

第 13 章

Linux 块设备驱动

本章导读

块设备是与字符设备并列的概念，这两类设备在 Linux 中的驱动结构有较大差异，总体而言，块设备驱动比字符设备驱动要复杂得多，在 I/O 操作上也表现出极大的不同。缓冲、I/O 调度、请求队列等都是与块设备驱动相关的概念。本章将详细讲解 Linux 块设备驱动的编程方法。

13.1 节讲解块设备的 I/O 操作特点，分析字符设备与块设备在 I/O 操作上的差异。

13.2 节从整体上描述 Linux 块设备驱动的结构，分析主要的数据结构、函数及其关系。

13.3 ~ 13.5 节分别讲解块设备驱动模块的加载与卸载、打开与释放和 ioctl() 函数。

13.6 节非常重要，讲述了块设备 I/O 操作所依赖的请求、bio 处理方法。

13.7 节在 13.1 ~ 13.6 节所讲解内容的基础上，总结在 Linux 下块设备的读写流程，实现了块设备驱动的一个具体实例，即 vmem_disk 驱动。

13.8 节简单地讲解了 Linux 的 MMC 子系统以及它与块设备的关系。

13.1 块设备的 I/O 操作特点

字符设备与块设备 I/O 操作的不同如下。

1) 块设备只能以块为单位接收输入和返回输出，而字符设备则以字节为单位。大多数设备是字符设备，因为它们不需要缓冲而且不以固定块大小进行操作。

2) 块设备对于 I/O 请求有对应的缓冲区，因此它们可以选择以什么顺序进行响应，字符设备无须缓冲且被直接读写。对于存储设备而言，调整读写的顺序作用巨大，因为在读写连续的扇区的存储速度比分离的扇区更快。

3) 字符设备只能被顺序读写，而块设备可以随机访问。

虽然块设备可随机访问，但是对于磁盘这类机械设备而言，顺序地组织块设备的访问可以提高性能，如图 13.1 所示，对扇区 1、10、3、2 的请求被调整为对扇区 1、2、3、10 的请求。

在 Linux 中，我们通常通过磁盘文件系统 EXT4、UBIFS 等访问磁盘，但是磁盘也有一种原始设备的访问方式，如直接访问 /dev/sdb1 等。所有的 EXT4、UBIFS、原始块设备又都

工作于VFS之下，而EXT4、UBIFS、原始块设备之下又包含块I/O调度层以进行排序和合并（见图13.2）。



图 13.1 调整块设备 I/O 操作的顺序

图 13.2 Linux 块设备子系统

I/O调度层的基本目的是将请求按照它们对应在块设备上的扇区号进行排列，以减少磁头的移动，提高效率。

13.2 Linux 块设备驱动结构

13.2.1 block_device_operations 结构体

在块设备驱动中，有一个类似于字符设备驱动中 file_operations 结构体的 block_device_operations 结构体，它是对块设备操作的集合，定义如代码清单 13.1 所示。

代码清单 13.1 block_device_operations 结构体

```

1 struct block_device_operations {
2     int (*open) (struct block_device *, fmode_t);
3     void (*release) (struct gendisk *, fmode_t);
4     int (*rw_page) (struct block_device *, sector_t, struct page *, int rw);
5     int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
6     int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
7     int (*direct_access) (struct block_device *, sector_t,
8                           void **, unsigned long *);
9     unsigned int (*check_events) (struct gendisk *disk,
10                                unsigned int clearing);
11    /* ->media_changed() is DEPRECATED, use ->check_events() instead */
12    int (*media_changed) (struct gendisk *);
13    void (*unlock_native_capacity) (struct gendisk *);
14    int (*revalidate_disk) (struct gendisk *);
15    int (*getgeo) (struct block_device *, struct hd_geometry *);
16    /* this callback is with swap_lock and sometimes page table lock held */
17    void (*swap_slot_free_notify) (struct block_device *, unsigned long);
18    struct module *owner;
19 };

```

下面对其主要成员函数进行分析。

1. 打开和释放

```
int (*open) (struct block_device *, fmode_t);
void (*release) (struct gendisk *, fmode_t);
```

与字符设备驱动类似，当设备被打开和关闭时将调用它们。

2. I/O 控制

```
int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
```

上述函数是 ioctl() 系统调用的实现，块设备包含大量的标准请求，这些标准请求由 Linux 通用块设备层处理，因此大部分块设备驱动的 ioctl() 函数相当短。当一个 64 位系统内的 32 位进程调用 ioctl() 的时候，调用的是 compat_ioctl()。

3. 介质改变

```
int (*media_changed) (struct gendisk *gd);
```

被内核调用以检查驱动器中的介质是否已经改变，如果是，则返回一个非 0 值，否则返回 0。这个函数仅适用于支持可移动介质的驱动器，通常需要在驱动中增加一个表示介质状态是否改变的标志变量，非可移动设备的驱动不需要实现这个方法。

```
unsigned int (*check_events) (struct gendisk *disk,
                             unsigned int clearing);
```

media_changed() 这个回调函数目前已经过时了，已被 check_events() 替代。Tejun Heo<tj@kernel.org> 在内核提交了一个补丁，完成了“implement in-kernel gendisk events handling”的工作，这个补丁对应的 commit ID 是 77ea887e。老的 Linux 在用户空间里轮询可移动磁盘介质是否存在，而新的内核则在内核空间里轮询。check_events() 函数检查有没有挂起的事件，如果有 DISK_EVENT_MEDIA_CHANGE 和 DISK_EVENT_EJECT_REQUEST 事件，就返回。

4. 使介质有效

```
int (*revalidate_disk) (struct gendisk *gd);
```

revalidate_disk() 函数被调用来响应一个介质改变，它给驱动一个机会来进行必要的工作以使新介质准备好。

5. 获得驱动器信息

```
int (*getgeo)(struct block_device *, struct hd_geometry *);
```

该函数根据驱动器的几何信息填充一个 hd_geometry 结构体，hd_geometry 结构体包含磁头、扇区、柱面等信息，其定义于 include/linux/hdreg.h 头文件中。

6. 模块指针

```
struct module *owner;
```

一个指向拥有这个结构体的模块的指针，它通常被初始化为 THIS_MODULE。

13.2.2 gendisk 结构体

在 Linux 内核中，使用 gendisk(通用磁盘) 结构体来表示一个独立的磁盘设备（或分区），这个结构体的定义如代码清单 13.2 所示。

代码清单 13.2 gendisk 结构体

```

1 struct gendisk {
2         /* major, first_minor and minors are input parameters only,
3          * don't use directly. Use disk_devt() and disk_max_parts().
4          */
5         int major;           /* major number of driver */
6         int first_minor;
7         int minors;          /* maximum number of minors, -1 for
8          * disks that can't be partitioned. */
9
10        char disk_name[DISK_NAME_LEN];    /* name of major driver */
11        char *(*devnode)(struct gendisk *gd, umode_t *mode);
12
13        unsigned int events;           /* supported events */
14        unsigned int async_events;     /* async events, subset of all */
15
16        /* Array of pointers to partitions indexed by partno.
17         * Protected with matching bdev lock but stat and other
18         * non-critical accesses use RCU. Always access through
19         * helpers.
20         */
21        struct disk_part_tbl __rcu *part_tbl;
22        struct hd_struct part0;
23
24        const struct block_device_operations *fops;
25        struct request_queue *queue;
26        void *private_data;
27
28        int flags;
29        struct device *driverfs_dev; //FIXME: remove
30        struct kobject *slave_dir;
31
32        struct timer_rand_state *random;
33        atomic_t sync_io;           /* RAID */
34        struct disk_events *ev;
35 #ifdef CONFIG_BLK_DEV_INTEGRITY
36        struct blk_integrity *integrity;
37 #endif
38        int node_id;
39 };

