

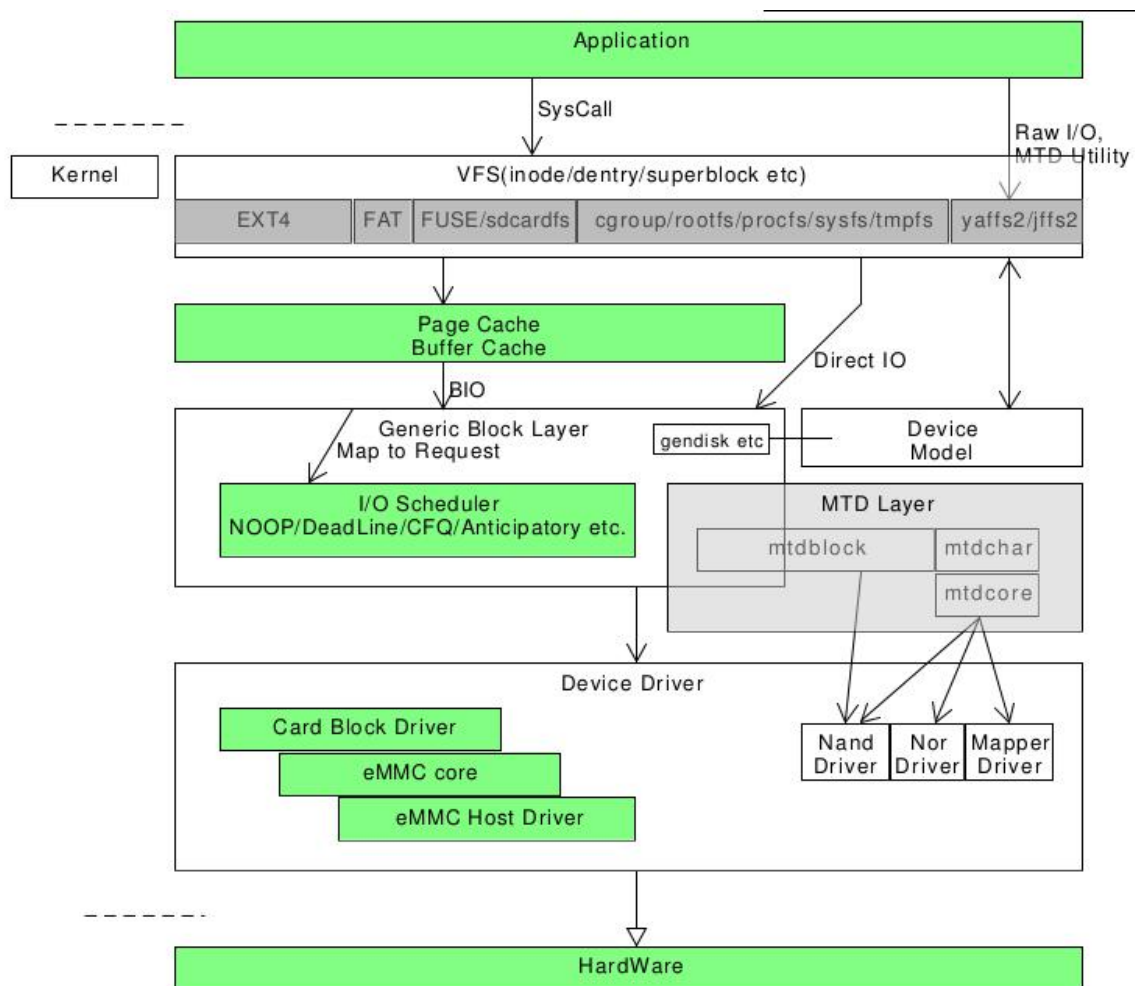
Linux内核中的 Bio 和 I/O Scheduler

Chentong
2016.06.20

Linux内核中的 Bio 和 I/O Scheduler

- Agenda
 - 存储子系统整体架构
 - 块设备的特点及优化考量
 - 块设备子系统相关数据结构
 - 块设备管理相关
 - I/O请求表示相关
 - 调度器实现相关
 - I/O请求的处理
 - I/O的生成
 - I/O的处理
 - I/O调度
 - I/O性能分析和调优

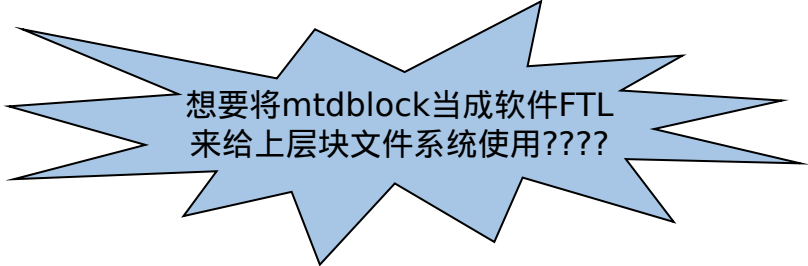
存储子系统整体架构



存储子系统整体架构

- 传统嵌入式设备

- 使用MTD(Memory Technology Devices)技术
 - 支持 NAND/NOR 等原始Flash设备
 - 生产/使用过程会产生坏块(wear-leveling)
 - 每次写入之前必须先进行擦除(all->1)
 - MTD在行为上和块设备类似，但又不能归类到块设备('r+w' VS 'r+w+e')
- 内核透过 MTd Subsystem 加以支持
 - mtdcore
 - 向上提供 mtd_read()/mtd_write() 等一致API
 - 向下通过 struct mtd_info 来封装底下 driver 的不同操作
 - mtdchar
 - 模拟成字符设备，透过 /dev/mtdX 等节点向上层提供服务
 - mtdblock
 - 模拟成块设备，但
 - » 不提供负载均衡(wear-leveling)
 - » 不提供坏块管理
 - » 掉电易丢数据(擦除块内数据实际缓存于RAM中，实际都读写于此)



想要将mtdblock当成软件FTL
来给上层块文件系统使用????

