# Old linux

## Page cache 层的处理

从上文得知：ext2\_readpage 函数是该层的入口点。该函数调用 mpage\_readpage 函数，清单6显示了 mpage\_readpage 函数的代码。

清单6 mpage\_readpage 函数的代码

int mpage\_readpage(struct page \*page, get\_block\_t get\_block)

{

struct bio \*bio = NULL;

sector\_t last\_block\_in\_bio = 0;

bio = do\_mpage\_readpage(bio, page, 1,

&last\_block\_in\_bio, get\_block);

if (bio)

mpage\_bio\_submit(READ, bio);

return 0;

}

该函数首先调用函数 do\_mpage\_readpage 函数创建了一个 bio 请求，该请求指明了要读取的数据块所在磁盘的位置、数据块的数量以及拷贝该数据的目标位置——缓存区中 page 的信息。然后调用 mpage\_bio\_submit 函数处理请求。 mpage\_bio\_submit 函数则调用 submit\_bio 函数处理该请求，后者最终将请求传递给函数 generic\_make\_request ，并由 generic\_make\_request 函数将请求提交给通用块层处理。

到此为止， page cache 层的处理结束。

## 通用块层的处理

generic\_make\_request 函数是该层的入口点，该层只有这一个函数处理请求。清单7显示了函数的部分代码

清单7 generic\_make\_request 函数部分代码

void generic\_make\_request(struct bio \*bio)

{

……

do {

char b[BDEVNAME\_SIZE];

q = bdev\_get\_queue(bio->bi\_bdev);

……

block\_wait\_queue\_running(q);

/\*

\* If this device has partitions, remap block n

\* of partition p to block n+start(p) of the disk.

\*/

blk\_partition\_remap(bio);

ret = q->make\_request\_fn(q, bio);

} while (ret);

}

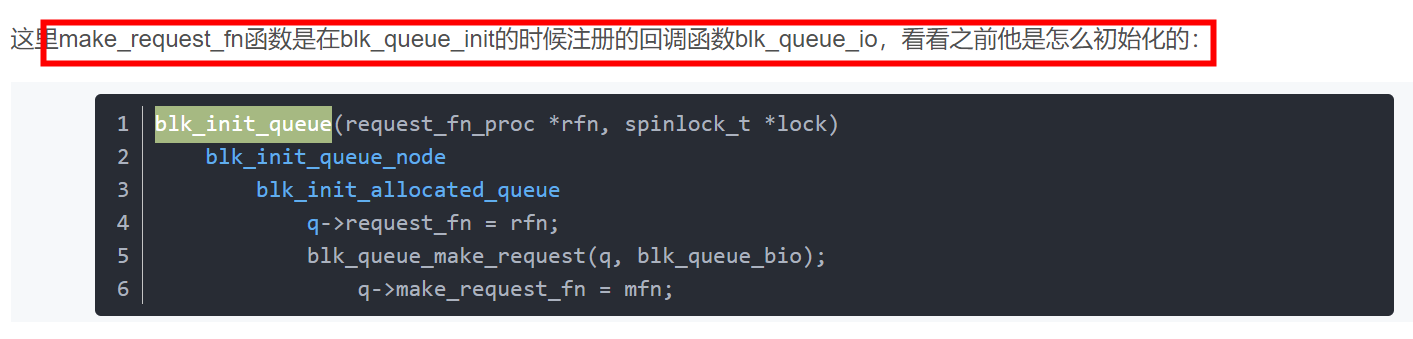
主要操作：

根据 bio 中保存的块设备号取得请求队列 q

检测当前 IO 调度器是否可用，如果可用，则继续；否则等待调度器可用

调用 q->make\_request\_fn 所指向的函数将该请求（bio）加入到请求队列中

到此为止，通用块层的操作结束。



## IO 调度层的处理

对 make\_request\_fn 函数的调用可以认为是 IO 调度层的入口，该函数用于向请求队列中添加请求。该函数是在创建请求队列时指定的，代码如下（blk\_init\_queue 函数中）：

q->request\_fn = rfn;

blk\_queue\_make\_request(q, \_\_make\_request);

函数 blk\_queue\_make\_request 将函数 \_\_make\_request 的地址赋予了请求队列 q 的 make\_request\_fn 成员，那么， \_\_make\_request 函数才是 IO 调度层的真实入口。

**I/O电梯调度算法**



# 新版linux多对列

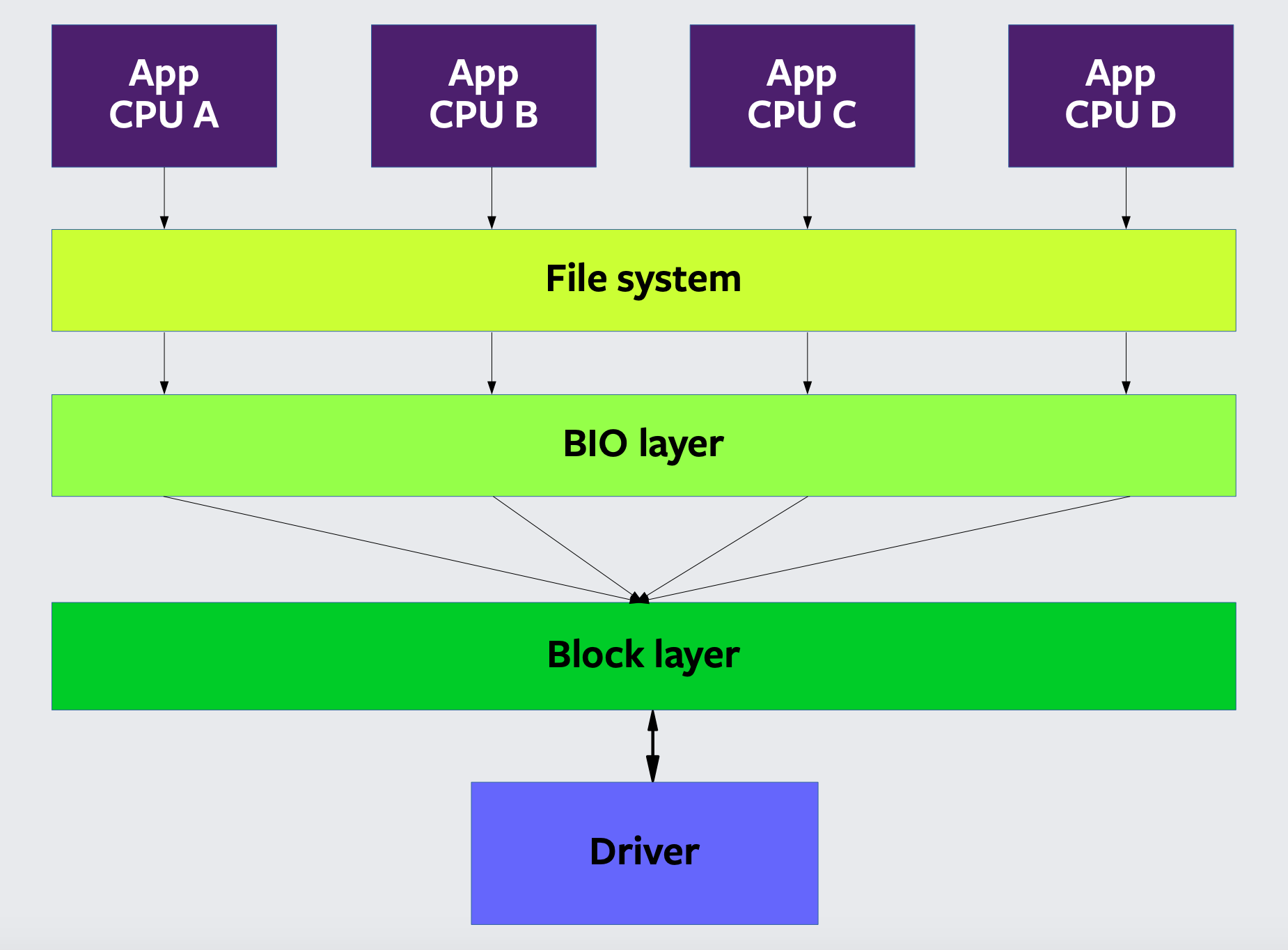
内核块层 (blk-mq)

Linux Block Layer中的I/O队列和调度器

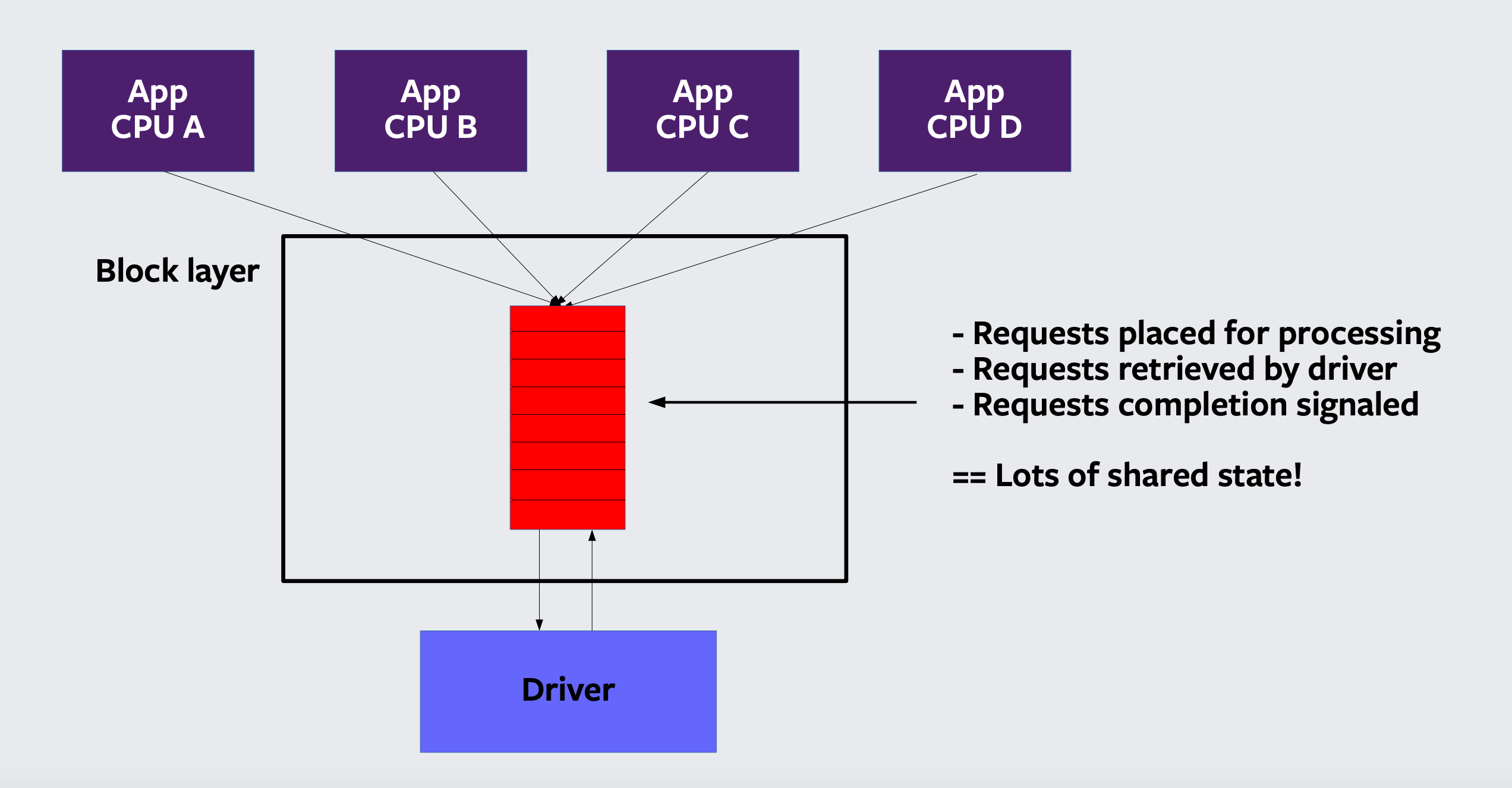
用于替代原先的块IO调度层，用多Queue代替单Queue，并且取消了电梯调度算法。