```

major、first_minor 和 minors 共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而次设备号则不同。fops 为 block_device_operations，即上节描述的块设备操作集合。queue 是内核用来管理这个设备的 I/O 请求队列的指针。private_data 可用于指向磁盘的任何私有数据，用法与字符设备驱动 file 结构体的 private_data 类似。hd_struct 成员表示一个分区，而 disk_part_tbl 成员用于容纳分区表，part0 和 part_tbl 两者的关系在于：

```
disk->part_tbl->part[0] = &disk->part0;
```

Linux 内核提供了一组函数来操作 gendisk，如下所示。

1. 分配 gendisk

gendisk 结构体是一个动态分配的结构体，它需要特别的内核操作来初始化，驱动不能自己分配这个结构体，而应该使用下列函数来分配 gendisk：

```
struct gendisk *alloc_disk(int minors);
```

minors 参数是这个磁盘使用的次设备号的数量，一般也就是磁盘分区的数量，此后 minors 不能被修改。

2. 增加 gendisk

gendisk 结构体被分配之后，系统还不能使用这个磁盘，需要调用如下函数来注册这个磁盘设备。

```
void add_disk(struct gendisk *disk);
```

特别要注意的是：对 add_disk() 的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3. 释放 gendisk

当不再需要磁盘时，应当使用如下函数释放 gendisk。

```
void del_gendisk(struct gendisk *gp);
```

4. gendisk 引用计数

通过 get_disk() 和 put_disk() 函数可操作 gendisk 的引用计数，这个工作一般不需要驱动亲自做。这两个函数的原型分别为：

```
struct kobject *get_disk(struct gendisk *disk);
void put_disk(struct gendisk *disk);
```

前者最终会调用 “kobject_get(&disk_to_dev(disk)->kobj);”，而后者则会调用 “kobject_put(&disk_to_dev(disk)->kobj);”。

13.2.3 bio、request 和 request_queue

通常一个 bio 对应上层传递给块层的 I/O 请求。每个 bio 结构体实例及其包含的 bvec_

iter、bio_vec 结构体实例描述了该 I/O 请求的开始扇区、数据方向（读还是写）、数据放入的页，其定义如代码清单 13.3 所示。

代码清单 13.3 bio 结构体

```

1 struct bvec_iter {
2     sector_t          bi_sector; /* device address in 512 byte
3                                sectors */
4     unsigned int       bi_size;   /* residual I/O count */
5
6     unsigned int       bi_idx;    /* current index into bvl_vec */
7
8     unsigned int       bi_bvec_done; /* number of bytes completed
9                                in current bvec */
10 };
11
12 /*
13  * main unit of I/O for the block layer and lower layers (ie drivers and
14  * stacking drivers)
15  */
16 struct bio {
17     struct bio         *bi_next;  /* request queue link */
18     struct block_device *bi_bdev;
19     unsigned long       bi_flags;   /* status, command, etc */
20     unsigned long       bi_rw;      /* bottom bits READ/WRITE,
21                                * top bits priority
22                                */
23
24     struct bvec_iter    bi_iter;
25
26     /* Number of segments in this BIO after
27      * physical address coalescing is performed.
28      */
29     unsigned int        bi_phys_segments;
30
31     ...
32
33     struct bio_vec      *bi_io_vec; /* the actual vec list */
34
35     struct bio_set      *bi_pool;
36
37     /*
38      * We can inline a number ofvecs at the end of the bio, to avoid
39      * double allocations for a small number of bio_vecs. This member
40      * MUST obviously be kept at the very end of the bio.
41      */
42     struct bio_vec       bi_inline_vecs[0];
43 };

```

与 bio 对应的数据每次存放的内存不一定是连续的，bio_vec 结构体用来描述与这个 bio

请求对应的所有的内存，它可能不总是在一个页面里面，因此需要一个向量，定义如代码清单 13.4 所示。向量中的每个元素实际是一个 [page, offset, len]，我们一般也称它为一个片段。

代码清单 13.4 bio_vec 结构体

```

1 struct bio_vec {
2     struct page      *bv_page;
3     unsigned int     bv_len;
4     unsigned int     bv_offset;
5 };

```

I/O 调度算法可将连续的 bio 合并成一个请求。请求是 bio 经由 I/O 调度进行调整后的结果，这是请求和 bio 的区别。因此，一个 request 可以包含多个 bio。当 bio 被提交给 I/O 调度器时，I/O 调度器可能会将这个 bio 插入现存的请求中，也可能生成新的请求。

每个块设备或者块设备的分区都对应有自身的 request_queue，从 I/O 调度器合并和排序出来的请求会被分发（Dispatch）到设备级的 request_queue。图 13.3 描述了 request_queue、request、bio、bio_vec 之间的关系。

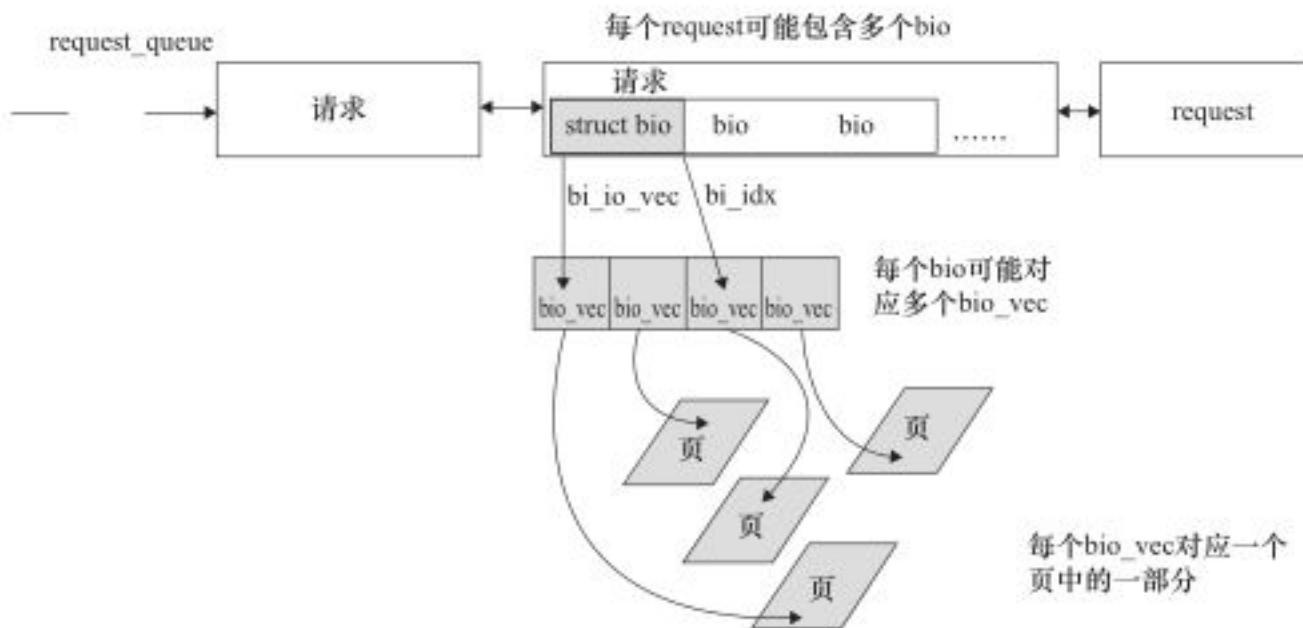


图 13.3 request_queue、request、bio 和 bio_vec

下面看一下驱动中涉及的处理 bio、request 和 request_queue 的主要 API。

(1) 初始化请求队列

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
```

该函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的初始化过程中调用。

(2) 清除请求队列

