Linux内核IO调度层

http://www.ilinuxkernel.com

本文链接：http://ilinuxkernel.com/?p=1693

目 录

图目录

1 概述

我们仍以块设备操作流程开始，分析内核块设备操作中的过程。在块设备上的操作，涉及内核中的多个组成部分，如图1所示。假设一个进程使用系统调用read（）读取磁盘上的文件。下面步骤是内核响应进程读请求的步骤；

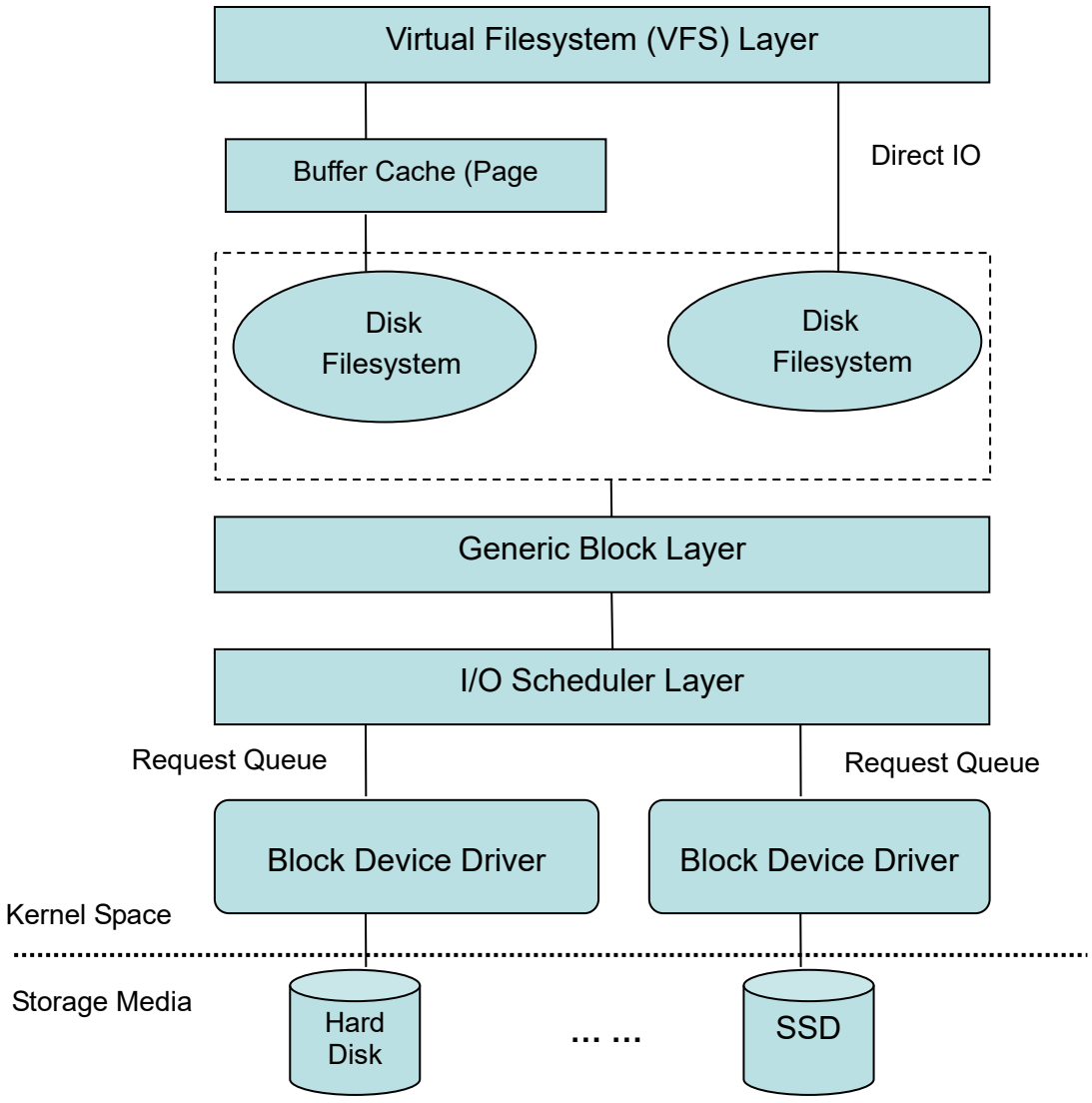

图1 内核中块设备操作流程

（1）系统调用read（）会触发相应的VFS（Virtual Filesystem Switch）函数，传递的参数有文件描述符和文件偏移量。

（2）VFS确定请求的数据是否已经在内存缓冲区中；若数据不在内存中，确定如何执行读操作。

（3）假设内核必须从块设备上读取数据，这样内核就必须**确定数据在物理设备上的位置**。这由映射层（Mapping Layer）来完成。

（4）内核通过通用块设备层（Generic Block Layer）在块**设备上执行读操作，启动I/O操作**，传输请求的数据。

（5）在通用块设备层之下是**I/O调度层（I/O Scheduler Layer）**，根据**内核的调度策略，对等待的I/O等待队列排序**。

（6）最后，块设备驱动（Block Device Driver）通过向磁盘控制器发送相应的命令，执行真正的数据传输。

对于（1）、（2）两个步骤，在《Linux虚拟文件系统》中，我们讨论了VFS（Virtual Filesystem Switch）主要数据结构和操作，结合相关系统调用（如sys_read（）、sys_write（）等）的源码，我们不难理解VFS层相关的操作和实现。而对于第（3）步中的Mapping Layer主要由具体的文件系统完成。

在《Linux通用块设备层》中，我们分析了第（4）步。上层的I/O请求发送到通用块设备层（Generic Block Layer）后，就会将请求继续传送到I/O调度层（I/O Scheduler Layer）。

本文以2.6.32-279内核源码为基础，分析图1中的第5步，即I/O调度层。首先分析请求队列（request queue）、请求（request）描述符；然后着重介绍两个个I/O调度算法；最后分析内核中相关源码。

2 请求和请求队列

尽管块设备驱动可以一次传输一个扇区（sector），但块I/O层并不会对一个扇区执行一次独立的I/O操作，因为在磁盘定位一个扇区的位置，是很耗时的工作，这会导致磁盘性能的下降。取而代之的是，内核尽可能聚集多个扇区，作为一次操作，这样就减少磁头的移动操作。

当内核读或写磁盘数据时，它就创建一个块设备请求（block device request），该请求主要描述请求的扇区、操作的类型（读还是写）。然而，当请求创建后，内核并不是立即执行请求，而是I/O操作仅是调度，且在稍后的时间才真正执行I/O操作。这种延迟是提高块

设备性能的关键机制。有新的数据块请求时，内核检查是否可以增大现有等待的请求来满足请求；因为访问磁盘往往是顺序的，于是这种简单的提高块设备性能机制非常有效。

延迟请求的执行，使对块设备的处理变得复杂。例如，假设一个进程打开一个普通的文件，进而文件系统驱动从磁盘上读取文件的inode信息。块设备驱动将请求挂在队列上，进程挂起直到inode数据传输完毕。但块设备驱动本身不能阻塞，因为这样会导致其他访问该磁盘的进程也会被阻塞。

为了避免块设备驱动阻塞，每个I/O操作的处理都是异步的。即块设备驱动是中断驱动的，通用块设备层触发I/O调度器来创建一个新块设备请求或者扩大已有的请求（该工作由通用块设备层完成还是I/O调度层？），然后结束。过一段时间后，块设备驱动激活并调用服务例程（strategy routine）来选择一个等待的请求，向磁盘控制器发送相应的命令来完成请求。当I/O操作结束时，磁盘控制器发起一个中断，必要时，相应的处理程序又调用服务例程，

每个块设备驱动维护自己的请求队列（request queue），队列中包含了对该设备的等待请求链表。若磁盘控制器处理多个磁盘，通常每个块设备都会有一个请求队列。在各自的请求队列上进行单独的I/O调度，从而提高磁盘性能。

