *
1971 blk_alloc_request(request_queue_t *q, int rw, int priv, gfp_t gfp_mask)
1972 {
1973         struct request *rq = mempool_alloc(q->rq.rq_pool, gfp_mask);
1974
1975         if (!rq)
1976                 return NULL;
1977
1978         /*
1979          * first three bits are identical in rq->cmd_flags and bio->bi_rw,
1980          * see bio.h and blkdev.h
1981          */
1982         rq->cmd_flags = rw | REQ_ALLOCED;
1983
1984         if (priv) {
1985                 if (unlikely(elv_set_request(q, rq, gfp_mask))) {
1986                         mempool_free(rq, q->rq.rq_pool);
1987                         return NULL;
1988                 }
1989                 rq->cmd_flags |= REQ_ELVPRIV;
1990         }
1991
1992         return rq;
1993 }
```

其它我们不懂没有关系,至少我们从 1972 行可以看出这里申请了一个 struct request 的结构体指针,换句话说,此前,我们已经有了请求队列,但是没有实质性的元素,从这一刻起,我们有了一个真正的 request.虽然现在还没有进入到队伍中去,但这只是早晚的事儿了.

下一个 rq_init().

```
static void rq_init(request_queue_t *q, struct request *rq)
{
        INIT_LIST_HEAD(&rq->queuelist);
        INIT_LIST_HEAD(&rq->donelist);

        rq->errors = 0;
        rq->bio = rq->biotail = NULL;
        INIT_HLIST_NODE(&rq->hash);
        RB_CLEAR_NODE(&rq->rb_node);
        rq->ioprio = 0;
        rq->buffer = NULL;
        rq->ref_count = 1;
        rq->q = q;
        rq->special = NULL;
        rq->data_len = 0;
        rq->data = NULL;
        rq->nr_phys_segments = 0;
        rq->sense = NULL;
        rq->end_io = NULL;
        rq->end_io_data = NULL;
        rq->completion_data = NULL;
}
```

这个函数在干什么不用我说,浦东金杨新村卖麻辣烫的大妈都知道,对刚申请的 rq 进行初始化.

然后,get_request()就开开心心的返回了,正常情况下,get_request_wait()也会跟着返回,再接着,blk_get_request()也就返回了.我们也带着申请好初始化好的 req 回到 scsi_execute()中去,而接下来一段代码就是我们最关心的,对 req 的真正的赋值.比如 req->cmd_len,req->cmd 等等,就是这样被赋上的.换言之,我们的 scsi 命令就是这样被 request 拖下水的,从此它们之间不再是以前那种"水留不住落花的漂泊,落花走不进水的世界"的关系,而是沦落到了一荣俱荣一损俱损狼狈为奸的关系.

至此,完成了第一次变身,从 scsi 命令到 request 的变身.

scsi 命令的前世今生(二)

一旦这种狼狈为奸的关系建立好了,就可以开始执行请求了.来看 blk_execute_rq(),来自 block/ll_rw_blk.c:

```
/**
 * blk_execute_rq - insert a request into queue for execution
```

```
 * @q:          queue to insert the request in
 * @bd_disk:    matching gendisk
 * @rq:         request to insert
 * @at_head:    insert request at head or tail of queue
 *
 * Description:
 *    Insert a fully prepared request at the back of the io scheduler queue
 *    for execution and wait for completion.
 */
int blk_execute_rq(request_queue_t *q, struct gendisk *bd_disk,
                   struct request *rq, int at_head)
{
        DECLARE_COMPLETION_ONSTACK(wait);
        char sense[SCSI_SENSE_BUFFERSIZE];
        int err = 0;

        /*
         * we need an extra reference to the request, so we can look at
         * it after io completion
         */
        rq->ref_count++;

        if (!rq->sense) {
                memset(sense, 0, sizeof(sense));
                rq->sense = sense;
                rq->sense_len = 0;
        }

        rq->end_io_data = &wait;
        blk_execute_rq_nowait(q, bd_disk, rq, at_head, blk_end_sync_rq);
        wait_for_completion(&wait);

        if (rq->errors)
                err = -EIO;

        return err;
}
```

抛去那些用于错误处理的代码,这个函数真正有意义的代码就是两行，blk_execute_rq_nowait 和 wait_for_completion.先看前者,来自 block/ll_rw_blk.c:

```
/**
 * blk_execute_rq_nowait - insert a request into queue for execution
 * @q:          queue to insert the request in
 * @bd_disk:    matching gendisk
 * @rq:         request to insert
```

```
 * @at_head:        insert request at head or tail of queue
 * @done:           I/O completion handler
 *
 * Description:
 *      Insert a fully prepared request at the back of the io scheduler queue
 *      for execution.   Don't wait for completion.
 */
void blk_execute_rq_nowait(request_queue_t *q, struct gendisk *bd_disk,
                           struct request *rq, int at_head,
                           rq_end_io_fn *done)
{
        int where = at_head ? ELEVATOR_INSERT_FRONT : ELEVATOR_INSERT_BACK;

        rq->rq_disk = bd_disk;
        rq->cmd_flags |= REQ_NOMERGE;
        rq->end_io = done;
        WARN_ON(irqs_disabled());
        spin_lock_irq(q->queue_lock);
        __elv_add_request(q, rq, where, 1);
        __generic_unplug_device(q);
        spin_unlock_irq(q->queue_lock);
}
```

首先 at_head 是表示往哪插.(汗…,该不会还有一个参数表示用什么姿势插吧.)

而 where 用来记录 at_head 的值.在我们这个上下文中,at_head 是从 scsi_execute()中调用 blk_execute_rq 的时候传递下来的,当时我们设置的是 1.于是 where 被设置为 ELEVATOR_INSERT_FRONT.这几个宏来自 include/linux/elevator.h:

```
/*
 * Insertion selection
 */
#define ELEVATOR_INSERT_FRONT     1
#define ELEVATOR_INSERT_BACK      2
#define ELEVATOR_INSERT_SORT      3
#define ELEVATOR_INSERT_REQUEUE 4
```

很明显,这是告诉我们从前面插,还算不是太变态.那么带着这个 where 我们进入下一个函数,即__elv_add_request.来自 block/elevator.c:

```
void __elv_add_request(request_queue_t *q, struct request *rq, int where,
                       int plug)
{
        if (q->ordcolor)
                rq->cmd_flags |= REQ_ORDERED_COLOR;

        if (rq->cmd_flags & (REQ_SOFTBARRIER | REQ_HARDBARRIER)) {
                /*
```

```
			 * toggle ordered color
			 */
			if (blk_barrier_rq(rq))
				q->ordcolor ^= 1;

			/*
			 * barriers implicitly indicate back insertion
			 */
			if (where == ELEVATOR_INSERT_SORT)
				where = ELEVATOR_INSERT_BACK;

			/*
			 * this request is scheduling boundary, update
			 * end_sector
			 */
			if (blk_fs_request(rq)) {
				q->end_sector = rq_end_sector(rq);
				q->boundary_rq = rq;
			}
		} else if (!(rq->cmd_flags & REQ_ELVPRIV) && where == ELEVATOR_INSERT_SORT)
			where = ELEVATOR_INSERT_BACK;

	if (plug)
		blk_plug_device(q);

	elv_insert(q, rq, where);
}
```

传入的参数 plug 等于 1,所以 blk_plug_device()会被执行.暂且先不管这个函数.

很明显,前面都和我们无关,直接跳到最后一行这个 elv_insert().

```
void elv_insert(request_queue_t *q, struct request *rq, int where)
{
	struct list_head *pos;
	unsigned ordseq;
	int unplug_it = 1;

	blk_add_trace_rq(q, rq, BLK_TA_INSERT);

	rq->q = q;

	switch (where) {
	case ELEVATOR_INSERT_FRONT:
		rq->cmd_flags |= REQ_SOFTBARRIER;

```

```
                list_add(&rq->queuelist, &q->queue_head);
                break;

        case ELEVATOR_INSERT_BACK:
                rq->cmd_flags |= REQ_SOFTBARRIER;
                elv_drain_elevator(q);
                list_add_tail(&rq->queuelist, &q->queue_head);
                /*
                 * We kick the queue here for the following reasons.
                 * - The elevator might have returned NULL previously
                 *   to delay requests and returned them now.  As the
                 *   queue wasn't empty before this request, ll_rw_blk
                 *   won't run the queue on return, resulting in hang.
                 * - Usually, back inserted requests won't be merged
                 *   with anything.  There's no point in delaying queue
                 *   processing.
                 */
                blk_remove_plug(q);
                q->request_fn(q);
                break;

        case ELEVATOR_INSERT_SORT:
                BUG_ON(!blk_fs_request(rq));
                rq->cmd_flags |= REQ_SORTED;
                q->nr_sorted++;
                if (rq_mergeable(rq)) {
                        elv_rqhash_add(q, rq);
                        if (!q->last_merge)
                                q->last_merge = rq;
                }

                /*
                 * Some ioscheds (cfq) run q->request_fn directly, so
                 * rq cannot be accessed after calling
                 * elevator_add_req_fn.
                 */
                q->elevator->ops->elevator_add_req_fn(q, rq);
                break;

        case ELEVATOR_INSERT_REQUEUE:
                /*
                 * If ordered flush isn't in progress, we do front
                 * insertion; otherwise, requests should be requeued
                 * in ordseq order.
```

```
606                  */
607                 rq->cmd_flags |= REQ_SOFTBARRIER;
608
609                 /*
610                  * Most requeues happen because of a busy condition,
611                  * don't force unplug of the queue for that case.
612                  */
613                 unplug_it = 0;
614
615                 if (q->ordseq == 0) {
616                         list_add(&rq->queuelist, &q->queue_head);
617                         break;
618                 }
619
620                 ordseq = blk_ordered_req_seq(rq);
621
622                 list_for_each(pos, &q->queue_head) {
623                         struct request *pos_rq = list_entry_rq(pos);
624                         if (ordseq <= blk_ordered_req_seq(pos_rq))
625                                 break;
626                 }
627
628                 list_add_tail(&rq->queuelist, pos);
629                 break;
630
631         default:
632                 printk(KERN_ERR "%s: bad insertion point %d\n",
633                        __FUNCTION__, where);
634                 BUG();
635         }
636
637         if (unplug_it && blk_queue_plugged(q)) {
638                 int nrq = q->rq.count[READ] + q->rq.count[WRITE]
639                         - q->in_flight;
640
641                 if (nrq >= q->unplug_thresh)
642                         __generic_unplug_device(q);
643         }
644 }
```

由于我们是从前面插,所以我们执行 562 行这个 list_add,struct request 有一个成员 struct list_head queuelist,而 struct request_queue 有一个成员 struct list_head queue_head,所以我们就把前者插入到后者所代表的这个队伍中来.然后咱们就返回了.

回到 blk_execute_rq_nowait()中,下一个被调用的函数是__generic_unplug_device,依然是来自 block/ll_rw_blk.c:

```
1586 /*
1587  * remove the plug and let it rip..
1588  */
1589 void __generic_unplug_device(request_queue_t *q)
1590 {
1591         if (unlikely(blk_queue_stopped(q)))
1592                 return;
1593
1594         if (!blk_remove_plug(q))
1595                 return;
1596
1597         q->request_fn(q);
1598 }
```

其实最有看点的就是 1597 行调用这个 request_fn,struct request_queue 中的一个成员 request_fn_proc *request_fn,而至于 request_fn_proc,其实又是 typedef 的小伎俩,来自 include/linux/blkdev.h:

```
typedef void (request_fn_proc) (request_queue_t *q);
```

那么这个 request_fn 是多少呢?还记得当初那个 scsi 子系统中申请队列的函数了么?没错,就是 __scsi_alloc_queue(),调用它的是 scsi_alloc_queue(),而在调用的时候就传递了这个参数:

```
1590 struct request_queue *scsi_alloc_queue(struct scsi_device *sdev)
1591 {
1592         struct request_queue *q;
1593
1594         q = __scsi_alloc_queue(sdev->host, scsi_request_fn);
1595         if (!q)
1596                 return NULL;
1597
1598         blk_queue_prep_rq(q, scsi_prep_fn);
1599         blk_queue_issue_flush_fn(q, scsi_issue_flush_fn);
1600         blk_queue_softirq_done(q, scsi_softirq_done);
1601         return q;
1602 }
```

对,就是这个 scsi_request_fn(),这么一个函数指针通过几次传递并最终在 blk_init_queue_node()中被赋予了 q->request_fn.所以我们真正需要关心的是 scsi_request_fn.

在看 scsi_request_fn 之前,注意这里 1598 行至 1560 行也是赋了三个函数指针,

```
/**
 * blk_queue_prep_rq - set a prepare_request function for queue
 * @q:          queue
 * @pfn:        prepare_request function
 *
 * It's possible for a queue to register a prepare_request callback which
 * is invoked before the request is handed to the request_fn. The goal of
 * the function is to prepare a request for I/O, it can be used to build a
 * cdb from the request data for instance.
```

```
 *
 */
void blk_queue_prep_rq(request_queue_t *q, prep_rq_fn *pfn)
{
        q->prep_rq_fn = pfn;
}
/**
 * blk_queue_issue_flush_fn - set function for issuing a flush
 * @q:     the request queue
 * @iff:   the function to be called issuing the flush
 *
 * Description:
 *   If a driver supports issuing a flush command, the support is notified
 *   to the block layer by defining it through this call.
 *
 **/
void blk_queue_issue_flush_fn(request_queue_t *q, issue_flush_fn *iff)
{
        q->issue_flush_fn = iff;
}
void blk_queue_softirq_done(request_queue_t *q, softirq_done_fn *fn)
{
        q->softirq_done_fn = fn;
}
```

分别是把 scsi_prep_fn 赋给了 q->prep_rq_fn,把 scsi_issue_flush_fn 赋给了 q->issue_flush_fn,把 scsi_softirq_done 赋给了 q->softirq_done_fn.尤其是 scsi_prep_fn 我们马上就会用到.

好,让我们继续前面的话题,来看 scsi_request_fn().