```
void blk_cleanup_queue(request_queue_t * q);
```

这个函数完成将请求队列返回给系统的任务，一般在块设备驱动卸载过程中调用。

(3) 分配请求队列

```
request_queue_t *blk_alloc_queue(int gfp_mask);
```

对于 RAMDISK 这种完全随机访问的非机械设备，并不需要进行复杂的 I/O 调度，这个时候，可以直接“踢开”I/O 调度器，使用如下函数来绑定请求队列和“制造请求”函数 (make_request_fn)。

```
void blk_queue_make_request(request_queue_t * q, make_request_fn * mfn);
```

blk_alloc_queue() 和 blk_queue_make_request() 结合起来使用的逻辑一般是：

```
xxx_queue = blk_alloc_queue(GFP_KERNEL);
blk_queue_make_request(xxx_queue, xxx_make_request);
```

(4) 提取请求

```
struct request * blk_peek_request(struct request_queue *q);
```

上述函数用于返回下一个要处理的请求（由 I/O 调度器决定），如果没有请求则返回 NULL。它不会清除请求，而是仍然将这个请求保留在队列上。原先的老的函数 elv_next_request() 已经不再存在。

(5) 启动请求

```
void blk_start_request(struct request *req);
```

从请求队列中移除请求。原先的老的 API blkdev_dequeue_request() 会在 blk_start_request() 内部被调用。

我们可以考虑使用 blk_fetch_request() 函数，它同时做完了 blk_peek_request() 和 blk_start_request() 的工作，如代码清单 13.5 所示。

代码清单 13.5 blk_fetch_request() 函数

```

1 struct request *blk_fetch_request(struct request_queue *q)
2 {
3     struct request *rq;
4
5     rq = blk_peek_request(q);
6     if (rq)
7         blk_start_request(rq);
8     return rq;
9 }
```

(6) 遍历 bio 和片段

```
#define __rq_for_each_bio(_bio, rq)
    if ((rq->bio))
        for (_bio = (rq)->bio; _bio; _bio = _bio->bi_next)
```

`__rq_for_each_bio()` 遍历一个请求的所有 bio。

```
#define __bio_for_each_segment(bvl, bio, iter, start)
    for (iter = (start);
        (iter).bi_size &&
        ((bvl = bio_iter iov((bio), (iter))), 1);
        bio_advance_iter((bio), &(iter), (bvl).bv_len))
```

```
#define bio_for_each_segment(bvl, bio, iter)
    __bio_for_each_segment(bvl, bio, iter, (bio)->bi_iter)
```

`bio_for_each_segment()` 遍历一个 bio 的所有 bio_vec。

```
#define rq_for_each_segment(bvl, _rq, _iter)
    __rq_for_each_bio(_iter.bio, _rq)
    bio_for_each_segment(bvl, _iter.bio, _iter.iter)
```

`rq_for_each_segment()` 迭代遍历一个请求所有 bio 中的所有 segment。

(7) 报告完成

```
void __blk_end_request_all(struct request *rq, int error);
void blk_end_request_all(struct request *rq, int error);
```

上述两个函数用于报告请求是否完成, error 为 0 表示成功, 小于 0 表示失败。`__blk_end_request_all()` 需要在持有队列锁的场景下调用。

类似的函数还有 `blk_end_request_cur()`、`blk_end_request_err()`、`__blk_end_request()`、`__blk_end_request_all()`、`__blk_end_request_cur()` 以及 `__blk_end_request_err()`。其中 `xxx_end_request_cur()` 只是表明完成了 request 中当前的那个 chunk, 也就是完成了当前的 `bio_cur_bytes(rq->bio)` 的传输。

若我们用 `blk_queue_make_request()` 绕开 I/O 调度, 但是在 bio 处理完成后应该使用 `bio_endio()` 函数通知处理结束:

```
void bio_endio(struct bio *bio, int error);
```

如果是 I/O 操作故障, 可以调用快捷函数 `bio_io_error()`, 它定义为:

```
#define bio_io_error(bio) bio_endio((bio), -EIO)
```

13.2.4 I/O 调度器

Linux 2.6 以后的内核包含 4 个 I/O 调度器, 它们分别是 Noop I/O 调度器、Anticipatory I/O 调度器、Deadline I/O 调度器和 CFQ I/O 调度器。

O 调度器、Deadline I/O 调度器与 CFQ I/O 调度器。其中，Anticipatory I/O 调度器算法已经在 2010 年从内核中去掉了。

Noop I/O 调度器是一个简化的调度程序，该算法实现了一个简单 FIFO 队列，它只进行最基本的合并，比较适合基于 Flash 的存储器。

Anticipatory I/O 调度器算法推迟 I/O 请求，以期能对它们进行排序，获得最高的效率。在每次处理完读请求之后，不是立即返回，而是等待几个微秒。在这段时间内，任何来自临近区域的请求都被立即执行。超时以后，继续原来的处理。

Deadline I/O 调度器是针对 Anticipatory I/O 调度器的缺点进行改善而得来的，它试图把每次请求的延迟降至最低，该算法重排了请求的顺序来提高性能。它使用轮询的调度器，简洁小巧，提供了最小的读取延迟和尚佳的吞吐量，特别适合于读取较多的环境（比如数据库）。

CFQ I/O 调度器为系统内的所有任务分配均匀的 I/O 带宽，提供一个公平的工作环境，在多媒体应用中，能保证音、视频及时从磁盘中读取数据。

内核 4.0-rc1 block 目录中的 noop-iosched.c、deadline-iosched.c 和 cfq-iosched.c 文件分别实现了 IOSCHED_NOOP、IOSCHED_DEADLINE 和 IOSCHED_CFQ 调度算法。as-iosched.c 这个文件目前已经不再存在。当前情况下，默认的调度器是 CFQ。

可以通过给内核添加启动参数，选择所使用的 I/O 调度算法，如：

```
kernel elevator=deadline
```

也可以通过类似如下的命令，改变一个设备的调度器：

```
echo SCHEDULER > /sys/block/DEVICE/queue/scheduler
```

13.3 Linux 块设备驱动的初始化

在块设备的注册和初始化阶段，与字符设备驱动类似，块设备驱动要注册它们自己到内核，申请设备号，完成这个任务的函数是 register_blkdev()，其原型为：

```
int register_blkdev(unsigned int major, const char *name);
```

major 参数是块设备要使用的主设备号，name 为设备名，它会显示在 /proc/devices 中。如果 major 为 0，内核会自动分配一个新的主设备号，register_blkdev() 函数的返回值就是这个主设备号。如果 register_blkdev() 返回一个负值，表明发生了一个错误。

与 register_blkdev() 对应的注销函数是 unregister_blkdev()，其原型为：

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，传递给 register_blkdev() 的参数必须与传递给 register_blkdev() 的参数匹配，否则这个函数返回 -EINVAL。

除此之外，在块设备驱动初始化过程中，通常需要完成分配、初始化请求队列，绑定请求队列和请求处理函数的工作，并且可能会分配、初始化 gendisk，给 gendisk 的 major、fops、queue 等成员赋值，最后添加 gendisk。

代码清单 13.6 演示了一个典型的块设备驱动的初始化过程，其中包含了 register_blkdev()、blk_init_queue() 和 add_disk() 的工作。

代码清单 13.6 块设备驱动的初始化