## 为什么要用mq

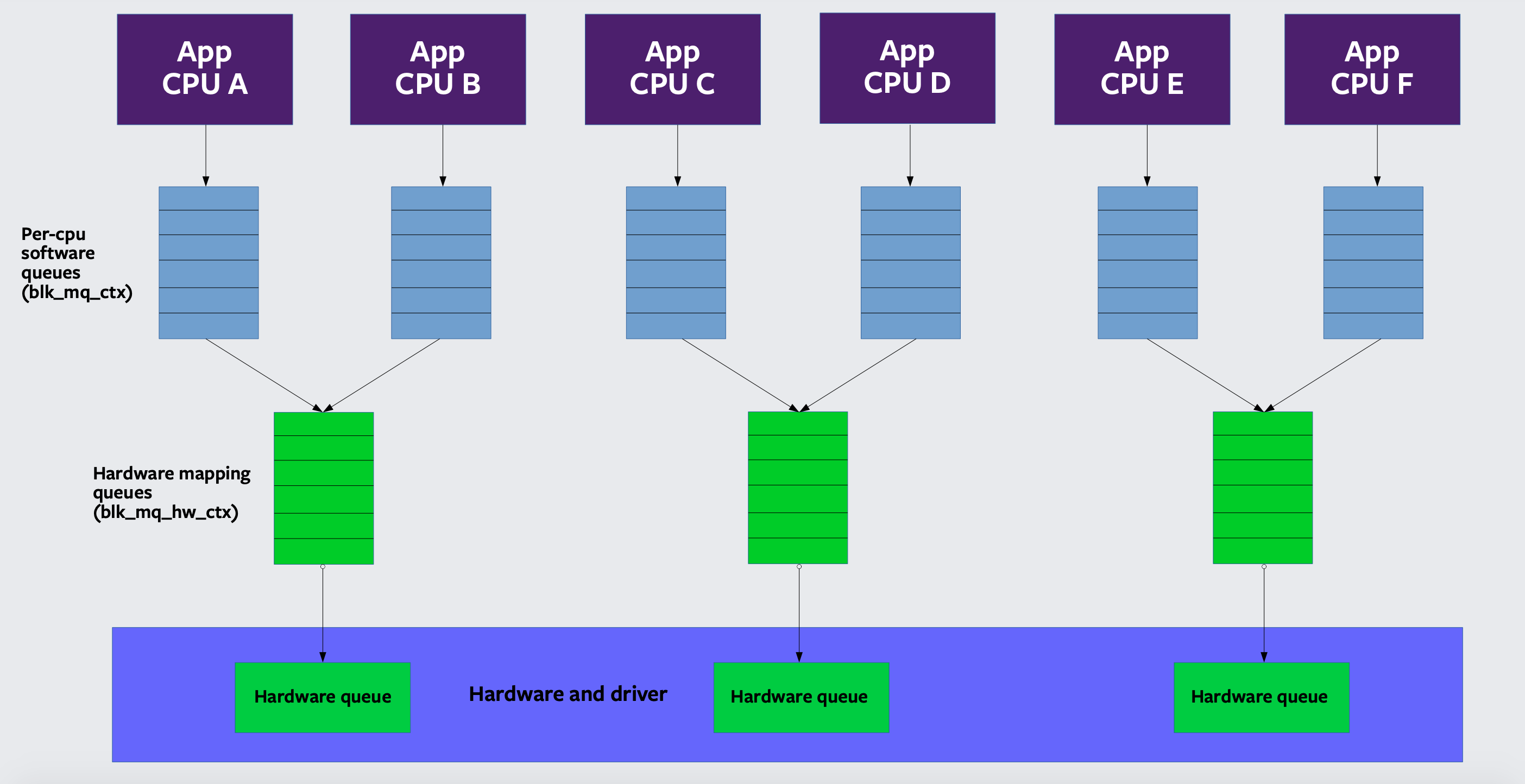
在主机中，多cpu运行多个线程，每个线程都能和文件系统交互，文件系统层也是用多线程和bio层交互，但是，块设备层只有一个队列：