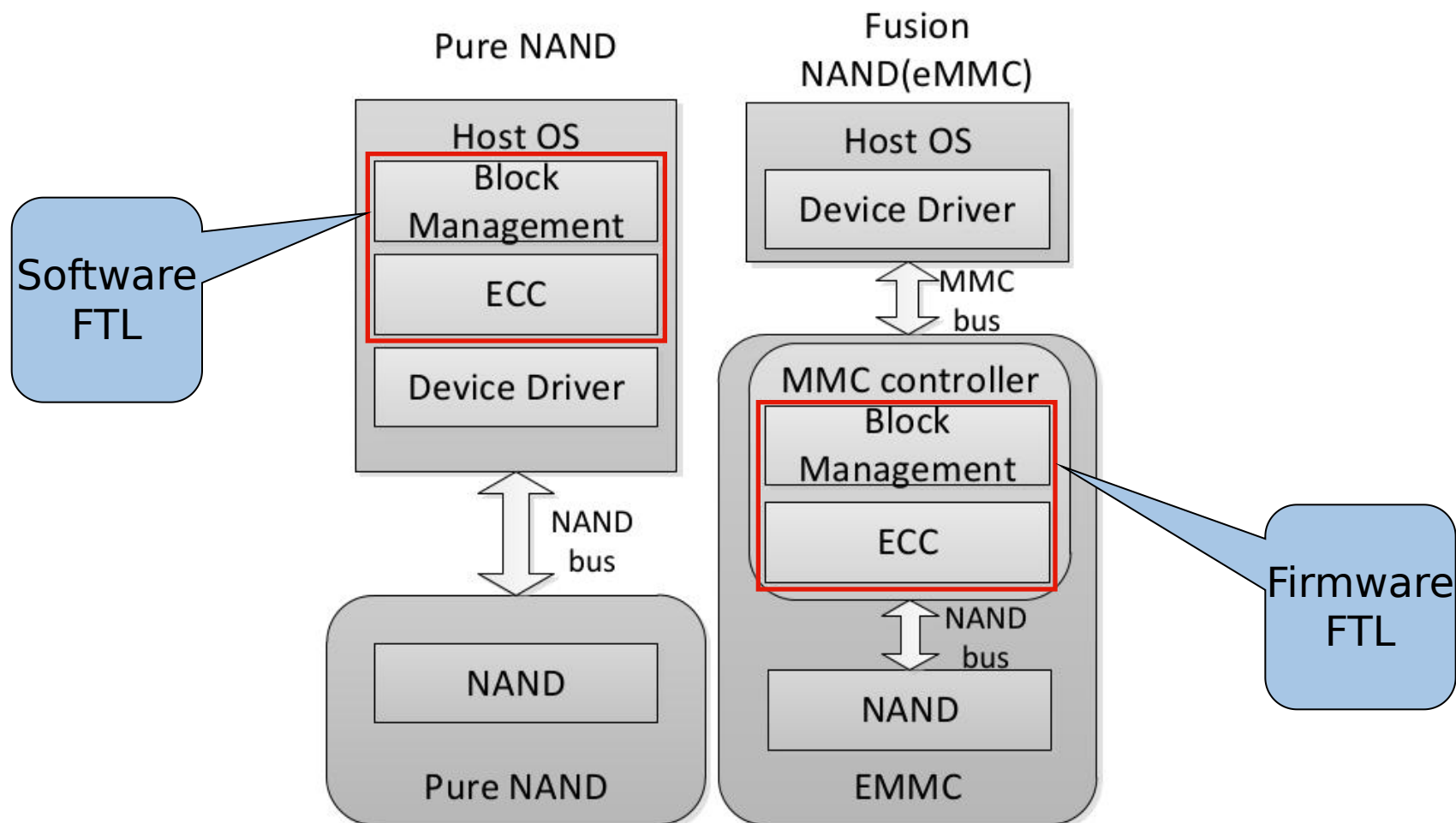
存储子系统整体架构

- 现代手机及嵌入式设备
 - 使用 eMMC (embedded MultiMediaCard)技术
 - 一个BGA封装的芯片，包含：
 - MMC总线接口
 - Flash Memory (NAND)
 - Flash Memory Controller (FTL)
 - » 作为 Firmware 运行
 - » 完成坏块管理、负载均衡、ECC等功能
 - 表现为一个“典型的”块设备
 - 内核将其放入块设备子系统进行管理
 - 向上直接与GBL()交互，接受来自 IO_Scheduler 处理过的I/O请求
 - 向下透过三个 driver 构成的 stack 操作硬件
 - Card Block Driver
 - » 抽象成一个块设备
 - MMC core
 - » 抽象 mmc/sd/sdio 等各种 host 的共性
 - eMMC Host Driver
 - » MMC 总线上主控的 driver

这三部分也
构成
MMC子系统

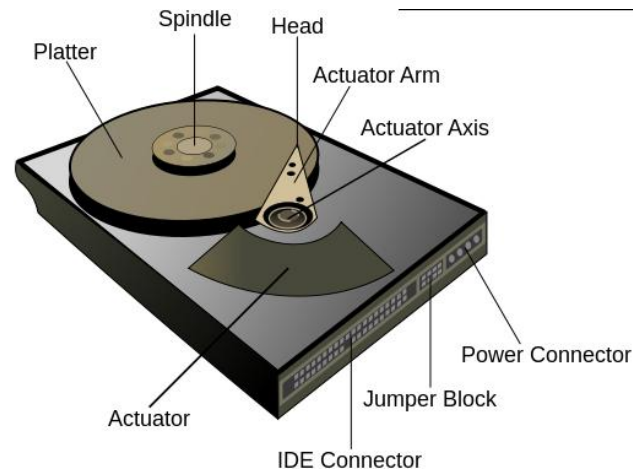
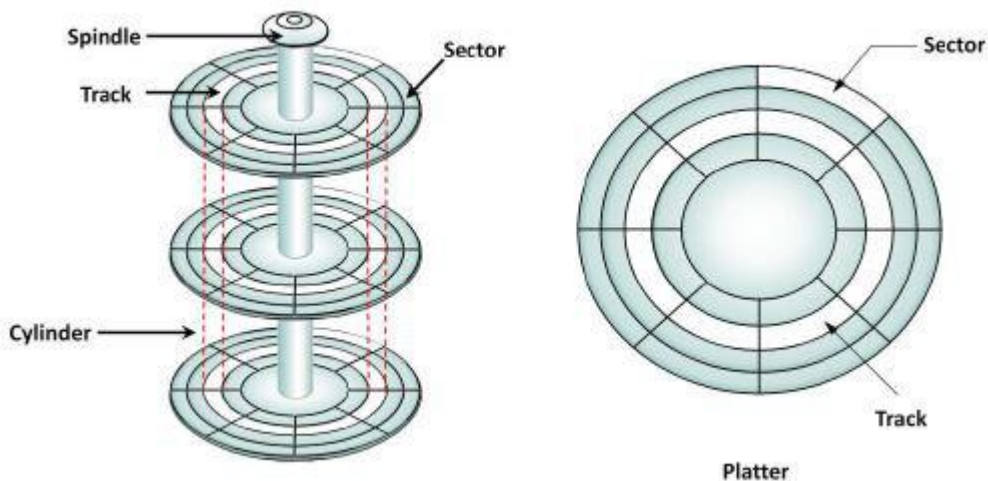
存储子系统整体架构