```
/*
 * Function:    scsi_request_fn()
 *
 * Purpose:     Main strategy routine for SCSI.
 *
 * Arguments:   q       - Pointer to actual queue.
 *
 * Returns:     Nothing
 *
 * Lock status: IO request lock assumed to be held when called.
 */
static void scsi_request_fn(struct request_queue *q)
{
        struct scsi_device *sdev = q->queuedata;
        struct Scsi_Host *shost;
        struct scsi_cmnd *cmd;
        struct request *req;
```

```

	if (!sdev) {
		printk("scsi: killing requests for dead queue\n");
		while ((req = elv_next_request(q)) != NULL)
			scsi_kill_request(req, q);
		return;
	}

	if(!get_device(&sdev->sdev_gendev))
		/* We must be tearing the block queue down already */
		return;

	/*
	 * To start with, we keep looping until the queue is empty, or until
	 * the host is no longer able to accept any more requests.
	 */
	shost = sdev->host;
	while (!blk_queue_plugged(q)) {
		int rtn;
		/*
		 * get next queueable request.  We do this early to make sure
		 * that the request is fully prepared even if we cannot
		 * accept it.
		 */
		req = elv_next_request(q);
		if (!req || !scsi_dev_queue_ready(q, sdev))
			break;

		if (unlikely(!scsi_device_online(sdev))) {
			sdev_printk(KERN_ERR, sdev,
				    "rejecting I/O to offline device\n");
			scsi_kill_request(req, q);
			continue;
		}


		/*
		 * Remove the request from the request list.
		 */
		if (!(blk_queue_tagged(q) && !blk_queue_start_tag(q, req)))
			blkdev_dequeue_request(req);
		sdev->device_busy++;

		spin_unlock(q->queue_lock);
```

```
		cmd = req->special;
		if (unlikely(cmd == NULL)) {
			printk(KERN_CRIT "impossible request in %s.\n"
					 "please mail a stack trace to "
					 "linux-scsi@vger.kernel.org\n",
					 __FUNCTION__);
			blk_dump_rq_flags(req, "foo");
			BUG();
		}
		spin_lock(shost->host_lock);

		if (!scsi_host_queue_ready(q, shost, sdev))
			goto not_ready;
		if (sdev->single_lun) {
			if (scsi_target(sdev)->starget_sdev_user &&
			    scsi_target(sdev)->starget_sdev_user != sdev)
				goto not_ready;
			scsi_target(sdev)->starget_sdev_user = sdev;
		}
		shost->host_busy++;

		/*
		 * XXX(hch): This is rather suboptimal, scsi_dispatch_cmd will
		 *		take the lock again.
		 */
		spin_unlock_irq(shost->host_lock);

		/*
		 * Finally, initialize any error handling parameters, and set up
		 * the timers for timeouts.
		 */
		scsi_init_cmd_errh(cmd);

		/*
		 * Dispatch the command to the low-level driver.
		 */
		rtn = scsi_dispatch_cmd(cmd);
		spin_lock_irq(q->queue_lock);
		if(rtn) {
			/* we're refusing the command; because of
			 * the way locks get dropped, we need to
			 * check here if plugging is required */
			if(sdev->device_busy == 0)
				blk_plug_device(q);
```

```

                        break;
                }
        }

        goto out;

 not_ready:
        spin_unlock_irq(shost->host_lock);

        /*
         * lock q, handle tag, requeue req, and decrement device_busy. We
         * must return with queue_lock held.
         *
         * Decrementing device_busy without checking it is OK, as all such
         * cases (host limits or settings) should run the queue at some
         * later time.
         */
        spin_lock_irq(q->queue_lock);
        blk_requeue_request(q, req);
        sdev->device_busy--;
        if(sdev->device_busy == 0)
                blk_plug_device(q);
 out:
        /* must be careful here...if we trigger the ->remove() function
         * we cannot be holding the q lock */
        spin_unlock_irq(q->queue_lock);
        put_device(&sdev->sdev_gendev);
        spin_lock_irq(q->queue_lock);
}
```

首先关注 elv_next_request().来自 block/elevator.c:

```
struct request *elv_next_request(request_queue_t *q)
{
        struct request *rq;
        int ret;

        while ((rq = __elv_next_request(q)) != NULL) {
                if (!(rq->cmd_flags & REQ_STARTED)) {
                        /*
                         * This is the first time the device driver
                         * sees this request (possibly after
                         * requeueing).   Notify IO scheduler.
                         */
                        if (blk_sorted_rq(rq))
```

```
			elv_activate_rq(q, rq);

		/*
		 * just mark as started even if we don't start
		 * it, a request that has been delayed should
		 * not be passed by new incoming requests
		 */
		rq->cmd_flags |= REQ_STARTED;
		blk_add_trace_rq(q, rq, BLK_TA_ISSUE);
	}

	if (!q->boundary_rq || q->boundary_rq == rq) {
		q->end_sector = rq_end_sector(rq);
		q->boundary_rq = NULL;
	}

	if ((rq->cmd_flags & REQ_DONTPREP) || !q->prep_rq_fn)
		break;

	ret = q->prep_rq_fn(q, rq);
	if (ret == BLKPREP_OK) {
		break;
	} else if (ret == BLKPREP_DEFER) {
		/*
		 * the request may have been (partially) prepped.
		 * we need to keep this request in the front to
		 * avoid resource deadlock.   REQ_STARTED will
		 * prevent other fs requests from passing this one.
		 */
		rq = NULL;
		break;
	} else if (ret == BLKPREP_KILL) {
		int nr_bytes = rq->hard_nr_sectors << 9;

		if (!nr_bytes)
			nr_bytes = rq->data_len;

		blkdev_dequeue_request(rq);
		rq->cmd_flags |= REQ_QUIET;
		end_that_request_chunk(rq, 0, nr_bytes);
		end_that_request_last(rq, 0);
	} else {
			printk(KERN_ERR "%s: bad return=%d\n",
__FUNCTION__,
```

```
                                                                   ret);
                                break;
                        }
                }

                return rq;
}
```

它调用的__elv_next_request()仍然来自 block/elevator.c:

```
static inline struct request *__elv_next_request(request_queue_t *q)
{
        struct request *rq;

        while (1) {
                while (!list_empty(&q->queue_head)) {
                        rq = list_entry_rq(q->queue_head.next);
                        if (blk_do_ordered(q, &rq))
                                return rq;
                }

                if (!q->elevator->ops->elevator_dispatch_fn(q, 0))
                        return NULL;
        }
}
```

由于我们刚才那个精彩的插入动作,这里 q->queue_head 不可能为空.所以从中取出一个 request 来.

首先是 blk_do_ordered(),来自 block/ll_rw_blk.c:

```
int blk_do_ordered(request_queue_t *q, struct request **rqp)
{
        struct request *rq = *rqp;
        int is_barrier = blk_fs_request(rq) && blk_barrier_rq(rq);

        if (!q->ordseq) {
                if (!is_barrier)
                        return 1;

                if (q->next_ordered != QUEUE_ORDERED_NONE) {
                        *rqp = start_ordered(q, rq);
                        return 1;
                } else {
                        /*
                         * This can happen when the queue switches to
                         * ORDERED_NONE while this request is on it.
                         */
                        blkdev_dequeue_request(rq);
```

```
                        end_that_request_first(rq, -EOPNOTSUPP,
                                               rq->hard_nr_sectors);
                        end_that_request_last(rq, -EOPNOTSUPP);
                        *rqp = NULL;
                        return 0;
                }
        }

        /*
         * Ordered sequence in progress
         */

        /* Special requests are not subject to ordering rules. */
        if (!blk_fs_request(rq) &&
            rq != &q->pre_flush_rq && rq != &q->post_flush_rq)
                return 1;

        if (q->ordered & QUEUE_ORDERED_TAG) {
                /* Ordered by tag.  Blocking the next barrier is enough. */
                if (is_barrier && rq != &q->bar_rq)
                        *rqp = NULL;
        } else {
                /* Ordered by draining.  Wait for turn. */
                WARN_ON(blk_ordered_req_seq(rq) < blk_ordered_cur_seq(q));
                if (blk_ordered_req_seq(rq) > blk_ordered_cur_seq(q))
                        *rqp = NULL;
        }

        return 1;
}
```

首先看一下 blk_fs_request,

```
#define blk_fs_request(rq)      ((rq)->cmd_type == REQ_TYPE_FS)
```

很显然,咱们的情况和这个不一样.

所以在咱们这个上下文里,is_barrier 一定是 0.所以,blk_do_ordered 二话不说,直接返回 1.那么回到__elv_next_request 以后,703 行这个 if 条件是满足的,所以也就是返回 rq.而下面那个 elevator_dispatch_fn 实际上在我们这个上下文中是不会执行的.另一方面,我们从__elv_next_request 返回,回到 elv_next_request()的时候,只要 request queue 不是空的,那么返回值就是队列头的那个 request.

继续往下走,cmd_flags 其实整个故事中设置 REQ_STARTED 的也就是这里,732 行,所以在我们执行 732 行之前,这个 flag 是没有设置的.因此,if 条件是满足的.

而 blk_sorted_rq 又是一个宏,来自 include/linux/blkdev.h:

```
#define blk_sorted_rq(rq)       ((rq)->cmd_flags & REQ_SORTED)
```

很显然,咱们也从来没有设置过这个 flag,所以这里不关我们的事.

当然了,对于 noop,即便执行下一个函数也没有意义,因为这个 elv_activate_rq()来自

block/elevator.c:

```
static void elv_activate_rq(request_queue_t *q, struct request *rq)
{
        elevator_t *e = q->elevator;

        if (e->ops->elevator_activate_req_fn)
                e->ops->elevator_activate_req_fn(q, rq);
}
```

而我们知道,对于 noop 来说,根本就没有这个指针,所以我们不准不开心.

这时候,我们设置 REQ_STARTED 这个 flag.

最开始我们在 elevator_init()中,有这么一句:

```
        q->boundary_rq = NULL;
```

于是 rq_end_sector 会被执行,这其实也只是一个很简单的宏.

```
#define rq_end_sector(rq)       ((rq)->sector + (rq)->nr_sectors)
```

同时,boundary_rq 还是被置为 NULL.

接下来,由于我们把 prep_rq_fn 赋上了 scsi_prep_fn,所以我们要看一下这个 scsi_prep_fn(),这个来自 drivers/scsi/scsi_lib.c 的函数.

```
static int scsi_prep_fn(struct request_queue *q, struct request *req)
{
        struct scsi_device *sdev = q->queuedata;
        int ret = BLKPREP_OK;

        /*
         * If the device is not in running state we will reject some
         * or all commands.
         */
        if (unlikely(sdev->sdev_state != SDEV_RUNNING)) {
                switch (sdev->sdev_state) {
                case SDEV_OFFLINE:
                        /*
                         * If the device is offline we refuse to process any
                         * commands.   The device must be brought online
                         * before trying any recovery commands.
                         */
                        sdev_printk(KERN_ERR, sdev,
                                    "rejecting I/O to offline device\n");
                        ret = BLKPREP_KILL;
                        break;
                case SDEV_DEL:
                        /*
                         * If the device is fully deleted, we refuse to
                         * process any commands as well.
                         */
                        sdev_printk(KERN_ERR, sdev,
```

```
                                        "rejecting I/O to dead device\n");
                        ret = BLKPREP_KILL;
                        break;
                case SDEV_QUIESCE:
                case SDEV_BLOCK:
                        /*
                         * If the devices is blocked we defer normal commands.
                         */
                        if (!(req->cmd_flags & REQ_PREEMPT))
                                ret = BLKPREP_DEFER;
                        break;
                default:
                        /*
                         * For any other not fully online state we only allow
                         * special commands.  In particular any user initiated
                         * command is not allowed.
                         */
                        if (!(req->cmd_flags & REQ_PREEMPT))
                                ret = BLKPREP_KILL;
                        break;
                }

                if (ret != BLKPREP_OK)
                        goto out;
        }

        switch (req->cmd_type) {
        case REQ_TYPE_BLOCK_PC:
                ret = scsi_setup_blk_pc_cmnd(sdev, req);
                break;
        case REQ_TYPE_FS:
                ret = scsi_setup_fs_cmnd(sdev, req);
                break;
        default:
                /*
                 * All other command types are not supported.
                 *
                 * Note that these days the SCSI subsystem does not use
                 * REQ_TYPE_SPECIAL requests anymore.  These are only used
                 * (directly or via blk_insert_request) by non-SCSI drivers.
                 */
                blk_dump_rq_flags(req, "SCSI bad req");
                ret = BLKPREP_KILL;
                break;
```

```
1247             }
1248
1249    out:
1250             switch (ret) {
1251             case BLKPREP_KILL:
1252                     req->errors = DID_NO_CONNECT << 16;
1253                     break;
1254             case BLKPREP_DEFER:
1255                     /*
1256                      * If we defer, the elv_next_request() returns NULL, but the
1257                      * queue must be restarted, so we plug here if no returning
1258                      * command will automatically do that.
1259                      */
1260                     if (sdev->device_busy == 0)
1261                             blk_plug_device(q);
1262                     break;
1263             default:
1264                     req->cmd_flags |= REQ_DONTPREP;
1265             }
1266
1267             return ret;
1268 }
```

按正路,我们会走到 1229 行这个 switch 语句,并且会根据 scsi 命令的类型而执行不同的函数, scsi_setup_blk_pc_cmnd 或者 scsi_setup_fs_cmnd.那么我们 cmd_type 究竟是什么呢?回首那如烟的往事,犹记当初在 scsi_execute()中有这么一行,

```
 199             req->cmd_type = REQ_TYPE_BLOCK_PC;
```

所以,没什么好说的.我们会执行 scsi_setup_blk_pc_cmnd,来自 drivers/scsi/scsi_lib.c:

```
1090 static int scsi_setup_blk_pc_cmnd(struct scsi_device *sdev, struct request *req)
1091 {
1092             struct scsi_cmnd *cmd;
1093
1094             cmd = scsi_get_cmd_from_req(sdev, req);
1095             if (unlikely(!cmd))
1096                     return BLKPREP_DEFER;
1097
1098             /*
1099              * BLOCK_PC requests may transfer data, in which case they must
1100              * a bio attached to them.    Or they might contain a SCSI command
1101              * that does not transfer data, in which case they may optionally
1102              * submit a request without an attached bio.
1103              */
1104             if (req->bio) {
1105                     int ret;
1106
```

```
			BUG_ON(!req->nr_phys_segments);

			ret = scsi_init_io(cmd);
			if (unlikely(ret))
				return ret;
		} else {
			BUG_ON(req->data_len);
			BUG_ON(req->data);

			cmd->request_bufflen = 0;
			cmd->request_buffer = NULL;
			cmd->use_sg = 0;
			req->buffer = NULL;
		}