在块设备层，来自多个cpu的bio请求被放在同一个队列中，造成阻塞：

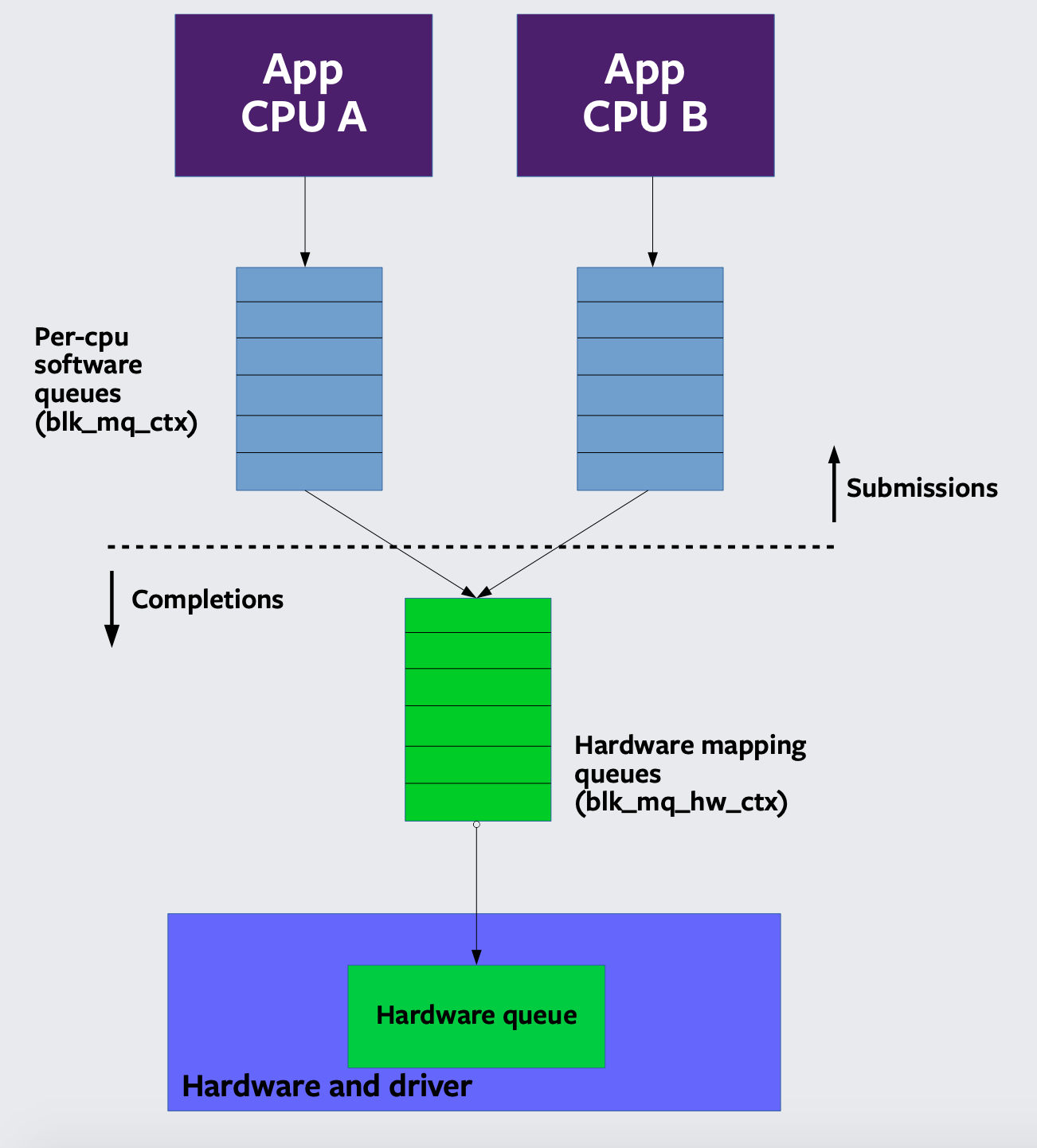


因此，提出了多队列的方法，在块设备层也做成多线程：

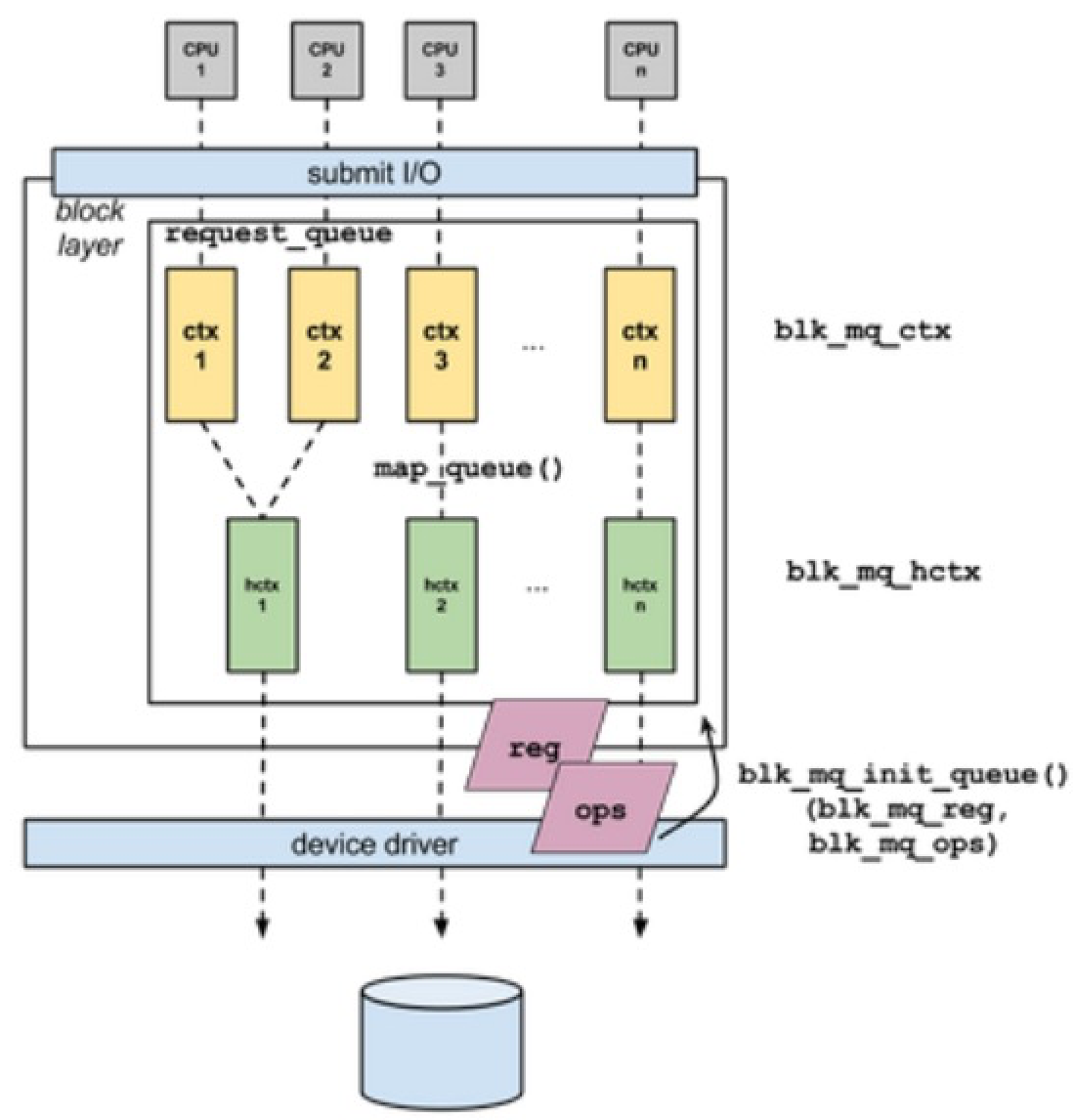


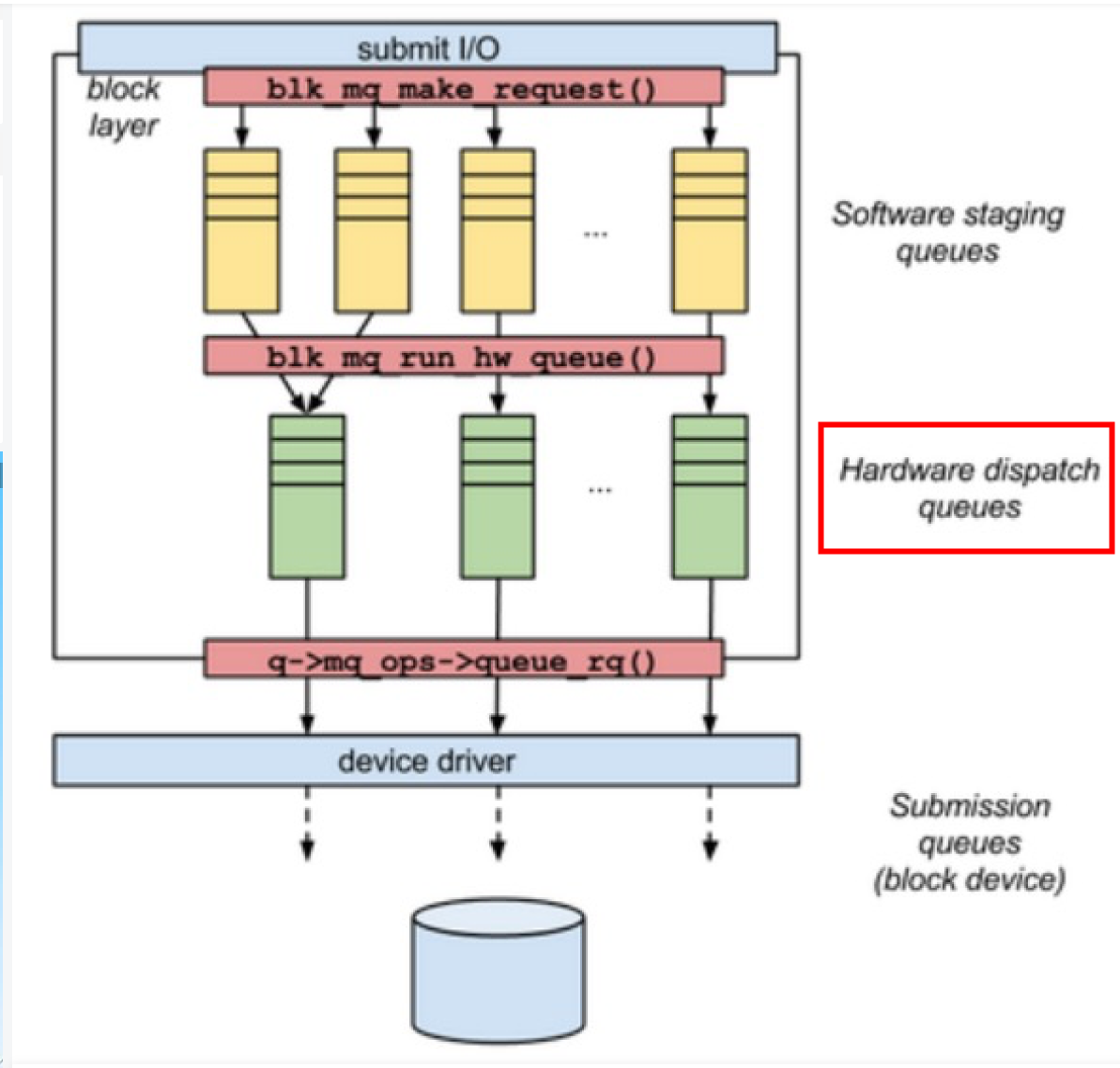
但是，在块设备层实现多个队列并不能像文件系统一样考虑，因为块设备层需要与硬件交互，这需要硬件也支持多队列，最理想的情况是，硬件支持的队列足够多，上层的每个队列（基于软件的队列），都有硬件队列和其关联。但有些时候，硬件支持的队列有限，就形成如上图的关联关系：上图中有3个硬件队列，但是，上层总共形成了6个队列（cpu到文件系统到bio层，都是6个基于软件的队列），因此，到达块设备层时，块设备会将2个基于软件的队列和1个硬件队列关联起来。

