# 总体结构

与字符设备以字节为单位进行读写不同，块设备以块为单位进行读写操作。字符设备只能顺序读写，但是块设备可以随机访问。块设备对于I/O请求有相应的缓冲区，字符设备无需缓冲直接读写。

块设备在Linux系统中的总体结构如下：

用户空间

内核空间

硬件

文件I/O

文件I/O

I/O调度(I/O schedule)

块设备驱动

块设备驱动

磁盘

flash

虚拟文件系统(VFS)

磁盘缓存(cache)

文件系统(EXT4/VFAT)

通用块层(Generic Block Layer)

VFS：文件系统的I/O请求首先会经过虚拟文件系统，虚拟文件系统是对各种具体的文件系统的统一封装，为用户空间提供了统一的接口。

cache：如果为缓存写，则写入page cache，然后会返回，后续会有刷脏页的操作，如果为缓存读，则会先判断数据是否在cache中，如果在则直接从cache中读取，如果不在则会到具体的文件系统中读取。当然如果选择使用Direct I/O则会绕过cache。

通用块层：把上层的读写请求构造成一个或多个bio。

I/O调度：将读写区域连续的bio合并成request。

块驱动层：完成与具体物理设备相关的操作，将request进行处理。

# 驱动初始化

内核使用gendisk结构来表示一个独立的磁盘设备或者分区：

struct gendisk {

//major、first\_minor、minors为输入参数，勿直接使用，使用disk\_devt()/disk\_max\_parts()访问

int major;

int first\_minor;

int minors;

char disk\_name[DISK\_NAME\_LEN];

char \*(\*devnode)(struct gendisk \*gd, umode\_t \*mode);

unsigned int events;

unsigned int async\_events;

struct disk\_part\_tbl \_\_rcu \*part\_tbl;

struct hd\_struct part0;

const struct block\_device\_operations \*fops;

struct request\_queue \*queue;

void \*private\_data;

int flags;

struct kobject \*slave\_dir;

struct timer\_rand\_state \*random;

atomic\_t sync\_io; /\* RAID \*/

struct disk\_events \*ev;

int node\_id;

struct badblocks \*bb;

};

.major：磁盘的主设备号，由register\_blkdev()函数申请。

.first\_minor：起始次设备号，同一磁盘的各个分区共享一个主设备号，次设备号不同。

.minors：次设备号的数量，即分区的数量，为1时表示无法分区，在alloc\_disk()时填充。

.disk\_name：磁盘名称，用于在sysfs和/proc/partitions中表示该磁盘。

.events：支持的事件。

.async\_events：异步事件。

.part\_tbl：磁盘分区表。

.part0：磁盘的第一个分区，disk->part\_tbl->part[0] = &disk->part0。

.fops：块设备的操作函数集，下面会详细介绍。

.queue：用于管理此设备的I/O请求队列，使用blk\_init\_queue()初始化。

.private\_data：指向私有数据。

.flags：磁盘标志

GENHD\_FL\_REMOVABLE：可移动介质

GENHD\_FL\_CD：CD\_ROM

GENHD\_FL\_SUPPRESS\_PARTITION\_INFO：不需要分区信息出现在 /proc/partitions

对块设备的操作方法使用block\_device\_operations结构描述：

struct block\_device\_operations {

int (\*open) (struct block\_device \*, fmode\_t);

void (\*release) (struct gendisk \*, fmode\_t);

int (\*rw\_page)(struct block\_device \*, sector\_t, struct page \*, bool);

int (\*ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);

int (\*compat\_ioctl) (struct block\_device \*, fmode\_t, unsigned, unsigned long);

long (\*direct\_access)(struct block\_device \*, sector\_t, void \*\*, pfn\_t \*, long);

unsigned int (\*check\_events) (struct gendisk \*disk, unsigned int clearing);

/\* ->media\_changed() is DEPRECATED, use ->check\_events() instead \*/

int (\*media\_changed) (struct gendisk \*);

void (\*unlock\_native\_capacity) (struct gendisk \*);

int (\*revalidate\_disk) (struct gendisk \*);

int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);

/\* this callback is with swap\_lock and sometimes page table lock held \*/

void (\*swap\_slot\_free\_notify) (struct block\_device \*, unsigned long);

struct module \*owner;

const struct pr\_ops \*pr\_ops;

};