```

1 static int xxx_init(void)
2 {
3     /* 块设备驱动注册 */
4     if (register_blkdev(XXX_MAJOR, "xxx")) {
5         err = -EIO;
6         goto out;
7     }
8
9     /* 请求队列初始化 */
10    xxx_queue = blk_init_queue(xxx_request, xxx_lock);
11    if (!xxx_queue)
12        goto out_queue;
13    blk_queue_max_hw_sectors(xxx_queue, 255);
14    blk_queue_logical_block_size(xxx_queue, 512);
15
16    /* gendisk 初始化 */
17    xxx_disks->major = XXX_MAJOR;
18    xxx_disks->first_minor = 0;
19    xxx_disks->fops = &xxx_op;
20    xxx_disks->queue = xxx_queue;
21    sprintf(xxx_disks->disk_name, "xxx%d", i);
22    set_capacity(xxx_disks, xxx_size *2);
23    add_disk(xxx_disks); /* 添加gendisk */
24
25    return 0;
26    out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
27    out: put_disk(xxx_disks);
28    blk_cleanup_queue(xxx_queue);
29
30    return -ENOMEM;
31 }
```

上述代码第 13 行的 blk_queue_max_hw_sectors() 用于通知通用块层和 I/O 调度器该请求队列支持的每个请求中能够包含的最大扇区数，第 14 行 blk_queue_logical_block_size() 则用于告知该请求队列的逻辑块大小。

在块设备驱动的卸载过程中完成与模块加载函数相反的工作。

- 1) 清除请求队列，使用 blk_cleanup_queue()。
- 2) 删除对 gendisk 的引用，使用 put_disk()。
- 3) 删除对块设备的引用，注销块设备驱动，使用 unregister_blkdev()。

13.4 块设备的打开与释放

块设备驱动的 open() 函数和其字符设备驱动的对等体不太相似，前者不以相关的 inode 和 file 结构体指针作为参数（因为 file 和 inode 概念位于文件系统层中）。在 open() 中我们可以通过 block_device 参数 bdev 获取 private_data，在 release() 函数中则通过 gendisk 参数 disk 获取，如代码清单 13.7 所示。

代码清单 13.7 在块设备的 open()/release() 函数中获取 private_data

```

1 static int xxx_open(struct block_device *bdev, fmode_t mode)
2 {
3     struct xxx_dev *dev = bdev->bd_disk->private_data;
4     ...
5     return 0;
6 }
7
8 static void xxx_release(struct gendisk *disk, fmode_t mode)
9 {
10    struct xxx_dev *dev = disk->private_data;
11    ...
12 }
```

13.5 块设备驱动的 ioctl 函数

与字符设备驱动一样，块设备可以包含一个 ioctl() 函数以提供对设备的 I/O 控制能力。实际上，高层的块设备层代码处理了绝大多数 I/O 控制，如 BLKFLSBUF、BLKROSET、BLKDISCARD、HDIO_GETGEO、BLKROGET 和 BLKSECTGET 等，因此，在具体的块设备驱动中通常只需要实现与设备相关的特定 ioctl 命令。例如，源代码文件为 drivers/block/floppy.c 实现了与软驱相关的命令（如 FDEJECT、FDSETPRM、FDFMTTRK 等）。

Linux MMC 子系统支持一个 IOCTL 命令 MMC_IOC_CMD，drivers/mmc/card/block.c 实现了这个命令的处理，如代码清单 13.8 所示。

代码清单 13.8 Linux MMC 块设备的 ioctl() 函数

```

1 static int mmc_blk_ioctl(struct block_device *bdev, fmode_t mode,
2                           unsigned int cmd, unsigned long arg)
3 {
4     int ret = -EINVAL;
5     if (cmd == MMC_IOC_CMD)
6         ret = mmc_blk_ioctl_cmd(bdev, (struct mmc_ioc_cmd __user *)arg);
7     return ret;
8 }
```

13.6 块设备驱动的 I/O 请求处理

13.6.1 使用请求队列

块设备驱动在使用请求队列的场景下，会用 `blk_init_queue()` 初始化 `request_queue`，而该函数的第一个参数就是请求处理函数的指针。`request_queue` 会作为参数传递给我们在调用 `blk_init_queue()` 时指定的请求处理函数，块设备驱动请求处理函数的原型为：

```
static void xxx_req(struct request_queue *q)
```

这个函数不能由驱动自己调用，只有当内核认为是时候让驱动处理对设备的读写等操作时，它才调用这个函数。该函数的主要工作就是发起与 `request` 对应的块设备 I/O 动作（但是具体的 I/O 工作不一定要在该函数内同步完成）。代码清单 13.9 给出了一个简单的请求处理函数的例子，它来源于 `drivers/memstick/core/ms_block.c`。

代码清单 13.9 块设备驱动请求函数例程

```

1 static void msb_submit_req(struct request_queue *q)
2 {
3     struct memstick_dev *card = q->queuedata;
4     struct msb_data *msb = memstick_get_drvdata(card);
5     struct request *req = NULL;
6
7     dbg_verbose("Submit request");
8
9     if (msb->card_dead) {
10         dbg("Refusing requests on removed card");
11
12         WARN_ON(!msb->io_queue_stopped);
13
14         while ((req = blk_fetch_request(q)) != NULL)
15             __blk_end_request_all(req, -ENODEV);
16         return;
17     }
18
19     if (msb->req)
20         return;
21
22     if (!msb->io_queue_stopped)
23         queue_work(msb->io_queue, &msb->io_work);
24 }
```

上述代码第 14 行使用 `blk_fetch_request()` 获得队列中第一个未完成的请求，由于 `msb->card_dead` 成立，实际上我们处理不了该请求，所以就直接通过 `__blk_end_request_all(req, -ENODEV)` 返回错误了。

正常的情况下，通过 `queue_work(msb->io_queue, &msb->io_work)` 启动工作队列执行 `msb_io_work(struct work_struct *work)` 这个函数，它的原型如代码清单 13.10 所示。

代码清单 13.10 msb_io_work() 完成请求处理

```

1 static void msb_io_work(struct work_struct *work)
2 {
3     struct msb_data *msb = container_of(work, struct msb_data, io_work);
4     int page, error, len;
5     sector_t lba;
6     unsigned long flags;
7     struct scatterlist *sg = msb->prealloc_sg;
8
9     dbg_verbose("IO: work started");
10
11    while (1) {
12        spin_lock_irqsave(&msb->q_lock, flags);
13
14        if (msb->need_flush_cache) {
15            msb->need_flush_cache = false;
16            spin_unlock_irqrestore(&msb->q_lock, flags);
17            msb_cache_flush(msb);
18            continue;
19        }
20
21        if (!msb->req) {
22            msb->req = blk_fetch_request(msb->queue);
23            if (!msb->req) {
24                dbg_verbose("IO: no more requests exiting");
25                spin_unlock_irqrestore(&msb->q_lock, flags);
26                return;
27            }
28        }
29
30        spin_unlock_irqrestore(&msb->q_lock, flags);
31
32        /* If card was removed meanwhile */
33        if (!msb->req)
34            return;
35
36        /* process the request */
37        dbg_verbose("IO: processing new request");
38        blk_rq_map_sg(msb->queue, msb->req, sg);
39
40        lba = blk_rq_pos(msb->req);
41
42        sector_div(lba, msb->page_size / 512);
43        page = do_div(lba, msb->pages_in_block);
44
45        if (rq_data_dir(msb->req) == READ)
46            error = msb_do_read_request(msb, lba, page, sg,
47                                         blk_rq_bytes(msb->req), &len);
48        else
49            error = msb_do_write_request(msb, lba, page, sg,

```