		BUILD_BUG_ON(sizeof(req->cmd) > sizeof(cmd->cmnd));
		memcpy(cmd->cmnd, req->cmd, sizeof(cmd->cmnd));
		cmd->cmd_len = req->cmd_len;
		if (!req->data_len)
			cmd->sc_data_direction = DMA_NONE;
		else if (rq_data_dir(req) == WRITE)
			cmd->sc_data_direction = DMA_TO_DEVICE;
		else
			cmd->sc_data_direction = DMA_FROM_DEVICE;

		cmd->transfersize = req->data_len;
		cmd->allowed = req->retries;
		cmd->timeout_per_command = req->timeout;
		cmd->done = scsi_blk_pc_done;
		return BLKPREP_OK;
}
```

如果曾经的你还对 scsi cmd 是如何形成的颇有疑义的话,那么相信此刻,你应该会明白了吧,尤其是当你在usb-storage那个故事中看到对它sc_data_direction的判断的时候,你不理解这个值是如何设定的,那么此刻,这代码活生生的展现在你面前,想必已经揭开了你心中那谜团吧.

最终,正常的话,函数返回 BLKPREP_OK.prep 表示 prepare 的意思,用我们的母语说就是准备的意思,最后 BLKPREP_OK 就说明准备好了,或者说准备就绪.而 scsi_prep_fn()也将返回这个值,返回之前还设置了 cmd_flags 中的 REQ_DONTPREP.(注意 elv_next_request()函数 741 行判断的就是设没设这个 flag.)

回到 elv_next_request()中,由于返回值是 BLKPREP_OK,所以 746 行我们就 break 了.换言之,我们取到了一个request,我们为之准备好了scsi命令,我们下一步就该是执行这个命令了.所以我们不需要再在 elv_next_request()中滞留.我们终于回到了 scsi_request_fn(),汤唯姐姐曾坦言拍床戏的经验让她恍如在地狱走了一趟,而看代码的我又何尝不是如此呢?而且汤唯姐姐说虽然过程好似地狱,但过后就是天堂.而我们则永远陷在这代码中,不知何时才是个头,这不,结束了 elv_next_request,又要看下一个,不只是一个,而是两个,1467 行,一个宏加一个函数,宏是

blk_queue_tagged,来自 include/linux/blkdev.h:

```
#define    blk_queue_tagged(q)                test_bit(QUEUE_FLAG_QUEUED,
&(q)->queue_flags)
```

而函数是 blk_queue_start_tag,来自 block/ll_rw_blk.c:

```
/**
 * blk_queue_start_tag - find a free tag and assign it
 * @q:  the request queue for the device
 * @rq:  the block request that needs tagging
 *
 *  Description:
 *    This can either be used as a stand-alone helper, or possibly be
 *    assigned as the queue &prep_rq_fn (in which case &struct request
 *    automagically gets a tag assigned). Note that this function
 *    assumes that any type of request can be queued! if this is not
 *    true for your device, you must check the request type before
 *    calling this function.  The request will also be removed from
 *    the request queue, so it's the drivers responsibility to readd
 *    it if it should need to be restarted for some reason.
 *
 *  Notes:
 *   queue lock must be held.
 **/
int blk_queue_start_tag(request_queue_t *q, struct request *rq)
{
        struct blk_queue_tag *bqt = q->queue_tags;
        int tag;

        if (unlikely((rq->cmd_flags & REQ_QUEUED))) {
                printk(KERN_ERR
                       "%s: request %p for device [%s] already tagged %d",
                       __FUNCTION__, rq,
                       rq->rq_disk ? rq->rq_disk->disk_name : "?", rq->tag);
                BUG();
        }

        /*
         * Protect against shared tag maps, as we may not have exclusive
         * access to the tag map.
         */
        do {
                tag = find_first_zero_bit(bqt->tag_map, bqt->max_depth);
                if (tag >= bqt->max_depth)
                        return 1;

```

```
        } while (test_and_set_bit(tag, bqt->tag_map));

        rq->cmd_flags |= REQ_QUEUED;
        rq->tag = tag;
        bqt->tag_index[tag] = rq;
        blkdev_dequeue_request(rq);
        list_add(&rq->queuelist, &bqt->busy_list);
        bqt->busy++;
        return 0;
}
```

对于我们大多数人来说,这两个函数的返回值都是 0.

也因此,下一个函数 blkdev_dequeue_request()就会被执行.来自 include/linux/blkdev.h:

```
static inline void blkdev_dequeue_request(struct request *req)
{
        elv_dequeue_request(req->q, req);
}
```

而 elv_dequeue_request 来自 block/elevator.c:

```
void elv_dequeue_request(request_queue_t *q, struct request *rq)
{
        BUG_ON(list_empty(&rq->queuelist));
        BUG_ON(ELV_ON_HASH(rq));

        list_del_init(&rq->queuelist);

        /*
         * the time frame between a request being removed from the lists
         * and to it is freed is accounted as io that is in progress at
         * the driver side.
         */
        if (blk_account_rq(rq))
                q->in_flight++;
}
```

现在这个社会就是利用与被利用的关系,既然这个 request 已经没有了利用价值,我们已经从它身上得到了我们想要的 scsi 命令,那么我们完全可以过河拆桥卸磨杀驴了.list_del_init 把这个 request 从 request queue 队列里删除掉.

而下面这个 blk_account_rq 也是一个来自 include/linux/blkdev.h 的宏:

```
#define blk_account_rq(rq)      (blk_rq_started(rq) && blk_fs_request(rq))
```

很显然,至少第二个条件我们是不满足的.所以不用多说,结束这个 elv_dequeue_request.

现在是时候去执行 scsi 命令了.所以调用 scsi_dispatch_cmd().

scsi 命令的前世今生(三)

下一个更为重要的函数是 scsi_dispatch_cmd,来自 drivers/scsi/scsi.c:

```
/*
 * Function:    scsi_dispatch_command
 *
 * Purpose:     Dispatch a command to the low-level driver.
 *
 * Arguments:   cmd - command block we are dispatching.
 *
 * Notes:
 */
int scsi_dispatch_cmd(struct scsi_cmnd *cmd)
{
        struct Scsi_Host *host = cmd->device->host;
        unsigned long flags = 0;
        unsigned long timeout;
        int rtn = 0;

        /* check if the device is still usable */
        if (unlikely(cmd->device->sdev_state == SDEV_DEL)) {
                /* in SDEV_DEL we error all commands. DID_NO_CONNECT
                 * returns an immediate error upwards, and signals
                 * that the device is no longer present */
                cmd->result = DID_NO_CONNECT << 16;
                atomic_inc(&cmd->device->iorequest_cnt);
                __scsi_done(cmd);
                /* return 0 (because the command has been processed) */
                goto out;
        }

        /* Check to see if the scsi lld put this device into state SDEV_BLOCK. */
        if (unlikely(cmd->device->sdev_state == SDEV_BLOCK)) {
                /*
                 * in SDEV_BLOCK, the command is just put back on the device
                 * queue.  The suspend state has already blocked the queue so
                 * future requests should not occur until the device
                 * transitions out of the suspend state.
                 */
                scsi_queue_insert(cmd, SCSI_MLQUEUE_DEVICE_BUSY);

                    SCSI_LOG_MLQUEUE(3, printk("queuecommand : device blocked \n"));
```

```

		/*
		 * NOTE: rtn is still zero here because we don't need the
		 * queue to be plugged on return (it's already stopped)
		 */
		goto out;
	}

	/*
	 * If SCSI-2 or lower, store the LUN value in cmnd.
	 */
	if (cmd->device->scsi_level <= SCSI_2 &&
	    cmd->device->scsi_level != SCSI_UNKNOWN) {
		cmd->cmnd[1] = (cmd->cmnd[1] & 0x1f) |
			       (cmd->device->lun << 5 & 0xe0);
	}

	/*
	 * We will wait MIN_RESET_DELAY clock ticks after the last reset so
	 * we can avoid the drive not being ready.
	 */
	timeout = host->last_reset + MIN_RESET_DELAY;

	if (host->resetting && time_before(jiffies, timeout)) {
		int ticks_remaining = timeout - jiffies;
		/*
		 * NOTE: This may be executed from within an interrupt
		 * handler!  This is bad, but for now, it'll do.  The irq
		 * level of the interrupt handler has been masked out by the
		 * platform dependent interrupt handling code already, so the
		 * sti() here will not cause another call to the SCSI host's
		 * interrupt handler (assuming there is one irq-level per
		 * host).
		 */
		while (--ticks_remaining >= 0)
			mdelay(1 + 999 / HZ);
		host->resetting = 0;
	}

	/*
	 * AK: unlikely race here: for some reason the timer could
	 * expire before the serial number is set up below.
	 */
	scsi_add_timer(cmd, cmd->timeout_per_command, scsi_times_out);
```

```

	scsi_log_send(cmd);

	/*
	 * We will use a queued command if possible, otherwise we will
	 * emulate the queuing and calling of completion function ourselves.
	 */
	atomic_inc(&cmd->device->iorequest_cnt);

	/*
	 * Before we queue this command, check if the command
	 * length exceeds what the host adapter can handle.
	 */
	if (CDB_SIZE(cmd) > cmd->device->host->max_cmd_len) {
		SCSI_LOG_MLQUEUE(3,
				printk("queuecommand : command too long.\n"));
		cmd->result = (DID_ABORT << 16);

		scsi_done(cmd);
		goto out;
	}

	spin_lock_irqsave(host->host_lock, flags);
	scsi_cmd_get_serial(host, cmd);

	if (unlikely(host->shost_state == SHOST_DEL)) {
		cmd->result = (DID_NO_CONNECT << 16);
		scsi_done(cmd);
	} else {
		rtn = host->hostt->queuecommand(cmd, scsi_done);
	}
	spin_unlock_irqrestore(host->host_lock, flags);
	if (rtn) {
		if (scsi_delete_timer(cmd)) {
			atomic_inc(&cmd->device->iodone_cnt);
			scsi_queue_insert(cmd,
					  (rtn == SCSI_MLQUEUE_DEVICE_BUSY) ?
					  rtn : SCSI_MLQUEUE_HOST_BUSY);
		}
		SCSI_LOG_MLQUEUE(3,
			printk("queuecommand : request rejected\n"));
```

```
583         }
584
585 out:
586         SCSI_LOG_MLQUEUE(3, printk("leaving scsi_dispatch_cmnd()\n"));
587         return rtn;
588 }
```

一路走来的兄弟一定会一眼就看出这里我们最期待的一行代码就是571那个queuecommand()的调用.因为这之后我们就知道该发生什么了.比如对于 U 盘驱动来说,命令就从这里接过去开始执行.而对于实际的 scsi 控制器,其对应的驱动中的 queuecommand 也会被调用,剩下的事情我们就不用操心了.正常情况下 queuecommand 返回 0.于是紧接着 scsi_dispatch_cmd 也返回 0.这样就算是执行了一条 scsi 命令了.

而 scsi_request_fn() 是否结束还得看 while 循环的条件是否满足,而这就得看 blk_queue_plugged()的脸色了.那么我们从字面上来分析,什么叫 queue plugged?我那盗版金山词霸告诉我 plugged 就是塞紧的意思,你说队列塞紧的是什么意思?比如说,北四环上上下班高峰期,许许多多的车辆排成一队又一队,但是可能半天都前进不了,这就叫塞紧,或者说堵车,也叫塞车.为此咱们使用一个 flag 来标志堵车与否,来自 include/linux/blkdev.h:

```
#define blk_queue_plugged(q)        test_bit(QUEUE_FLAG_PLUGGED, &(q)->queue_flags)
```

改变这个这个 flag 的函数有两个,一个是设置,一个是取消.

负责设置的是 blk_plug_device.

```
/*
 * "plug" the device if there are no outstanding requests: this will
 * force the transfer to start only after we have put all the requests
 * on the list.
 *
 * This is called with interrupts off and no requests on the queue and
 * with the queue lock held.
 */
void blk_plug_device(request_queue_t *q)
{
        WARN_ON(!irqs_disabled());

        /*
     * don't plug a stopped queue, it must be paired with blk_start_queue()
         * which will restart the queueing
         */
        if (blk_queue_stopped(q))
                return;

        if (!test_and_set_bit(QUEUE_FLAG_PLUGGED, &q->queue_flags)) {
                mod_timer(&q->unplug_timer, jiffies + q->unplug_delay);
                blk_add_trace_generic(q, NULL, 0, BLK_TA_PLUG);
        }
}
```

负责取消的是 blk_remove_plug().