- raw-flash 和带 FTL flash 的比较



块设备的特点及优化考量

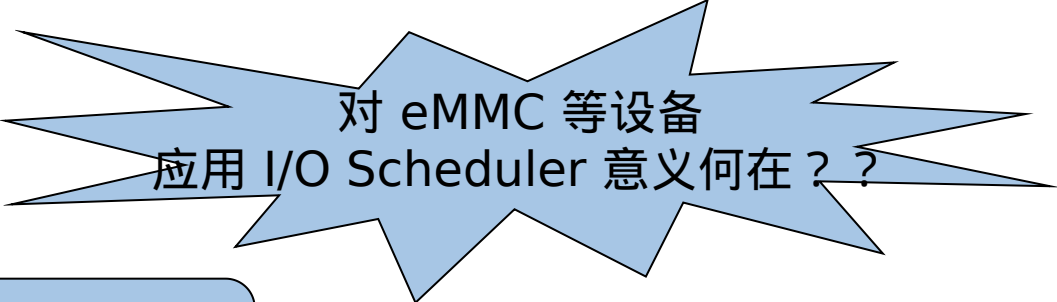
- 机械式磁盘的构造
 - 盘片|转轴|磁头
 - 磁道|柱面|扇区



块设备的 特点及优化考量

- I/O调度器的优化考量

- 对读写请求进行排序，用于尽量保证磁头在同一个方向移动
- 对物理扇区层面连续的多个读写请求，加以合并后进行传输



对 eMMC 等设备
应用 I/O Scheduler 意义何在??

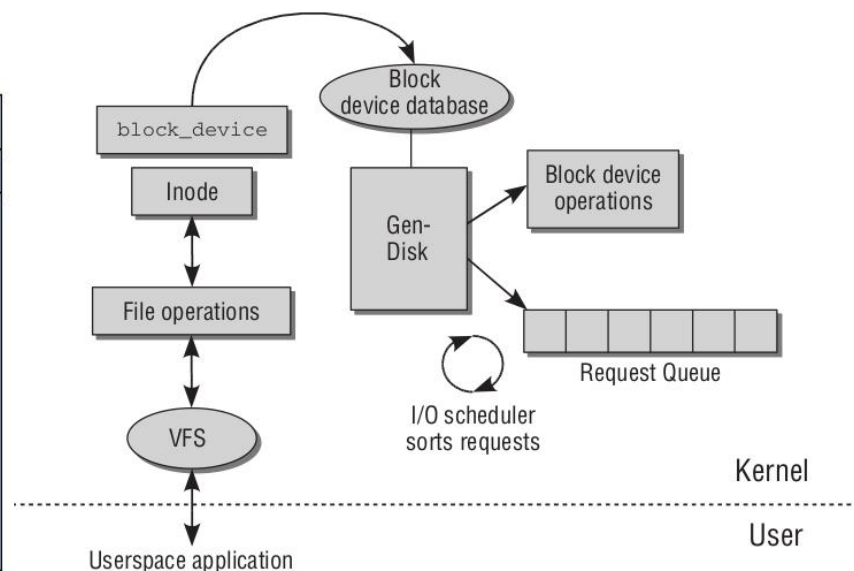


机械式磁盘 VS eMMC设备

块设备子系统相关数据结构

- 整个子系统相对复杂，牵涉的**DS**也比较多，按照功能基本上可以分成三类

块设备子系统相关数据结构		
块设备管理相关	块I/O请求的表示相关	I/O调度器的实现相关
1, gendisk 2, hd_struct 3, block_device 4, inode	1, buffer_header 2, bio 3, bio_vec 4, page 5, request 6, request_queue	1, elevator_queue 2, elevator_type 3, elevator_ops