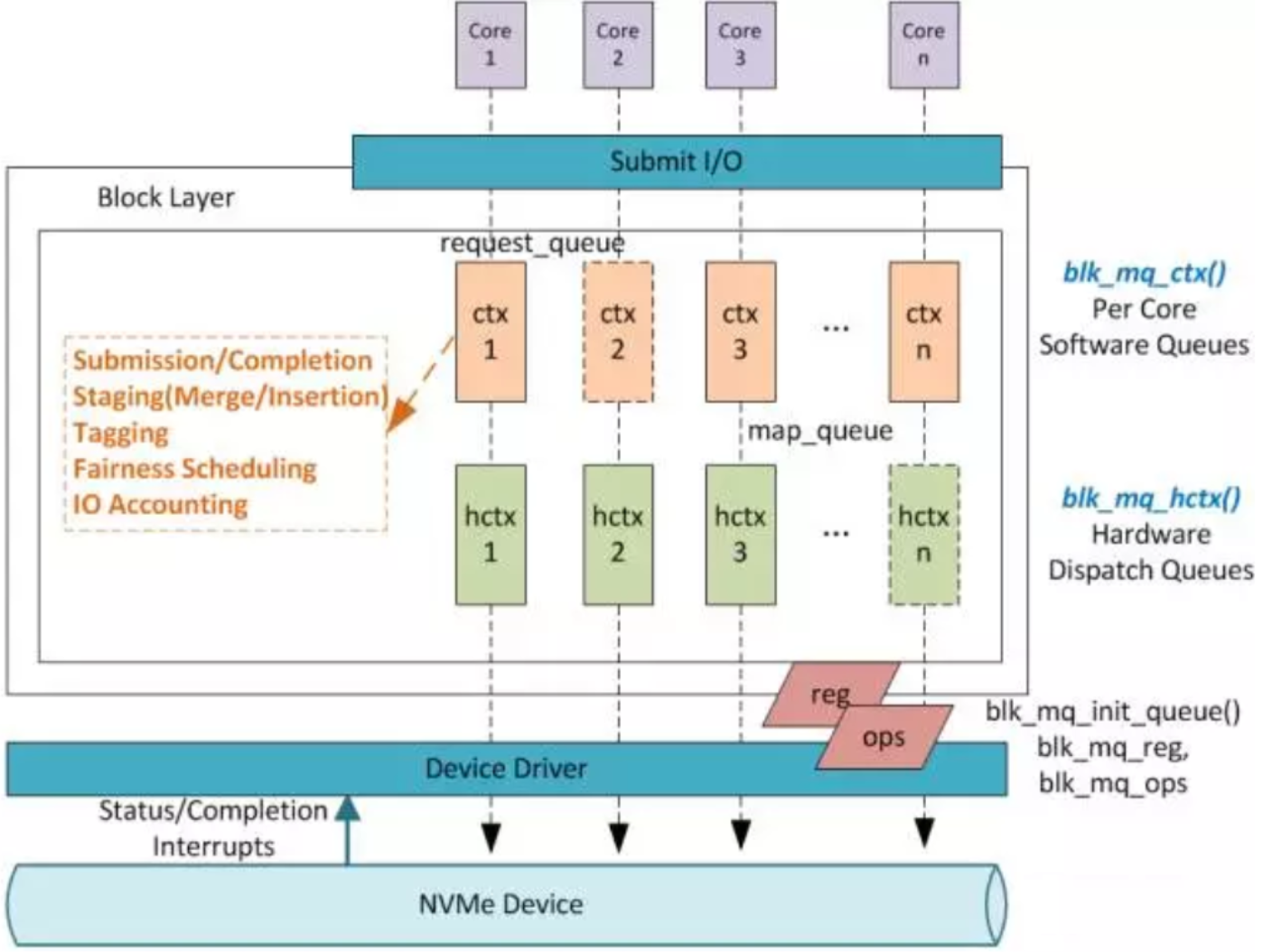
以下是细节图：



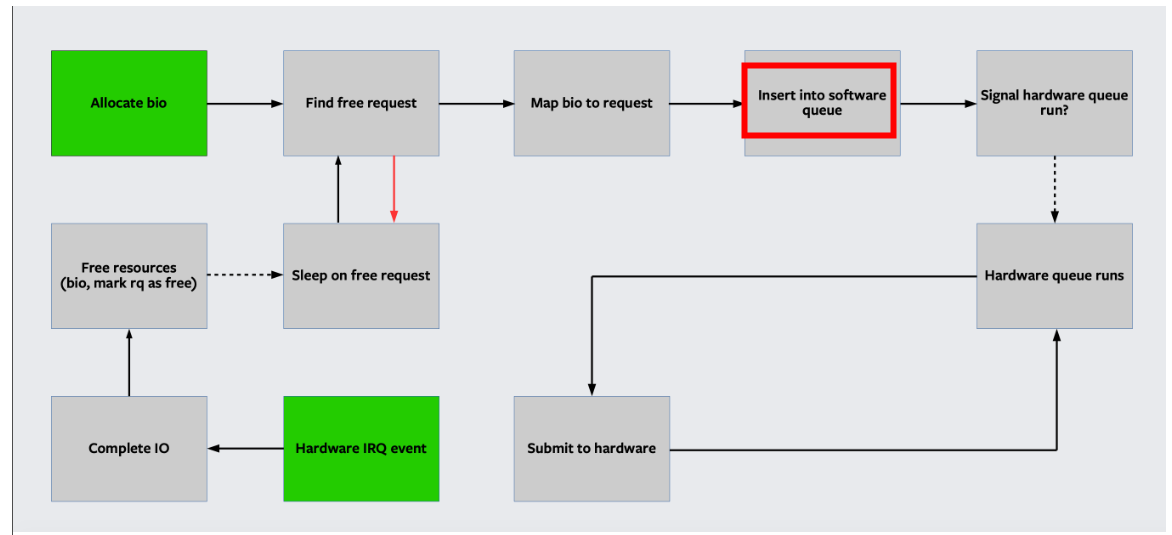
在编程中，一般用到的变量名和blk\_mq 中各部分的对应关系：



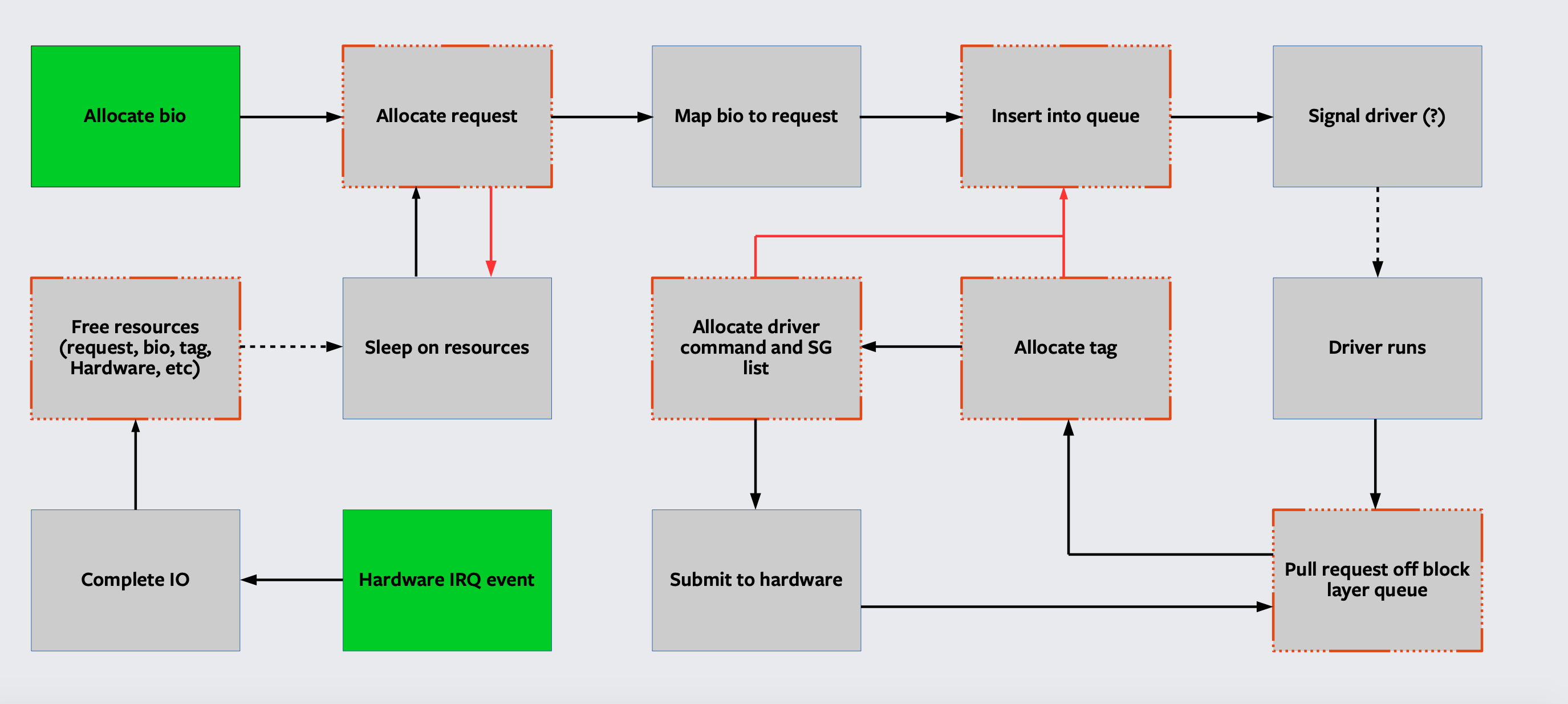




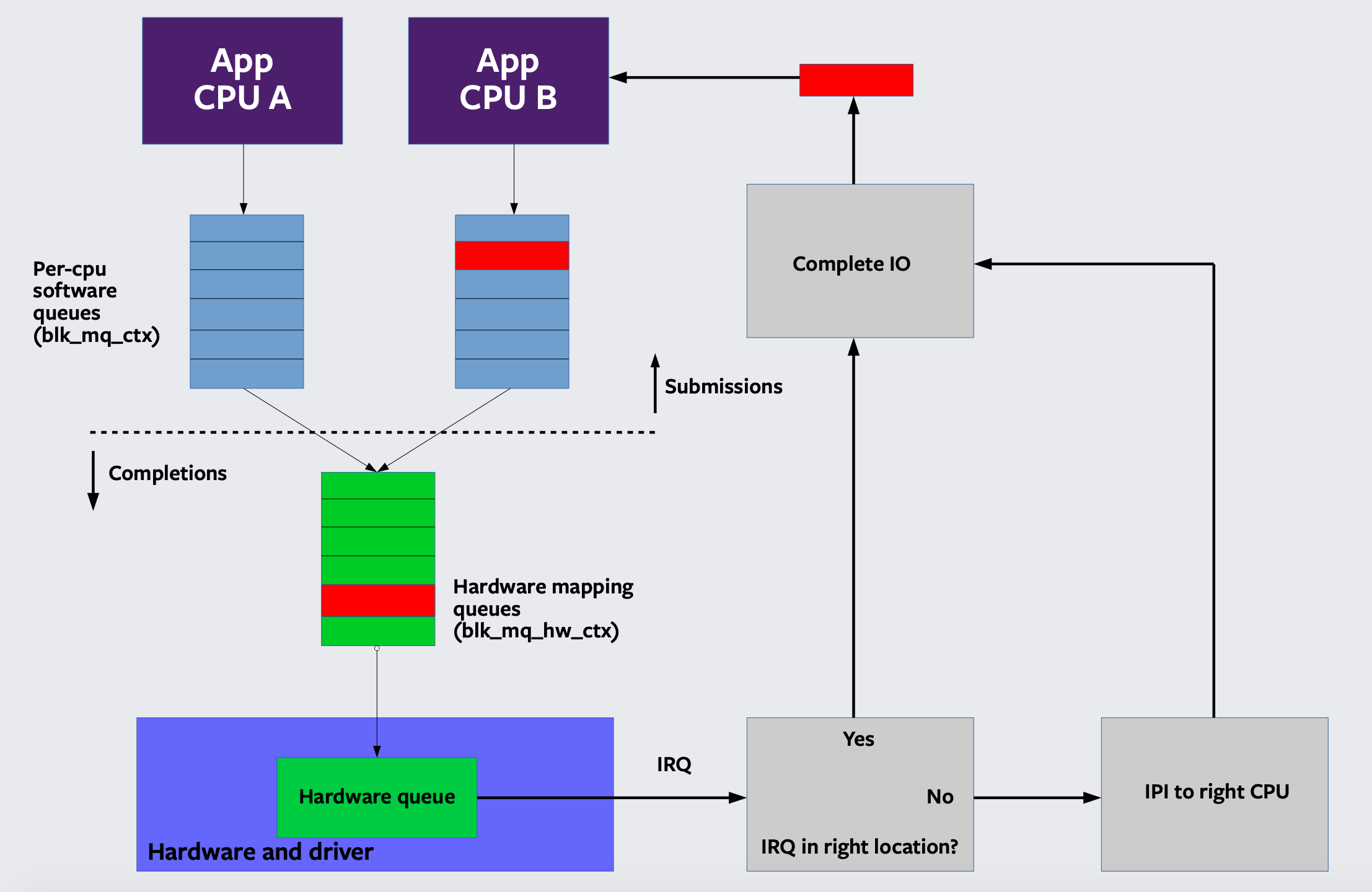
**blk\_mq 中的 io 流：**



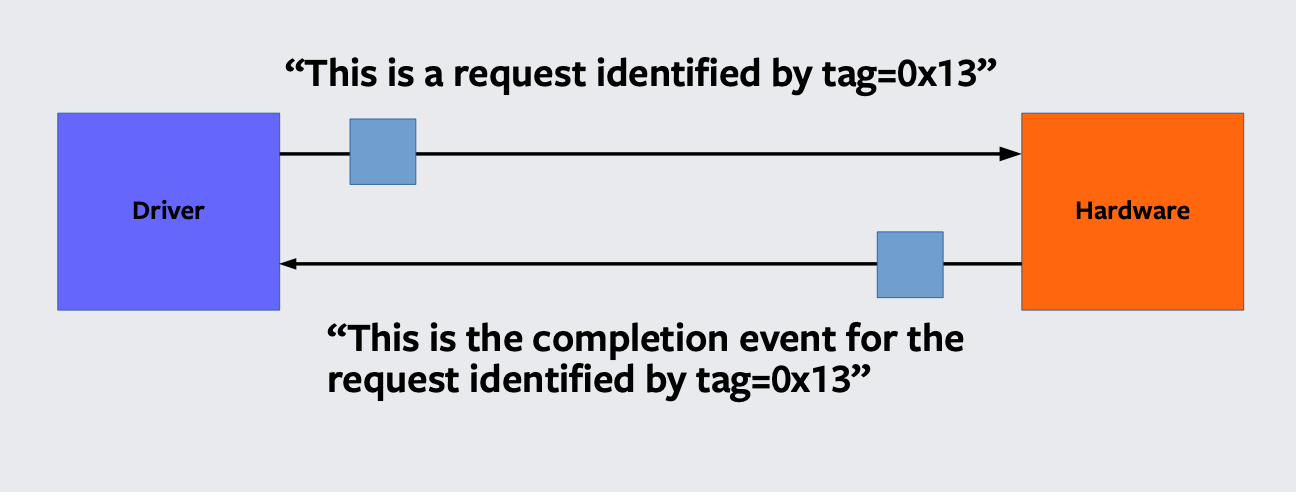
**块设备层的io流：**



因为来自上层的多个队列在块设备层被放在不同的队列中，一个需要解决的问题是：当块设备的层提交给下层设备的bio请求完成后，如何从返回的请求（complete IO）中区分其来自块设备中的哪一个队列。一种方法是使用IPI（处理器间中断处理），示意图如下：



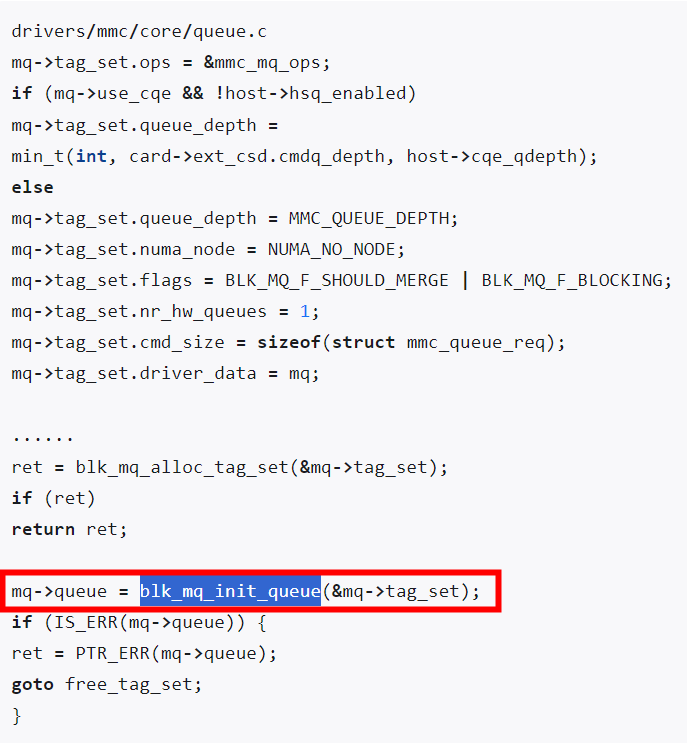
另一个方法是使用一串数字来标记请求来自哪个队列，即：硬件从请求中获得 tag，请求完成后，在返回给 completion IO 池的的请求中也带上该 tag，示意图如下：



### tag number

struct request\_queue之于struct request的关系，非常像struct gendisk之于struct bio的关系：一个代表具体的设备，一个代表IO请求。需要注意的是，每个gendisk都有一个关联的request\_queue结构体，但是只有那些使用了"request层"的设备才会分配request结构体。

struct request\_queue \*blk\_mq\_init\_queue(struct blk\_mq\_tag\_set \*set)



与single-queue相比有另一个重要区别，multi-queue使用的request结构体都是预分配的。每个request结构体都关联着一个不同tag number，这个tag number会随着请求传递到硬件，再随着请求完成通知传递回来。早点为一个请求分配tag number，在时机到来的时候，request 层可随时向底层发送请求。

## struct blk\_mq\_ops 实现

1. hctx->run\_work在blk\_mq\_init\_queue()初始化硬件队列时绑定为blk\_mq\_run\_work\_fn
2. blk\_mq\_run\_work\_fn -> \_\_blk\_mq\_run\_hw\_queue ->

blk\_mq\_sched.c: blk\_mq\_sched\_dispatch\_requests -> blk\_mq\_do\_dispatch\_sched or blk\_mq\_do\_dispatch\_ctx ->

blk\_map.c: blk\_mq\_dispatch\_rq\_list -> [q->mq\_ops->queue\_rq(hctx, &bd)];