```
/*
 * remove the queue from the plugged list, if present. called with
 * queue lock held and interrupts disabled.
 */
int blk_remove_plug(request_queue_t *q)
{
        WARN_ON(!irqs_disabled());

        if (!test_and_clear_bit(QUEUE_FLAG_PLUGGED, &q->queue_flags))
                return 0;

        del_timer(&q->unplug_timer);
        return 1;
}
```

而调用前者的地方不少,比如我们见到的__elv_add_request,其第四个参数 int plug 就可以控制是否调用 blk_plug_device(),而当我们在 blk_execute_rq_nowait()中调用__elv_add_request()的时候传递的 plug 就是 1.

另一方面,调用 blk_remove_plug 的地方也有多处.其中__generic_unplug_device()就是之一.所以在咱们这个上下文里,实际上并没有设置这个 flag,因此 scsi_request_fn()就会被执行.

那么编写这两个函数究竟是为了什么呢?这年头,有人做贼,我可以理解是为了劫富济贫,有人杀人,我可以理解是为了伸张正义,甚至有女人红杏出墙,我还可以理解是为了繁荣经济.然而,很长一段时间我都没办法理解有人编写这两个函数是为了什么?

后来我想,不妨这样理解,假设你经常开车经过长安街,你会发现经常有戒严的现象发生,比如某位领导人要出行,比如某位领导人要来访,而你可以把 blk_plug_device()想象成戒严,把 blk_remove_plug 想象成开放.车流要想行进,前提条件是没有戒严,换言之,没有设卡,而 QUEUE_FLAG_PLUGGED 这个 flag 就相当于”卡”,设了它队列就不能前进了,没有设才有可能前进.之所以需要设卡,是因为确实有这个需求,有时候确实不想让队列前进.

那么这里我们还看到两个函数被调用了,mod_timer 和 del_timer,这是干嘛使的?还记得 kblockd 么?最早咱们创建了那个工作队列 kblockd_workqueue,现在是它该出场的时间了.让我们把镜头拉回到函数 blk_init_queue_node().这个函数我们曾经看过,所以这里只贴出其中跟我们这里密切相关的几行:

```
        q->request_fn           = rfn;
        q->prep_rq_fn           = NULL;
        q->unplug_fn            = generic_unplug_device;
        q->queue_flags          = (1 << QUEUE_FLAG_CLUSTER);
        q->queue_lock           = lock;

        blk_queue_segment_boundary(q, 0xffffffff);

        blk_queue_make_request(q, __make_request);
```

首先 q->unplug_fn 被赋上了 generic_unplug_device.这一点很重要,稍后会用到.

然后来看 blk_queue_make_request().这个函数当时咱们并没有讲过.来自 block/ll_rw_block.c:

```
/**
```

```
 * blk_queue_make_request - define an alternate make_request function for a device
 * @q:    the request queue for the device to be affected
 * @mfn: the alternate make_request function
 *
 * Description:
 *       The normal way for &struct bios to be passed to a device
 *       driver is for them to be collected into requests on a request
 *       queue, and then to allow the device driver to select requests
 *       off that queue when it is ready.    This works well for many block
 *       devices. However some block devices (typically virtual devices
 *       such as md or lvm) do not benefit from the processing on the
 *       request queue, and are served best by having the requests passed
 *       directly to them.    This can be achieved by providing a function
 *       to blk_queue_make_request().
 *
 * Caveat:
 *       The driver that does this *must* be able to deal appropriately
 *       with buffers in "highmemory". This can be accomplished by either calling
 *       __bio_kmap_atomic() to get a temporary kernel mapping, or by calling
 *       blk_queue_bounce() to create a buffer in normal memory.
 **/
void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn)
{
        /*
         * set defaults
         */
        q->nr_requests = BLKDEV_MAX_RQ;
        blk_queue_max_phys_segments(q, MAX_PHYS_SEGMENTS);
        blk_queue_max_hw_segments(q, MAX_HW_SEGMENTS);
        q->make_request_fn = mfn;
        q->backing_dev_info.ra_pages = (VM_MAX_READAHEAD * 1024) / PAGE_CACHE_SIZE;
        q->backing_dev_info.state = 0;
        q->backing_dev_info.capabilities = BDI_CAP_MAP_COPY;
        blk_queue_max_sectors(q, SAFE_MAX_SECTORS);
        blk_queue_hardsect_size(q, 512);
        blk_queue_dma_alignment(q, 511);
        blk_queue_congestion_threshold(q);
        q->nr_batching = BLK_BATCH_REQ;

        q->unplug_thresh = 4;               /* hmm */
        q->unplug_delay = (3 * HZ) / 1000;       /* 3 milliseconds */
        if (q->unplug_delay == 0)
                q->unplug_delay = 1;
```

```

        INIT_WORK(&q->unplug_work, blk_unplug_work);

        q->unplug_timer.function = blk_unplug_timeout;
        q->unplug_timer.data = (unsigned long)q;

        /*
         * by default assume old behaviour and bounce for any highmem page
         */
        blk_queue_bounce_limit(q, BLK_BOUNCE_HIGH);
}
```

这里重点关注几个”unplug”为名字的成员.尤其是INIT_WORK,它使得一旦unplug_work这项工作被执行,blk_unplug_work 这个函数就会被执行.而 unplug_timer 这么一赋值,我们就知道,一旦设了闹钟,一旦闹钟时间到了,blk_unplug_timeout 这个函数就会被执行.并且因为这里设置了 unplug_delay 为 3ms,使得闹钟的 timeout 就是 3ms,一旦激活闹钟,3ms 之后 blk_unplug_timeout 就会被执行.这个函数来自 block/ll_rw_blk.c:

```
static void blk_unplug_timeout(unsigned long data)
{
        request_queue_t *q = (request_queue_t *)data;

        blk_add_trace_pdu_int(q, BLK_TA_UNPLUG_TIMER, NULL,
                                q->rq.count[READ] + q->rq.count[WRITE]);

        kblockd_schedule_work(&q->unplug_work);
}
```

可以看到,其实就是执行 kblockd_schedule_work,换言之,真正被调用的函数就是 blk_unplug_work().

```
static void blk_unplug_work(struct work_struct *work)
{
        request_queue_t *q = container_of(work, request_queue_t, unplug_work);

        blk_add_trace_pdu_int(q, BLK_TA_UNPLUG_IO, NULL,
                                q->rq.count[READ] + q->rq.count[WRITE]);

        q->unplug_fn(q);
}
```

而刚才我们说了,unplug_fn 被赋上了 generic_unplug_device.所以真正要执行的是 generic_unplug_device.而这个函数又长成什么样呢?

```
/**
 * generic_unplug_device - fire a request queue
 * @q:    The &request_queue_t in question
 *
 * Description:
 *   Linux uses plugging to build bigger requests queues before letting
```

```
1607     *      the device have at them. If a queue is plugged, the I/O scheduler
1608     *      is still adding and merging requests on the queue. Once the queue
1609     *      gets unplugged, the request_fn defined for the queue is invoked and
1610     *      transfers started.
1611     **/
1612 void generic_unplug_device(request_queue_t *q)
1613 {
1614             spin_lock_irq(q->queue_lock);
1615             __generic_unplug_device(q);
1616             spin_unlock_irq(q->queue_lock);
1617 }
```

哦,扭扭捏捏大半天,其实就是调用__generic_unplug_device.而回过头去看这个函数,我们知道,它也无非就是调用了两个函数,blk_remove_plug 和 request_fn.这下子我们基本上就明白了.总结一下就是:

1. blk_plug_device()负责戒严.
2. blk_remove_plug()负责解禁.
3. 但是戒严这东西吧,也是有时间限制的,毕竟长安街就算有重大活动也是短时间的,一年中毕竟大多数时间还是得保证道路畅通.所以在戒严的时候,设了一个定时器,unplug_timer,(即 mod_timer),一旦时间到了就自动执行 blk_remove_plug 去解禁.
4. 而在解禁的时候就不要忘记把这个定时器给关掉.(即 del_timer)
5. 解禁之后调用 request_fn()开始处理队列中的下一个请求,或者说车流开始恢复前行.

Ok,这样我们就算是明白这两个戒严与解禁的函数了.最后,题外话,关于 unplug 和 plug,我觉得更贴切的单词是 activate 和 deactivate,或者说激活与冻结,或者简单的说,开与关.

scsi 命令的前世今生(四)

当然,while 循环结束也可能是因为 1453 行的这两个判断.首先 req 如果没有了,另一个得看 scsi_dev_queue_ready()的返回值,如果返回值为 0,那么 break 也会被执行,从而结束循环.

```
1270 /*
1271     * scsi_dev_queue_ready: if we can send requests to sdev, return 1 else
1272     * return 0.
1273     *
1274     * Called with the queue_lock held.
1275     */
1276 static inline int scsi_dev_queue_ready(struct request_queue *q,
1277                                                  struct scsi_device *sdev)
1278 {
1279             if (sdev->device_busy >= sdev->queue_depth)
1280                     return 0;
1281             if (sdev->device_busy == 0 && sdev->device_blocked) {
1282                     /*
```

```
                 * unblock after device_blocked iterates to zero
                 */
                if (--sdev->device_blocked == 0) {
                        SCSI_LOG_MLQUEUE(3,
                                  sdev_printk(KERN_INFO, sdev,
                                  "unblocking device at zero depth\n"));
                } else {
                        blk_plug_device(q);
                        return 0;
                }
        }
        if (sdev->device_blocked)
                return 0;

        return 1;
}
```

这里需要判断的是 device_busy.这个 flag 如果设置了,说明命令正在执行中,或者说命令已经传递到了底层驱动.因此,我们在调用 scsi_dispatch_cmd 之前先增加 device_busy,即 1469 行.另一个 flag 是 device_blocked.这个 flag 是告诉世人这个设备不能再接收新的命令了,因为它十有八九是正在处理命令.正常情况下这个 flag 的值为 0.除非你调用了 scsi_queue_insert()函数.友情提示一下,scsi 设备的这个 flag 是提供了 sysfs 的接口的,因此我们可以通过 sysfs 的接口看一下设备的这个值,下面列举了两个 scsi 设备的这个变量的值,可以看到都是 0,应该说这是它的常态.

```
[root@localhost ~]# ls /sys/bus/scsi/devices/
0:0:8:0/ 0:2:0:0/ 1:0:0:0/ 2:0:0:0/
[root@localhost ~]# ls /sys/bus/scsi/devices/2\:0\:0\:0/
block:sdb/   iocounterbits   modalias   rev   subsystem/
bus/   iodone_cnt   model   scsi_device:2:0:0:0/   timeout   delete
ioerr_cnt   queue_depth   scsi_disk:2:0:0:0/   type   device_blocked
iorequest_cnt   queue_type   scsi_level   uevent   driver/
max_sectors   rescan   state   vendor
[root@localhost ~]# cat /sys/bus/scsi/devices/2\:0\:0\:0/device_blocked
0
[root@localhost ~]# cat /sys/bus/scsi/devices/0\:0\:8\:0/device_blocked
0
```

所以正常情况下,scsi_dev_queue_ready()函数的返回值就是 1,这一点正如其注释里说的那样.但是所谓的常态,指的是单独执行一个命令,如果要执行多个命令,或者说我们提交了多个 request,那么 device_busy 就会一次次的在 1469 行增加,从而使得 device_busy 有可能将超过 queue_depth,这样子 scsi_dev_queue_ready()就会返回 0,从而 scsi_request_fn()就有可能结束,这之后,__generic_unplug_device 也将返回,之后 blk_execute_rq_nowait()返回,回到 blk_execute_rq()中,执行 wait_for_completion(),于是就睡眠了,等待了,按照游戏规则,我们应该能找到一条 complete()语句来唤醒它,那么这条语句在哪里呢?答案是 blk_end_sync_rq.

网友"宁失身不失眠"非常好奇我是怎么知道的.说来话长,还记得我们当时在 usb-storage 中说的那个 scsi_done 么?命令执行完了就会 call scsi_done.而 scsi_done 来自 drivers/scsi/scsi.c,很显

然这个函数是我们的突破口,我们找到了这个函数就好比国民党找到了甫志高,就好比王佳芝找到了易先生:

```
/**
 * scsi_done - Enqueue the finished SCSI command into the done queue.
 * @cmd: The SCSI Command for which a low-level device driver (LLDD) gives
 * ownership back to SCSI Core -- i.e. the LLDD has finished with it.
 *
 * This function is the mid-level's (SCSI Core) interrupt routine, which
 * regains ownership of the SCSI command (de facto) from a LLDD, and enqueues
 * the command to the done queue for further processing.
 *
 * This is the producer of the done queue who enqueues at the tail.
 *
 * This function is interrupt context safe.
 */
static void scsi_done(struct scsi_cmnd *cmd)
{
        /*
         * We don't have to worry about this one timing out any more.
         * If we are unable to remove the timer, then the command
         * has already timed out.  In which case, we have no choice but to
         * let the timeout function run, as we have no idea where in fact
         * that function could really be.  It might be on another processor,
         * etc, etc.
         */
        if (!scsi_delete_timer(cmd))
                return;
        __scsi_done(cmd);
}
```

躲躲闪闪的是来自同一文件的__scsi_done,

```
/* Private entry to scsi_done() to complete a command when the timer
 * isn't running --- used by scsi_times_out */
void __scsi_done(struct scsi_cmnd *cmd)
{
        struct request *rq = cmd->request;

        /*
         * Set the serial numbers back to zero
         */
        cmd->serial_number = 0;

        atomic_inc(&cmd->device->iodone_cnt);
        if (cmd->result)
                atomic_inc(&cmd->device->ioerr_cnt);
```

```

        BUG_ON(!rq);

        /*
         * The uptodate/nbytes values don't matter, as we allow partial
         * completes and thus will check this in the softirq callback
         */
        rq->completion_data = cmd;
        blk_complete_request(rq);
}
```

别的我们都不关心,就关心最后这个 blk_complete_request().

```
/**
 * blk_complete_request - end I/O on a request
 * @req:      the request being processed
 *
 * Description:
 *     Ends all I/O on a request. It does not handle partial completions,
 *     unless the driver actually implements this in its completion callback
 *     through requeueing. Theh actual completion happens out-of-order,
 *     through a softirq handler. The user must have registered a completion
 *     callback through blk_queue_softirq_done().
 **/

void blk_complete_request(struct request *req)
{
        struct list_head *cpu_list;
        unsigned long flags;

        BUG_ON(!req->q->softirq_done_fn);

        local_irq_save(flags);

        cpu_list = &__get_cpu_var(blk_cpu_done);
        list_add_tail(&req->donelist, cpu_list);
        raise_softirq_irqoff(BLOCK_SOFTIRQ);

        local_irq_restore(flags);
}
```

其它的咱们不管,就管一管这个 raise_softirq_irqoff().在很久很久以前,有一个函数,它的名字叫做 blk_dev_init().它是我们这个故事的起源.在这个函数中我们曾经见过这么一行,

```
        open_softirq(BLOCK_SOFTIRQ, blk_done_softirq, NULL);
```

当时咱们就说过,它所做的就是初始化了一个 softirq,即 BLOCK_SOFTIRQ.并且绑定了 softirq 函数 blk_done_softirq,而要触发这个软中断,咱们当时也说了,只要调用 raise_softirq_irqoff()即可.所以现在我们也就这样做了.这也就意味着,blk_done_softirq 会被调用.

```
/*
 * splice the completion data to a local structure and hand off to
 * process_completion_queue() to complete the requests
 */
static void blk_done_softirq(struct softirq_action *h)
{
        struct list_head *cpu_list, local_list;

        local_irq_disable();
        cpu_list = &__get_cpu_var(blk_cpu_done);
        list_replace_init(cpu_list, &local_list);
        local_irq_enable();

        while (!list_empty(&local_list)) {
                struct request *rq = list_entry(local_list.next, struct request, donelist);

                list_del_init(&rq->donelist);
                rq->q->softirq_done_fn(rq);
        }
}
```

而这个 softirq_done_fn 是什么呢?不要说你不知道,其实我们也讲过.不过忘记了也不要紧,人最大的烦恼便是记忆太好,健忘的人容易快乐.在 scsi_alloc_queue 中,我们调用 blk_queue_softirq_done 把 scsi_softirq_done 赋给了 q->softirq_done_fn,所以到了这里,被调用的就是 scsi_softirq_done.