2.1 请求队列request_queue

每个请求队列由request_queue数据结构表示，其定义在文件include/linux/blkdev.h中。

```
00288: struct request_queue
00289: {
00290:     /*
00291:      * Together with queue_head for cacheline sharing
00292:      */
00293:     struct list_head    queue_head;
00294:     struct request      *last_merge;
00295:     struct elevator_queue *elevator;
00296:
00297:     /*
00298:      * the queue request freelist, one for reads and one for writes
00299:      */
00300:     struct request_listrq;
00301:
00302:     request_fn_proc         *request_fn;
00303:     make_request_fn         *make_request_fn;
00304:     prep_rq_fn          *prep_rq_fn;
00305:     unprep_rq_fn        *unprep_rq_fn;
00306:     unplug_fn       *unplug_fn;
00307:     merge_bvec_fn           *merge_bvec_fn;
00308:     prepare_flush_fn    *prepare_flush_fn;
00309:     softirq_done_fn         *softirq_done_fn;
00310:     rq_timed_out_fn         *rq_timed_out_fn;
00311:     dma_drain_needed_fn     *dma_drain_needed;
```

```
00312:     lld_busy_fn         *lld_busy_fn;
00313:
00314:     /*
00315:      * Dispatch queue sorting
00316:      */
00317:     sector_t            end_sector;
00318:     struct request      *boundary_rq;
00319:
00320:     /*
00321:      * Auto- unplugging state
00322:      */
00323:     struct timer_list       unplug_timer;
00324:     int             unplug_thresh;      /* After this many requests */
00325:     unsigned long       unplug_delay;   /* After this many jiffies */
00326:     struct work_struct      unplug_work;
00327:
00328:     struct backing_dev_info     backing_dev_info;
00329:
00330:     /*
00331:      * The queue owner gets to use this for whatever they like.
00332:      * ll_rw_blk doesn't touch it.
00333:      */
00334:     void                *queuedata;
00335:
00336:     /*
00337:      * queue needs bounce pages for pages above this limit
00338:      */
00339:     gfp_t               bounce_gfp;
00340:
00341:     /*
00342:      * various queue flags, see QUEUE_* below
00343:      */
00344:     unsigned long       queue_flags;
00345:
00346:     /*
00347:      * protects queue structures from reentrancy. ->__queue_lock should
00348:      * _never_ be used directly, it is queue private. always use
00349:      * ->queue_lock.
00350:      */
00351:     spinlock_t      __queue_lock;
00352:     spinlock_t      *queue_lock;
00353:
00354:     /*
00355:      * queue kobject
00356:      */
00357:     struct kobject kobj;
00358:
00359:     /*
00360:      * queue settings
00361:      */
00362:     unsigned long       nr_requests; /* Max # of requests */
00363:     unsigned int        nr_congestion_on;
00364:     unsigned int        nr_congestion_off;
00365:     unsigned int        nr_batching;
00366:
00367:     void                *dma_drain_buffer;
00368:     unsigned int        dma_drain_size;
00369:     unsigned int        dma_pad_mask;
00370:     unsigned int        dma_alignment;
00371:
00372:     struct blk_queue_tag    *queue_tags;
00373:     struct list_head    tag_busy_list;
00374:
00375:     unsigned int        nr_sorted;
00376:     unsigned int        in_flight[2];
00377:
```

```
00378:     unsigned int        rq_timeout;
00379:     struct timer_list      timeout;
00380:     struct list_head    timeout_list;
00381:
00382:     struct queue_limits    limits;
00383:
00384:     /*
00385:      * sg stuff
00386:      */
00387:     unsigned int        sg_timeout;
00388:     unsigned int        sg_reserved_size;
00389:     int             node;
00390: #ifdef CONFIG_BLK_DEV_IO_TRACE
00391:   struct blk_trace    *blk_trace;
00392: #endif
00393:     /*
00394:      * DEPRECATED: please use "for flush operations" members below!
00395:      * These members (through orig_bar_rq) are preserved purely
00396:      * to maintain kABI.
00397:      */
00398:     unsigned int        ordered, next_ordered, ordseq;
00399:     int             orderr, ordcolor;
00400:     struct request      pre_flush_rq, bar_rq, post_flush_rq;
00401:     struct request      *orig_bar_rq;
00402:
00403:     struct mutex        sysfs_lock;
00404:
00405: #if defined(CONFIG_BLK_DEV_BSG)
00406:     struct bsg_class_device bsg_dev;
00407: #endif
00408:     /* For future extensions */
00409:     void    *pad;
00410:
00411: #ifndef __GENKSYMS__
00412:     /*
00413:      * for flush operations
00414:      */
00415:     unsigned int        flush_flags;
00416:     unsigned int        flush_not_queueable:1;
00417:     unsigned int        flush_queue_delayed:1;
00418:     unsigned int        flush_pending_idx:1;
00419:     unsigned int        flush_running_idx:1;
00420:     unsigned long       flush_pending_since;
00421:     struct list_head    flush_queue[2];
00422:     struct list_head    flush_data_in_flight;
00423:     struct request      flush_rq;
00424:
00425: #ifdef CONFIG_BLK_DEV_THROTTLING
00426:     /* Throttle data */
00427:     struct throtl_data *td;
00428: #endif
00429:     /*
00430:      * Delayed queue handling
00431:      */
00432:     struct delayed_work     delay_work;
00433: #endif /* ____GENKSYMS____ */
00434: } « end request_queue » ;
00435:
```

表1是request_queue数据结构中主要成员变量的含义。

表1 request_queue数据结构主要成员变量含义

成员变量	含义
queue_head	等待请求的链表头
last_merge	指向队列中可能被合并的请求
elevator	指向elevator队列
rq	空闲请求链表，共有两个，一个为读请求，另外一个写请求
request_fn	驱动服务例程（strategy routine）的实现方法
make_request_fn	当新请求加入队列中调用的方法
prep_rq_fn	生成硬件处理请求指令的方法
unplug_fn	unplug块设备的方法
merge_bvec_fn	合并bvec到bio中的方法，通常未定义
unplug_timer	执行设备plugging的动态定时器
unplug_thresh	当队列上等待的请求超过这个值时，设备立刻unplug。默认值为4
unplug_delay	设备unplugging的时间间隔，默认值为3ms
unplug_work	用于unplug设备的工作队列
backing_dev_info	backing_dev_info描述了backing_dev的所有信息
queuedata	指向块设备驱动的私有数据
bounce_gfp	bounce buffer内存分配标志
queue_flags	队列状态标志
queue_lock	队列锁
kobj	嵌入在请求队列的kobject
nr_requests	队列中最大请求数量
nr_congestion_on	当队列中等待的请求数量高于某个值时，就认为队列拥塞
nr_congestion_off	当队列中等待的请求数量低于某个值时，就认为队列不再拥塞
nr_batching	最大可以提交的请求数量（通常为32）
dma_alignment	初始地址和DMA缓冲区的对齐bitmap，默认值为511
queue_tags	空闲/繁忙标志位图，用在有tagged请求
refcnt	Reference counter of the queue
in_flight	队列中，处于pending状态的请求数量
sg_timeout	用户自定义的命令time-out时间，仅在通用SCSI设备中使用
sg_reserved_size	未使用

事实上，一个请求队列就是双链表，其中的每个元素是请求（request）描述符。成员变量queue_head是请求队列中保存链表头，请求（request）描述符中的queuelist成员变量将各个请求连接起来。请求队列中各个元素的顺序与各自的设备驱动程序相关。I/O调度器有多种方式来对请求排序，这部分内容在第3部分I/O调度器中介绍。

2.2 请求

2.2.1 数据结构request

对每个块设备的等待请求表示为请求描述符（request discriptor），保存在request数据结构中。其定义在include/linux/blkdev.h中。

```
00095: struct request {
00096:     struct list_head queuelist;
00097:     struct call_single_data csd;
00098:
00099:     struct request_queue *q;
00100:
00101:     unsigned int cmd_flags;
00102:     enum rq_cmd_type_bits cmd_type;
00103:     unsigned long atomic_flags;
00104:
00105:     int cpu;
00106:
00107:     /* the following two fields are internal, NEVER access directly */
00108:     unsigned int __data_len;      /* total data len */
00109:     sector_t __sector;            /* sector cursor */
00110:
00111:     struct bio *bio;
00112:     struct bio *biotail;
00113:
00114:     struct hlist_node hash;       /* merge hash */
00115:     /*
00116:      * The rb_node is only used inside the io scheduler, requests
00117:      * are pruned when moved to the dispatch queue. So let the
00118:      * completion_data share space with the rb_node.
00119:      */
00120:     union {
00121:         struct rb_node rb_node;     /* sort/lookup */
00122:         void *completion_data;
00123:     };
00124:
00125:     /*
00126:      * Three pointers are available for the IO schedulers, if they need
00127:      * more they have to dynamically allocate it.  Flush requests are
00128:      * never put on the IO scheduler. So let the flush fields share
00129:      * space with the three elevator_private pointers.
00130:      */
00131: #ifdef ___GENKSYMS__
00132:     void *elevator_private;
00133:     void *elevator_private2;
00134:     void *elevator_private3;
00135: #else
00136:     union {
00137:         void *elevator_private[3];
00138:         struct {
00139:             unsigned int        seq;
00140:             struct list_head    list;
00141:         } flush;
00142:     };
00143: #endif /* ____GENKSYMS____ */
00144:
00145:     struct gendisk *rq_disk;
00146:     unsigned long start_time;
00147: #ifdef CONFIG_BLK_CGROUP
00148:     unsigned long long start_time_ns;
00149:     unsigned long long io_start_time_ns;        /* when passed to hardware */
```

```
00150: #endif
00151:     /* Number of scatter- gather DMA addr+len pairs after
00152:      * physical address coalescing is performed.
00153:      */
00154:     unsigned short nr_phys_segments;
00155:
00156:     unsigned short ioprio;
00157:
00158:     int ref_count;
00159:
00160:     void *special;        /* opaque pointer available for LLD use */
00161:     char *buffer;         /* kaddr of the current segment if available */
00162:
00163:     int tag;
00164:     int errors;
00165:
00166:     /*
00167:      * when request is used as a packet command carrier
00168:      */
00169:     unsigned char __cmd[BLK_MAX_CDB];
00170:     unsigned char *cmd;
00171:     unsigned short cmd_len;
00172:
00173:     unsigned int extra_len;       /* length of alignment and padding */
00174:     unsigned int sense_len;
00175:     unsigned int resid_len;    /* residual count */
00176:     void *sense;
00177:
00178:     unsigned long deadline;
00179:     struct list_head timeout_list;
00180:     unsigned int timeout;
00181:     int retries;
00182:
00183:     /*
00184:      * completion callback.
00185:      */
00186:     rq_end_io_fn *end_io;
00187:     void *end_io_data;
00188:
00189:     /* for bidi */
00190:     struct request *next_rq;
00191:     /* For future extensions */
00192:     void *pad;
00193: } « end request » ;
```

request数据结构中各成员变量的含义如表2所示。

表2 request数据结构主要成员变量的含义

成员变量	含义
queuelist	指向请求队列链表
q	指向所属的请求队列
cmd_flags	请求的标志
__data_len	request要传输的数据字节数
__sector	下一个要传输的bio的起始扇区号
bio	请求中第一个未完成的bios
biotail	请求链表中最后一个bio
elevator_private	指向I/O调度器的私有数据

rq_disk	该请求访问的磁盘
errors	当前数据传输出现的I/O错误数量
start_time	请求的起始时间（jiffies）
nr_phys_segments	该请求所包含的物理segment数量
tag	该请求的标记
buffer	当前数据传输所在的内存缓冲区（当缓冲区在高端内存时，值为NULL）
ref_count	请求的引用计数
special	当请求包含硬件特殊的指令时，special指向该特殊命令使用的数据
cmd_len	cmd域的命令长度
cmd	请求队列中prep_rq_fn生成的pre-built命令
sense_len	sense域的缓冲区长度（当sense为NULL时，该值为0）
sense	sense命令输出结果所在的缓冲区
timeout	请求time-out时间