```

50             blk_rq_bytes(msb->req), &len);
51
52         spin_lock_irqsave(&msb->q_lock, flags);
53
54         if (len)
55             if (!__blk_end_request(msb->req, 0, len))
56                 msb->req = NULL;
57
58         if (error && msb->req) {
59             dbg_verbose("IO: ending one sector of the request with error");
60             if (!__blk_end_request(msb->req, error, msb->page_size))
61                 msb->req = NULL;
62         }
63
64         if (msb->req)
65             dbg_verbose("IO: request still pending");
66
67         spin_unlock_irqrestore(&msb->q_lock, flags);
68     }
69 }
```

在读写无错的情况下，第 55 行调用的 __blk_end_request(msb->req, 0, len) 实际上告诉了上层该请求处理完成。如果读写有错，则调用 __blk_end_request(msb->req, error, msb->page_size)，把出错原因作为第 2 个参数传入上层。

第 38 行调用的 blk_rq_map_sg() 函数实现于 block/blk-merge.c 文件。代码清单 13.11 列出了该函数的实现中比较精华的部分，它通过 rq_for_each_bio()、bio_for_each_segment() 来遍历所有的 bio，以及所有的片段，将所有与某请求相关的页组成一个 scatter/gather 的列表。

代码清单 13.11 blk_rq_map_sg() 函数

```

1 int blk_rq_map_sg(struct request_queue *q, struct request *rq,
2                     struct scatterlist *sglist)
3 {
4     struct scatterlist *sg = NULL;
5     int nsegs = 0;
6
7     if (rq->bio)
8         nsegs = __blk_bios_map_sg(q, rq->bio, sglist, &sg);
9     ...
10 }
11
12 static int __blk_bios_map_sg(struct request_queue *q, struct bio *bio,
13                             struct scatterlist *sglist,
14                             struct scatterlist **sg)
15 {
16     struct bio_vec bvec, bvprv = { NULL };
17     struct bvec_iter iter;
18     int nsegs, cluster;
```

```

19
20     nsegs = 0;
21     cluster = blk_queue_cluster(q);
22     ...
23     for_each_bio(bio)
24         bio_for_each_segment(bvec, bio, iter)
25             __blk_segment_map_sg(q, &bvec, sglist, &bvprv, sg,
26                                     &nsegs, &cluster);
27
28     return nsegs;
29 }
30
31 static inline void
32 __blk_segment_map_sg(struct request_queue *q, struct bio_vec *bvec,
33                      struct scatterlist *sglist, struct bio_vec *bvprv,
34                      struct scatterlist **sg, int *nsegs, int *cluster)
35 {
36
37     int nbytes = bvec->bv_len;
38
39     if (*sg && *cluster) {
40         if ((*sg)->length + nbytes > queue_max_segment_size(q))
41             goto new_segment;
42
43         if (!BIOVEC_PHYS_MERGEABLE(bvprv, bvec))
44             goto new_segment;
45         if (!BIOVEC_SEG_BOUNDARY(q, bvprv, bvec))
46             goto new_segment;
47
48         (*sg)->length += nbytes;
49     } else {
50     new_segment:
51         if (!*sg)
52             *sg = sglist;
53         else {
54             /*
55              * If the driver previously mapped a shorter
56              * list, we could see a termination bit
57              * prematurely unless it fully inits the sg
58              * table on each mapping. We KNOW that there
59              * must be more entries here or the driver
60              * would be buggy, so force clear the
61              * termination bit to avoid doing a full
62              * sg_init_table() in drivers for each command.
63              */
64             sg_unmark_end(*sg);
65             *sg = sg_next(*sg);
66         }
67     }

```

```

68             sg_set_page(*sg, bvec->bv_page, nbytes, bvec->bv_offset);
69             (*nsegs)++;
70         }
71     *bvprv = *bvec;

```

一般情况下，若外设支持 scatter/gather 模式的 DMA 操作，紧接着，它就会执行 pci_map_sg() 或者 dma_map_sg() 来进行上述 scatter/gather 列表的 DMA 映射了，之后进行硬件的访问。

13.6.2 不使用请求队列

使用请求队列对于一个机械磁盘设备而言的确有助于提高系统的性能，但是对于 RAMDISK、ZRAM (Compressed RAM Block Device) 等完全可真正随机访问的设备而言，无法从高级的请求队列逻辑中获益。对于这些设备，块层支持“无队列”的操作模式，为使用这个模式，驱动必须提供一个“制造请求”函数，而不是一个请求处理函数，“制造请求”函数的原型为：

```
static void xxx_make_request(struct request_queue *queue, struct bio *bio);
```

块设备驱动初始化的时候不再调用 blk_init_queue()，而是调用 blk_alloc_queue() 和 blk_queue_make_request()，xxx_make_request 则会成为 blk_queue_make_request() 的第 2 个参数。

xxx_make_request() 函数的第一个参数仍然是“请求队列”，但是这个“请求队列”实际不包含任何请求，因为块层没有必要将 bio 调整为请求。因此，“制造请求”函数的主要参数是 bio 结构体。代码清单 13.12 所示为一个“制造请求”函数的例子，它取材于 drivers/block/zram/zram_drv.c。

代码清单 13.12 “制造请求”函数例程

```

1 static void zram_make_request(struct request_queue *queue, struct bio *bio)
2 {
3     ...
4     __zram_make_request(zram, bio);
5     ...
6 }
7
8 static void __zram_make_request(struct zram *zram, struct bio *bio)
9 {
10    int offset;
11    u32 index;
12    struct bio_vec bvec;
13    struct bvec_iter iter;
14
15    index = bio->bi_iter.bi_sector >> SECTORS_PER_PAGE_SHIFT;
16    offset = (bio->bi_iter.bi_sector &
17               (SECTORS_PER_PAGE - 1)) << SECTOR_SHIFT;
18

```