```
static void scsi_softirq_done(struct request *rq)
{
        struct scsi_cmnd *cmd = rq->completion_data;
        unsigned long wait_for = (cmd->allowed + 1) * cmd->timeout_per_command;
        int disposition;

        INIT_LIST_HEAD(&cmd->eh_entry);

        disposition = scsi_decide_disposition(cmd);
        if (disposition != SUCCESS &&
            time_before(cmd->jiffies_at_alloc + wait_for, jiffies)) {
                sdev_printk(KERN_ERR, cmd->device,
                            "timing out command, waited %lus\n",
                            wait_for/HZ);
                disposition = SUCCESS;
        }

        scsi_log_completion(cmd, disposition);

```

```
        switch (disposition) {
                case SUCCESS:
                        scsi_finish_command(cmd);
                        break;
                case NEEDS_RETRY:
                        scsi_queue_insert(cmd, SCSI_MLQUEUE_EH_RETRY);
                        break;
                case ADD_TO_MLQUEUE:
                        scsi_queue_insert(cmd, SCSI_MLQUEUE_DEVICE_BUSY);
                        break;
                default:
                        if (!scsi_eh_scmd_add(cmd, 0))
                                scsi_finish_command(cmd);
        }
}
```

不用我多说,你也知道,scsi_softirq_done 会调用 scsi_finish_command,来自 drivers/scsi/scsi.c:

```
/*
 * Function:    scsi_finish_command
 *
 * Purpose:     Pass command off to upper layer for finishing of I/O
 *              request, waking processes that are waiting on results,
 *              etc.
 */
void scsi_finish_command(struct scsi_cmnd *cmd)
{
        struct scsi_device *sdev = cmd->device;
        struct Scsi_Host *shost = sdev->host;

        scsi_device_unbusy(sdev);

        /*
         * Clear the flags which say that the device/host is no longer
         * capable of accepting new commands.   These are set in scsi_queue.c
         * for both the queue full condition on a device, and for a
         * host full condition on the host.
         *
         * XXX(hch): What about locking?
         */
        shost->host_blocked = 0;
        sdev->device_blocked = 0;

        /*
         * If we have valid sense information, then some kind of recovery
```

```
         * must have taken place.    Make a note of this.
         */
        if (SCSI_SENSE_VALID(cmd))
                cmd->result |= (DRIVER_SENSE << 24);

        SCSI_LOG_MLCOMPLETE(4, sdev_printk(KERN_INFO, sdev,
                                "Notifying upper driver of completion "
                                "(result %x)\n", cmd->result));

        cmd->done(cmd);
}
```

也就是说,cmd->done 会被调用,从而真正的幕后工作者 scsi_blk_pc_done 会被调用.因为,当初在 scsi_setup_blk_pc_cmnd()中有这么一行,

```
        cmd->done = scsi_blk_pc_done;
```

而 scsi_blk_pc_done 来自 drivers/scsi/scsi_lib.c:

```
static void scsi_blk_pc_done(struct scsi_cmnd *cmd)
{
        BUG_ON(!blk_pc_request(cmd->request));
        /*
         * This will complete the whole command with uptodate=1 so
         * as far as the block layer is concerned the command completed
         * successfully. Since this is a REQ_BLOCK_PC command the
         * caller should check the request's errors value
         */
        scsi_io_completion(cmd, cmd->request_bufflen);
}
```

来自 drivers/scsi/scsi_lib.c:

```
/*
 * Function:    scsi_io_completion()
 *
 * Purpose:     Completion processing for block device I/O requests.
 *
 * Arguments:   cmd   - command that is finished.
 *
 * Lock status: Assumed that no lock is held upon entry.
 *
 * Returns:     Nothing
 *
 * Notes:       This function is matched in terms of capabilities to
 *              the function that created the scatter-gather list.
 *              In other words, if there are no bounce buffers
 *              (the normal case for most drivers), we don't need
 *              the logic to deal with cleaning up afterwards.
 *
```

```
 *              We must do one of several things here:
 *
 *              a) Call scsi_end_request.   This will finish off the
 *                 specified number of sectors.   If we are done, the
 *                 command block will be released, and the queue
 *                 function will be goosed.   If we are not done, then
 *                 scsi_end_request will directly goose the queue.
 *
 *              b) We can just use scsi_requeue_command() here.   This would
 *                 be used if we just wanted to retry, for example.
 */
void scsi_io_completion(struct scsi_cmnd *cmd, unsigned int good_bytes)
{
        int result = cmd->result;
        int this_count = cmd->request_bufflen;
        request_queue_t *q = cmd->device->request_queue;
        struct request *req = cmd->request;
        int clear_errors = 1;
        struct scsi_sense_hdr sshdr;
        int sense_valid = 0;
        int sense_deferred = 0;

        scsi_release_buffers(cmd);

        if (result) {
                sense_valid = scsi_command_normalize_sense(cmd, &sshdr);
                if (sense_valid)
                        sense_deferred = scsi_sense_is_deferred(&sshdr);
        }

        if (blk_pc_request(req)) { /* SG_IO ioctl from block level */
                req->errors = result;
                if (result) {
                        clear_errors = 0;
                        if (sense_valid && req->sense) {
                                /*
                                 * SG_IO wants current and deferred errors
                                 */
                                int len = 8 + cmd->sense_buffer[7];

                                if (len > SCSI_SENSE_BUFFERSIZE)
                                        len = SCSI_SENSE_BUFFERSIZE;
                                memcpy(req->sense, cmd->sense_buffer,   len);
                                req->sense_len = len;
```

```
                        }
                }
                req->data_len = cmd->resid;
        }

        /*
         * Next deal with any sectors which we were able to correctly
         * handle.
         */
        SCSI_LOG_HLCOMPLETE(1, printk("%ld sectors total, "
                                      "%d bytes done.\n",
                                      req->nr_sectors, good_bytes));
        SCSI_LOG_HLCOMPLETE(1, printk("use_sg is %d\n", cmd->use_sg));

        if (clear_errors)
                req->errors = 0;

        /* A number of bytes were successfully read.  If there
         * are leftovers and there is some kind of error
         * (result != 0), retry the rest.
         */
        if (scsi_end_request(cmd, 1, good_bytes, result == 0) == NULL)
                return;

        /* good_bytes = 0, or (inclusive) there were leftovers and
         * result = 0, so scsi_end_request couldn't retry.
         */
        if (sense_valid && !sense_deferred) {
                switch (sshdr.sense_key) {
                case UNIT_ATTENTION:
                        if (cmd->device->removable) {
                                /* Detected disc change.  Set a bit
                                 * and quietly refuse further access.
                                 */
                                cmd->device->changed = 1;
                                scsi_end_request(cmd, 0, this_count, 1);
                                return;
                        } else {
                                /* Must have been a power glitch, or a
                                 * bus reset.  Could not have been a
                                 * media change, so we just retry the
                                 * request and see what happens.
                                 */
                                scsi_requeue_command(q, cmd);
```

```
                        return;
                }
                break;
        case ILLEGAL_REQUEST:
                /* If we had an ILLEGAL REQUEST returned, then
                 * we may have performed an unsupported
                 * command.  The only thing this should be
                 * would be a ten byte read where only a six
                 * byte read was supported.  Also, on a system
                 * where READ CAPACITY failed, we may have
                 * read past the end of the disk.
                 */
                if ((cmd->device->use_10_for_rw &&
                    sshdr.asc == 0x20 && sshdr.ascq == 0x00) &&
                    (cmd->cmnd[0] == READ_10 ||
                     cmd->cmnd[0] == WRITE_10)) {
                        cmd->device->use_10_for_rw = 0;
                        /* This will cause a retry with a
                         * 6-byte command.
                         */
                        scsi_requeue_command(q, cmd);
                        return;
                } else {
                        scsi_end_request(cmd, 0, this_count, 1);
                        return;
                }
                break;
        case NOT_READY:
                /* If the device is in the process of becoming
                 * ready, or has a temporary blockage, retry.
                 */
                if (sshdr.asc == 0x04) {
                        switch (sshdr.ascq) {
                        case 0x01: /* becoming ready */
                        case 0x04: /* format in progress */
                        case 0x05: /* rebuild in progress */
                        case 0x06: /* recalculation in progress */
                        case 0x07: /* operation in progress */
                        case 0x08: /* Long write in progress */
                        case 0x09: /* self test in progress */
                                scsi_requeue_command(q, cmd);
                                return;
                        default:
                                break;
```

```
                                }
                        }
                        if (!(req->cmd_flags & REQ_QUIET)) {
                                scmd_printk(KERN_INFO, cmd,
                                            "Device not ready: ");
                                scsi_print_sense_hdr("", &sshdr);
                        }
                        scsi_end_request(cmd, 0, this_count, 1);
                        return;
                case VOLUME_OVERFLOW:
                        if (!(req->cmd_flags & REQ_QUIET)) {
                                scmd_printk(KERN_INFO, cmd,
                                            "Volume overflow, CDB: ");
                                __scsi_print_command(cmd->cmnd);
                                scsi_print_sense("", cmd);
                        }
                        /* See SSC3rXX or current. */
                        scsi_end_request(cmd, 0, this_count, 1);
                        return;
                default:
                        break;
                }
        }
        if (host_byte(result) == DID_RESET) {
                /* Third party bus reset or reset for error recovery
                 * reasons.   Just retry the request and see what
                 * happens.
                 */
                scsi_requeue_command(q, cmd);
                return;
        }
        if (result) {
                if (!(req->cmd_flags & REQ_QUIET)) {
                        scsi_print_result(cmd);
                        if (driver_byte(result) & DRIVER_SENSE)
                                scsi_print_sense("", cmd);
                }
        }
        scsi_end_request(cmd, 0, this_count, !result);
}
```

又是一个令人发指的函数.但我什么都不想多说了.直接跳到最后一行,scsi_end_request().来自 drivers/scsi_lib.c:

```
/*
 * Function:    scsi_end_request()
```

```
 *
 * Purpose:        Post-processing of completed commands (usually invoked at end
 *                 of upper level post-processing and scsi_io_completion).
 *
 * Arguments:      cmd      - command that is complete.
 *                 uptodate - 1 if I/O indicates success, <= 0 for I/O error.
 *                 bytes    - number of bytes of completed I/O
 *                 requeue  - indicates whether we should requeue leftovers.
 *
 * Lock status: Assumed that lock is not held upon entry.
 *
 * Returns:        cmd if requeue required, NULL otherwise.
 *
 * Notes:          This is called for block device requests in order to
 *                 mark some number of sectors as complete.
 *
 *                 We are guaranteeing that the request queue will be goosed
 *                 at some point during this call.
 * Notes:          If cmd was requeued, upon return it will be a stale pointer.
 */
static struct scsi_cmnd *scsi_end_request(struct scsi_cmnd *cmd, int uptodate,
                                          int bytes, int requeue)
{
        request_queue_t *q = cmd->device->request_queue;
        struct request *req = cmd->request;
        unsigned long flags;

        /*
         * If there are blocks left over at the end, set up the command
         * to queue the remainder of them.
         */
        if (end_that_request_chunk(req, uptodate, bytes)) {
                int leftover = (req->hard_nr_sectors << 9);

                if (blk_pc_request(req))
                        leftover = req->data_len;

                /* kill remainder if no retrys */
                if (!uptodate && blk_noretry_request(req))
                        end_that_request_chunk(req, 0, leftover);
                else {
                        if (requeue) {
                                /*
                                 * Bleah.  Leftovers again.  Stick the
```

```
678                                      * leftovers in the front of the
679                                      * queue, and goose the queue again.
680                                      */
681                                     scsi_requeue_command(q, cmd);
682                                     cmd = NULL;
683                             }
684                             return cmd;
685                     }
686             }
687
688             add_disk_randomness(req->rq_disk);
689
690             spin_lock_irqsave(q->queue_lock, flags);
691             if (blk_rq_tagged(req))
692                     blk_queue_end_tag(q, req);
693             end_that_request_last(req, uptodate);
694             spin_unlock_irqrestore(q->queue_lock, flags);
695
696             /*
697              * This will goose the queue request function at the end, so we don't
698              * need to worry about launching another command.
699              */
700             scsi_next_command(cmd);
701             return NULL;
702 }
```

而我们最需要关心的,是 693 行 end_that_request_last.

```
/*
 * queue lock must be held
 */
void end_that_request_last(struct request *req, int uptodate)
{
        struct gendisk *disk = req->rq_disk;
        int error;

        /*
         * extend uptodate bool to allow < 0 value to be direct io error
         */
        error = 0;
        if (end_io_error(uptodate))
                error = !uptodate ? -EIO : uptodate;

        if (unlikely(laptop_mode) && blk_fs_request(req))
                laptop_io_completion();

```

```
3636            /*
3637             * Account IO completion.   bar_rq isn't accounted as a normal
3638             * IO on queueing nor completion.   Accounting the containing
3639             * request is enough.
3640             */
3641            if (disk && blk_fs_request(req) && req != &req->q->bar_rq) {
3642                    unsigned long duration = jiffies - req->start_time;
3643                    const int rw = rq_data_dir(req);
3644
3645                    __disk_stat_inc(disk, ios[rw]);
3646                    __disk_stat_add(disk, ticks[rw], duration);
3647                    disk_round_stats(disk);
3648                    disk->in_flight--;
3649            }
3650            if (req->end_io)
3651                    req->end_io(req, error);
3652            else
3653                    __blk_put_request(req->q, req);
3654 }
```

好了,3651 行这个 end_io 是最关键的代码.也许你早已忘记我们曾经见过 end_io,但是不要紧,有我在.在 blk_execute_rq_nowait()中,曾经有一行

```
2596            rq->end_io = done;
```

而 done 是这个函数的第四个参数.当初我们在调用这个函数的时候,在 blk_execute_rq 中,我们是这样写的:

```
2636            blk_execute_rq_nowait(q, bd_disk, rq, at_head, blk_end_sync_rq);
```

也就是说,rq->end_io 被赋上了 blk_end_sync_rq.