每个请求由一个或多个bio结构组成。如图2所示。最初，通用块设备层创建一个请求，请求只包含一个bio。之后，通过加一个新段（segment）到原来的bio中，或者将其他的bio链接到该请求中；I/O调度器可能会扩展该请求。这种情形可能性比较大，因为新请求的数据可能与已存在请求的数据相邻。request数据结构中的成员变量bio指向该请求中的第一个bio结构，biotail指向最后一个bio结构。宏__rq_for_each_bio可以遍历请求中的所有bio结构。

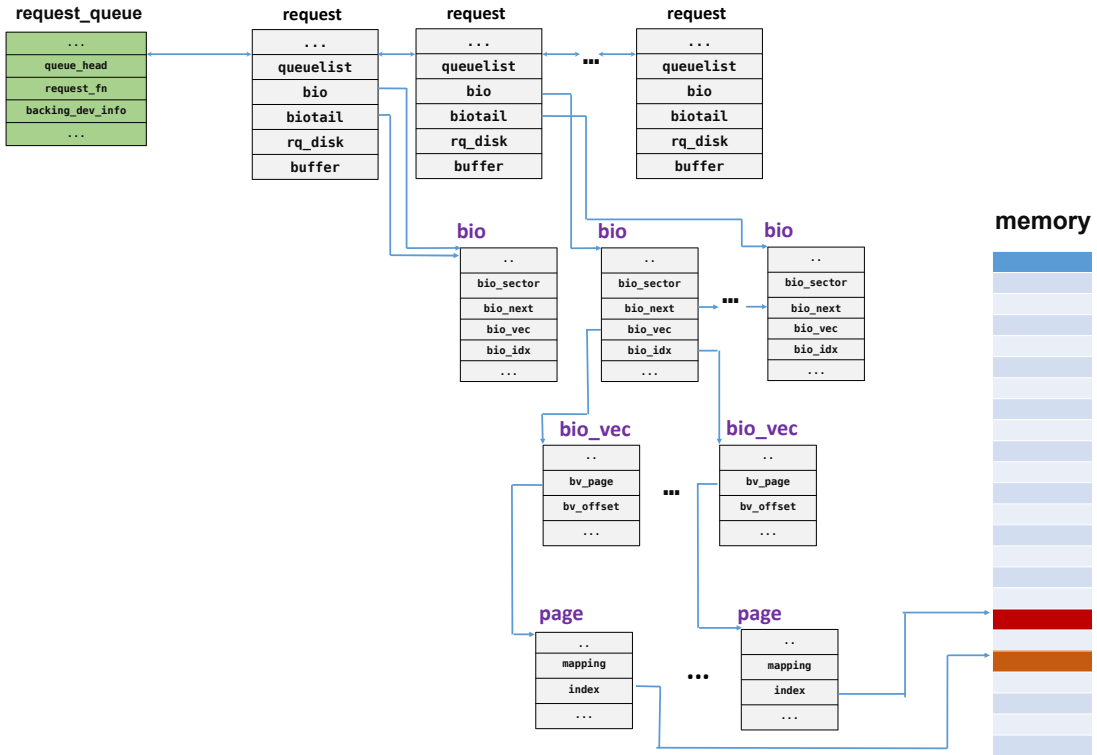

图2 请求和请求队列

```
00725: /* This should not be used directly -  use rq_for_each_segment */
00726: #define for_each_bio(_bio)      \
00727:     for (; _bio; _bio = _bio->bi_next)
00728: #define __rq_for_each_bio(_bio, rq) \
00729:     if ((rq->bio))          \
00730:         for (_bio = (rq)->bio; _bio; _bio = _bio->bi_next)
```

request数据结构中的某些成员变量，在运行中可能会动态地发生变化。例如，当一个bio结构中的数据传输完毕后，request中的成员变量bio就会更新，指向请求中的下一个bio。同时新的bio可以加入到请求中，也即biotail也会发生变化。

在磁盘扇区传输的过程中，request数据结构中的其他一些成员变量可能会被I/O调度器或设备驱动改变。例如，nr_sectors是请求中所有仍需要传输的扇区数，current_nr_sectors保存当前bio中待传输的扇区数。

request数据结构中的成员变量cmd_flags保存一些标志。最重要的标志是REQ_WRITE，它决定数据传输的方向（读还是写）。

2.2.2 请求标志

请求的标志位定义在文件include/linux/blk_types.h中。

```
00098: /*
00099:  * request type modified bits. first four bits match BIO_RW* bits, important
00100:  */
00101: enum rq_flag_bits {
00102:     __REQ_WRITE,        /* was ____REQ_RW, not set, read. set, write */
00103:     __REQ_FAILFAST_DEV,     /* no driver retries of device errors */
00104:     __REQ_FAILFAST_TRANSPORT, /* no driver retries of transport errors */
00105:     __REQ_FAILFAST_DRIVER, /* no driver retries of driver errors */
00106:     /* above flags must match BIO_RW_* */
00107:     __REQ_DISCARD,          /* request to discard sectors */
00108:     __REQ_SORTED,           /* elevator knows about this request */
00109:     __REQ_SOFTBARRIER,      /* may not be passed by ioscheduler */
00110:     __REQ_HARDBARRIER,      /* DEPRECATED: may not be passed by drive either */
00111:     __REQ_FUA,              /* forced unit access */
00112:     __REQ_NOMERGE,          /* don't touch this for merging */
00113:     __REQ_STARTED,          /* drive already may have started this one */
00114:     __REQ_DONTPREP,         /* don't call prep for this one */
00115:     __REQ_QUEUED,           /* uses queueing */
00116:     __REQ_ELVPRIV,          /* elevator private data attached */
00117:     __REQ_FAILED,           /* set if the request failed */
00118:     __REQ_QUIET,        /* don't worry about errors */
00119:     __REQ_PREEMPT,          /* set for "ide_preempt" requests */
00120:     __REQ_ORDERED_COLOR,    /* DEPRECATED: is before or after barrier */
00121:     __REQ_SYNC,         /* was ____REQ_RW_SYNC, request is sync (sync write or read) */
00122:     __REQ_ALLOCED,          /* request came from our alloc pool */
00123:     __REQ_META,         /* was ____REQ_RW_META, metadata io request */
00124:     __REQ_COPY_USER,    /* contains copies of user pages */
00125:     __REQ_INTEGRITY,    /* DEPRECATED: integrity metadata has been remapped */
00126:     __REQ_NOIDLE,           /* don't anticipate more IO after this one */
00127:     __REQ_IO_STAT,          /* account I/O stat */
00128:     __REQ_MIXED_MERGE,  /* merge of different types, fail separately */
00129:     __REQ_FLUSH,        /* request for cache flush */
```

```
00130:      __REQ_FLUSH_SEQ,      /* request for flush sequence */
00131:      __REQ_NR_BITS,        /* stops here */
00132: };
```

表3 request主要标志位含义

成员变量	含义
__REQ_WRITE	区分读还是写。设置了该标志，则为写；否则为读。
__REQ_DISCARD	丢弃扇区。
__REQ_SORTED	请求已排序。
__REQ_SOFTBARRIER	告诉请求队列，其他请求不能排在当前请求前面。
__REQ_HARDBARRIER	该标志已废弃。
__REQ_FUA	Forced Unit Access写请求，即保证将数据写入存储介质。
__REQ_NOMERGE	不要合并该请求。
__REQ_STARTED	请求已经被设备开始处理。
__REQ_QUEUED	请求已加入队列。
__REQ_FLVPRIV	调度器（Elevator）私有数据。
__REQ_FAILED	请求失败。
__REQ_QUIET	不关心是否有错误发生（有错误，也不打印信息）。
__REQ_PREEMPT	ide_preempt请求。
__REQ_SYNC	请求是同步的，即同步写或同步读。
__REQ_ALLOCED	请求来源于分配池。
__REQ_META	metadata I/O请求。
__REQ_COPY_USER	包含用户页面的拷贝。
__REQ_NOIDLE	告知调度器该请求发送后，不要等待新请求。
__REQ_IOSTAT	I/O统计请求。
__REQ_FLUSH	Flush Cache数据请求。

2.3 调度器操作方法

调度器操作方法定义为elavator_ops，在文件include/linux/elevator.h中。

```
00036: struct elevator_ops
00037: {
00038:      elevator_merge_fn    *elevator_merge_fn;
00039:      elevator_merged_fn    *elevator_merged_fn;
00040:      elevator_merge_req_fn *elevator_merge_req_fn;
00041:      elevator_allow_merge_fn *elevator_allow_merge_fn;
00042:      elevator_bio_merged_fn *elevator_bio_merged_fn;
00043:
00044:      elevator_dispatch_fn    *elevator_dispatch_fn;
00045:      elevator_add_req_fn    *elevator_add_req_fn;
00046:      elevator_activate_req_fn *elevator_activate_req_fn;
00047:      elevator_deactivate_req_fn *elevator_deactivate_req_fn;
00048:
00049:      elevator_queue_empty_fn    *elevator_queue_empty_fn;
00050:      elevator_completed_req_fn *elevator_completed_req_fn;
00051:
00052:      elevator_request_list_fn *elevator_former_req_fn;
```

```
00053:      elevator_request_list_fn *elevator_latter_req_fn;
00054:
00055:      elevator_set_req_fn    *elevator_set_req_fn;
00056:      elevator_put_req_fn    *elevator_put_req_fn;
00057:
00058:      elevator_may_queue_fn *elevator_may_queue_fn;
00059:
00060:      elevator_init_fn  *elevator_init_fn;
00061:      elevator_exit_fn  *elevator_exit_fn;
00062:      void (*trim)(struct io_context *);
00063: } ? end elevator_ops ? ;
```

elevator_merge_fn：检查新的请求与已有的请求是否相邻，是否可以合并。

elevator_merge_req_fn：将两个请求合并。

elevator_dispatch_fn：从请求队列中选择下一个将要下发的请求。

elevator_add_req_fn：将一个新的request添加进调度器的队列中。

elevator_queue_empty_fn：检查调度器的队列是否为空。

elevator_set_req_fn和elevator_put_req_fn：分别在创建新请求和将请求所占的空间释放到内存时调用。

elevator_init_fn：用于初始化调度器实例。

3 I/O调度器

如果简单地以内核产生请求的次序直接将请求发给块设备的话，那么块设备性能肯定让人难以接受，磁盘寻址是整个计算机中最慢的操作之一，每一次寻址 - 定位硬盘磁头到特定块上的某个位置 - 需要花费不少时间。所以尽量缩短寻址时间无疑是提高系统性能的关键。