.open/release：打开与释放。

.ioctl/compat\_ioctl：I/O控制。

.check\_events：检查驱动器中的介质是否发生改变，是则返回非0值，用于可移动的设备，media\_changed是check\_events的过时函数。

.revalidate\_disk：介质改变时调用，它给驱动一个机会来进行必要的工作以使新介质准备好。

.getgeo：获得驱动器信息，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。

.owner：一般为THIS\_MODULE。

gendisk相关的操作方法如下：

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，使用如下函数来分配：

struct gendisk \* alloc\_disk(int minors);

.minors：磁盘的次设备号数量，即磁盘分区的数量，此后此值不能被更改。

gendisk结构分配后，需要向系统进行设备的注册，注意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：

void add\_disk(struct gendisk \*disk);

与add\_disk()相应的从系统注销gendisk的函数为：

void del\_gendisk(struct gendisk \*gp);

gendisk的引用计数通过get\_disk()和put\_disk()加或者减，这个工作一般不需要驱动自己做，但是注意add\_disk()增加的引用计数，需要我们自己在del\_gendisk()后或者错误处理中调用put\_disk()。

块设备的最小寻址单元为扇区，扇区大小一般为512字节，需要通过如下函数设置扇区数目：

void set\_capacity(struct gendisk \*disk, sector\_t size);

典型的块设备驱动的初始化过程如下：

1. 通过register\_blkdev()注册块设备，申请块设备的主设备号。
2. 通过blk\_init\_queue()申请请求队列，通过blk\_queue\_max\_hw\_sectors()和blk\_queue\_logical\_block\_size()设置该请求队列支持的每个请求能够包含的最大扇区数目以及逻辑块大小即硬件扇区大小。
3. 通过alloc\_disk()分配gendisk，并对其成员进行初始化。
4. 通过add\_disk()将磁盘添加到块I/O层。

相应的反初始化过程就为：

1. 通过del\_disk()注销gendisk。
2. 通过put\_disk()删除对gendisk的引用。
3. 通过blk\_cleanup\_queue()清除请求队列。
4. 通过unregister\_blkdev()注销块设备。

# I/O请求

在块设备驱动中，使用request结构来表示通过I/O调度以后的针对一个gendisk的I/O请求：

struct request {

struct list\_head queuelist; //链接到所属请求队列上

struct request\_queue \*q;//此request从属的request\_queue

struct blk\_mq\_ctx \*mq\_ctx;

int cpu;

unsigned cmd\_type;

u64 cmd\_flags;

unsigned long atomic\_flags;

unsigned int \_\_data\_len;

sector\_t \_\_sector;

struct bio \*bio;//请求的bio结构体的链表

struct bio \*biotail;//请求的bio结构体的链表尾

...

};

request结构很大，目前并不完全理解其所有成员的含义，重点先关注其bio成员，bio表示针对一个gendisk的I/O调度前的原始请求，I/O调度算法会将连续的bio合并成一个请求，因此一个request可能包含多个bio，bio是通用快层的主要数据结构，既描述了磁盘的位置同时也描述了内存的位置，是上层vfs与下层驱动的纽带：

struct bio {

struct bio \*bi\_next;//下一个bio

struct block\_device \*bi\_bdev;//发起请求的块设备

int bi\_error;

unsigned int bi\_opf;

unsigned short bi\_flags;

unsigned short bi\_ioprio;

struct bvec\_iter bi\_iter;//用于记录当前bvec被处理的状况

unsigned int bi\_phys\_segments;//合并后不相邻的物理段的数目

unsigned int bi\_seg\_front\_size;

unsigned int bi\_seg\_back\_size;

atomic\_t \_\_bi\_remaining;

bio\_end\_io\_t \*bi\_end\_io;

void \*bi\_private;

unsigned short bi\_vcnt; //列表数目bio\_vec

unsigned short bi\_max\_vecs; //bio中bio\_vec的最大数目

atomic\_t \_\_bi\_cnt;

struct bio\_vec \*bi\_io\_vec; // bio\_vec列表

struct bio\_set \*bi\_pool;

};

其中的bio\_vec描述了与这个bio请求对应的所有内存，这些内存可能分布在不同页中，所以用数组来表示：

struct bio\_vec {

struct page \*bv\_page;//页指针

unsigned int bv\_len;//传输的字节数

unsigned int bv\_offset;//偏移地址

};