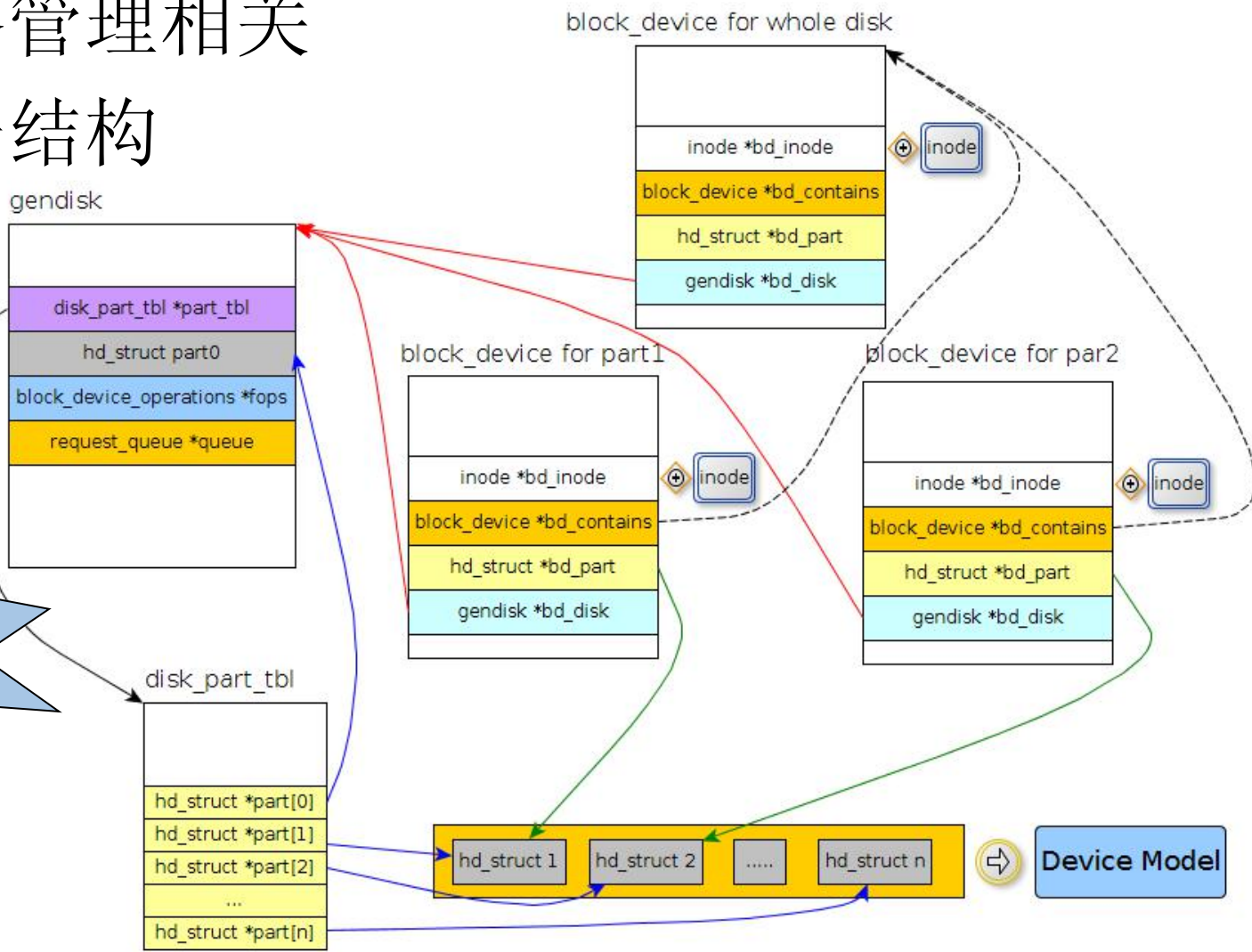


块设备子系统相关数据结构

- 块设备管理相关的数据结构
 - **gendisk**
 - 用于静态得表示任何一个“带有某类分区结构”的通用块存储设备
 - **hd_struct**
 - 用于表示一个“分区结构”，内部镶嵌有**struct device** 实例
 - **block_device**
 - 用于抽象已经打开的块设备，既可表示单个分区，也可表示整个块设备
 - **inode**
 - “Everything is a file”，用于在文件系统中抽象块设备对应的设备文件

块设备子系统相关数据结构

- 块设备管理相关的数据结构



gendisk 与
block_device
之间的异同??

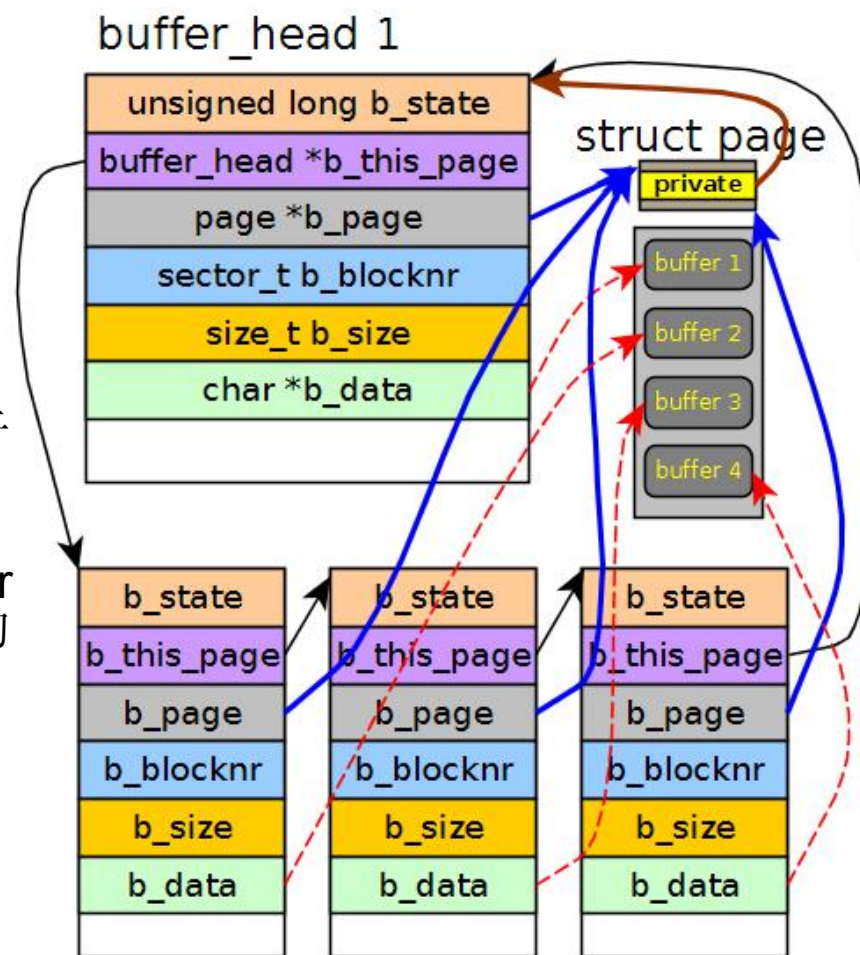
块设备子系统相关数据结构

- 块I/O请求表示相关的数据结构

- **buffer_head**

- 用于描述物理磁盘块被读入内存页后所形成的缓冲区；
 - 2.6之前的内核也将其用作完成I/O操作的数据载体
 - 本身相对比较庞大，不便于直接用于I/O操作
 - 一个**buffer_head**只能描述小于**page**大小的单个**buffer**，所以假如要传输较多量的数据，会如何？