为了优化寻址操作，内核不会一旦接收到I/O请求后，就按照请求的次序发起块I/O请求。而是，它在提交前，先执行合并与排序（merging and sorting），这种操作可以极大地提高系统的整体性能。在内核子系统中，负责以上操作的被称为I/O调度器（I/O scheduler）。

I/O调度器将磁盘I/O资源分配给系统中所有挂起的块I/O请求。具体的说，这种资源分配是通过将请求队列中挂起的请求合并和排序来完成的。注意不要将I/O调度器和进程调度器混淆。进程调度器的作用是将处理器资源分配给系统运行的进程。这两种子系统看起来非常相似，但并不相同。进程调度器和I/O调度器都是将虚拟资源分配给多个对象，对进程调度器来说，处理器被系统中运行的进程共享、这种虚拟提供给用户的就是多任务和分时操作系

统，像Unix系统、相反，I/O调度器虚拟块设备给多个磁盘请求，以便降低磁盘寻址时间，确保磁盘性能的最优化。

3.1 I/O调度器的工作

I/O调度器的工作是**管理块设备的请求队列**。它决定队列中的请求排列顺序及在什么时候派发请求到块设备。这样做有利于减少磁盘寻址时间，从而提高全局吞吐量。注意“全局”的的含义，I/O调度器可能为了提高系统整体性能，而对某些请求不公。

I/O调度器通过两种方法减少磁盘寻址时间：合并（merging）与排序 （sorting）。合并指将两个或多个请求结合成一个新请求。考虑一下这种情况，文件系统提交请求到请求队列—从文件中读取一个数据区，如果这时队列中已经存在一个请求，它访问的磁盘扇区和当前请求访问的磁盘扇区相邻，那么这两个请求就可以合并为一个对单个或多个相邻磁盘扇区操作的新请求。通过合并请求，I/O调度器将多次请求的开销压缩为一次请求的开销。更重要的是，请求合并后只需要传递给磁盘一条寻址命令，就可以访问到请求合并前必须多次寻址才能访问完的磁盘区域了，因此合并请求显然能减少系统开销和磁盘寻址次数。

假设在读请求被提交给请求队列的时候，队列中并没有其他请求需要操作相邻的扇区，此时就无法将当前请求和其他请求合并，当然，可以将其插入请求队列的尾部。但是如果有其他请求需要操作磁盘上类似的位置呢？如果存在一个请求，它要操作的磁盘扇区位置与当前请求的比较接近，那么是不是该让这两个请求在请求队列上也相邻呢？事实上，I/O调度器的确是这样处理上述情况的，整个请求队列将按扇区增长方向有序排列。使所有请求按硬盘上扇区的排列顺序有序排列（尽可能的），目的不仅是为了缩短单独一次请求的寻址时间，更重要的优化在于，通过保持磁盘头以直线方向移动，缩短了所有请求的磁盘寻址时间。该排序算法类似于电梯调度—电梯不能随意的从一层跳到另一层，它应该向一个方向移动，当抵达了同一方向的最后一层后，再掉头向另一个方向移动。处于这种相似特性，I/O调度器也被称为电梯调度。

3.2 调度算法

3.2.1 CFQ调度器

完全公正排队I/O调度器（Complete Fair Queuing）是为专有工作负荷设计的，不过在实际中，也为多种工作负荷提供了良好的性能。

CFQ I/O调度器把进入的I/O请求放入特定的队列中，这种队列是根据引起I/O请求的进程组织的。例如，来自foo进程的I/O请求进入foo队列，而来自bar进程的请求进入bar队列。在每个队列中，刚进入的请求与相邻请求合并在一起，并进行插入分类。队列由此按扇区方式分类，这与其他I/O调度器队列相似。**CFQ I/O调度器的差异在于每一个提交I/O的进程都有自己的队列**。

CFQ I/O调度器以时间片轮转调度队列，从每个队列中选取请求数（默认值为4，可以进行配置），然后进行下一轮调度。这就在进程级提供了公平，确保在每个进程接收公平的磁盘带宽片段。预定的工作负荷是多媒体，在这种媒体中，这种公平的算法可以得到保证，比如，音频播放器总能够及时从磁盘再填满它的音频缓冲区。不过，CFQ I/O调度器在很多场合都能很好地执行。

完全公正排队I/O调度器源码位于block/cfq-iosched.c。尽管这主要推荐给桌面工作负荷使用，但是如果没有其他特殊情况，它在几乎所有的工作负荷中都能很好地执行。

3.2.2 Deadline调度器

Deadline调度器对一个请求的多方面特性进行权衡来进行调度，对读写request进行了分类管理，并且在调度处理的过程中读请求具有较高优先级。因为读请求往往是同步操作，对延迟时间比较敏感，而写操作往往是异步操作，可以尽可能的将相邻访问地址的请求进行合并，但是，合并的效率越高，延迟时间会越长。因此，为了区别对待读写请求类型，deadline采用两条链表对读写请求进行分类管理。

引入分类管理之后，在读优先的情况下，写请求如果长时间得到不到调度，会出现饿死的情况，如当应用程序频繁访问文件的一部分而此时若有另一个远端的请求，那么这个请求将会在很长一段时间内得不到响应，这显然是不合理的。因此，deadline算法考虑了写饿死的情况，从而保证在读优先调度的情况下，写请求不会被饿死。即能满足块设备扇区的顺寻访问又兼顾到一个请求不会在队列中等待太久导致饿死。

Deadline调度器引入了四个队列，这四个队列可分为两类，每一类都由读和写两种队列组成。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。

与deadline相关的请求调度发送函数是deadline_dispatch_requests（），源码在文件

block/deadline-iosched.c中。

```
00239: /*
00240:  * deadline_dispatch_requests selects the best request according to
00241:  * read/write expire, fifo_batch, etc
00242:  */
00243: static int deadline_dispatch_requests(struct request_queue *q, int force)
00244: {
00245:     struct deadline_data *dd = q->elevator->elevator_data;
00246:     const int reads = !list_empty(&dd->fifo_list[READ]);
00247:     const int writes = !list_empty(&dd->fifo_list[WRITE]);
00248:     struct request *rq;
00249:     int data_dir;
00250:
00251:     /*
00252:      * batches are currently reads XOR writes
00253:      */
00254:     if (dd->next_rq[WRITE])
00255:         rq = dd->next_rq[WRITE];
00256:     else
00257:         rq = dd->next_rq[READ];
00258:
00259:     if (rq && dd->batching < dd->fifo_batch)
00260:         /* we have a next request are still entitled to batch */
00261:         goto ↓dispatch_request;
00262:
00263:     /*
00264:      * at this point we are not running a batch. select the appropriate
00265:      * data direction (read / write)
00266:      */
00267:
00268:     if (reads) {
00269:         BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[READ]));
00270:
00271:         if (writes && (dd->starved++ >= dd->writes_starved))
00272:             goto ↓dispatch_writes;
00273:
00274:         data_dir = READ;
00275:
00276:         goto ↓dispatch_find_request;
00277:     }
00278:
00279:     /*
00280:      * there are either no reads or writes have been starved
00281:      */
00282:
00283:     if (writes) {
00284: dispatch_writes:
00285:         BUG_ON(RB_EMPTY_ROOT(&dd->sort_list[WRITE]));
00286:
00287:         dd->starved = 0;
00288:
00289:         data_dir = WRITE;
00290:
00291:         goto ↓dispatch_find_request;
00292:     }
00293:
00294:     return 0;
00295:
00296: dispatch_find_request:
00297:     /*
00298:      * we are not running a batch, find best request for selected data_dir
```

```
00299:      */
00300:     if (deadline_check_fifo(dd, data_dir) || !dd->next_rq[data_dir]) {
00301:         /*
00302:          * A deadline has expired, the last request was in the other
00303:          * direction, or we have run out of higher- sectored requests.
00304:          * Start again from the request with the earliest expiry time.
00305:          */
00306:         rq = rq_entry_fifo(dd->fifo_list[data_dir].next);
00307:     } else {
00308:         /*
00309:          * The last req was the same dir and we have a next request in
00310:          * sort order. No expired requests so continue on from here.
00311:          */
00312:         rq = dd->next_rq[data_dir];
00313:     }
00314:
00315:     dd->batching = 0;
00316:
00317: dispatch_request:
00318:     /*
00319:     * rq is the selected appropriate request.
00320:     */
00321:     dd->batching++;
00322:     deadline_move_request(dd, rq);
00323:
00324:     return 1;
00325: } ? end deadline_dispatch_requests ?
```

读写请求被分成了两个队列，并且采用两种方式将这些request管理起来。一种是采用红黑树（RB tree）的方式将所有request组织起来，通过request的访问地址作为索引；另一种方式是采用队列的方式将request管理起来，所有的request采用先来后到的方式进行排序，即FIFO队列。每个request会被分配一个time stamp，这样就可以知道这个request是否已经长时间没有得到调度，需要优先处理。在请求调度的过程中，读队列是优先得到处理的，除非写队列长时间没有得到调度，存在饿死的状况。

在请求处理的过程中，deadline算法会优先处理那些访问地址临近的请求，这样可以最大程度的减少磁盘抖动的可能性。只有在有些request即将被饿死的时候，或者没有办法进行磁盘顺序化操作的时候，deadline才会放弃地址优先策略，转而处理那些即将被饿死的request。

总之，deadline算法对request进行了优先权控制调度，包括以下几个方面：

（1）读写请求分离，读请求具有高优先调度权，除非写请求即将被饿死的时候，才会去调度处理写请求。这种处理可以保证读请求的延迟时间最小化。

（2）对请求的顺序批量处理。对那些地址临近的顺序化请求，deadline给予了高优先级处理权。例如一个写请求得到调度后，其临近的request会在紧接着的调度过程中被处理掉。这种顺序批量处理的方法可以最大程度的减少磁盘抖动。