```
2786 /**
2787  * blk_end_sync_rq - executes a completion event on a request
2788  * @rq: request to complete
2789  * @error: end io status of the request
2790  */
2791 void blk_end_sync_rq(struct request *rq, int error)
2792 {
2793            struct completion *waiting = rq->end_io_data;
2794
2795            rq->end_io_data = NULL;
2796            __blk_put_request(rq->q, rq);
2797
2798            /*
2799             * complete last, if this is a stack request the process (and thus
2800             * the rq pointer) could be invalid right after this complete()
2801             */
2802            complete(waiting);
2803 }
```

终于我们找到了亲爱的可爱的相爱的深爱的最爱的 complete().那么如何确定此 waiting 就是彼 wait 呢?对照一下这个 waiting,当时在 blk_execute_rq 中我们有:

```
		rq->end_io_data = &wait;
```

而眼下我们又有:

```
		struct completion *waiting = rq->end_io_data;
```

由此可知我们没有搞错对象,毕竟我们深知,接吻可以搞错对象,发脾气则不可以,写代码则更加不可以.

至此,blk_execute_rq 被唤醒,然后迅速返回.紧随其后的是 scsi_execute 的返回和 scsi_execute_req 的返回.这一刻,一个 scsi 命令终于从无到有最终到有,它经历了 scsi 命令到 request 的蜕变,也经历了 request 到 scsi 命令的历练.最终它完成了它的使命.对它来说,生命是一场幻觉,别离或者死亡是唯一的结局.

传说中的内存映射(上)

“如果这次有机会与中央首长握了手,能不能不要洗掉,这样等回去之后与他们握手,就如同首长与他们握手了.” 2007 年 10 月 17 日,参加十七大的福建三明市特殊教育学校校长黄金莲如此转述学生的嘱托.

网络暴民们对这一事件进行了强烈的讽刺和抨击,然而我觉得大可不必如此,事实上,学生们的想法看似纯朴,实则蕴含了一种深刻的思想,这就是 Linux 中的内存映射的思想.Linux 中经常有这样的情况,一个是用户空间的 buffer,一个是内核空间的 buffer,一个是属于应用程序,一个属于设备驱动,它们原本没有联系,它们只是永远的相提并论,只是永恒的擦肩而过,就仿佛天上的小鸟和水里的鱼,也许可以相恋,但是它们在哪里筑巢呢?

解决这一问题的方法就是映射,看似并不相连的世界,通过映射,就使得它们有关系了.但是为什么要让前者和后者联系起来呢?如果我把 user buffer 比作上例中的学生,而把 kernel buffer 比作黄金莲校长,那么你很快就能知道,之所以学生要和黄校长握手,并不是因为黄校长多么有明星气质,而是因为她和中央首长握了手,那么这里谁可以被比作中央首长呢?仔细一想就知道,设备驱动干嘛用的?用来驱动设备,没错,真正的主角不是设备驱动,而是设备.所以,应用程序之所以愿意把它的 user buffer 和 kernel buffer 映射起来,恰恰是因为 kernel buffer 和设备本身有联系.所以,和 kernel buffer 握手,就如同和设备握手.

我们拿 Block 层的两个函数来举例.这两个函数就是 blk_rq_map_user 和 blk_rq_map_kern.两者都来自 block/ll_rw_block.c.在我们分析 sd 模块时,说到 ioctl 时,我们最后实际上调用的是 sg_io(),而 sg_io()中我们需要调用 blk_rq_map_user 函数,所以我们先来看这个函数.

```
/**
 * blk_rq_map_user - map user data to a request, for REQ_BLOCK_PC usage
 * @q:             request queue where request should be inserted
 * @rq:            request structure to fill
 * @ubuf:          the user buffer
 * @len:           length of user data
 *
 * Description:
 *    Data will be mapped directly for zero copy io, if possible. Otherwise
 *    a kernel bounce buffer is used.
```

```
 *
 *    A matching blk_rq_unmap_user() must be issued at the end of io, while
 *    still in process context.
 *
 *    Note: The mapped bio may need to be bounced through blk_queue_bounce()
 *    before being submitted to the device, as pages mapped may be out of
 *    reach. It's the callers responsibility to make sure this happens. The
 *    original bio must be passed back in to blk_rq_unmap_user() for proper
 *    unmapping.
 */
int blk_rq_map_user(request_queue_t *q, struct request *rq, void __user *ubuf,
                    unsigned long len)
{
        unsigned long bytes_read = 0;
        struct bio *bio = NULL;
        int ret;

        if (len > (q->max_hw_sectors << 9))
                return -EINVAL;
        if (!len || !ubuf)
                return -EINVAL;

        while (bytes_read != len) {
                unsigned long map_len, end, start;

                        map_len = min_t(unsigned long, len - bytes_read,
BIO_MAX_SIZE);
                end = ((unsigned long)ubuf + map_len + PAGE_SIZE - 1)
                                                                        >>
PAGE_SHIFT;
                start = (unsigned long)ubuf >> PAGE_SHIFT;

                /*
                 * A bad offset could cause us to require BIO_MAX_PAGES + 1
                 * pages. If this happens we just lower the requested
                 * mapping len by a page so that we can fit
                 */
                if (end - start > BIO_MAX_PAGES)
                        map_len -= PAGE_SIZE;

                ret = __blk_rq_map_user(q, rq, ubuf, map_len);
                if (ret < 0)
                        goto unmap_rq;
                if (!bio)
```

```
                        bio = rq->bio;
                bytes_read += ret;
                ubuf += ret;
        }

        rq->buffer = rq->data = NULL;
        return 0;
unmap_rq:
        blk_rq_unmap_user(bio);
        return ret;
}
```

这个函数的参数 ubuf 不是别人,正是从用户空间传下来的那个 user buffer,或曰 user-space buffer,而 len 则是该 buffer 的长度.

也许我们早就该讲 struct bio 了.毫无疑问这个结构体是 Generic Block Layer 中最基础最核心最拉风最潇洒最酷的结构体之一.它表征的是一次正在进行的块设备 I/O 操作.经典的 Linux 书籍中无一例外的都对这个结构体进行了详细的介绍,但作为 80 后我们并不需要跟风,并不需要随波逐流,我们要追求自己的个性,所以这里我们并不过多地讲这个结构体,只是告诉你,它来自 include/linux/bio.h:

```
/*
 * main unit of I/O for the block layer and lower layers (ie drivers and
 * stacking drivers)
 */
struct bio {
        sector_t                bi_sector;      /* device address in 512 byte
                                                   sectors */
        struct bio              *bi_next;       /* request queue link */
        struct block_device     *bi_bdev;
        unsigned long           bi_flags;       /* status, command, etc */
        unsigned long           bi_rw;          /* bottom bits READ/WRITE,
                                                 * top bits priority
                                                 */

        unsigned short          bi_vcnt;        /* how many bio_vec's */
        unsigned short          bi_idx;         /* current index into bvl_vec */

        /* Number of segments in this BIO after
         * physical address coalescing is performed.
         */
        unsigned short          bi_phys_segments;

        /* Number of segments after physical and DMA remapping
         * hardware coalescing is performed.
         */
        unsigned short          bi_hw_segments;
```

```

	unsigned int			bi_size;		/* residual I/O count */

	/*
	 * To keep track of the max hw size, we account for the
	 * sizes of the first and last virtually mergeable segments
	 * in this bio
	 */
	unsigned int			bi_hw_front_size;
	unsigned int			bi_hw_back_size;

	unsigned int			bi_max_vecs;	/* max bvl_vecs we can hold */

	struct bio_vec		*bi_io_vec;	/* the actual vec list */

	bio_end_io_t			*bi_end_io;
	atomic_t			bi_cnt;		/* pin count */

	void				*bi_private;

	bio_destructor_t		*bi_destructor;	/* destructor */
};
```

而它的存在并非是孤立的,它和 request 是有联系的.struct request 中有一个成员 struct bio *bio,表征的就是这个 request 的 bio 们,因为一个 request 包含多个 I/O 操作.而 blk_rq_map_user 的主要工作就是建立 user buffer 和 bio 之间的映射,具体工作是由__blk_rq_map_user 来完成的.

```
static int __blk_rq_map_user(request_queue_t *q, struct request *rq,
				     void __user *ubuf, unsigned int len)
{
	unsigned long uaddr;
	struct bio *bio, *orig_bio;
	int reading, ret;

	reading = rq_data_dir(rq) == READ;

	/*
	 * if alignment requirement is satisfied, map in user pages for
	 * direct dma. else, set up kernel bounce buffers
	 */
	uaddr = (unsigned long) ubuf;
	if (!(uaddr & queue_dma_alignment(q)) && !(len & queue_dma_alignment(q)))
		bio = bio_map_user(q, NULL, uaddr, len, reading);
	else
		bio = bio_copy_user(q, uaddr, len, reading);
```

```

	if (IS_ERR(bio))
		return PTR_ERR(bio);

	orig_bio = bio;
	blk_queue_bounce(q, &bio);

	/*
	 * We link the bounce buffer in and could have to traverse it
	 * later so we have to get a ref to prevent it from being freed
	 */
	bio_get(bio);

	if (!rq->bio)
		blk_rq_bio_prep(q, rq, bio);
	else if (!ll_back_merge_fn(q, rq, bio)) {
		ret = -EINVAL;
		goto unmap_bio;
	} else {
		rq->biotail->bi_next = bio;
		rq->biotail = bio;

		rq->data_len += bio->bi_size;
	}

	return bio->bi_size;

unmap_bio:
	/* if it was boucned we must call the end io function */
	bio_endio(bio, bio->bi_size, 0);
	__blk_rq_unmap_user(orig_bio);
	bio_put(bio);
	return ret;
}
```

但至少目前为止,bio 还只是一个虚无缥缈的指针,华而不实,谁为它申请了内存呢?让我们接着深入,进一步我们需要关注的是 bio_map_user().uaddr 是 ubuf 的虚拟地址,如果其满足所在队列的字节对齐要求,则 bio_map_user()会被调用.(否则需要调用 bio_copy_user()来建立所谓的 bounce buffer,不表.)该函数来自 fs/bio.c:

```
/**
 *	bio_map_user	-	map user address into bio
 *	@q: the request_queue_t for the bio
 *	@bdev: destination block device
 *	@uaddr: start of user address
 *	@len: length in bytes
```

```
 *      @write_to_vm: bool indicating writing to pages or not
 *
 *      Map the user space address into a bio suitable for io to a block
 *      device. Returns an error pointer in case of error.
 */
struct bio *bio_map_user(request_queue_t *q, struct block_device *bdev,
                         unsigned long uaddr, unsigned int len, int write_to_vm)
{
        struct sg_iovec iov;

        iov.iov_base = (void __user *)uaddr;
        iov.iov_len = len;

        return bio_map_user_iov(q, bdev, &iov, 1, write_to_vm);
}
```

走到这里,struct sg_iovec 似曾相识,仔细回忆一下,在 sd 中讲 ioctl 的时候曾经讲过这个结构体,描述的就是一个 scatter-gather 数组成员.iovec 就是 io vector 的意思,即 IO 向量,或者说一个由基地址和长度组成的结构体.

关于函数的各个参数,注释里说得很清楚,而且注释也说了这个函数的目的,不难知道这个函数将返回一个描述了一次 IO 操作的 bio 指针.不过真正干活的是 bio_map_user_iov().于是再转战至 bio_map_user_iov().同样来自 fs/bio.c:

```
/**
 *      bio_map_user_iov - map user sg_iovec table into bio
 *      @q: the request_queue_t for the bio
 *      @bdev: destination block device
 *      @iov:   the iovec.
 *      @iov_count: number of elements in the iovec
 *      @write_to_vm: bool indicating writing to pages or not
 *
 *      Map the user space address into a bio suitable for io to a block
 *      device. Returns an error pointer in case of error.
 */
struct bio *bio_map_user_iov(request_queue_t *q, struct block_device *bdev,
                             struct sg_iovec *iov, int iov_count,
                             int write_to_vm)
{
        struct bio *bio;

        bio = __bio_map_user_iov(q, bdev, iov, iov_count, write_to_vm);

        if (IS_ERR(bio))
                return bio;

        /*
```

```
         * subtle -- if __bio_map_user() ended up bouncing a bio,
         * it would normally disappear when its bi_end_io is run.
         * however, we need it for the unmap, so grab an extra
         * reference to it
         */
        bio_get(bio);

        return bio;
}
```

还不是终点,继续走入__bio_map_user_iov().

```
static struct bio *__bio_map_user_iov(request_queue_t *q,
                                      struct block_device *bdev,
                                      struct sg_iovec *iov, int iov_count,
                                      int write_to_vm)
{
        int i, j;
        int nr_pages = 0;
        struct page **pages;
        struct bio *bio;
        int cur_page = 0;
        int ret, offset;

        for (i = 0; i < iov_count; i++) {
                unsigned long uaddr = (unsigned long)iov[i].iov_base;
                unsigned long len = iov[i].iov_len;
                unsigned long end = (uaddr + len + PAGE_SIZE - 1) >> PAGE_SHIFT;
                unsigned long start = uaddr >> PAGE_SHIFT;

                nr_pages += end - start;
                /*
                 * buffer must be aligned to at least hardsector size for now
                 */
                if (uaddr & queue_dma_alignment(q))
                        return ERR_PTR(-EINVAL);
        }

        if (!nr_pages)
                return ERR_PTR(-EINVAL);

        bio = bio_alloc(GFP_KERNEL, nr_pages);
        if (!bio)
                return ERR_PTR(-ENOMEM);

```

```
	ret = -ENOMEM;
	pages = kcalloc(nr_pages, sizeof(struct page *), GFP_KERNEL);
	if (!pages)
		goto out;

	for (i = 0; i < iov_count; i++) {
		unsigned long uaddr = (unsigned long)iov[i].iov_base;
		unsigned long len = iov[i].iov_len;
		unsigned long end = (uaddr + len + PAGE_SIZE - 1) >> PAGE_SHIFT;
		unsigned long start = uaddr >> PAGE_SHIFT;
		const int local_nr_pages = end - start;
		const int page_limit = cur_page + local_nr_pages;

		down_read(&current->mm->mmap_sem);
		ret = get_user_pages(current, current->mm, uaddr,
					local_nr_pages,
					write_to_vm, 0, &pages[cur_page], NULL);
		up_read(&current->mm->mmap_sem);

		if (ret < local_nr_pages) {
			ret = -EFAULT;
			goto out_unmap;
		}

		offset = uaddr & ~PAGE_MASK;
		for (j = cur_page; j < page_limit; j++) {
			unsigned int bytes = PAGE_SIZE - offset;

			if (len <= 0)
				break;

			if (bytes > len)
				bytes = len;