```

19         if (unlikely(bio->bi_rw & REQ_DISCARD)) {
20             zram_bio_discard(zram, index, offset, bio);
21             bio_endio(bio, 0);
22             return;
23         }
24
25         bio_for_each_segment(bvec, bio, iter) {
26             int max_transfer_size = PAGE_SIZE - offset;
27
28             if (bvec.bv_len > max_transfer_size) {
29                 /*
30                  * zram_bvec_rw() can only make operation on a single
31                  * zram page. Split the bio vector.
32                  */
33                 struct bio_vec bv;
34
35                 bv.bv_page = bvec.bv_page;
36                 bv.bv_len = max_transfer_size;
37                 bv.bv_offset = bvec.bv_offset;
38
39                 if (zram_bvec_rw(zram, &bv, index, offset, bio) < 0)
40                     goto out;
41
42                 bv.bv_len = bvec.bv_len - max_transfer_size;
43                 bv.bv_offset += max_transfer_size;
44                 if (zram_bvec_rw(zram, &bv, index + 1, 0, bio) < 0)
45                     goto out;
46             } else
47                 if (zram_bvec_rw(zram, &bvec, index, offset, bio) < 0)
48                     goto out;
49
50             update_position(&index, &offset, &bvec);
51         }
52
53         set_bit(BIO_UPTODATE, &bio->bi_flags);
54         bio_endio(bio, 0);
55         return;
56
57     out:
58         bio_io_error(bio);
59 }

```

上述代码通过 `bio_for_each_segment()` 迭代 `bio` 中的每个 `segement`，最终调用 `zram_bvec_rw()` 完成内存的压缩、解压、读取和写入。

ZRAM 是 Linux 的一种内存优化技术，它划定一片内存区域作为 SWAP 的交换分区，但是它本身具备自动压缩功能，从而可以达到辅助 Linux 匿名页的交换效果，变相“增大”了内存。

13.7 实例：vmem_disk 驱动

13.7.1 vmem_disk 的硬件原理

vmem_disk 是一种模拟磁盘，其数据实际上存储在 RAM 中。它使用通过 `vmalloc()` 分配出来的内存空间来模拟出一个磁盘，以块设备的方式来访问这片内存。该驱动是对字符设备驱动章节中 `globalmem` 驱动的块方式改造。

加载 `vmem_disk.ko` 后，在使用默认模块参数的情况下，系统会增加 4 个块设备节点：

```
# ls -l /dev/vmem_disk*
brw-rw---- 1 root disk 252, 0 2月 25 14:00 /dev/vmem_diska
brw-rw---- 1 root disk 252, 16 2月 25 14:00 /dev/vmem_diskb
brw-rw---- 1 root disk 252, 32 2月 25 14:00 /dev/vmem_diskc
brw-rw---- 1 root disk 252, 48 2月 25 14:00 /dev/vmem_diskd
```

其中，`mkfs.ext2/dev/vmem_diska` 命令的执行会回馈如下信息：

```
$ sudo mkfs.ext2 /dev/vmem_diska
mke2fs 1.42.9 (4-Feb-2014)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
64 inodes, 512 blocks
25 blocks (4.88%) reserved for the super user
First data block=1
Maximum filesystem blocks=524288
1 block group
8192 blocks per group, 8192 fragments per group
64 inodes per group

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

它将 `/dev/vmem_diska` 格式化为 EXT2 文件系统。之后我们可以 `mount` 这个分区并在其中进行文件读写。

13.7.2 vmem_disk 驱动模块的加载与卸载

`vmem_disk` 驱动的模块加载函数完成的工作与 13.3 节给出的模板完全一致，它支持“制造请求”（对应于代码清单 13.9）、请求队列（对应于代码清单 13.10）两种模式（请注意在请求队列方面又支持简、繁两种模式），使用模块参数 `request_mode` 进行区分。代码清单 13.13 给出了 `vmem_disk` 设备驱动的模块加载与卸载函数。

代码清单 13.13 vmem_disk 设备驱动的模块加载与卸载函数

```

1 static void setup_device(struct vmem_disk_dev *dev, int which)
2 {
3     memset (dev, 0, sizeof (struct vmem_disk_dev));
4     dev->size = NSECTORS*HARDSECT_SIZE;
5     dev->data = vmalloc(dev->size);
6     if (dev->data == NULL) {
7         printk (KERN_NOTICE "vmalloc failure.\n");
8         return;
9     }
10    spin_lock_init(&dev->lock);
11
12    /*
13     * The I/O queue, depending on whether we are using our own
14     * make_request function or not.
15     */
16    switch (request_mode) {
17    case VMEMD_NOQUEUE:
18        dev->queue = blk_alloc_queue(GFP_KERNEL);
19        if (dev->queue == NULL)
20            goto out_vfree;
21        blk_queue_make_request(dev->queue, vmem_disk_make_request);
22        break;
23    default:
24        printk(KERN_NOTICE "Bad request mode %d, using simple\n", request_mode);
25    case VMEMD_QUEUE:
26        dev->queue = blk_init_queue(vmem_disk_request, &dev->lock);
27        if (dev->queue == NULL)
28            goto out_vfree;
29        break;
30    }
31    blk_queue_logical_block_size(dev->queue, HARDSECT_SIZE);
32    dev->queue->queuedata = dev;
33
34    dev->gd = alloc_disk(VMEM_DISK_MINORS);
35    if (!dev->gd) {
36        printk (KERN_NOTICE "alloc_disk failure\n");
37        goto out_vfree;
38    }
39    dev->gd->major = vmem_disk_major;
40    dev->gd->first_minor = which*VMEM_DISK_MINORS;
41    dev->gd->fops = &vmem_disk_ops;
42    dev->gd->queue = dev->queue;
43    dev->gd->private_data = dev;
44    snprintf (dev->gd->disk_name, 32, "vmem_disk%c", which + 'a');
45    set_capacity(dev->gd, NSECTORS*(HARDSECT_SIZE/KERNEL_SECTOR_SIZE));
46    add_disk(dev->gd);
47    return;
48
49 out_vfree:
50     if (dev->data)

```

```

51         vfree(dev->data);
52     }
53
54
55 static int __init vmem_disk_init(void)
56 {
57     int i;
58
59     vmem_disk_major = register_blkdev(vmem_disk_major, "vmem_disk");
60     if (vmem_disk_major <= 0) {
61         printk(KERN_WARNING "vmem_disk: unable to get major number\n");
62         return -EBUSY;
63     }
64
65     devices = kmalloc(NDEVICES*sizeof (struct vmem_disk_dev), GFP_KERNEL);
66     if (!devices)
67         goto out_unregister;
68     for (i = 0; i < NDEVICES; i++)
69         setup_device(devices + i, i);
70
71     return 0;
72
73 out_unregister:
74     unregister_blkdev(vmem_disk_major, "sbd");
75     return -ENOMEM;
76 }
77 module_init(vmem_disk_init);

```

注意上述代码的第 16 ~ 30 行，我们实际上支持两种 I/O 请求模式，一种是 make_request，另一种是 request_queue。make_request 的版本直接使用 vmem_disk_make_request() 来处理 bio，而 request_queue 的版本则使用 vmem_disk_request 来处理请求队列。

13.7.3 vmem_disk 设备驱动的 block_device_operations

vmem_disk 提供 block_device_operations 结构体中的 getgeo() 成员函数，代码清单 13.14 给出了 vmem_disk 设备驱动的 block_device_operations 结构体定义及其成员函数的实现。

代码清单 13.14 vmem_disk 设备驱动的 block_device_operations 结构体及成员函数

```

1 static int vmem_disk_getgeo(struct block_device *bdev, struct hd_geometry *geo)
2 {
3     long size;
4     struct vmem_disk_dev *dev = bdev->bd_disk->private_data;
5
6     size = dev->size*(HARDSECT_SIZE/KERNEL_SECTOR_SIZE);
7     geo->cylinders = (size & ~0x3f) >> 6;
8     geo->heads = 4;
9     geo->sectors = 16;
10    geo->start = 4;

```

```

11     return 0;
12 }
13
14
15 static struct block_device_operations vmem_disk_ops = {
16     .getgeo      = vmem_disk_getgeo,
17 };

```

13.7.4 vmem_disk 的 I/O 请求处理

在 vmem_disk 驱动中，通过模块参数 request_mode 的方式来支持 3 种不同的请求处理模式以加深读者对它们的理解，代码清单 13.15 列出了 vmem_disk 设备驱动的请求处理代码。