（3）保证每个请求的延迟时间。每个请求都赋予了一个最大延迟时间，如果达到延迟时间的上限，那么这个请求就会被提前处理掉，此时，会破坏磁盘访问的顺序化特征，回影响性能，但是，保证了每个请求的最大延迟时间。

3.2.3 I/O调度器的选择

除了前面介绍的两种调度器外，Linux内核中还存在其他调度器，如预测（anticipatory）I/O调度器、空操作（Noop）I/O调度器等。

在2.6内核中有四种不同的I/O调度器。其中的每一种I/O调度器都可以被启用，并内置在内核中。作为默认，块设备使用预测I/O调度器。在内核启动时，可以通过命令选项elevator=foo来覆盖默认，这里foo是一个有效而激活的I/O调度器。

可以通过修改内核参数来动态调整块设备的调度算法。

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

表4 给定elevator选项的参数

参数	调度器
as	Anticipatory
cfq	Complete Fair Queuing
deadline	Deadline
noop	Noop

4 发送请求到I/O调度器

4.1 __make_request（）

在Linux通用块设备层介绍向通用块设备发送请求，generic_make_request（）触发请求队列的make_request_fn方法来发送一个请求到I/O调度器。对于块设备，该方法通常由__make_request（）实现，代码在文件driver/block/ll_rw_blk.c中。它接收的参数有request_queue描述符q和bio描述符bio。

```
01365: static int __make_request(struct request_queue *q, struct bio *bio)
01366: {
01367:     struct request *req;
01368:     int el_ret;
01369:     unsigned int bytes = bio->bi_size;
01370:     const unsigned short prio = bio_prio(bio);
01371:     const bool sync = bio_rw_flagged(bio, BIO_RW_SYNCIO);
01372:     const bool unplug = bio_rw_flagged(bio, BIO_RW_UNPLUG);
01373:     const unsigned int ff = bio->bi_rw & REQ_FAILFAST_MASK;
01374:     int where = ELEVATOR_INSERT_SORT;
01375:     int rw_flags;
01376:
01377:     /* BIO_RW_BARRIER is deprecated */
01378:     if (WARN_ONCE(bio_rw_flagged(bio, BIO_RW_BARRIER),
01379:         "block: BARRIER is deprecated, use FLUSH/FUA instead\n")) {
01380:         bio_endio(bio, -EOPNOTSUPP);
01381:         return 0;
01382:     }
01383:
01384:     /*
01385:      * low level driver can indicate that it wants pages above a
01386:      * certain limit bounced to low memory (ie for highmem, or even
01387:      * ISA dma in theory)
01388:      */
01389:     blk_queue_bounce(q, &bio);
01390:
01391:     spin_lock_irq(q->queue_lock);
01392:
01393:     if (bio->bi_rw & (BIO_FLUSH | BIO_FUA)) {
01394:         where = ELEVATOR_INSERT_FLUSH;
01395:         goto ↓get_rq;
01396:     }
01397:
01398:     if (elv_queue_empty(q))
01399:         goto ↓get_rq;
01400:
01401:     el_ret = elv_merge(q, &req, bio);
01402:     switch (el_ret) {
01403:     case ELEVATOR_BACK_MERGE:
01404:         BUG_ON(!rq_mergeable(req));
01405:
01406:         if (!ll_back_merge_fn(q, req, bio))
01407:             break;
01408:
01409:         trace_block_bio_backmerge(q, bio);
01410:
01411:         if ((req->cmd_flags & REQ_FAILFAST_MASK) != ff)
01412:             blk_rq_set_mixed_merge(req);
01413:
01414:         req->biotail->bi_next    = bio;
01415:         req->biotail = bio;
01416:         req->__data_len += bytes;
01417:         req->ioprio = ioprio_best(req->ioprio, prio);
01418:         if (!blk_rq_cpu_valid(req))
01419:             req->cpu = bio->bi_comp_cpu;
01420:         drive_stat_acct(req, 0);
01421:         elv_bio_merged(q,    req, bio);
01422:         if (!attempt_back_merge(q, req))
01423:             elv_merged_request(q, req, el_ret);
01424:         goto ↓out;
01425:
01426:     case ELEVATOR_FRONT_MERGE:
01427:         BUG_ON(!rq_mergeable(req));
```

```
01428:
01429:  if (!ll_front_merge_fn(q, req, bio))
01430:              break;
01431:
01432:          trace_block_bio_frontmerge(q, bio);
01433:
01434:          if ((req->cmd_flags & REQ_FAILFAST_MASK) != ff) {
01435:              blk_rq_set_mixed_merge(req);
01436:              req->cmd_flags &= ~REQ_FAILFAST_MASK;
01437:              req->cmd_flags |= ff;
01438:          }
01439:
01440:          bio->bi_next = req->bio;
01441:          req->bio = bio;
01442:
01443:          /*
01444:           * may not be valid. if the low level driver said
01445:           * it didn't need a bounce buffer then it better
01446:           * not touch req->buffer either...
01447:           */
01448:          req->buffer = bio_data(bio);
01449:          /*
01450:           * The merge may happen accross partitions
01451:           * We must update in_flight value accordingly
01452:           */
01453:          blk_account_io_front_merge(req, bio->bi_sector);
01454:          req->__sector = bio->bi_sector;
01455:          req->__data_len += bytes;
01456:          req->ioprio = ioprio_best(req->ioprio, prio);
01457:          if (!blk_rq_cpu_valid(req))
01458:              req->cpu = bio->bi_comp_cpu;
01459:          drive_stat_acct(req, 0);
01460:          elv_bio_merged(q,    req, bio);
01461:          if (!attempt_front_merge(q, req))
01462:              elv_merged_request(q, req, el_ret);
01463:          goto ↓out;
01464:
01465:      /* ELV_NO_MERGE: elevator says don't/can't merge. */
01466:      default:
01467:          ;
01468:      } « end switch el_ret »
01469:
01470: get_rq:
01471:      /*
01472:       * This sync check and mask will be re-done in init_request_from_bio(),
01473:       * but we need to set it earlier to expose the sync flag to the
01474:       * rq allocator and io schedulers.
01475:       */
01476:      rw_flags = bio_data_dir(bio);
01477:      if (sync)
01478:          rw_flags |= REQ_SYNC;
01479:
01480:      /*
01481:       * Grab a free request. This is might sleep but can not fail.
01482:       * Returns with the queue unlocked.
01483:       */
01484:      req = get_request_wait(q, rw_flags, bio);
01485:      if (unlikely(! req)) {
01486:          bio_endio(bio, -ENODEV);      /* @q is dead */
01487:          goto ↓out_unlock;
01488:      }
01489:
01490:      /*
01491:       * After dropping the lock and possibly sleeping here, our request
01492:       * may now be mergeable after it had proven unmergeable (above).
01493:       * We don't worry about that case for efficiency. It won't happen
```

```
01494:      * often, and the elevators are able to handle it.
01495:     */
01496:     init_request_from_bio(req, bio);
01497:
01498:     spin_lock_irq(q->queue_lock);
01499:     if (test_bit(QUEUE_FLAG_SAME_COMP, &q->queue_flags) ||
01500:       bio_flagged(bio,   BIO_CPU_AFFINE))
01501:         req->cpu = raw_smp_processor_id();
01502:
01503:     if (queue_should_plug(q) && elv_queue_empty(q))
01504:         blk_plug_device(q);
01505:
01506:     /* insert the request into the elevator */
01507:     drive_stat_acct(req, 1);
01508:     __elv_add_request(q, req, where, 0);
01509: out:
01510:     if (unplug || !queue_should_plug(q))
01511:         __generic_unplug_device(q);
01512: out_unlock:
01513:     spin_unlock_irq(q->queue_lock);
01514:     return 0;
01515: } « end ____make_request »
```

__make_request（）主要的操作如下：

（1）若需要，调用blk_queue_bounce（）设置一个弹性缓冲区（bounce buffer）。如果创建了弹性缓冲区，__make_request()就在该缓冲区上操作，而不是原来的bio。（1389行）

（2）调用I/O调度器的的函数elv_queue_empty（）（1398行），检查请求队列中是否有等待的请求。注意：派遣队列可能是空的，但I/O调度器中的其他队列可能包含等待的请求。若没有等待的请求，则调用blk_plug_device（）来插入请求队列，然后跳转到步骤（5）。

（3）此时，请求队列中有等待的请求。调用I/O调度器请求合并函数elv_merge（）（1401行）来检查当前的bio可以合并到已存在的请求中。合并时，根据相应的调度器e->ops->elevator_merge_fn(q, req, bio)进行合并请求，该函数有三个可能返回值。

- ELEVATOR_NO_MERGE：bio不能与已存在的请求合并，这种情况下，跳转到步骤（5）；
- ELEVATOR_BACK_MERGE：bio可以作为最后一个bio加入到某个请求req。这种情况下，函数调用q->back_merge_fn来检查请求是否可以扩展。若不能扩展，则跳转到步骤（5）；否则，就将bio插入到请求req链表的尾部并更新req的相关成员变量。然后，尝试合并该请求和下面的请求（新的bio可能是两个请求的空洞）。
- ELEVATOR_FRONT_MERGE：bio可以加入到请求req，作为第一个bio。这

种情况下，调用q->front_merger_fn方法来检查请求是否可以扩展。若不能，则跳转到步骤（5）；否则将新的bio插入到请求req链表的头部，并更新req的相关成员变量。

（4）bio被合并到已存在的请求中，跳转到步骤8（1509行），函数结束。

（5）此时，bio必须插入到一个新请求中。分配一个新请求描述符。若没有空闲的内存，则挂起当前进程，除非bio->bi_rw中的BIO_RW_AHEAD标志设置。该标志的函数是预读（read-ahead）。这种情形下，调用bio_endio（）并结束：数据传输并不会执行。bio_endio（）函数已在《Linux通用块设备层》中介绍。