1 block
= 1 buffer
= n sectors



块设备子系统相关数据结构

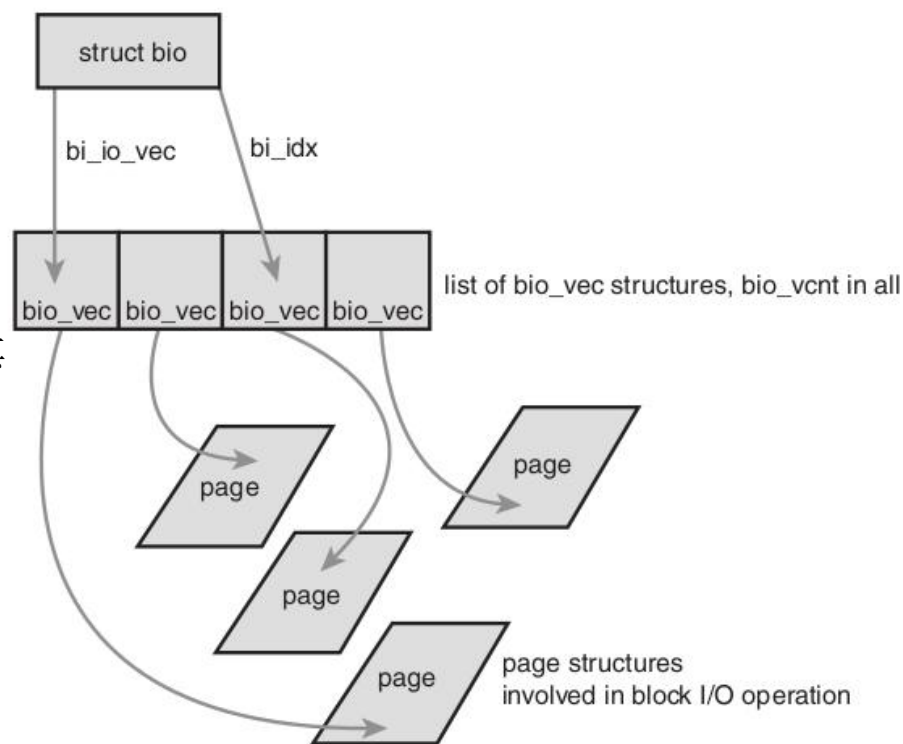
- 块I/O请求表示相关的数据结构(续1)

- bio

- 用做上层发起I/O操作的数据载体
 - 能支持大数据量的块I/O，把要I/O的所有数据“分割成”一个个的段(segment)来管理
 - 段(segment)为不可再分的最小粒度，其存储于不同物理页的不同位置

- bio_vec

- 由三元组<page, offset, len>构成，用于表示一个段
 - 地位/功能/粒度上等价于buffer_head



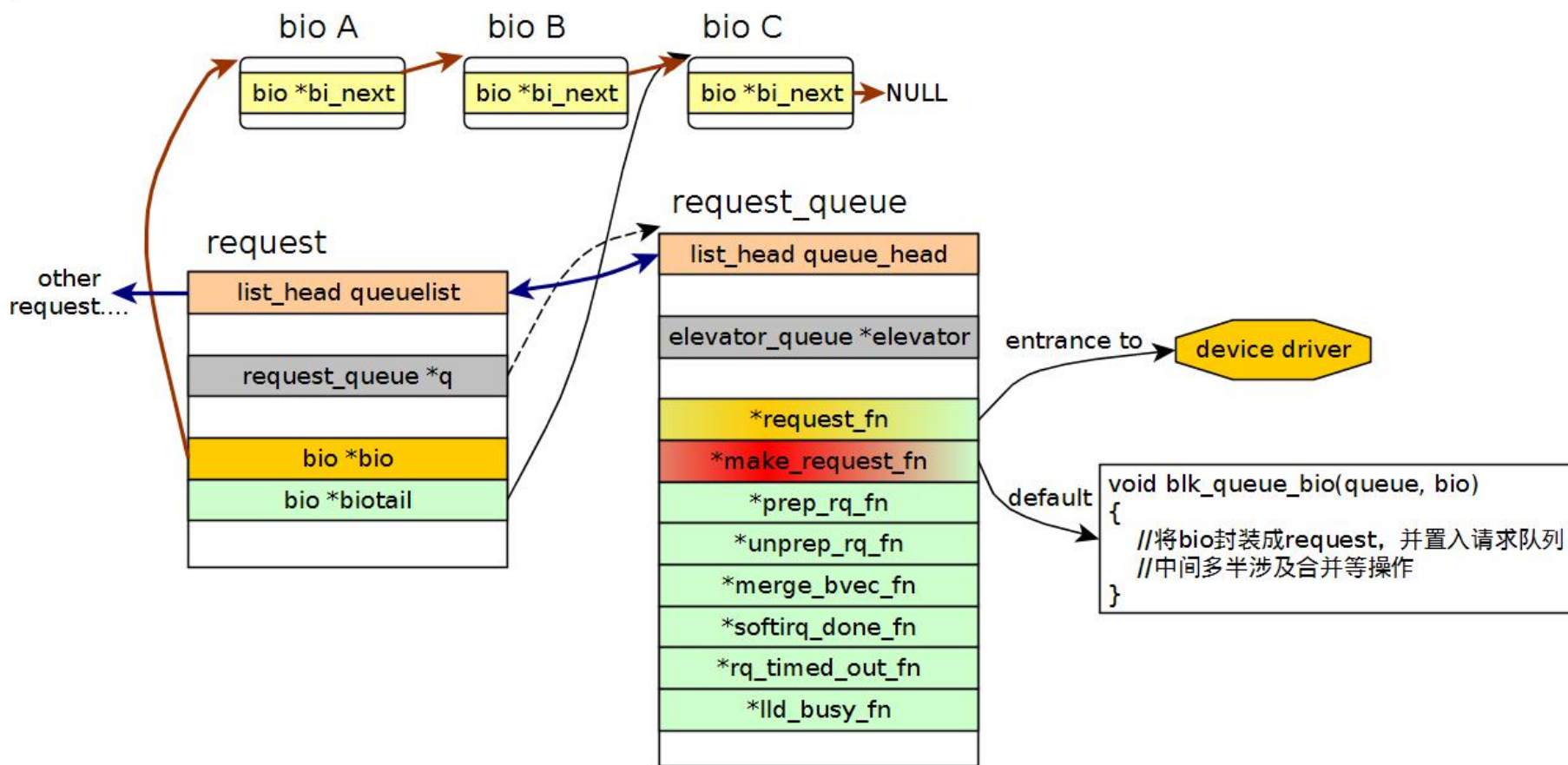
像 bio/bio_vec 这样的向量式I/O被称为
Scatter-Gather I/O

块设备子系统相关数据结构

- 块I/O请求表示相关的数据结构(续2)
 - request
 - 设备驱动程序所处理的I/O请求，内部可能封装有多个物理上连续的bio
 - request_queue
 - request在正式被交给驱动程序之前，它们会被放在request_queue 中，并按照一定的策略排序和归并