代码清单 13.15 vmem_disk 设备驱动的请求处理函数

```

1  /*
2   * Handle an I/O request.
3   */
4  static void vmem_disk_transfer(struct vmem_disk_dev *dev, unsigned long sector,
5      unsigned long nsect, char *buffer, int write)
6  {
7      unsigned long offset = sector*KERNEL_SECTOR_SIZE;
8      unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
9
10     if ((offset + nbytes) > dev->size) {
11         printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
12         return;
13     }
14     if (write)
15         memcpy(dev->data + offset, buffer, nbytes);
16     else
17         memcpy(buffer, dev->data + offset, nbytes);
18 }
19
20 /*
21  * Transfer a single BIO.
22  */
23 static int vmem_disk_xfer_bio(struct vmem_disk_dev *dev, struct bio *bio)
24 {
25     struct bio_vec bvec;
26     struct bvec_iter iter;
27     sector_t sector = bio->bi_iter.bi_sector;
28
29     bio_for_each_segment(bvec, bio, iter) {
30         char *buffer = __bio_kmap_atomic(bio, iter);
31         vmem_disk_transfer(dev, sector, bio_cur_bytes(bio) >> 9,
32             buffer, bio_data_dir(bio) == WRITE);
33         sector += bio_cur_bytes(bio) >> 9;
34         __bio_kunmap_atomic(buffer);
35     }

```

```

36     return 0;
37 }
38 /*
39 * The request_queue version.
40 */
41 static void vmem_disk_request(struct request_queue *q)
42 {
43     struct request *req;
44     struct bio *bio;
45
46     while ((req = blk_peek_request(q)) != NULL) {
47         struct vmem_disk_dev *dev = req->rq_disk->private_data;
48         if (req->cmd_type != REQ_TYPE_FS) {
49             printk (KERN_NOTICE "Skip non-fs request\n");
50             blk_start_request(req);
51             __blk_end_request_all(req, -EIO);
52             continue;
53         }
54
55         blk_start_request(req);
56         __rq_for_each_bio(bio, req)
57             vmem_disk_xfer_bio(dev, bio);
58             __blk_end_request_all(req, 0);
59     }
60 }
61 */
62
63 /*
64 * The direct make request version.
65 */
66 static void vmem_disk_make_request(struct request_queue *q, struct bio *bio)
67 {
68     struct vmem_disk_dev *dev = q->queuedata;
69     int status;
70
71     status = vmem_disk_xfer_bio(dev, bio);
72     bio_endio(bio, status);
73 }

```

第4行的 vmem_disk_transfer() 完成真实的硬件 I/O 操作（对于本例而言，就是一个 memcpy），第23行的 vmem_disk_xfer_bio() 函数调用它来完成一个与 bio 对应的硬件操作，在完成的过程中通过第29行的 bio_for_each_segment() 展开了该 bio 中的每个 segment。

vmem_disk_make_request() 直接调用 vmem_disk_xfer_bio() 来完成一个 bio 操作，而 vmem_disk_request() 则通过第47行的 blk_peek_request() 先从 request_queue 拿出一个请求，再通过第57行的 __rq_for_each_bio() 从该请求中取出一个 bio，之后调用 vmem_disk_xfer_bio() 来完成该 I/O 请求，图 13.4 描述了这个过程。

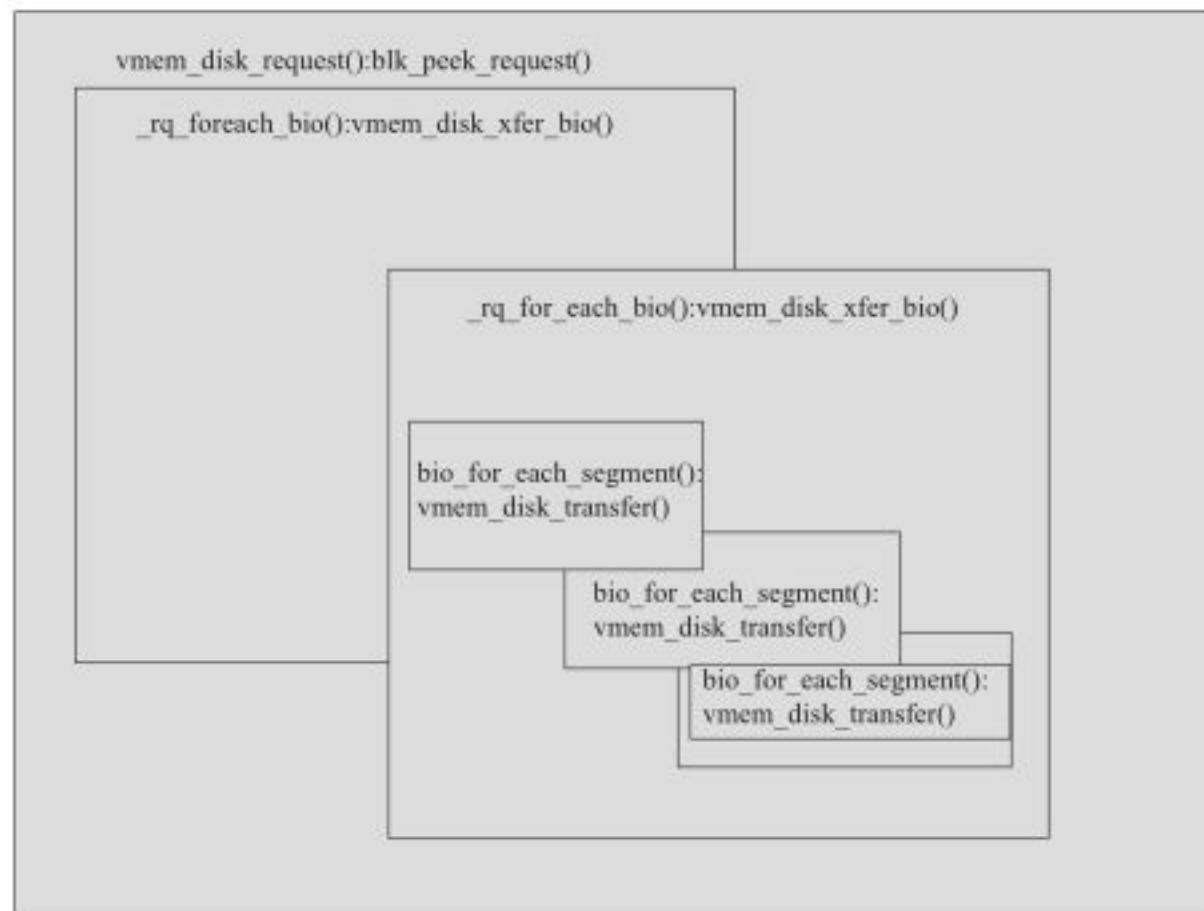


图 13.4 vmem_disk 的 I/O 处理过程

13.8 Linux MMC 子系统

Linux MMC/SD 存储卡是一种典型的块设备，它的实现位于 `drivers/mmc`。`drivers/mmc` 下又分为 `card`、`core` 和 `host` 这 3 个子目录。`card` 实际上跟 Linux 的块设备子系统对接，实现块设备驱动以及完成请求，但是具体的协议经过 `core` 层的接口，最终通过 `host` 完成传输，因此整个 MMC 子系统的框架结构如图 13.5 所示。另外，`card` 目录除了实现标准的 MMC/SD 存储卡以外，该目录还包含一些 SDIO 外设的卡驱动，如 `drivers/mmc/card/sdio_uart.c`。`core` 目录除了给 `card` 提供接口外，实际上也定义好了 `host` 驱动的框架。