（6）初始化新请求描述符中的成员变量：

- 初始化存储扇区数、当前bio、当前段（segment）。
- 设置flags的REQ_CMD（是一个正常的读或写操作）。
- 若页面中的第一个bio段是在低端内存，则设置成员变量buffer的值为该缓冲区的线性地址。
- 设置成员变量rq_disk的值为bio->bdev->bd_disk。
- 将bio插入到请求链表中。
- 设置成员变量start_time的值为jiffies。

（7）更新硬盘统计信息（如读写扇区数量），然后将请求加入调度器（elevator）中（1507～1508 行）。

（8）所有工作结束。但在结束前，检查bio->bi_rw的BIO_RW_UNPLUG标志是否设置且队列不应激活（1510行）。若条件满足，则调用generic_unplug_device（）在请求队列上断开（unplug）队列。

（9）结束。

若在调用__make_request（）之前，请求队列非空；则请求队列要么已经断开（unplug），要么即将被断开；因为每个插入（plug）的请求队列q有一个动态计时器q->unplug_timer。同时，若请求队列为空，__make_request（）插入它。请求队列也迟早会断开。不管处于何种情况，块设备驱动的服务例程（strategy routine）会处理派遣队列上的请求。

4.2 __elv_add_request（）

在4.1节，我们分析了__make_request（），假设执行到第（7）步，即调用__elv_add_request（）将bio插入到一个新的请求中，（__make_request（）中的1508行），

代码在文件block/elevator.c中。

```
00676: void __elv_add_request(struct request_queue *q, struct request *rq,
00677:                int where, int plug)
00678: {
00679:     if (rq->cmd_flags & (REQ_SOFTBARRIER | REQ_HARDBARRIER))    {
00680:         /* barriers are scheduling boundary, update end_sector */
00681:         if (rq->cmd_type == REQ_TYPE_FS ||
00682:             (rq->cmd_flags & REQ_DISCARD)) {
00683:             q->end_sector = rq_end_sector(rq);
00684:             q->boundary_rq = rq;
00685:         }
00686:     } else if (!(rq->cmd_flags & REQ_ELVPRIV) &&
00687:             where == ELEVATOR_INSERT_SORT)
00688:         where = ELEVATOR_INSERT_BACK;
00689:
00690:     if (plug)
00691:         blk_plug_device(q);
00692:
00693:     elv_insert(q, rq, where);
00694: }
00695: EXPORT_SYMBOL(__elv_add_request);
00696:
```

__elv_add_request（）主要为对elv_insert（）函数进行封装，该函数也在文件block/elevator.c中，通过该函数可以将request加入到elevator request queue或者device request queue中。

elv_insert（）函数根据请求插入的位置来处理，where的值定义在文件include/linux/elevator.h中。

```
00170: /*
00171:  * Insertion selection
00172:  */
00173: #define ELEVATOR_INSERT_FRONT    1
00174: #define ELEVATOR_INSERT_BACK     2
00175: #define ELEVATOR_INSERT_SORT     3
00176: #define ELEVATOR_INSERT_REQUEUE  4
00177: #define ELEVATOR_INSERT_FLUSH    5
```

在__make_request（）中，首先将where的默认值设为ELEVATOR_INSERT_SORT，即请求需要排序。若bio请求的标志为BIO_FLUSH或BIO_FUA（__make_request（）1393行），则将where的值更改为ELEVATOR_INSERT_FLUSH。

ELEVATOR_INSERT_FRONT：将request加入到device request queue的队列前。

ELEVATOR_INSERT_BACK：将request加入到device request queue的队列尾部。

ELEVATOR_INSERT_SORT：尝试对request进行合并操作。

ELEVATOR_INSERT_REQUEUE：将request重新加入队列中。

ELEVATOR_INSERT_FLUSH：插入新的FLUSH/FUA request。

```
00598: void elv_insert(struct request_queue *q, struct request *rq, int where)
00599: {
00600:     int unplug_it = 1;
00601:
00602:     trace_block_rq_insert(q, rq);
00603:
00604:     rq->q = q;
00605:
00606:     switch (where) {
00607:     case ELEVATOR_INSERT_REQUEUE:
00608:     /*
00609:     * Most requeues happen because of a busy condition,
00610:          * don't force unplug of the queue for that case.
00611:          * Clear unplug_it and fall through.
00612:          */
00613:         unplug_it = 0;
00614:
00615:     case ELEVATOR_INSERT_FRONT:
00616:         rq->cmd_flags |= REQ_SOFTBARRIER;
00617:         list_add(&rq->queuelist,    &q->queue_head);
00618:         break;
00619:
00620:     case ELEVATOR_INSERT_BACK:
00621:         rq->cmd_flags |= REQ_SOFTBARRIER;
00622:         elv_drain_elevator(q);
00623:         list_add_tail(&rq->queuelist, &q->queue_head);
00624:         /*
00625:          * We kick the queue here for the following reasons.
00626:          * - The elevator might have returned NULL previously
00627:          *     to delay requests and returned them now.    As the
00628:          *     queue wasn't empty before this request, ll_rw_blk
00629:          *     won't run the queue on return, resulting in hang.
00630:          * - Usually, back inserted requests won't be merged
00631:          *     with anything.    There's no point in delaying queue
00632:          *     processing.
00633:          */
00634:         __blk_run_queue(q);
00635:         break;
00636:
00637:     case ELEVATOR_INSERT_SORT:
00638:         BUG_ON(rq->cmd_type    != REQ_TYPE_FS &&
00639:             !(rq->cmd_flags & REQ_DISCARD));
00640:         rq->cmd_flags |= REQ_SORTED;
00641:         q->nr_sorted++;
00642:         if (rq_mergeable(rq)) {
00643:             elv_rqhash_add(q,    rq);
00644:             if (!q->last_merge)
00645:                 q->last_merge = rq;
00646:         }
0647:
00648:         /*
00649:          * Some ioscheds (cfq) run q->request_fn directly, so
00650:          * rq cannot be accessed after calling
00651:          * elevator_add_req_fn.
00652:          */
00653:         q->elevator->ops->elevator_add_req_fn(q, rq);
00654:         break;
00655:
00656:     case ELEVATOR_INSERT_FLUSH:
00657:         rq->cmd_flags |= REQ_SOFTBARRIER;
00658:         blk_insert_flush(rq);
00659:         break;
00660:
00661:     default:
```

```
00662:          printk(KERN_ERR   "%s: bad insertion point %d\n",
00663:                 __func__, where);
00664:          BUG();
00665:      } « end switch where »
00666:
00667:      if (unplug_it && blk_queue_plugged(q)) {
00668:          int nrq = q->rq.count[BLK_RW_SYNC] + q->rq.count[BLK_RW_ASYNC]
00669:                  - queue_in_flight(q);
00670:
00671:          if (nrq >= q->unplug_thresh)
00672:              __generic_unplug_device(q);
00673:      }
00674: } « end elv_insert »
00675:
```

elv_drain_elevator（）循环调用elevator_dispatch_fn。__blk_run_queue（）运行设备的队列，执行块设备驱动的请求队列处理例程，如SCSI设备的请求队列处理例程为scsi_request_fn（）。

blk_insert_flush（）插入新的FLUSH/FUA请求。

4.3 __blk_run_queue（）

```
00442: /**
00443: * _____blk_run_queue - run a single device queue
00444: * @q:       The queue to run
00445: *
00446: * Description:
00447: *      See @blk_run_queue. This variant must be called with the queue lock
00448: *      held and interrupts disabled.
00449: *
00450: */
00451: void __blk_run_queue(struct request_queue *q)
00452: {
00453:      blk_remove_plug(q);
00454:
00455:      if (unlikely(blk_queue_stopped(q)))
00456:          return;
00457:
00458:      if (elv_queue_empty(q))
00459:          return;
00460:
00461:      q->request_fn(q);
00462: }
00463: EXPORT_SYMBOL(__blk_run_queue);
```

__blk_run_queue（）函数调用块设备驱动服务例程q->request_fn（）来直接处理请求，进行数据读写。

4.4 驱动服务例程request_fn

请求队列处理例程通常是在空的请求队列上插入新的请求后启动。一旦处理例程被激活，块设备驱动程序就会处理请求队列中的请求，直到队列为空。

对于SCSI设备（SAS或RAID控制器下面的硬盘，都为SCSI方式），请求处理函数为scsi_request_fn（），即由该函数来负责向SAS/RAID控制器发送上层的I/O请求。我们以LSISAS2308为例，了解SCSI设备请求处理函数scsi_request_fn（）初始化位置。以下是LSISAS2308驱动初始化时，__scsi_alloc_queue（）函数调用栈。

```
[<ffffffff8136b5ec>] ? __scsi_alloc_queue+0x13c/0x160
[<ffffffff8136b62d>] ? scsi_alloc_queue+0x1d/0x70
[<ffffffff8136e027>] ? scsi_alloc_sdev+0x1a7/0x250
[<ffffffff8136e6c2>] ? scsi_probe_and_add_lun+0x522/0xed0
[<ffffffff81162fb5>] ? kmem_cache_alloc_notrace+0x115/0x130
[<ffffffffa0029801>] ? _scsih_target_alloc+0x131/0x240 [mpt2sas]
[<ffffffff8134bd09>] ? get_device+0x19/0x20
[<ffffffff8136de4a>] ? scsi_alloc_target+0x29a/0x2d0
[<ffffffff8136f191>] ? __scsi_scan_target+0x121/0x750
[<ffffffff8134dc41>] ? device_create+0x31/0x40
[<ffffffff812623ea>] ? bsg_register_queue+0x14a/0x1d0
[<ffffffff8125783d>] ? blk_init_queue_node+0x4d/0x80
[<ffffffff8136ff05>] ? scsi_scan_target+0xd5/0xf0
[<ffffffffa000a68e>] ? sas_rphy_add+0x10e/0x160 [scsi_transport_sas]
[<ffffffffa002f2ea>] ? mpt2sas_transport_port_add+0x33a/0xc60 [mpt2sas]
[<ffffffffa00240f2>] ? _scsih_probe_sas+0x82/0x130 [mpt2sas]
[<ffffffffa00242aa>] ? _scsih_scan_finished+0x10a/0x2c0 [mpt2sas]
[<ffffffff8136fa87>] ? do_scsi_scan_host+0x77/0xa0
[<ffffffff8136facc>] ? do_scan_async+0x1c/0x140
[<ffffffff8136fab0>] ? do_scan_async+0x0/0x140
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20
```

```
[<ffffffff8136b5ec>] ? __scsi_alloc_queue+0x13c/0x160
[<ffffffff8136b62d>] ? scsi_alloc_queue+0x1d/0x70
[<ffffffff8136e027>] ? scsi_alloc_sdev+0x1a7/0x250
[<ffffffff8136e6c2>] ? scsi_probe_and_add_lun+0x522/0xed0
[<ffffffff81162fb5>] ? kmem_cache_alloc_notrace+0x115/0x130
[<ffffffffa0029801>] ? _scsih_target_alloc+0x131/0x240 [mpt2sas]
[<ffffffff8134bd09>] ? get_device+0x19/0x20
[<ffffffff8136de4a>] ? scsi_alloc_target+0x29a/0x2d0
```

```
[<ffffffff81370022>] ? __scsi_add_device+0x102/0x120
[<ffffffff81370051>] ? scsi_add_device+0x11/0x30
[<ffffffffa0024271>] ? _scsih_scan_finished+0xd1/0x2c0 [mpt2sas]
[<ffffffff8136fa87>] ? do_scsi_scan_host+0x77/0xa0
[<ffffffff8136facc>] ? do_scan_async+0x1c/0x140
[<ffffffff8136fab0>] ? do_scan_async+0x0/0x140
[<ffffffff81091d66>] ? kthread+0x96/0xa0
[<ffffffff8100c14a>] ? child_rip+0xa/0x20
[<ffffffff81091cd0>] ? kthread+0x0/0xa0
[<ffffffff8100c140>] ? child_rip+0x0/0x20
```