块设备子系统相关数据结构

- 块I/O请求表示相关的数据结构(续3)

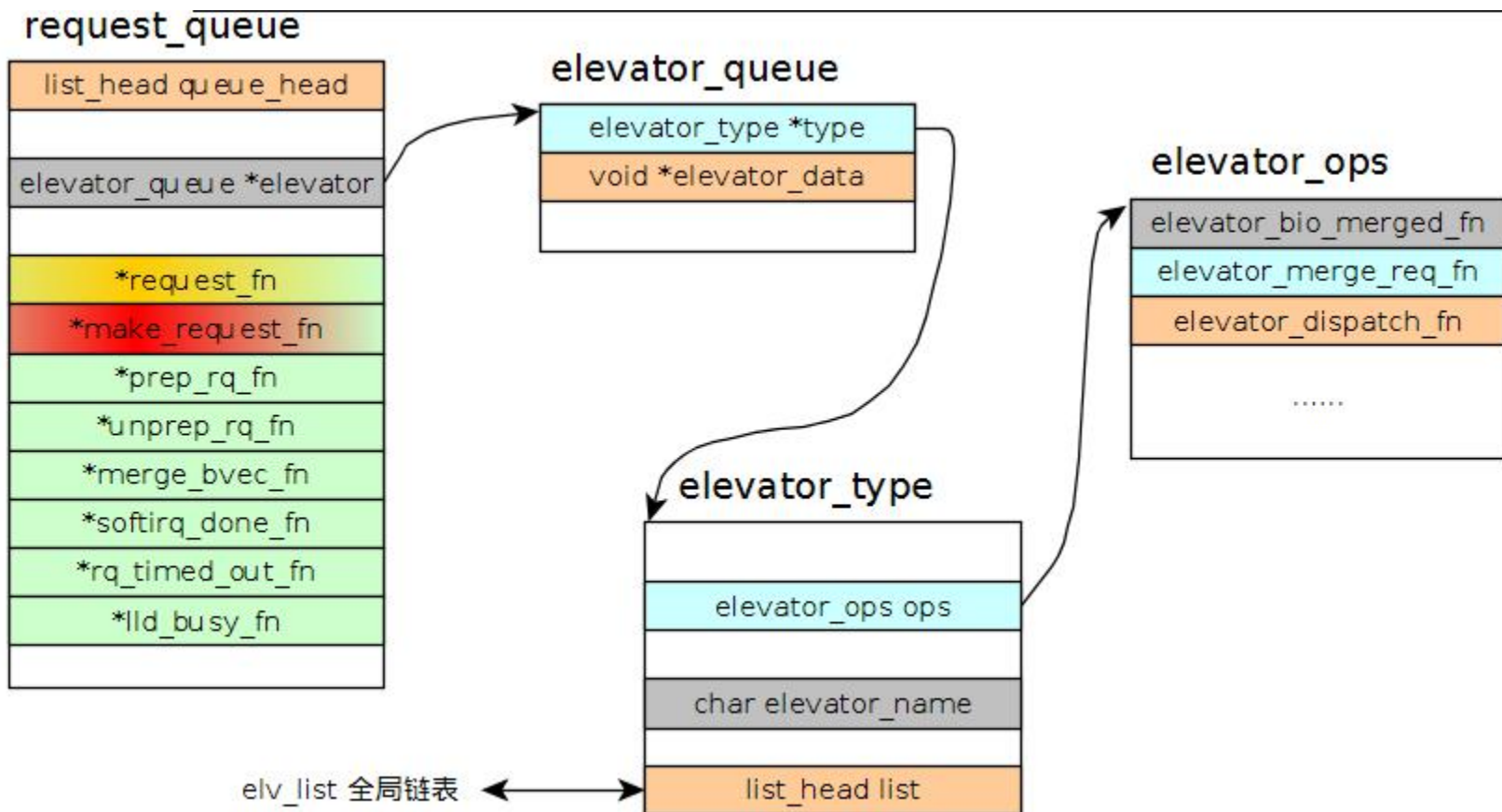


块设备子系统相关数据结构

- I/O调度器实现相关的数据结构
 - `elevator_queue`
 - 用于在请求队列这个层面封装调度器
 - `elevator_type`
 - 真正核心的调度器抽象
 - `elevator_ops`
 - 特定调度器依不同策略所实现的系列操作，诸如如何排序合并等等
 - 这些操作由 `.../block/elevator.c` 文件中的 `elevator` 抽象层接口所调用

块设备子系统相关数据结构

- I/O调度器实现相关的数据结构(续1)



块设备子系统相关数据结构

- I/O调度器实现相关的数据结构(续2)

elevator_ops 中封装的系列操作			
	操作名称	elevator 抽象层中的 wrapper	描述
1	elevator_merge_fn	elv_merge()	调度器这边查找已有的请求中是否存在和 bio 可合并的 request, 并给出可能的合并方式
2	elevator_allow_merge_fn	elv_iosched_allow_merge()	给定一个 request 后, 判断bio是否可以和之合并, 并给出可能的合并方式
3	elevator_bio_merged_fn	elv_bio_merged()	用于进行在 bio 和一个 request 合并后的后处理
4	elevator_merge_req_fn	elv_merge_requests()	用于合并新 request 和队列中的目标 request
5	elevator_dispatch_fn	elv_dispatch_sort()	用于将 request 从调度器自己所维护的数据结构中移出, 并dispatch 到块设备对应的 request_queue 中去
6	elevator_add_req_fn	__elv_add_request()	用于将 request 添加到调度器自己内部维护的数据结构中
7	elevator_activate_rq_fn / elevator_deactivate_rq_fn	----	I/O 调度器可通过前者来跟踪请求于何时被真正开始传输; 后者用于在 requeue request 时做一些 bookkeeping 的工作
8	elevator_completed_req_fn	elv_completed_request()	用于在 request 处理完毕时, 通知调度器做一些后处理
9	elevator_former_req_fn / elevator_latter_req_fn	----	这两个指针用于在调度器自己维护的数据结构中取得前一个/后一个request
10	elevator_set_req_fn / elevator_put_req_fn	----	这两个指针用于在调度器内部分配/释放与一个 request 对应的某些资源
11	elevator_init_fn / elevator_exit_fn	elvator_init() / elevator_exit()	这两个指针用于调度器自身的构造和析构