每个gendisk都包含一个request\_queue，其中包含着所有request的链表头，同时包含设备支持的请求信息，如最大大小、硬件扇区大小、对齐要求等，这就能保证设备接收到的请求都是可被处理的：

struct request\_queue {

struct list\_head queue\_head;//所有request的链表头

struct request \*last\_merge;

struct elevator\_queue \*elevator;//请求队列使用的IO调度算法

request\_fn\_proc \*request\_fn;//队列初始化时提供的用于处理队列中请求的回调，这是实现块设备驱动的核心

make\_request\_fn \*make\_request\_fn;

prep\_rq\_fn \*prep\_rq\_fn;

unprep\_rq\_fn \*unprep\_rq\_fn;

softirq\_done\_fn \*softirq\_done\_fn;

rq\_timed\_out\_fn \*rq\_timed\_out\_fn;

dma\_drain\_needed\_fn \*dma\_drain\_needed;

lld\_busy\_fn \*lld\_busy\_fn;

struct blk\_mq\_ops \*mq\_ops;

unsigned int \*mq\_map;

unsigned long nr\_requests; //最大请求数量

...

};

request\_queue/request/bio处理的相关API如下：

初始化请求队列：

struct request\_queue \*blk\_init\_queue(request\_fn\_proc \*rfn, spinlock\_t \*lock);

.rfn：请求处理函数指针

.lock：控制访问队列权限的自旋锁

对应的清除请求队列：

void blk\_cleanup\_queue(struct request\_queue \*q);

对于flash/RAMDISK这种完全随机访问的非机械设备，无需进行复杂的I/O调度，可使用如下函数分配请求队列：

struct request\_queue \*blk\_alloc\_queue(gfp\_t gfp\_mask);

对于没有I/O调度的request\_queue使用如下函数绑定请求队列和制造请求函数，在制造请求函数中就会完成请求的处理：

void blk\_queue\_make\_request(struct request\_queue \*q, make\_request\_fn \*mfn);

提取请求，可返回下一个需要处理的请求，如果没有请求则返回NULL，此函数并不会移除请求：

struct request \*blk\_peek\_request(struct request\_queue \*q);

启动请求，会将请求从请求队列中移除：

void blk\_start\_request(struct request \*req);

可使用如下函数同时完成blk\_peek\_request()和blk\_start\_request():

struct request \*blk\_fetch\_request(struct request\_queue \*q);

报告请求是否完成，error为0表示成功，小于0表示失败，其中带\_\_前缀的需要在持有队列锁的场景下调用：

报告整个request完成：

void blk\_end\_request\_all(struct request \*rq, int error);

void \_\_blk\_end\_request\_all(struct request \*rq, int error);

报告request中当前的chunk完成，返回值为false代表request的全部数据都处理完成，返回true则代表request中还有未处理完毕的数据：

bool blk\_end\_request\_cur(struct request \*rq, int error);

bool \_\_blk\_end\_request\_cur(struct request \*rq, int error);

获取当前扇区位置：

sector\_t blk\_rq\_pos(const struct request \*rq);

获取整个请求中剩余的字节数：

unsigned int blk\_rq\_bytes(const struct request \*rq);

获取当前segment剩余的字节数：

int blk\_rq\_cur\_bytes(const struct request \*rq);

获取请求的读写方向，返回WRITE或READ：

rq\_data\_dir();

获取bio的数据地址：

void \*bio\_data(struct bio \*bio);

获取bio数据方向：

int bio\_data\_dir(struct bio \* bio);

遍历一个请求的所有bio:

\_\_rq\_for\_each\_bio(\_bio, rq);

遍历一个请求的所有bio中的所有segment：

rq\_for\_each\_segment(bvl, \_rq, \_iter);

遍历一个bio的所有bio\_vev：

bio\_for\_each\_segment(bvl, bio, iter);

对于绕开I/O调度的情况，应该在bio处理完成后使用如下函数通知处理结束：

void bio\_endio(struct bio \*bio);