1. nvme的驱动中queue\_rq都指向了nvme\_queue\_rq ，初始化如下，

static const struct blk\_mq\_ops nvme\_mq\_admin\_ops = {

.queue\_rq = nvme\_queue\_rq,

.complete = nvme\_pci\_complete\_rq,

.init\_hctx = nvme\_admin\_init\_hctx,

.exit\_hctx = nvme\_admin\_exit\_hctx,

.init\_request = nvme\_init\_request,

.timeout = nvme\_timeout,

};​​​​​​​

static const struct blk\_mq\_ops nvme\_mq\_ops = {

.queue\_rq = nvme\_queue\_rq,

.complete = nvme\_pci\_complete\_rq,

.commit\_rqs = nvme\_commit\_rqs,

.init\_hctx = nvme\_init\_hctx,

.init\_request = nvme\_init\_request,

.map\_queues = nvme\_pci\_map\_queues,

.timeout = nvme\_timeout,

.poll = nvme\_poll,

};

1. nvme\_queue\_rq

nvme/host/pci.c:

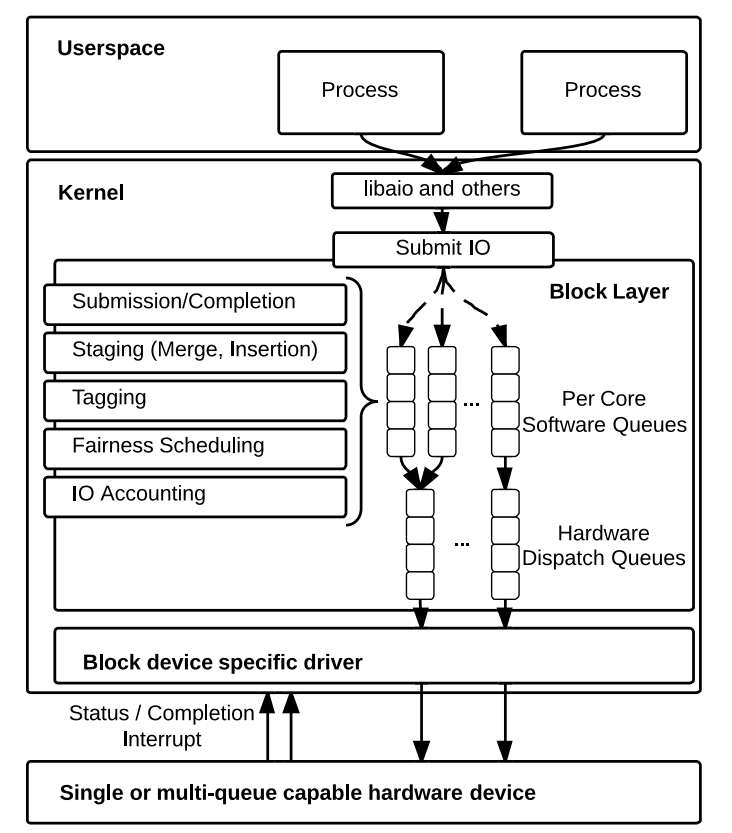
nvme\_queue\_rq

1. nvme\_setup\_cmd 使用rq中的cmd拷贝给新的cmd变量，并根据rq->cmd\_flags对cmd做调整，最后将cmd-> common.command\_id = rq->tag；rq->tag在nvme\_alloc\_request -> blk\_mq\_alloc\_request -> blk\_mq\_get\_request -> blk\_mq\_get\_tag -> \_\_blk\_mq\_get\_tag -> \_\_sbitmap\_queue\_get中分配，然后在 blk\_mq\_rq\_ctx\_init 函数中赋值给rq->tag。
2. nvme\_map\_data 组装数据
3. nvme\_submit\_cmd 拷贝cmd到sq的对应的dma地址nvmeq->sq\_cmds中，并写doorbell通知设备读取sq命令

## blk\_mq\_alloc\_tag\_set



# Nvme 框架



为了减少锁争用和尽可能利用局部性原理，blk-mq把同时负责提交和派发的单一队列拆分为多层级和多队列

blk-mq框架中有2种形式的队列：

* per-cpu级别的软件队列Software Staging Queue

一般称为software queue、ctx(context)