I/O请求的处理

- I/O请求的生成

- 生成缘由

- 文件读写/数据库事务操作
 - dirty data writeback(主动/被动)
 - swap 换入换出
 - 其他如 fsck、mount fs、fdisk 操作等等

- 生成方式

- 调用 submit_bio() 函数

```
int _submit_bh(int rw, struct buffer_head *bh, unsigned long bio_flags)
{
    struct bio *bio;
    int ret = 0;

    //...
    bio = bio_alloc(GFP_NOIO, 1);

    //... bio 初始化
    bio->bi_vcnt = 1;
    bio->bi_iter.bi_size = bh->b_size;
    //...

    bio_get(bio);
    submit_bio(rw, bio);

    //...
}
```

```
void submit_bio(int rw, struct bio *bio)
{
    bio->bi_rw |= rw;

    //...
    generic_make_request(bio);
}
EXPORT_SYMBOL(submit_bio);
```

I/O请求的处理

- I/O请求的生成(续1)
 - generic_make_request() 函数

```
void generic_make_request(struct bio *bio)
{
    struct bio_list bio_list_on_stack;

    if (!generic_make_request_checks(bio))
        return;

    if (current->bio_list) {
        bio_list_add(current->bio_list, bio);
        return;
    }

    BUG_ON(bio->bi_next);
    bio_list_init(&bio_list_on_stack);
    current->bio_list = &bio_list_on_stack;
    do {
        struct request_queue *q = bdev_get_queue(bio->bi_bdev);

        q->make_request_fn(q, bio);

        bio = bio_list_pop(current->bio_list);
    } while (bio);
    current->bio_list = NULL; /* deactivate */
}
```

```
struct bio_list {
    struct bio *head;
    struct bio *tail;
};
```

I/O请求的处理

- I/O请求的生成(续2)

- q->make_request_fn()

- 若块设备的特性决定了使用queue/I/O调度器是有益的
 - 如磁盘等典型设备
 - 让q->make_request_fn() 默认指向 blk_queue_bio()
 - 若块设备的特性认为使用queue无益处，或者中间需要某种特殊处理的
 - 如ramdisk、RAID及LVM等
 - 让q->make_request_fn() 指向自己的特殊版本

Note: request_queue 中的 make_request_fn 用 blk_queue_make_request() 来设置

I/O请求的处理

- I/O请求的生成(续3)
 - blk_queue_bio()函数，逻辑基本上有两部分
 - 第一部分，是“尽最大努力地”将所提交的bio合并入某一个request(@ A/B)
 - 看看传入的bio是否能和队列中已有的request相合并;
 - 看看被合并而有所改动的 request 能否与相邻的 request 继续合并。
 - 第二部分，是新分配一个request，并用bio去初始化，然后将其增加到队列中(to A/C)

I/O请求的处理

- I/O请求的生成(续4)

- request的存放位置

- A, current 进程的 plug list

- 为了得到更多优化，也就是在本地收集一批请求，让调度器在小范围内得到足够的优化(合并)空间
 - plug list 中的请求会在合适的时机被 `blk_flush_plug_list()` 刷新至 `request_queue` 中

- B, 调度器自己的私有数据结构中

- 调度器自己单独维护，保存其中的 request 会被 `elv_drain_elevator()` / `blk_peek_request()` 等函数取出

- C, gendisk 对应的 `request_queue` 中

- request 被交给驱动程序之前最后一刻被存储的地方，也被称为 `dispatch queue`

I/O请求的处理

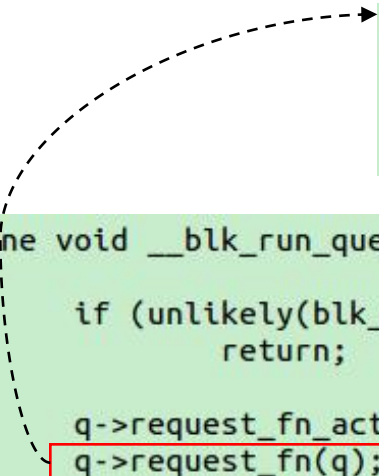
- I/O请求的处理

```
inline void __blk_run_queue_uncond(struct request_queue *q)
{
    if (unlikely(blk_queue_dead(q)))
        return;

    q->request_fn_active++;
    q->request_fn(q);
    q->request_fn_active--;
}

void __blk_run_queue(struct request_queue *q)
{
    if (unlikely(blk_queue_stopped(q)))
        return;

    __blk_run_queue_uncond(q);
}
```



```
while ((req = blk_fetch_request(q)) != NULL) {
    //... processing of request(req)
    __blk_end_request_all(req, -EIO);
}
```


I/O调度

- I/O调度
 - 目的重在对I/O请求进行排序和合并，以优化性能；此外还考虑几点：
 - a, 防止请求长时间得不到处理
 - 设置deadline
 - b, 避免读请求因写请求的处理而受到耽搁
 - 读请求同步处理，而写请求则是异步操作
 - 读请求对性能影响更为关键
 - c, 兼顾系统内进行I/O各进程之间的公平性
 - request 被交给驱动程序之前最后一刻被存储的地方，也被称为 dispatch queue

I/O调度

- I/O调度(续1)

- 到目前(3.18)为止常见的I/O调度器:

- Noop

- 先来先服务, 合并但不排序
 - 适用于自己能处理排序的"智能"设备, 也适用于无需 seeking 的, 如 eMMC 之类的设备

- Deadline

- 不仅努力 seeking 次数, 也努力解决前述 a/b 两点问题
 - 给每个请求绑定一个定时器(read:500ms/write:5s), 并尽量确保在超时前得到处理
 - 尽量减小读请求的延迟, 以期提升性能

思考: 吞吐率
上面会有影响么?