总结gendisk/request\_queue/request/bio之间的关系，每个gendisk都包含一个request\_queue，request\_queue上又挂了request，而request中又包含了一个或多个bio，每个bio又包含了bio\_vec数组：

bio

bio\_vec

gendisk

queue

request\_queue

queue\_head

request

queuelist

bio

request

queuelist

bio

request

queuelist

bio

bio

bio\_vec

bio

bio\_vec

bio\_vec

bv\_page

bv\_len

bv\_offset

bio\_vec

bv\_page

bv\_len

bv\_offset

bio\_vec

bv\_page

bv\_len

bv\_offset

# I/O调度器

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，目前Linux中IO调度的电梯算法有3种，分别是cfq(Complete Fairness Queueing)、deadline以及noop(No Operation)，linux默认使用cfq。具体使用哪种算法可以进行更改：

echo deadline > /sys/block/mmcblk1/queue/scheduler

Noop：

Noop调度算法是内核中最简单的IO调度算法。它将IO请求放入到一个FIFO队列中，然后逐个执行这些IO请求，当然对于一些在磁盘上连续的IO请求，Noop算法会适当做一些合并。此算法适用于不希望调度器重新组织IO请求顺序的应用，如基于flash的存储器，所以使用固态硬盘（SSD）时优先使用Noop算法。

Deadline:

Deadline算法的核心在于保证每个IO请求在一定的时间内一定要被服务到，以此来避免某个请求饥饿。Deadline算法中引入了四个队列，这四个队列可以分为两类，每一类都由读和写两类队列组成，一类队列用来对请求按起始扇区序号进行排序，通过红黑树来组织，称为sort\_list；另一类对请求按它们的生成时间进行排序，由链表来组织，称为fifo\_list。每当确定了一个传输方向(读或写)，那么将会从相应的sort\_list中将一批连续请求dispatch到requst\_queue的请求队列里，具体的数目由fifo\_batch来确定。

只有下面三种情况才会导致一次批量传输的结束：

1）对应的sort\_list中已经没有请求了

2）下一个请求的扇区不满足递增的要求

3）上一个请求已经是批量传输的最后一个请求了。

所有的请求在生成时都会被赋上一个期限值(根据jiffies)，并按期限值排序在fifo\_list中，读请求的期限时长默认为为500ms，写请求的期限时长默认为5s，可以看出内核对读请求是十分偏心的，其实不仅如此，在deadline调度器中，还定义了一个starved和writes\_starved，writes\_starved默认为2，可以理解为写请求的饥饿线，内核总是优先处理读请求，starved表明当前处理的读请求批数，只有starved超过了writes\_starved后，才会去考虑写请求。因此，假如一个写请求的期限已经超过，该请求也不一定会被立刻响应，因为读请求的batch还没处理完，即使处理完，也必须等到starved超过writes\_starved才有机会被响应。为什么内核会偏袒读请求？这是从整体性能上进行考虑的。读请求和应用程序的关系是同步的，因为应用程序要等待读取的内容完毕，才能进行下一步工作，因此读请求会阻塞进程，而写请求则不一样，应用程序发出写请求后，内存的内容何时写入块设备对程序的影响并不大，所以调度器会优先处理读请求。所以Deadline适用于业务比较单一并且I/O压力比较重的情况，对于同步写多的情况Deadline是不太合适的。

CFQ：

CFQ（Completely Fair Queuing）算法，顾名思义，绝对公平算法。它试图为竞争块设备使用权的所有进程分配一个请求队列和一个时间片，在调度器分配给进程的时间片内，进程可以将其读写请求发送给底层块设备，当进程的时间片消耗完，进程的请求队列将被挂起，等待调度。每个进程的时间片和每个进程的队列长度取决于进程的IO优先级，每个进程都会有一个IO优先级，CFQ调度器将会将其作为考虑的因素之一，来确定该进程的请求队列何时可以获取块设备的使用权。所以CFQ适用于有大量进程的多用户系统。

# 示例说明

设备树：

virt\_disk: disk@0 {

compatible = "xm,virt-disk";

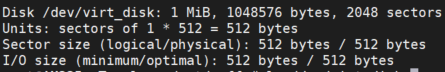
status = "okay";

};

宏NOUSE\_IO\_SCHEDULE控制是否使用I/O调度，决定了使用哪个函数来处理请求。

insmod virt-disk.ko后出现/dev/virt\_disk设备

fdisk -l可以看到磁盘信息：



mkfs.ext4 /dev/virt\_disk 格式化成ext4格式

mount /dev/virt\_disk test 挂载

此后可以对磁盘进行读写操作：