对应于数据结构blk\_mq\_ctx

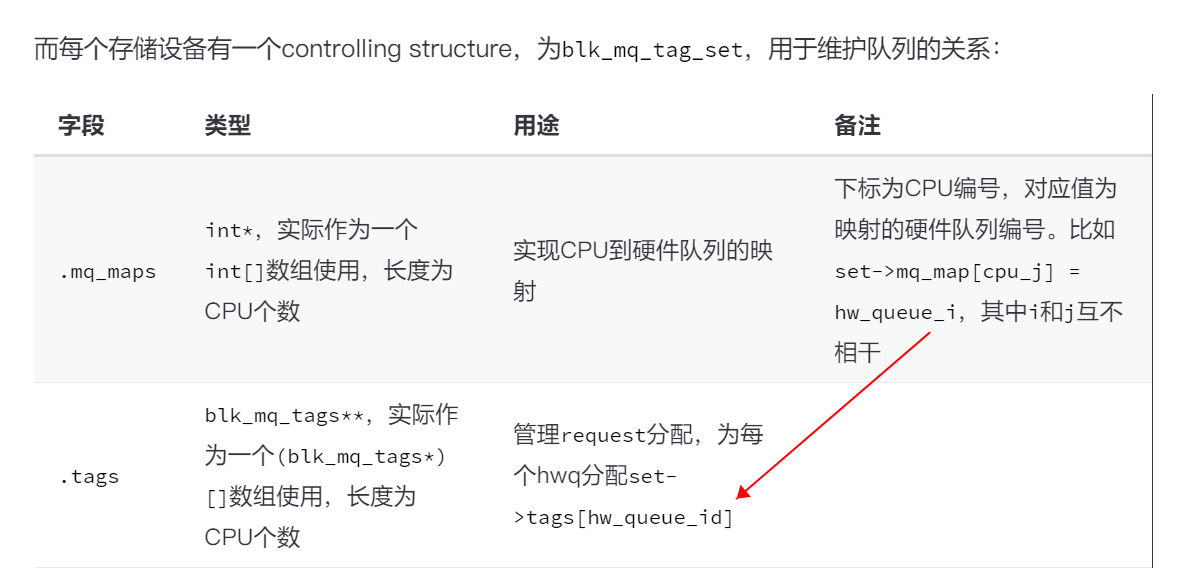
* 对应于存储设备硬件队列的Hardware Dispatch Queue

一般有hardware queue、hctx(hardware context)、hwq等奇怪命名

对应于数据结构blk\_mq\_hw\_ctx

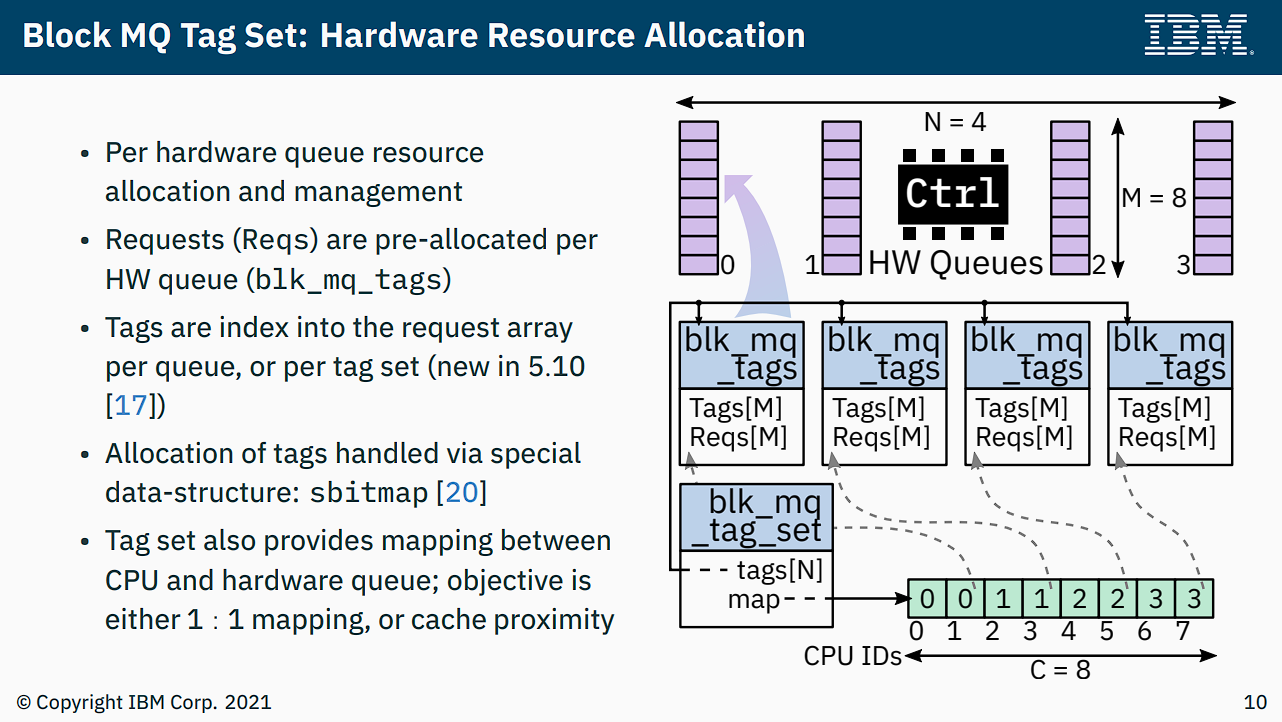
进入该队列的request意味着已经经过了调度

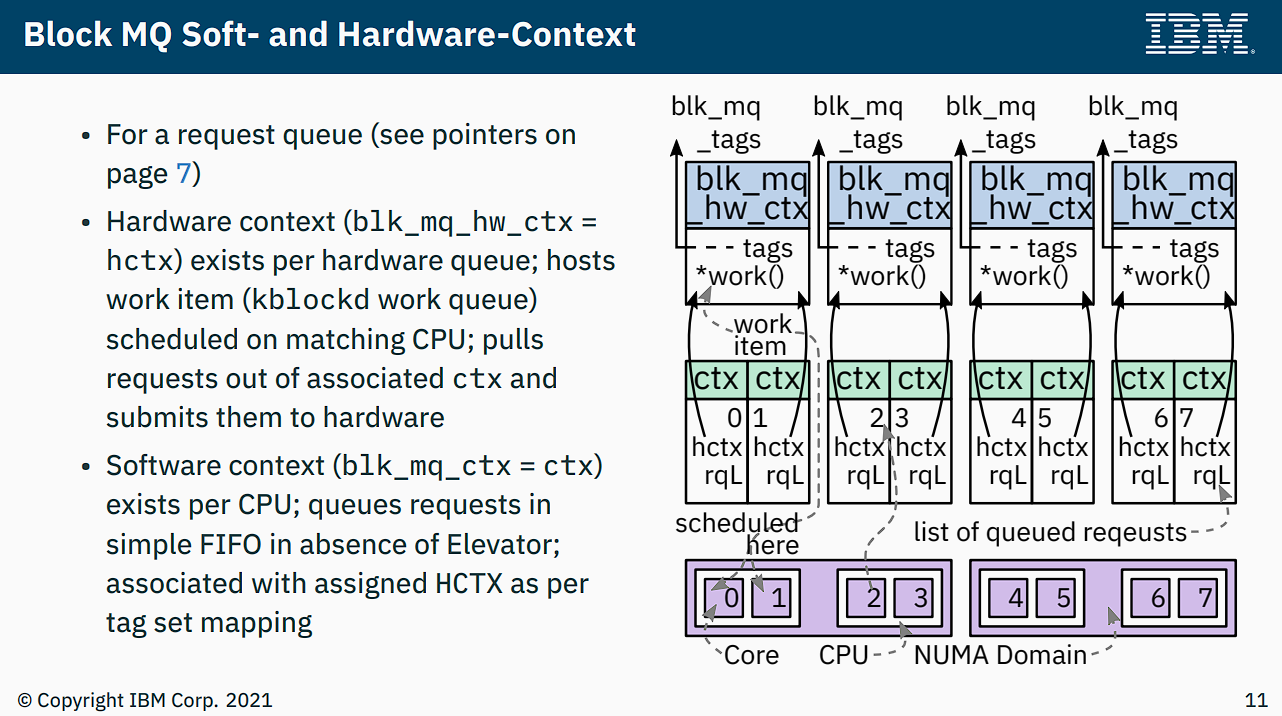
而每个存储设备有一个controlling structure，为blk\_mq\_tag\_set，用于维护队列的关系：





tag-set与ctx的关系可以看下图：





框架的初始化

## 流程之nvme\_probe

blk-mq框架在driver层完成初始化，以nvme设备为例，初始化阶段分为上下半部。上半部开始于nvme\_probe函数：

nvme\_probe(...)

dev = kzalloc\_node(...)

...

INIT\_WORK(..., nvme\_reset\_work)

在异步流程中

nvme\_reset\_work(work)

dev = container\_of(work, ...)

...

nvme\_dev\_add(dev)

...

nvme\_dev\_add(dev)

dev->tagset.ops = nvme\_mq\_ops

dev->tagset.nr\_hw\_queues = ...

hwq的数目最终会被限制到min(硬件队列数，CPU数)

dev->tagset.queue\_depth = min(dev->q\_dep, 10240) - 1

这里tagset的队列深度还不是最后确定的，如果后续过程构造失败，kernel会尝试深度折半继续重试，直到深度只有1

dev->tagset.flags = BLK\_MQ\_F\_SHOULD\_MERGE

blk\_mq\_alloc\_tag\_set(alias set = dev->tagset)

set->tags = kcalloc\_node(nr\_cpu\_ids, sizeof \*, ...)

这里说明tags是一个元素类型为blk\_mq\_tags\*，长度为CPU数的数组

set->mq\_map = kcalloc\_node(nr\_cpi\_ids, sizeof, ...)

mq\_map是一个元素类型为int，长度为CPU数的数组

blk\_mq\_update\_queue\_map(set)

这个过程完成了CPU到hw queue的映射

for-each cpu: set->mq\_map[cpu] = 0

set->ops->map\_queues(ret)

对应于实现nvme\_pci\_map\_queues

for-each hwq: for-each cpu-in-mask: set->mq\_map[cpu] = queue

blk\_mq\_alloc\_rq\_maps(set)

构造set->tags[0...hctx\_max-1]

忽略深度折半的特殊情况

for-each hwq, i: \_\_blk\_mq\_alloc\_rq\_map(set, alias hctx\_idx = i)

构造set->tags[hctx\_idx]

set->tags[hctx\_idx] = blk\_mq\_alloc\_rq\_map(set, hctx\_id, ...)