LSISAS2308 SAS控制器在驱动加载时，会调用scsi_scan_target（）函数来扫描设备，该函数会进而调用scsi_alloc_queue（）来初始化请求队列及处理函数scsi_request_fn（）（1697行），源码在driver/scsi/scsi_lib.c中。我们会在《Linux块设备驱动》中详细分析scsi_request_fn（）函数。

```
01693: struct request_queue *scsi_alloc_queue(struct scsi_device *sdev)
01694: {
01695:     struct request_queue *q;
01696:
01697:     q = __scsi_alloc_queue(sdev->host, scsi_request_fn);
01698:     if (!q)
01699:         return NULL;
01700:
01701:     blk_queue_prep_rq(q, scsi_prep_fn);
01702:     blk_queue_softirq_done(q, scsi_softirq_done);
01703:     blk_queue_rq_timed_out(q, scsi_times_out);
01704:     blk_queue_lld_busy(q, scsi_lld_busy);
01705:     return q;
01706: }
```

4.5 请求队列的由来

在第二部分内容请求队列中，已详细介绍数据结构，我们也知道每个块设备都会有自己的请求队列。可能有部分同事有一些疑问，块设备的请求队列是何时创建的？在通用块设备层又是如何从bio结构体从获取块设备的队列？假设系统中有多个块设备，那么通用块设备层和I/O调度层又是如何能将上层的I/O请求加入正确的块设备请求队列中？

4.5.1 块设备请求队列的创建

块设备的请求队列是在设备驱动初始化时创建。由于块设备驱动超出本文讨论范畴，只列出块设备驱动的部分源码。本节以MTD（Memory Technology Device）块设备为例，介绍块设备请求队列的创建。

一个请求队列就是一个动态的数据结构，该结构必须由块设备的I/O子系统创建。创建和初始化请求队列的函数是：

```
00666: struct request_queue *blk_init_queue(request_fn_proc   *rfn,
                                            spinlock_t *lock)
00667: {
00668:      return blk_init_queue_node(rfn, lock, -1);
00669: }
00670: EXPORT_SYMBOL(blk_init_queue);
```

该函数的参数是处理这个队列的rfn函数指针和控制访问队列权限的自旋锁。由于该函数主要负责分配内存（实际上是分配很多的内存），因此可能会失败；所以在使用队列前一定要检查返回值。

MTD块设备驱动的入口为：（drivers/mtd/mtdblock.c）

```
00389: static int __init init_mtdblock(void)
00390: {
00391:      mutex_init(&mtdblks_lock);
00392:
00393:      return register_mtd_blktrans(&mtdblock_tr);
00394: }
00395:
```

就是调用register_mtd_blktrans（）函数，而该函数的实现在drivers/mtd/mtd_blkdevs.c中。

```
00337: int  register_mtd_blktrans(struct mtd_blktrans_ops   *tr)
00338: {
00339:      int ret, i;
00340:
00341:      /* Register the notifier if/when the first device type is
00342:         registered, to prevent the link/init ordering from fucking
00343:         us over. */
00344:      if (!blktrans_notifier.list.next)
00345:          register_mtd_user(&blktrans_notifier);
00346:
        ... ...
00362:
00363:      tr->blkcore_priv->rq = blk_init_queue(mtd_blktrans_request,
&tr->blkcore_priv->queue_lock);
       ... ...
00397:
00398:      mutex_unlock(&mtd_table_mutex);
00399:
00400:      return 0;
00401: } ? end register_mtd_blktrans ?
00402:
```

363行就是MTD块设备请求队列的初始化。初始化过程，这里不再详细介绍。

4.5.2 块设备请求队列的获取

在《Linux通用块设备层》中，我们知道，内核对块设备的I/O请求，都会经过通用块设备层，即都会执行__generic_make_request（）函数。

```
01640: static inline void __generic_make_request(struct bio *bio)
01641: {
01642:     struct request_queue *q;
01643:     sector_t old_sector;
01644:     int ret, nr_sectors = bio_sectors(bio);
01645:     dev_t old_dev;
01646:     int err = -EIO;
01647:
   ...  ...
01666:         q = bdev_get_queue(bio->bi_bdev);
01667:         if (unlikely(! q)) {
01668:             printk(KERN_ERR
01669:                 "generic_make_request: Trying to access "
01670:                 "nonexistent block-device %s (%Lu)\n",
01671:                 bdevname(bio->bi_bdev, b),
01672:                 (long long) bio->bi_sector);
01673:             goto ↓end_io;
01674:         }
    ... ..
```

块设备请求队列的获取，非常简单，在1666行，通过bdev_get_queue（）函数即可（include/linux/blkdev.h）。

```
00814: static inline struct request_queue *bdev_get_queue(struct
            block_device *bdev)
00815: {
00816:     return bdev->bd_disk->queue;
00817: }
```

5 plug与unplug设备

我们理解了I/O调度器的主要作用就是尽可能合并I/O请求，提升存储的吞吐量。合并I/O请求，就意味着前面的请求还在排队，没有被块设备驱动程序处理。问题来了，**如何控制已有的I/O请求不立即被处理**呢？

I/O请求延时处理机制，是通过plugging和unplugging的方式来实现。当处于plugged状态时，块设备驱动不会被激活（可以理解为休眠状态），即使当前队列上有I/O请求，也不会去处理它。unplugged状态就是块设备驱动程序处理激活状态，立即处理请求队列上的所有请求。

plug和unplug的命名容易让人产生误解，unplug的准确含义是activate，而不是简单的

理解拔掉；plug的准确含义是deactivate。我们可以拿开关灯为例来解释plug/unplug；plug就是关掉块设备驱动，相当于按下照明灯的“关”按钮，灯灭了；而unplug就是打开块设备驱动，相当于按下照明灯的“开”按钮，灯亮了。从这个比喻上来说，内核中的unplug/plug的作用和英文含义恰恰相反，这导致我过去一直困惑。

plug就是让块设备驱动休息一会，unplug让块设备驱动马上干活。

5.1 blk_plug_device（）与blk_remove_plug（）

blk_plug_device（）函数设置请求队列的标志为QUEUE_FLAG_PLUGGED，并重启定时器q->unplug_timer。目的是plug一个块设备，即让块设备驱动休息一会。

块设备驱动不能无限期休息，若休息超过q->unplug_delay（通常为3ms），q->unplug_timer定时器就会超时，接着调用blk_unplug_timeout（），触发内核线程kblockd来唤醒kblockd_workqueue。

klockd内核线程执行路径为q->unplug_work（）-> q->unplug_fn（） -> generic_unplug_device（）。接下来我们来分析generic_unplug_device（）。

```
00197: /*
00198:  * "plug" the device if there are no outstanding requests: this will
00199:  * force the transfer to start only after we have put all the requests
00200:  * on the list.
00201:  *
00202:  * This is called with interrupts off and no requests on the queue and
00203:  * with the queue lock held.
00204:  */
00205: void blk_plug_device(struct request_queue *q)
00206: {
00207:     WARN_ON(!irqs_disabled());
00208:
00209:     /*
00210:      * don't plug a stopped queue, it must be paired with blk_start_queue()
00211:      * which will restart the queueing
00212:      */
00213:     if (blk_queue_stopped(q))
00214:         return;
00215:
00216:     if (!queue_flag_test_and_set(QUEUE_FLAG_PLUGGED, q)) {
00217:         mod_timer(&q->unplug_timer, jiffies + q->unplug_delay);
00218:         trace_block_plug(q);
00219:     }
00220: }

00221: EXPORT_SYMBOL(blk_plug_device);
```

blk_remove_plug（）就是清除队列QUEUE_FLAG_PLUGGED标志，并取消q->unplug_timer定时器。当所有可合并的请求加入到队列后，内核可以直接调用这个函数。

当请求队列上等待的I/O请求数量超过q-> unplug_thresh时（默认值为4），调度器就会unplug请求队列。

```
00241: /*
00242:  * remove the queue from the plugged list, if present. called with
00243:  * queue lock held and interrupts disabled.
00244:  */
00245: int blk_remove_plug(struct request_queue *q)
00246: {
00247:     WARN_ON(!irqs_disabled());
00248:
00249:     if (!queue_flag_test_and_clear(QUEUE_FLAG_PLUGGED, q))
00250:         return 0;
00251:
00252:     del_timer(&q->unplug_timer);
00253:     return 1;
00254: }
00255: EXPORT_SYMBOL(blk_remove_plug);
```