			/*
			 * sorry...
			 */
			if (bio_add_pc_page(q, bio, pages[j], bytes, offset) <
					    bytes)
				break;

			len -= bytes;
```

```
678                             offset = 0;
679                     }
680
681                     cur_page = j;
682                     /*
683                      * release the pages we didn't map into the bio, if any
684                      */
685                     while (j < page_limit)
686                             page_cache_release(pages[j++]);
687             }
688
689             kfree(pages);
690
691             /*
692              * set data direction, and check if mapped pages need bouncing
693              */
694             if (!write_to_vm)
695                     bio->bi_rw |= (1 << BIO_RW);
696
697             bio->bi_bdev = bdev;
698             bio->bi_flags |= (1 << BIO_USER_MAPPED);
699             return bio;
700
701     out_unmap:
702             for (i = 0; i < nr_pages; i++) {
703                     if(!pages[i])
704                             break;
705                     page_cache_release(pages[i]);
706             }
707     out:
708             kfree(pages);
709             bio_put(bio);
710             return ERR_PTR(ret);
711 }
```

632 行,bio_alloc(),看到了吧,很明显,内存是在这里申请的,bio 从此站了起来.
我们本可以不再深入,但是阿信告诉我们看代码不淋漓尽致不痛快.
所以继续深入 bio_alloc,来自 fs/bio.c:

```
187 struct bio *bio_alloc(gfp_t gfp_mask, int nr_iovecs)
188 {
189             struct bio *bio = bio_alloc_bioset(gfp_mask, nr_iovecs, fs_bio_set);
190
191             if (bio)
192                     bio->bi_destructor = bio_fs_destructor;
193
```

```
        return bio;
}
```

其实就是调用 bio_alloc_bioset(),来自同一个文件:

```
/**
 * bio_alloc_bioset - allocate a bio for I/O
 * @gfp_mask:   the GFP_ mask given to the slab allocator
 * @nr_iovecs:  number of iovecs to pre-allocate
 * @bs:         the bio_set to allocate from
 *
 * Description:
 *   bio_alloc_bioset will first try it's on mempool to satisfy the allocation.
 *   If %__GFP_WAIT is set then we will block on the internal pool waiting
 *   for a &struct bio to become free.
 *
 *   allocate bio and iovecs from the memory pools specified by the
 *   bio_set structure.
 **/
struct bio *bio_alloc_bioset(gfp_t gfp_mask, int nr_iovecs, struct bio_set *bs)
{
        struct bio *bio = mempool_alloc(bs->bio_pool, gfp_mask);

        if (likely(bio)) {
                struct bio_vec *bvl = NULL;

                bio_init(bio);
                if (likely(nr_iovecs)) {
                        unsigned long idx = 0; /* shut up gcc */

                        bvl = bvec_alloc_bs(gfp_mask, nr_iovecs, &idx, bs);
                        if (unlikely(!bvl)) {
                                mempool_free(bio, bs->bio_pool);
                                bio = NULL;
                                goto out;
                        }
                        bio->bi_flags |= idx << BIO_POOL_OFFSET;
                        bio->bi_max_vecs = bvec_slabs[idx].nr_vecs;
                }
                bio->bi_io_vec = bvl;
        }
out:
        return bio;
}
```

看到这儿基本上就明白怎么回事了.mempool_alloc 很明确的告诉我们,为 bio 申请了内存,紧接着 bio_init()为它做了初始化.更多细节不再说了,唯一需要关注的是,nr_iovecs,一路传过来

的, __bio_map_user_iov()中把 nr_pages 传递了给了 bio_alloc(),而 615 行到 627 行对 nr_pages 进行了计算,通过一个 for 循环累加,循环次数是 iov_count,每次雷加的是 end 和 start 的差值.很显然,最终的 nr_pages 就是 iov 数组所对应的 page 的数量,而 iov 是__bio_map_user_iov 的第三个参数,另一方面,很显然,iov_count 表征的是 iov 数组的元素个数,而在 bio_map_user 中调用 bio_map_user_iov 时传递的第三个参数是 1,所以 iov_count 就是 1.不过这些都不重要,重要的是我们现在有 bio 了.我们结束 bio_alloc,回到__bio_map_user_iov 中继续往下走,637 行,申请了另一个东西,pages,一个二级指针,冥冥中感觉到这将代表一个指针数组.
而紧接着,又是另一个 for 循环.而 get_user_pages 是获得 page 描述符.这一行代码应该是灵魂性质的代码.从这一刻起,用户空间的 buffer 和内核空间建立了姻缘.让我们从下面这幅图说起.

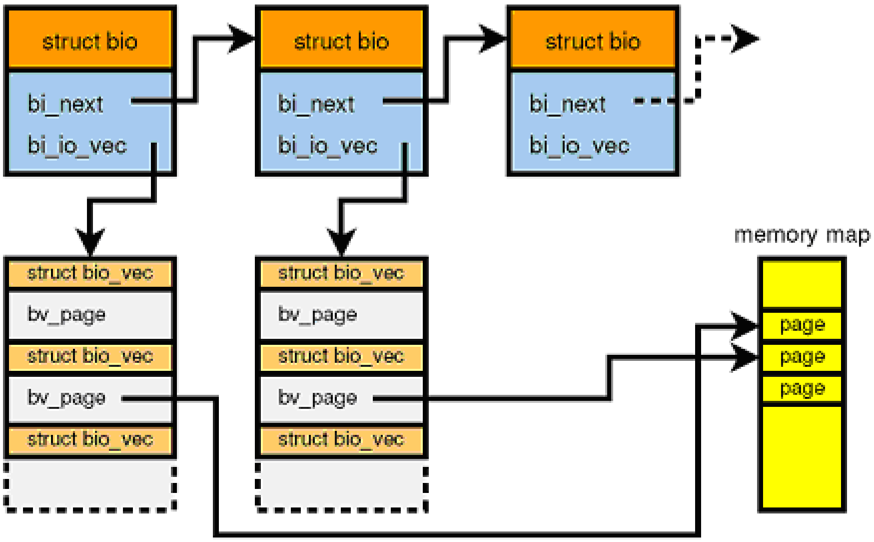

Figure 4.3.: A request holds a pointer to a list of bio structures, whereas each BIO has a pointer to a vector array with the corresponding memory page information. The kernel uses these structures to transfer data blocks from a block device to memory or vice versa.

Bio 中最重要的成员就是 bi_io_vec 和 bi_vcnt.bi_io_vec 是一个 struct bio_vec 指针,后者的定义在 include/linux/bio.h 中:

```
54 /*
55    * was unsigned short, but we might as well be ready for > 64kB I/O pages
56    */
57 struct bio_vec {
58              struct page        *bv_page;
59              unsigned int       bv_len;
```

```
        unsigned int        bv_offset;
};
```

而 bi_io_vec 实际上则是代表了一个 struct bio_vec 的数组,bi_vcnt 是这个数组的元素个数.如图中看到的那样,bio_vec 中的成员 bv_page 指向的是一个个映射的 page.而建立映射的恰恰就是刚才看到的这个伟大的 get_user_pages()函数,是它让这些个 page 和用户空间的 buffer 联系了起来.而 bio_add_pc_page()则是让 bv_page 指向相应的 page.之所以要把 page 和用户空间的 buffer 映射起来,其原因在于 block 层只认 bio 不认用户空间的 user buffer,block 层的那些个函数都是针对 bio 来操作的,它们可不管你什么用户空间不用户空间,它们就管自己的 bio,它们就知道每一个 request 对应一个 bio.

关于 get_user_pages 函数,其原型在 include/linux/mm.h 中:

```
int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start,
                        int len, int write, int force, struct page **pages, struct vm_area_struct **vmas);
```

这其中,start 和 len 这两个参数描述的是 user-space buffer,(其中 len 的单位是 page,即 len 如果为 3 就表示 3 个 page.)本函数的目的就是把这个 user-space buffer 映射到内核空间,而 pages 和 vmas 是这个函数的输出.其中 pages 是一个二级指针,换言之它其实就是一个指针数组,包含的是一群 page 指针,这群 page 指针指向的正是这个 user-space buffer.这个函数的返回值是实际映射了几个 pages.(The return value is the number of pages actually mapped.)而 vmas 咱们不用管了,至少咱们这里传递进去的是 NULL,所以它不会起什么作用.

继续对 get_user_pages 多八卦几句,正如每一个成功的男人背后都有一个(或多个)女人,比如张斌老师,比如赵忠祥老师,比如李金斗老师,每一个 Linux 进程背后都有一个页表.在进程创建的时候会在其地址空间中建立自己的页表,对于 x86 而言,页表中一共有 1024 项,每一项可以表征一个 page,而该 page 是否存在于物理内存中呢?这就很难说了.我们不妨把 page table 中的 1024 项说成 1024 个指针,这 1024 个指针都是 32 个 bits,这其中就有一位被叫做 Present 位,它为 1 则说明该 page 存在于物理内存中,它为 0 则说明不存在物理内存中.

那么这和我们这个 get_user_pages 有什么关系呢?get_user_pages 的参数 start 和 len 表征的是线性地址,拿 x86 来说,线性地址一共 32 个 bits,这三十二个 bits 分为三段,bit31-bit22 称为 Directory,或者说 Page Directory 中的索引,bit21-bit12 称为 Table,或者说 Page Table 中的索引,bit11-bit0 则是 Offset.给定了一个虚拟地址,或者说线性地址,就相当于给定了它在 Page Directory 中的位置,给定了它在 Page Table 中的位置,也就是说给定了一个 Page.假如这个 Page 在物理内存中,那么好说,但是如果不在呢?如果不在,这时候 get_user_pages()方显英雄本色,它会申请一个 Page Frame,会相应的设置页表.这之后,这段虚拟地址就属于有后台的虚拟地址了,因为有物理地址给它撑腰,这样你应用程序就可以访问它了,而设备驱动也可以访问它了,只不过设备驱动并不是直接访问这些个地址,还是前面说的,Block 层只认 bio,不认 page,不认虚拟地址,所以有下面这个函数 bio_add_pc_page(),负责把 page 和 bio 联系起来.

我们来看 bio_add_pc_page,它来自 fs/bio.c:

```
/**
 *      bio_add_pc_page -       attempt to add page to bio
 *      @q: the target queue
 *      @bio: destination bio
 *      @page: page to add
 *      @len: vec entry length
 *      @offset: vec entry offset
 *
```

```
 *      Attempt to add a page to the bio_vec maplist. This can fail for a
 *      number of reasons, such as the bio being full or target block
 *      device limitations. The target block device must allow bio's
 *      smaller than PAGE_SIZE, so it is always possible to add a single
 *      page to an empty bio. This should only be used by REQ_PC bios.
 */
int bio_add_pc_page(request_queue_t *q, struct bio *bio, struct page *page,
                    unsigned int len, unsigned int offset)
{
        return __bio_add_page(q, bio, page, len, offset, q->max_hw_sectors);
}
```

而__bio_add_pages 来自同一个文件.

```
static int __bio_add_page(request_queue_t *q, struct bio *bio, struct page
                          *page, unsigned int len, unsigned int offset,
                          unsigned short max_sectors)
{
        int retried_segments = 0;
        struct bio_vec *bvec;

        /*
         * cloned bio must not modify vec list
         */
        if (unlikely(bio_flagged(bio, BIO_CLONED)))
                return 0;

        if (((bio->bi_size + len) >> 9) > max_sectors)
                return 0;

        /*
         * For filesystems with a blocksize smaller than the pagesize
         * we will often be called with the same page as last time and
         * a consecutive offset.  Optimize this special case.
         */
        if (bio->bi_vcnt > 0) {
                struct bio_vec *prev = &bio->bi_io_vec[bio->bi_vcnt - 1];

                if (page == prev->bv_page &&
                    offset == prev->bv_offset + prev->bv_len) {
                        prev->bv_len += len;
                        if (q->merge_bvec_fn &&
                            q->merge_bvec_fn(q, bio, prev) < len) {
                                prev->bv_len -= len;
                                return 0;
                        }
```

```
				goto done;
			}
		}

		if (bio->bi_vcnt >= bio->bi_max_vecs)
			return 0;

		/*
		 * we might lose a segment or two here, but rather that than
		 * make this too complex.
		 */

		while (bio->bi_phys_segments >= q->max_phys_segments
		       || bio->bi_hw_segments >= q->max_hw_segments
		       || BIOVEC_VIRT_OVERSIZE(bio->bi_size)) {

			if (retried_segments)
				return 0;

			retried_segments = 1;
			blk_recount_segments(q, bio);
		}

		/*
		 * setup the new entry, we might clear it again later if we
		 * cannot add the page
		 */
		bvec = &bio->bi_io_vec[bio->bi_vcnt];
		bvec->bv_page = page;
		bvec->bv_len = len;
		bvec->bv_offset = offset;

		/*
		 * if queue has other restrictions (eg varying max sector size
		 * depending on offset), it can specify a merge_bvec_fn in the
		 * queue to get further control
		 */
		if (q->merge_bvec_fn) {
			/*
			 * merge_bvec_fn() returns number of bytes it can accept
			 * at this offset
			 */
			if (q->merge_bvec_fn(q, bio, bvec) < len) {
```