获取numa\_node node

定义（对应队列的）tags = blk\_mq\_init\_tags(...)

tags->nr\_tags和tags\_nr\_reserved\_tags的确认

以及sbitmap的构造

tags->rqs = kcalloc\_node(nr\_tags, sizeof \*)

rqs是一个元素类型为request\*的长度为队列深度的数组

tags->static\_rqs = kcalloc\_node(nr\_tgags, sizeof \*)

return tags

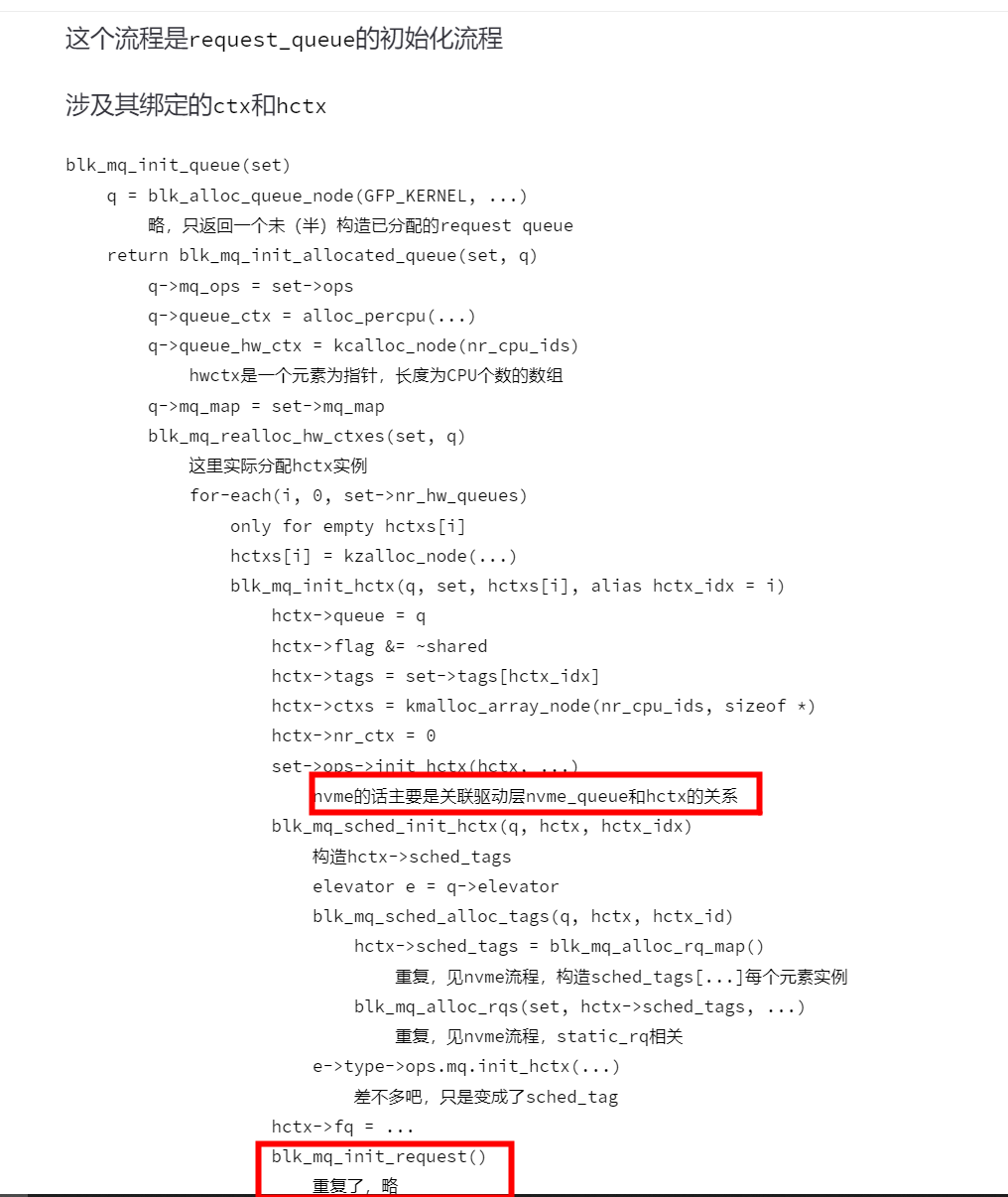
blk\_mq\_alloc\_rqs(set, set->tags[hctx\_id], hctx\_id, queue\_depth)

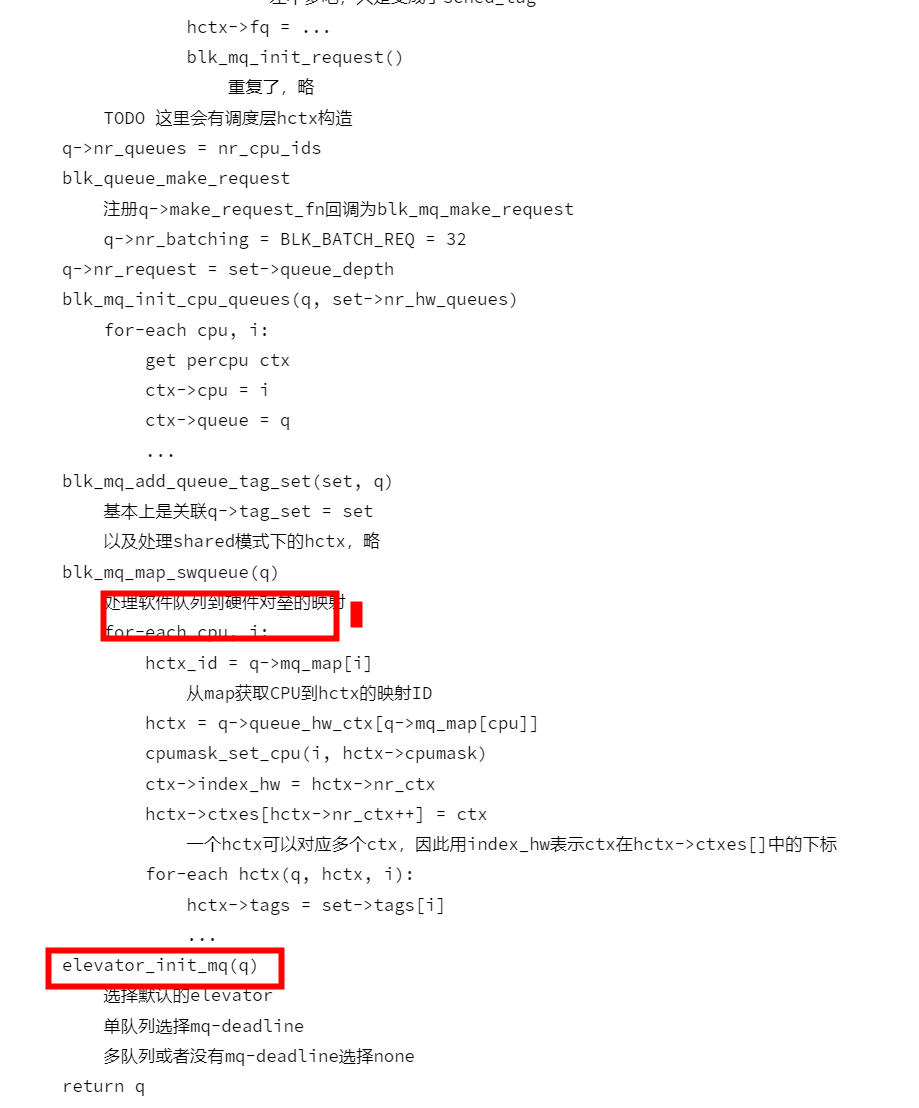
构造tags->page\_list，按队列深度d将带payload的request大小乘上d，从buddy分配对应的page，并且用page的虚拟地址存放到static\_rqs[...]，其中多个page可以通过page\_list遍历到