`drivers/mmc/card/queue.c` 的 `mmc_init_queue()` 函数通过 `blk_init_queue(mmc_request_fn, lock)` 绑定了请求处理函数 `mmc_request_fn()`：

```

int mmc_init_queue(struct mmc_queue *mq, struct mmc_card *card,
                   spinlock_t *lock, const char *subname)
{
    ...
}
  
```

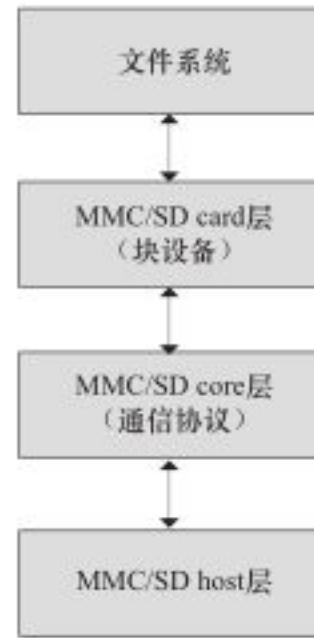


图 13.5 Linux MMC 子系统

```

    mq->queue = blk_init_queue(mmc_request_fn, lock);
    ...
}

```

而 mmc_request_fn() 函数会唤醒与 MMC 对应的内核线程来处理请求，与该线程对应的处理函数 mmc_queue_thread() 执行与 MMC 对应的 mq->issue_fn(mq, req):

```

static int mmc_queue_thread(void *d)
{
    ...
    do {
        ...
        req = blk_fetch_request(q);
        mq->mqrq_cur->req = req;
        ...
        if (req || mq->mqrq_prev->req) {
            set_current_state(TASK_RUNNING);
            cmd_flags = req ? req->cmd_flags : 0;
            mq->issue_fn(mq, req);
            ...
        }
        return 0;
    }
}

```

对于存储设备而言，mq->issue_fn() 函数指向 drivers/mmc/card/block.c 中的 mmc_blk_issue_rq():

```

static struct mmc_blk_data *mmc_blk_alloc_req(struct mmc_card *card,
                                              struct device *parent,
                                              sector_t size,
                                              bool default_ro,
                                              const char *subname,
                                              int area_type)
{
    ...
    md->queue.issue_fn = mmc_blk_issue_rq;
    md->queue.data = md;
    ...
}

```

其中的 mmc_blk_issue_rw_rq() 等函数最终会调用 drivers/mmc/core/core.c 中的 mmc_start_req() 这样的函数：

```

static int mmc_blk_issue_rw_rq(struct mmc_queue *mq, struct request *rqc)
{
    ...
    areq = mmc_start_req(card->host, areq, (int *) &status);
    ...
}

```

`mmc_start_req()`反过来又调用host驱动的`host->ops->pre_req()`、`host->ops->enable()`、`host->ops->disable()`、`host->ops->request()`等成员函数，这些函数实现于`drivers/mm/host`目录中。

`host->ops`实际上是一个MMC host操作的集合，对应的结构体为`mmc_host_ops`，它的定义如代码清单13.16所示。MMC主机驱动的主体工作就是实现该结构体的成员函数，如`drivers/mmc/host/mmc_spi.c`、`drivers/mmc/host/bfin_sdh.c`、`drivers/mmc/host/sdhci.c`等。

代码清单13.16 `mmc_host_ops`结构体

```

1 struct mmc_host_ops {
2     /*
3      * 'enable' is called when the host is claimed and 'disable' is called
4      * when the host is released. 'enable' and 'disable' are deprecated.
5      */
6     int (*enable)(struct mmc_host *host);
7     int (*disable)(struct mmc_host *host);
8     /*
9      * It is optional for the host to implement pre_req and post_req in
10     * order to support double buffering of requests (prepare one
11     * request while another request is active).
12     * pre_req() must always be followed by a post_req().
13     * To undo a call made to pre_req(), call post_req() with
14     * a nonzero err condition.
15     */
16     void (*post_req)(struct mmc_host *host, struct mmc_request *req,
17                      int err);
18     void (*pre_req)(struct mmc_host *host, struct mmc_request *req,
19                     bool is_first_req);
20     void (*request)(struct mmc_host *host, struct mmc_request *req);
21     /*
22      * Avoid calling these three functions too often or in a "fast path",
23      * since underlaying controller might implement them in an expensive
24      * and/or slow way.
25      *
26      * Also note that these functions might sleep, so don't call them
27      * in the atomic contexts!
28      *
29      * Return values for the get_ro callback should be:
30      *   0 for a read/write card
31      *   1 for a read-only card
32      *   -ENOSYS when not supported (equal to NULL callback)
33      *   or a negative errno value when something bad happened
34      *
35      * Return values for the get_cd callback should be:
36      *   0 for a absent card
37      *   1 for a present card
38      *   -ENOSYS when not supported (equal to NULL callback)
39      *   or a negative errno value when something bad happened
40      */

```

```

41      void      (*set_ios)(struct mmc_host *host, struct mmc_ios *ios);
42      int       (*get_ro)(struct mmc_host *host);
43      int       (*get_cd)(struct mmc_host *host);
44
45      void      (*enable_sdio_irq)(struct mmc_host *host, int enable);
46
47      /* optional callback for HC quirks */
48      void      (*init_card)(struct mmc_host *host, struct mmc_card *card);
49
50      int       (*start_signal_voltage_switch)(struct mmc_host *host, struct
51                                              mmc_ios *ios);
52
53      /* Check if the card is pulling dat[0:3] low */
54      int       (*card_busy)(struct mmc_host *host);
55
56      /* The tuning command opcode value is different for SD and eMMC cards */
57      int       (*execute_tuning)(struct mmc_host *host, u32 opcode);
58
59      /* Prepare HS400 target operating frequency depending host driver */
60      int       (*prepare_hs400_tuning)(struct mmc_host *host, struct mmc_ios *ios);
61      int       (*select_drive_strength)(unsigned int max_dtr, int host_drv,
62                                         int card_drv);
62      void      (*hw_reset)(struct mmc_host *host);
63      void      (*card_event)(struct mmc_host *host);
63 };

```

由于目前大多数 SoC 内嵌的 MMC/SD/SDIO 控制器是 SDHCI (Secure Digital Host Controller Interface)，所以更多是直接重用 drivers/mmc/host/sdhci.c 驱动，很多芯片甚至还可以进一步使用基于 drivers/mmc/host/sdhci.c 定义的 drivers/mmc/host/sdhci-pltfm.c 框架。

13.9 总结

块设备的 I/O 操作方式与字符设备的存在较大的不同，因而引入了 request_queue、request、bio 等一系列数据结构。在整个块设备的 I/O 操作中，贯穿始终的就是“请求”，字符设备的 I/O 操作则是直接进行不绕弯，块设备的 I/O 操作会排队和整合。

驱动的任务是处理请求，对请求的排队和整合由 I/O 调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。

尽管在块设备驱动中仍然存在 block_device_operations 结构体及其成员函数，但不再包含读写类的成员函数，而只是包含打开、释放及 I/O 控制等与具体读写无关的函数。

块设备驱动的结构相对复杂，但幸运的是，块设备不像字符设备那样包罗万象，它通常就是存储设备，而且驱动的主体已经由 Linux 内核提供，针对一个特定的硬件系统，驱动工程师所涉及的工作往往只是编写极其少量的与硬件平台相关的代码。