I/O调度

- I/O调度(续2)

- 到目前(3.18)为止常见的I/O调度器:
(cont)

- Anticipatory Scheduler

- 是在读请求被提交处理之后,它并不马上回去处理前面的写请求,而是等待几个ms的时间,看是否有新的读请求过来,若有则处理之,从而避免耗时的seeking操作
 - 等待期间,无读请求过来,则浪费几ms的时间,但通常不是问题,只要能有比较精准的预测
 - 实现复杂,因为要统计和预测

思考: 这么做有何理论基础?
期望每次的带的平均时间要小于(因不等待)
而每次花掉的成对seeking时间

I/O调度

- I/O调度(续3)

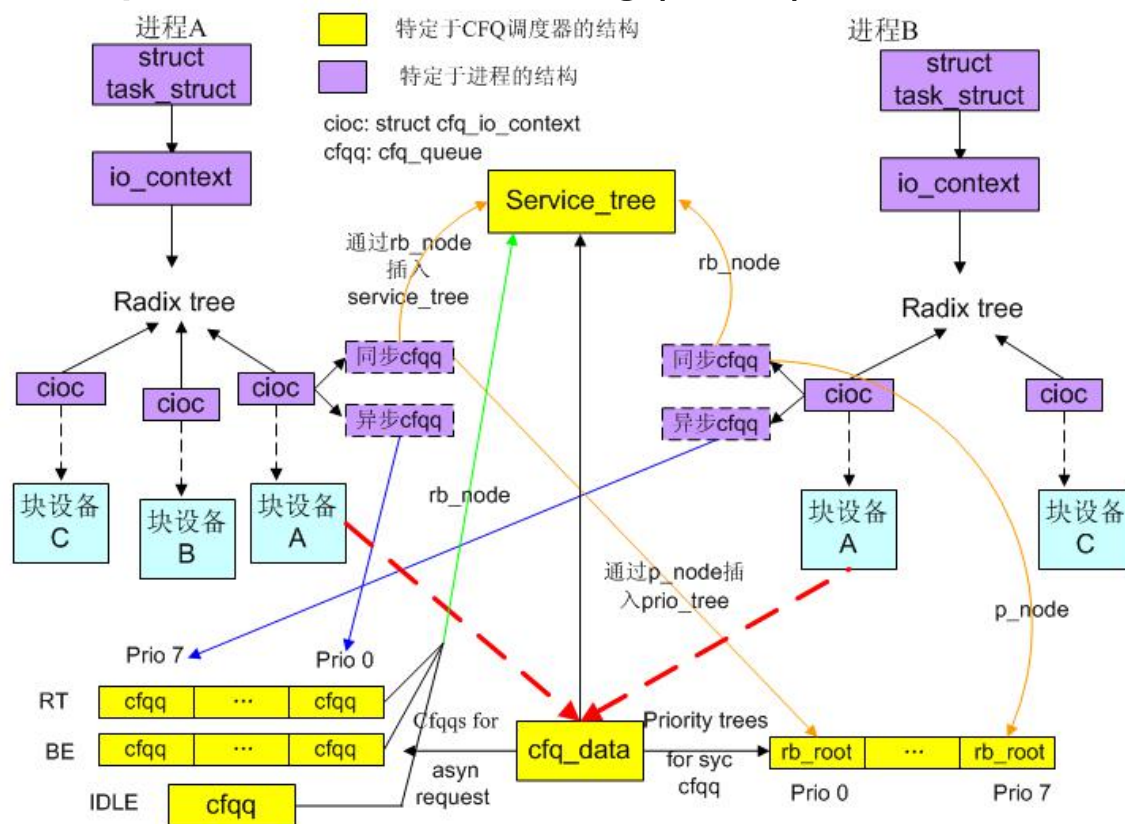
- 到目前(3.18)为止常见的I/O调度器: (cont)

- Complete Fair Queuing

- 源于 CFS 调度算法, 目的在于把I/O带宽平均分配给系统中参与I/O的所有进程
 - 每个进程可安排RT/BE/IDLE等三类共17种优先级($2*8+1$)
 - 请求有按照类型分为两种(同步[读|写]和异步写)
 - 在不同进程内给同步请求单独安排一个队列, 在全局范围内给异步请求按等级不同各安排一个队列
 - CFQ以 round-robin 的方式来服务这些队列, 即将其中的 request dispatch 到 request_queue 中去
 - 较新版本中引入 cgroup 后能做到 I/O 带宽的精细分配, 但是也带来较大的复杂性
 - 现在是很多发行版默认使用的调度器, 但....

I/O调度

- I/O调度(续4)
 - 到目前(3.18)为止常见的I/O调度器: (cont)
 - Complete Fair Queuing(cont)



I/O性能分析和调优

- 何以要调优
 - 慢速的机械磁盘 -> 带有FTL的eMMC
 - 通常pc/server上的应用 -> android上的应用
- 调优原则与步骤
 - a, 用某些 **benchmark** 工具来对不同场景下的I/O处理情况, 诸如I/O请求的生成、处理、以及结束时的不同参数做跟踪采集和记录

可能的工具:

IOZone

fio

blktrace(blkparse)

btt

Androbench

自写工具

可能的参数:

I/O发起者进程

I/O数据在设备中的位置

I/O类型

I/O数据量大小

各阶段动作的时间戳

I/O性能分析和调优

- 调优原则与步骤(续)

- **b**, 按照**a**步中的方式，多次采集数据后用统计学原理加以分析
 - 找出隐藏于数据之中的规律
 - 大部分是小于4K的请求么？
 - 是随机读写多，还是顺序读写多？
 - 是读请求占多数，还是写请求占多数？
 - 写请求里面是同步写居多，还是异步写居多？
 - 定位整个I/O Path上面的瓶颈所在
 - fs? 还是 I/O Scheduler? 还是 eMMC这边？
- **c**, 根据**b**步中分析出的I/O特征，再想办法做一些针对性的优化

I/O性能分析和调优

- Andoird上面必须更整体来考虑I/O性能
 - **sqlite+ext4+cfq**
 - **sqlite** 一次update/insert会产生多达11个写操作
 - 大量很小的临时文件
 - 更新数据库表格
 - 操作 **ext4 journal**
 - 为了保证数据完整性，**sqlite**事务和**ext4**日志单个本身就能产生大量的写操作，更何况它们双剑合璧
 - **cfq** 内部逻辑太多，对**eMMC**这样的介质来说，坏处可能多过收益
 - 考虑其他更简单的调度器，如**noop**

<https://www.sqlite.org/tempfiles.html>

I/O性能分析和调优

- Andoird上面必须更整体来考虑I/O性能
 - 其他特征
 - 超过50%的写操作是ext4 journal更新
 - 60-80%的写操作都是随机的
 - 超过50%的写操作都是同步的写
 - 4K以下的写操作占了70%