5.2 generic_unplug_device（）

generic_unplug_device（）函数功能就是unplug一个块设备，即激活块设备驱动，让它来处理请求队列上的I/O请求。

函数__generic_unplug_device（）是对blk_remove_plug（）和q->request_fn（）的封装。

```
00317: /**
00318:  * generic_unplug_device - fire a request queue
00319:  * @q:      The &struct request_queue in question
00320:  *
00321:  * Description:
00322:  *     Linux uses plugging to build bigger requests queues before letting
00323:  *     the device have at them. If a queue is plugged, the I/O scheduler
00324:  *     is still adding and merging requests on the queue. Once the queue
00325:  *     gets unplugged, the request_fn defined for the queue is invoked and
00326:  *     transfers started.
00327:  **/
00328: void generic_unplug_device(struct request_queue *q)
00329: {
00330:     if (blk_queue_plugged(q)) {
00331:         spin_lock_irq(q->queue_lock);
00332:         __generic_unplug_device(q);
00333:         spin_unlock_irq(q->queue_lock);
00334:     }
00335: }
00336: EXPORT_SYMBOL(generic_unplug_device);

00307: void __generic_unplug_device(struct request_queue *q)
00308: {
00309:     if (unlikely(blk_queue_stopped(q)))
00310:         return;
00311:     if (!blk_remove_plug(q) && !blk_queue_nonrot(q))
```

```
00312:          return;
00313:
00314:      q->request_fn(q);
00315: }
00316:
```

6 块设备请求队列拥塞的处理

想象一下这种场景，用户进程向文件中不断写入数据，不断下发I/O请求。

问题：

（1）用户进程可以无限制地、快速下发I/O请求吗？

用户进程下发I/O请求是毫不费力气，但硬盘响应请求就没那么快了。

（2）内核检测到拥塞时，是如何处理的？

当块设备请求队列被填满时，就会迫使让进程处于I/O等待状态。当等待的I/O请求数量超过request_queue->nr_congestion_on（默认值为113）时，内核就设置队列为拥塞状态，并且尝试降低新请求的创建速度。当拥塞队列上的请求数量低于request_queue->nr_congestion_off（默认值为103）时，解除队列的拥塞状态。

设置队列拥塞的函数为blk_set_queue_full（），检测函数为blk_queue_full（），源码均在include/linux/blkdev.h。

```
00631: static inline int blk_queue_full(struct request_queue *q, int sync)
00632: {
00633:      if (sync)
00634:          return test_bit(QUEUE_FLAG_SYNCFULL, &q->queue_flags);
00635:      return test_bit(QUEUE_FLAG_ASYNCFULL, &q->queue_flags);
00636: }

00638: static inline void blk_set_queue_full(struct request_queue *q,
            int sync)
00639: {
00640:      if (sync)
00641:          queue_flag_set(QUEUE_FLAG_SYNCFULL, q);
00642:      else
00643:          queue_flag_set(QUEUE_FLAG_ASYNCFULL, q);
00644: }

00646: static inline void blk_clear_queue_full(struct request_queue *q,
              int sync)
00647: {
00648:      if (sync)
00649:          queue_flag_clear(QUEUE_FLAG_SYNCFULL, q);
00650:      else
00651:          queue_flag_clear(QUEUE_FLAG_ASYNCFULL, q);
```

```
00652: }
```

每次有新的I/O请求，都通过get_request（）（文件block/blk-core.c）来创建新的request数据结构。893行就是检测块设备请求队列是否超过了阀值，若条件成立，就设置队列拥塞（917行）。 接下来，我们分析队列拥塞后，对内核其他部分的影响。

```
00877: static struct request *get_request(struct request_queue *q,
00878:                 int rw_flags, struct bio *bio, gfp_t gfp_mask)
00879: {
00880:     struct request *rq = NULL;
00881:     struct request_list *rl = &q->rq;
00882:     struct io_context *ioc = NULL;
00883:     const bool is_sync = rw_is_sync(rw_flags) != 0;
00884:     int may_queue;

    ... ...
00893:     if (rl->count[is_sync]+1 >= queue_congestion_on_threshold(q)) {

    ... ...
00917:         blk_set_queue_congested(q, is_sync);
00918:     } « end if rl- >count[is_sync]+1>... »

    ... ...
00973:     if (ioc_batching(q, ioc))
00974:         ioc->nr_batch_requests--;
00975:
00976:     trace_block_getrq(q, bio, rw_flags & 1);
00977: out:
00978:     return rq;
00979: } « end get_request »
```

blk_set_queue_congested（）仅是对set_bdi_congested（）的封装。函数主要工作就是设置块设备对应数据结构backing_dev_info的状态标志，并增加nr_bdi_congested[ ]计数。

设置块设备状态为BDI_sync_congested或BDI_async_congested之后呢？

```
00784: /*
00785:  * A queue has just entered congestion.  Flag that in the queue's VM- visible
00786:  * state flags and increment the global gounter of congested queues.
00787:  */
00788: static inline void blk_set_queue_congested(struct
                request_queue *q, int sync)
00789: {
00790:     set_bdi_congested(&q->backing_dev_info, sync);
00791: }
00792:
```

```
00734: void set_bdi_congested(struct backing_dev_info * bdi, int sync)
00735: {
00736:     enum bdi_state bit;
00737:
00738:     bit = sync ? BDI_sync_congested : BDI_async_congested;
```

```
00739:      if (!test_and_set_bit(bit, &bdi->state))
00740:          atomic_inc(&nr_bdi_congested[sync]);
00741: }
00742: EXPORT_SYMBOL(set_bdi_congested);
```

设置块设备状态为拥塞之后呢？没有然后了！就像向领导汇报一件重要的事情，领导说我知道了，但接下来啥也没做。

对于**异步写**来说，I/O调度器任务就是对来的I/O请求就行排队、合并，但对队列拥塞情况，只能报告，并不能做什么。那么用户进程是否可以无限制下发请求呢？答案是否定的。用户进程每次写入数据，都要消耗内存的（参见Linux内核延迟写机制），随着写入数据量的增加，内存逐渐耗尽；事实上，每次写数据generic_perform_write（）都会调用balance_dirty_pages_ratelimited(mapping)（2385行）来调整内存中的脏页数量。脏页达到一定比例，就会调用io_schedule_timeout（），让用户进程等一会再发新的I/O请求。兄弟，你歇一会，不能再无节制发I/O请求了。

```
02302: static ssize_t generic_perform_write(struct file *file,
           struct iov_iter *i,     loff_t  pos)
02304: {
02305:     struct address_space *mapping = file->f_mapping;
02306:     const struct address_space_operations *a_ops = mapping->a_ops;
02307:     long status = 0;
02308:     ssize_t written = 0;
02309:     unsigned int flags = 0;

       ...

02385:         balance_dirty_pages_ratelimited(mapping);
02386:         if (fatal_signal_pending(current)) {
02387:             status = -EINTR;
02388:             break;
02389:         }
02390:     } « end do » while (iov_iter_count(i));
02391:
02392:     return written ? written : status;
02393: } « end generic_perform_write »
```

对于同步写或读来说，就不会存在请求队列拥塞问题。因为进程会忙等I/O请求的完成。

7 常见问题

7.1 SSD经过I/O调度层吗？

SSD有多种接口和类型，主要有以下三类：

- SATA/SAS
- PCIe

Proprietary、*NVMe*、*SATAe*、*SCSIe*

- DIMM

最常见的SSD为SATA或SAS接口，使用时，要接在SAS或RAID控制器下。这种情况下，Linux内核看到的仍是SCSI设备，SSD读写时，仍然要经过I/O调度层。

我们要清楚，I/O调度器的主要功能是**合并I/O请求，提升读写带宽**。因为传统磁介质硬盘读写，需要移动磁头，且随机与顺序读写性能差别很大；合并I/O请求，可以减少磁头移动次数，提升读写性能。I/O调度器的设计主要针对传统磁介质存储，而SSD的特性和传统磁介质有很大不同，随机与顺序读写性能差别不大；延迟更低、读写带宽更高。I/O调度器对SSD性能提升几乎没有帮助，反而可能带来负面影响（如读写诶延时），因此非SATA/SAS接口的SSD，都不使用I/O调度层。如下图是Linux内核中NVMe SSD的驱动架构，从图中可以看出NVMe SSD I/O路径并不经传统的I/O调度层。

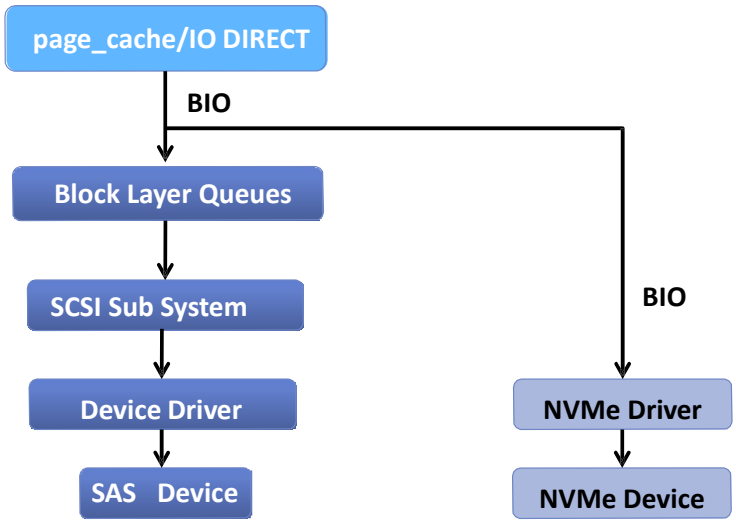

图3 NVMe SSD驱动架构

7.2 I/O请求放到块设备请求队列上后，是否立即被执行呢？

I/O请求放到请求队列上后，只有unplug（activate）块设备驱动，I/O请求才会被真正执行。若块设备处于plug（deactivate）状态，I/O请求就在请求队列上排队等待。