```
394                         bvec->bv_page = NULL;
395                         bvec->bv_len = 0;
396                         bvec->bv_offset = 0;
397                         return 0;
398                 }
399         }
400
401         /* If we may be able to merge these biovecs, force a recount */
402         if (bio->bi_vcnt && (BIOVEC_PHYS_MERGEABLE(bvec-1, bvec) ||
403             BIOVEC_VIRT_MERGEABLE(bvec-1, bvec)))
404                 bio->bi_flags &= ~(1 << BIO_SEG_VALID);
405
406         bio->bi_vcnt++;
407         bio->bi_phys_segments++;
408         bio->bi_hw_segments++;
409  done:
410         bio->bi_size += len;
411         return len;
412 }
```

Block 层很多东西都是为 Raid 服务的,比如这里的这个 merge_bvec_fn 函数指针,对于普通的硬盘驱动来说,是没有这么一个破指针的,或者说这个指针指向的是空气.不过有意思的是没有这个函数的话,__bio_add_pages 这个函数就变得很简单了,所以我们很开心.这个函数最有意义的代码就是 378 行到 381 行对 bvec 的赋值,以及 406 行到 410 行对 bio 的赋值.友情提醒一下,注意 410 行这个赋值,bio->bi_size 就是 len 的累加,如果你仔细追踪一下就会发现,其实兜来转去,这个 bio->bi_size 就是最初用户空间传下来那个 len.

函数__bio_map_user_iov()中,661 行到 679 行这个 for 循环,就是让这所有的那些 pages 一个个的全都加入到 bio 的那张 bi_io_vec 表里去,让每一个 bv_page 都有所指.

然后,在 699 行,__bio_map_user_iov()函数返回,返回的就是 bio.紧接着,bio_map_user_iov()和 bio_map_user()也先后返回,返回值也都是这个 bio.我们于是回到了__blk_rq_map_user()中.

不过,我们刚才也看到了,bio 是有了,bio 和 pages 也有了暧昧关系,bio 和 user buffer 也有了暧昧关系,可是这就够了吗?很显然 bio 还应该和 request 建立关系吧,没加入到 request 中的 bio 可不是有用的 bio,request 和 bio 之间的关系如下图所示:

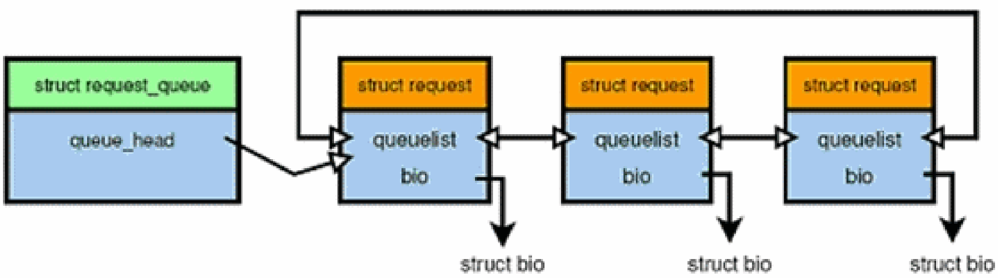

Figure 4.2.: Read and write requests are collected in *request queues*. This structure includes a pointer to a doubly-linked list which contains the requests. Each request has a pointer to a so-called `bio` (block I/O) structure which maps a block to a page instance in memory (figure 4.3).

完成这项工作的就是 2373 行调用的 blk_rq_bio_prep()函数,来自 block/ll_rw_blk.c:

```
void blk_rq_bio_prep(request_queue_t *q, struct request *rq, struct bio *bio)
{
        /* first two bits are identical in rq->cmd_flags and bio->bi_rw */
        rq->cmd_flags |= (bio->bi_rw & 3);

        rq->nr_phys_segments = bio_phys_segments(q, bio);
        rq->nr_hw_segments = bio_hw_segments(q, bio);
        rq->current_nr_sectors = bio_cur_sectors(bio);
        rq->hard_cur_sectors = rq->current_nr_sectors;
        rq->hard_nr_sectors = rq->nr_sectors = bio_sectors(bio);
        rq->buffer = bio_data(bio);
        rq->data_len = bio->bi_size;

        rq->bio = rq->biotail = bio;
}
```

到这里 bio 正式嫁入 rq.

回到__blk_rq_map_user(),也该返回了,2384 行,返回的是 bio->bi_size.刚才说过了,这个就是用户空间传过来那个 user buffer 的长度.

而回到 blk_rq_map_user()中,发现这个函数也该结束了,正常的话这个函数返回 0.于是这个浩大的映射工程就算是结束了.然而网友”贱男村村长”提出质疑,这些个 bio 什么时候被用到的?当时在讲 scsi 命令的时候好像没怎么说起?其实当时在讲 scsi 命令的时候,有这么一个函数,scsi_setup_blk_pc_cmnd,这个函数 1104 行就是判断 req->bio 是否为 NULL,如果不为 NULL,则会对它进行相应的处理,一个叫做 scsi_init_io()的函数会被调用,会建立一个 scatter-gather 数组来和这个 bio 中的向量 bi_io_vec 相对应.

传说中的内存映射(下)

下面我们来看另一个”映射”函数,blk_rq_map_kern().当我们在设备驱动内部或者 scsi mid-level 要发送 scsi 命令给设备的时候,我们会调用这个函数.回首往事,当年在讲 scsi 命令的时候,在 scsi_execute_req() 调用了 scsi_execute() 之后,scsi_execute() 中就会调用 blk_rq_map_kern()函数.正常情况下它应该返回 0,在当年的 scsi_execute()中,189 行,判断如果 bufflen 不为 0 且 blk_rq_map_kern()也不为 0,就毫不犹豫的跳出函数,之所以如此果断,是因为,如果 bufflen 不为 0,则说明这次 scsi 命令需要传输数据,既然需要传输数据,就需要得到 bio 的支持,而 blk_rq_map_kern 的任务就是完成 rq 和 bio,bio 和 pages 的那种建交.它的返回值如果不为 0,本身就说明出错了,那么既然它出错了,scsi 命令也就没必要往下执行了.

Ok,来看具体的代码吧,blk_rq_map_kern(),来自 block/ll_rw_blk.c:

```
/**
 * blk_rq_map_kern - map kernel data to a request, for REQ_BLOCK_PC usage
 * @q:              request queue where request should be inserted
 * @rq:             request to fill
 * @kbuf:           the kernel buffer
```

```
 * @len:          length of user data
 * @gfp_mask:     memory allocation flags
 */
int blk_rq_map_kern(request_queue_t *q, struct request *rq, void *kbuf,
                    unsigned int len, gfp_t gfp_mask)
{
        struct bio *bio;

        if (len > (q->max_hw_sectors << 9))
                return -EINVAL;
        if (!len || !kbuf)
                return -EINVAL;

        bio = bio_map_kern(q, kbuf, len, gfp_mask);
        if (IS_ERR(bio))
                return PTR_ERR(bio);

        if (rq_data_dir(rq) == WRITE)
                bio->bi_rw |= (1 << BIO_RW);

        blk_rq_bio_prep(q, rq, bio);
        blk_queue_bounce(q, &rq->bio);
        rq->buffer = rq->data = NULL;
        return 0;
}
```

和 blk_rq_map_user()不同的是,这里的 kbuf 是内核空间的 buffer.这是一个让人大跌隐形眼镜的函数,因为既然 kbuf 是内核空间的 buffer,而 request 也是存在于内核空间,那么大家都是一条道上混的,何来映射之说?事实上,虽然这个函数自称”map”,但它和 map 根本没有关系,一个更合适的做法是把 map 这个词换成 associate,没必要用 map 这么一个欺骗性的词.不过写代码的人这么做我们也没办法,毕竟在这个很黄很暴力的时代,整个社会系统都在鼓励谎言,掩盖真相.就像 CCTV,虽然它声称自己代表民意,虽然它总是善于假借民意,但是它从来就没有代表过任何民意.它为了给<<互联网视听节目服务管理规定>>出台造势,不惜借助并诱导张殊凡小朋友向全国人民说谎,以此来说明它们所鼓吹的是伟大光荣正确的.但最终只是让这个 13 岁的孩子受到伤害,只是让网络暴民们同仇敌忾,只是让大家更清楚的认识到那个所谓的全国收视率最高的节目不过是由一帮骗子导演的谎言恶剧.

Ok,甭管假不假,只有看代码是王道.首先,bio_map_kern()来自 fs/bio.c:

```
/**
 *      bio_map_kern    -       map kernel address into bio
 *      @q: the request_queue_t for the bio
 *      @data: pointer to buffer to map
 *      @len: length in bytes
 *      @gfp_mask: allocation flags for bio allocation
 *
 *      Map the kernel address into a bio suitable for io to a block
```

```
 *      device. Returns an error pointer in case of error.
 */
struct bio *bio_map_kern(request_queue_t *q, void *data, unsigned int len,
                         gfp_t gfp_mask)
{
        struct bio *bio;

        bio = __bio_map_kern(q, data, len, gfp_mask);
        if (IS_ERR(bio))
                return bio;

        if (bio->bi_size == len)
                return bio;

        /*
         * Don't support partial mappings.
         */
        bio_put(bio);
        return ERR_PTR(-EINVAL);
}
```

__bio_map_kern()亦来自 fs/bio.c:

```
static struct bio *__bio_map_kern(request_queue_t *q, void *data,
                                  unsigned int len, gfp_t gfp_mask)
{
        unsigned long kaddr = (unsigned long)data;
        unsigned long end = (kaddr + len + PAGE_SIZE - 1) >> PAGE_SHIFT;
        unsigned long start = kaddr >> PAGE_SHIFT;
        const int nr_pages = end - start;
        int offset, i;
        struct bio *bio;

        bio = bio_alloc(gfp_mask, nr_pages);
        if (!bio)
                return ERR_PTR(-ENOMEM);

        offset = offset_in_page(kaddr);
        for (i = 0; i < nr_pages; i++) {
                unsigned int bytes = PAGE_SIZE - offset;

                if (len <= 0)
                        break;

                if (bytes > len)
                        bytes = len;
```

```
834
835                     if (bio_add_pc_page(q, bio, virt_to_page(data), bytes,
836                                                   offset) < bytes)
837                             break;
838
839                     data += bytes;
840                     len -= bytes;
841                     offset = 0;
842             }
843
844             bio->bi_end_io = bio_map_kern_endio;
845             return bio;
846 }
```

仔细对比一下这个函数与__bio_map_user_iov(),不难发现,本质的不同就是差了那个get_user_page()函数,而其它方面基本上是一样的.一样调用 bio_alloc 来申请 bio 的内存,一样调用 bio_add_pc_page()来把 bio 和 pages 们联系起来.

说点内存管理的题外话,virt_to_page(),它就是把一个虚拟地址转化为一个 page.注意这里的 data 实际上就是前面 blk_rq_map_kern()传下来的那个 kbuf,如果我们追溯过去,去看 scsi_execute()甚至回到 scsi_execute_req(),我们去看那些调用 scsi_execute_req()的地方,比如在 sd 模块中,sd_revalidate_disk()函数中,有这么一行,

```
1518            buffer = kmalloc(SD_BUF_SIZE, GFP_KERNEL | __GFP_DMA);
```

还有这么一行,

```
1540                    sd_read_capacity(sdkp, buffer);
```

而我们知道 sd_read_capacity()会调用 scsi_execute_req()来执行 Read Capacity 命令.所以这个 kernel-space 的 buffer 最初的来源就是这里这个 kmalloc.对于 x86 系统来说,这段内存就是永久映射在内核空间的那个896M 以下的内存.因为 virt_to_page 这个宏有硬性要求,它的参数必须是这个范围内的内存.

最后,844 行,bio 的成员 bi_end_io 指向的是一个函数,这个函数将在这个 bio 对应的 io 操作结束的时候被调用.所以我们知道,在不久的可以看见的将来的某一天,bio_map_kern_endio()函数会被调用.不过这个函数不干什么正经事罢了,来自 fs/bio.c:

```
801 static int bio_map_kern_endio(struct bio *bio, unsigned int bytes_done, int err)
802 {
803             if (bio->bi_size)
804                     return 1;
805
806             bio_put(bio);
807             return 0;
808 }
```

结束了 bio_map_kern()之后,回到 blk_rq_map_kern().一样要调用 blk_rq_bio_prep()来把 bio 和 rq 联系起来.而之后调用 blk_queue_bounce()是为了建立 bounce buffer,当 buffer pages 不适合这次 I/O 操作的时候需要利用 bounce buffer,比如设备本身有限制,只能访问某些 pages.

用我一个懂 Linux 的同事 Hugh Dickins 的话说就是,it is substituting bounce buffers if the buffer pages are unsuited to this I/O,e.g. device limited in the address range of pages it can access.关于 blk_queue_bounce 我们就不多说了.毕竟是少数情况需要用到.如果需要 bounce buffer,那么在

struct request_queue 中可以设置,因为它有一个成员,unsigned long bounce_pfn,需要设置的可以调用函数 blk_queue_bounce_limit()来设置.比如我们前面看到的__scsi_alloc_queue()函数,就调用了 blk_queue_bounce_limit().

```
        blk_queue_bounce_limit(q, scsi_calculate_bounce_limit(shost));
```

如果你具有十足的八卦精神,如果你具有专业的八卦水准,那么你可以去看看这个 scsi_calculate_bounce_limit,这个来自 drivers/scsi/scsi_lib.c 中的函数.

```
u64 scsi_calculate_bounce_limit(struct Scsi_Host *shost)
{
        struct device *host_dev;
        u64 bounce_limit = 0xffffffff;

        if (shost->unchecked_isa_dma)
                return BLK_BOUNCE_ISA;
        /*
         * Platforms with virtual-DMA translation
         * hardware have no practical limit.
         */
        if (!PCI_DMA_BUS_IS_PHYS)
                return BLK_BOUNCE_ANY;

        host_dev = scsi_get_device(shost);
        if (host_dev && host_dev->dma_mask)
                bounce_limit = *host_dev->dma_mask;

        return bounce_limit;
}
```

基本上对于 scsi 设备来说,需要不需要 bounce buffer,主要得由 scsi host 说了算,因为 scsi 的世界里,host 是一家之主,device 是从属于 host 的.就好比张斌的那些女人们能不能被扶正,能不能从第五者变成第四者,能不能从第四者变成第三者,关键还得张斌说了算,因为在紫薇大闹央视发布会这台戏后,真正的主角还是张斌.

最后总结一下,blk_rq_map_user()和 blk_rq_map_kern(),其实我还是那句话,map 这个词用得不是很合适,更好一点应该叫 associate,因为在这两个函数中,映射并不是最主要的,最主要的是联系,就是说甭管你是用户空间的 buffer 还是内核空间的 buffer,我 Block 层都不认,我只认 bio,我的这些函数只和 bio 打交道.这种情况生活中也很常见,就比如火车上的乘务员和列车长们在查票的时候,如果遇到残疾人,他们的态度一定是只认证不认人.我想我们没有理由忘记当年那辆开往西安的火车上,那位列车长面对那个只有半个脚掌,那个买了一张和残疾人票一样价格的票的中年人时,说的那句铿锵有力的话:”我们只认证不认人!有残疾证就是残疾人,没有残疾证怎么能证明你是残疾人啊?”

好在开源社区的人没有这么无情,在他们看来,虽然我们要的是 bio,不是 buffer,但是毕竟 bio 可以和 page 有联系,page 可以和线性地址有联系,所以最终我们的解决方案就是通过这两个函数让 buffer 或者说让 buffer 所对应的地址和 bio 联系起来,这才是根本,而映射只是达到这一目的所采取的手段,并且只是用户空间的 buffer 才有此需求.(当然如果你喜欢钻牛角尖,那你也可以说内核空间的 buffer 也是映射好了的,因为 kmalloc()申请的内存本身就是映射好了的内存,不过这都无所谓.)