分配request后，可以从set->ops->init\_request自定义初始化request

dev->ctrl.tagset = dev->tagset

## 流程之blk\_mq\_init\_queue

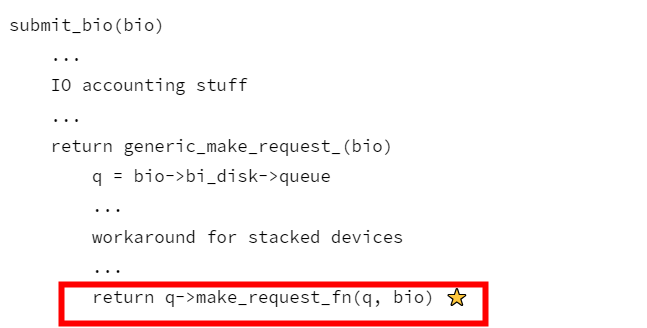




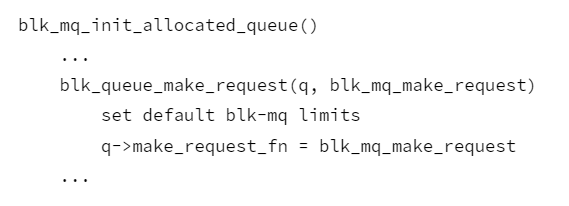
## 框架的IO处理

### 流程之提交IO

userspace进行的IO操作，具体的IO类型和操作对象会以bio结构体描述，并在内核中会通过submit\_bio统一接口进行提交



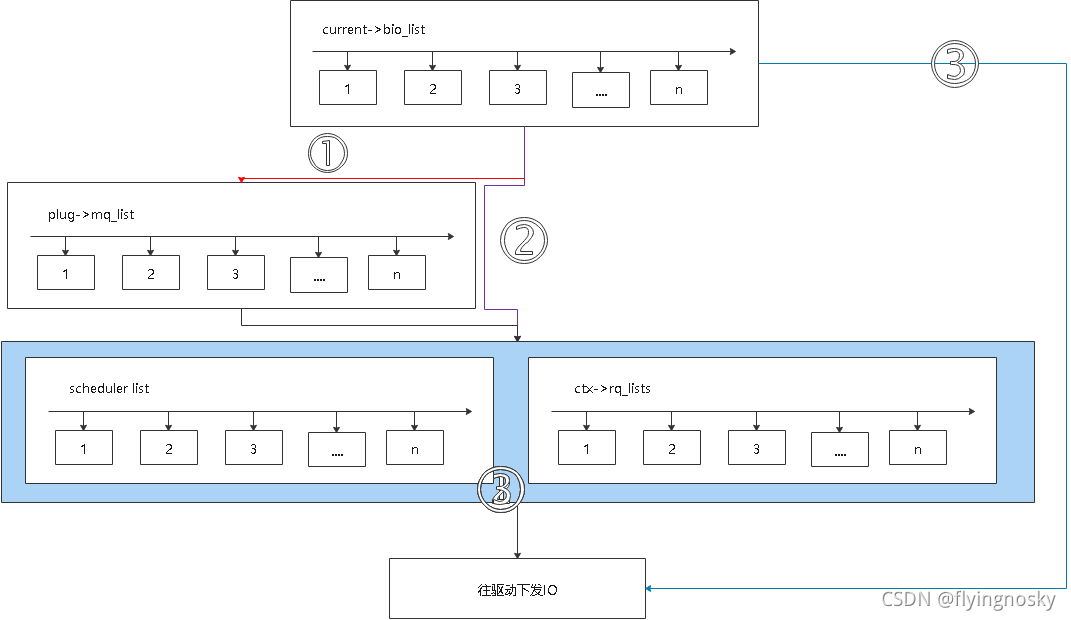
从前面的流程可以知道，blk-mq下的make\_request\_fn注册实例为blk\_mq\_make\_request



### Submit IO 的3条路径

对于每个线程，存在成员current->bio\_list来放置本线程发送的BIO。这里有三条路径发送IO到底层：

1. 路径一，使能了plug/unplug机制，此时会等待plug池中存取足够的IO后统一往调度器插入IO，并选取IO下发；
2. 路径二，没有使能plug/unplug机制，此时会将IO插入调度器中，并选取IO下发；
3. 路径三，跳过调度层，直接下发IO；



简单来讲，这里有三条链：

1. current->bio\_list存储在当前线程的所有bio;
2. plug->mq\_list使能plug/unplug机制时存放在缓存池的bio；
3. 若定义IO调度层，IO请求会发送到scheduler list中；
4. 若没有定义IO调度层，IO请求会发送到ctx->rq\_lists。

上述下发路径是IO提交下发的简述，代码流程如下：



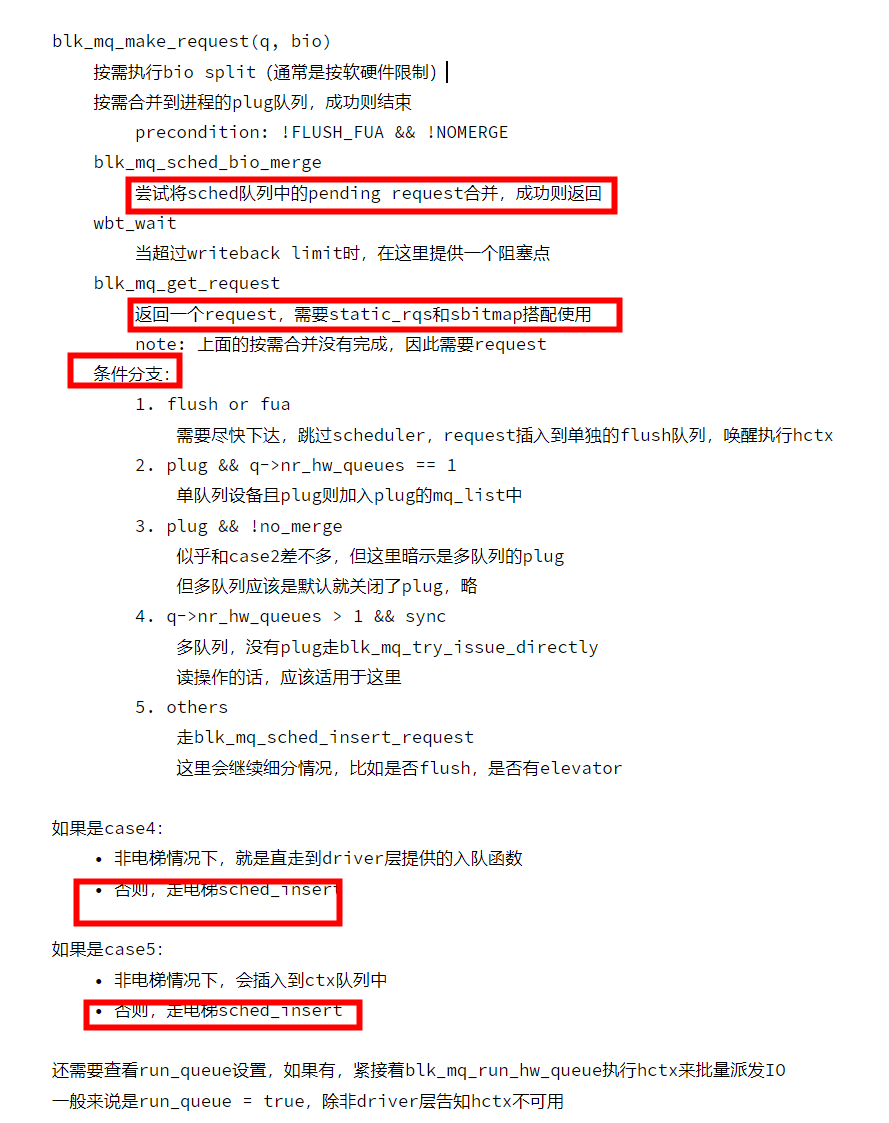
过程如下：

1. 每个线程若已经在执行blk\_mq\_submit\_bio()，将新下发BIO链入到线程current->bio\_list;
2. 依次处理current->list中的每个bio；
3. 若bio中存在数据在高端内存区，在低端内存区分配内存，将高端内存区数据拷贝到新分配的内存区，称为bounce过程，后面单独一节介绍；
4. 检查请求队列中的bio，若过大进行切分，称BIO的切分；
5. 尝试将bio合并到plug->mq\_list中，然后尝试合并到IO调度层链表或ctx->rq\_lists中；
6. 若没有合并，分配新的request；
7. 若定义plug，且没达到冲刷数目，加入到plug->mq\_list；若达到冲刷数目，将冲刷下发（plug/unplug机制）；
8. 若定义IO调度器，往IO调度器中插入新的request（对于机械硬盘，通过IO调度层座合并和排序，有利于提高性能）;
9. 若没有定义IO调度器，可以直接下发（对于较快的硬盘如nvme盘，进入调度层可能会浪费时间，跳过IO调度层有利于性能提升）；

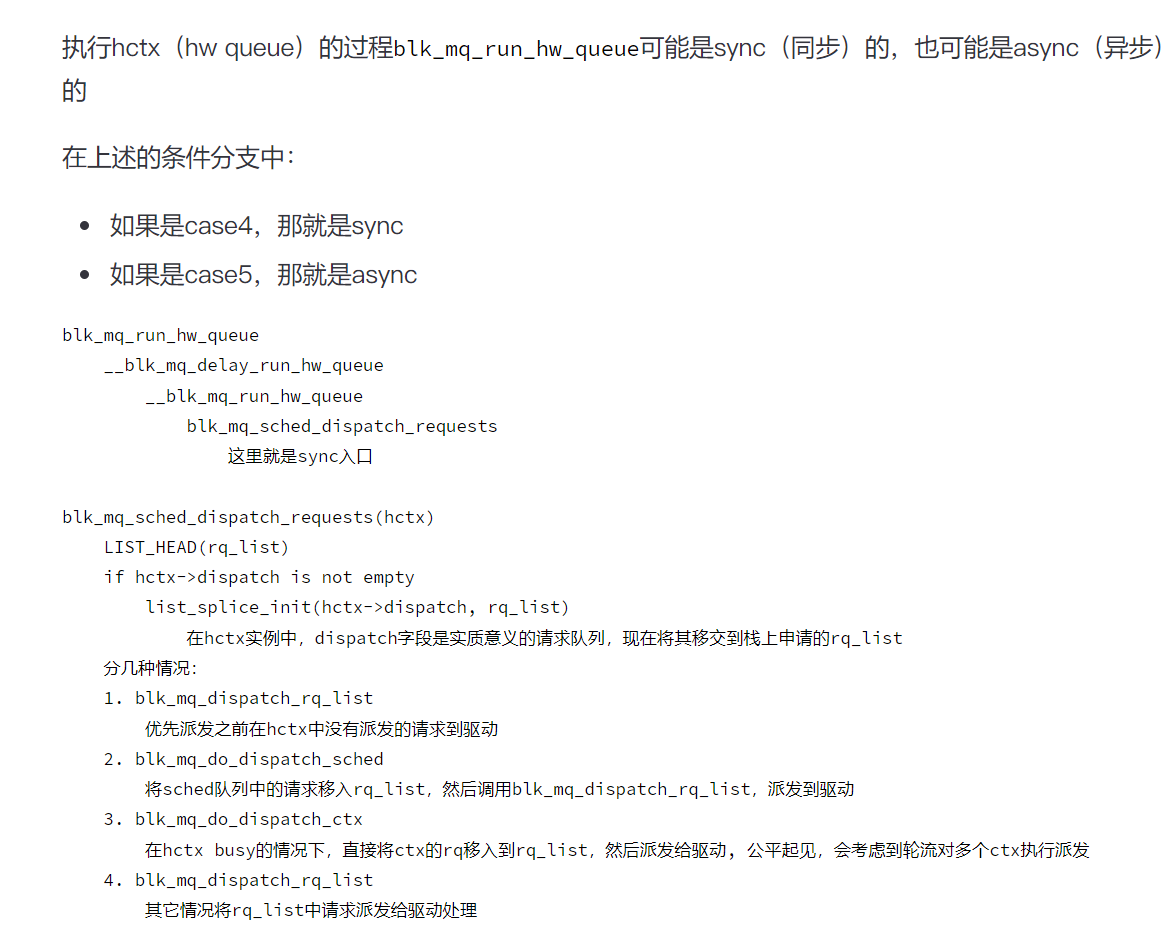
## 流程之处理IO

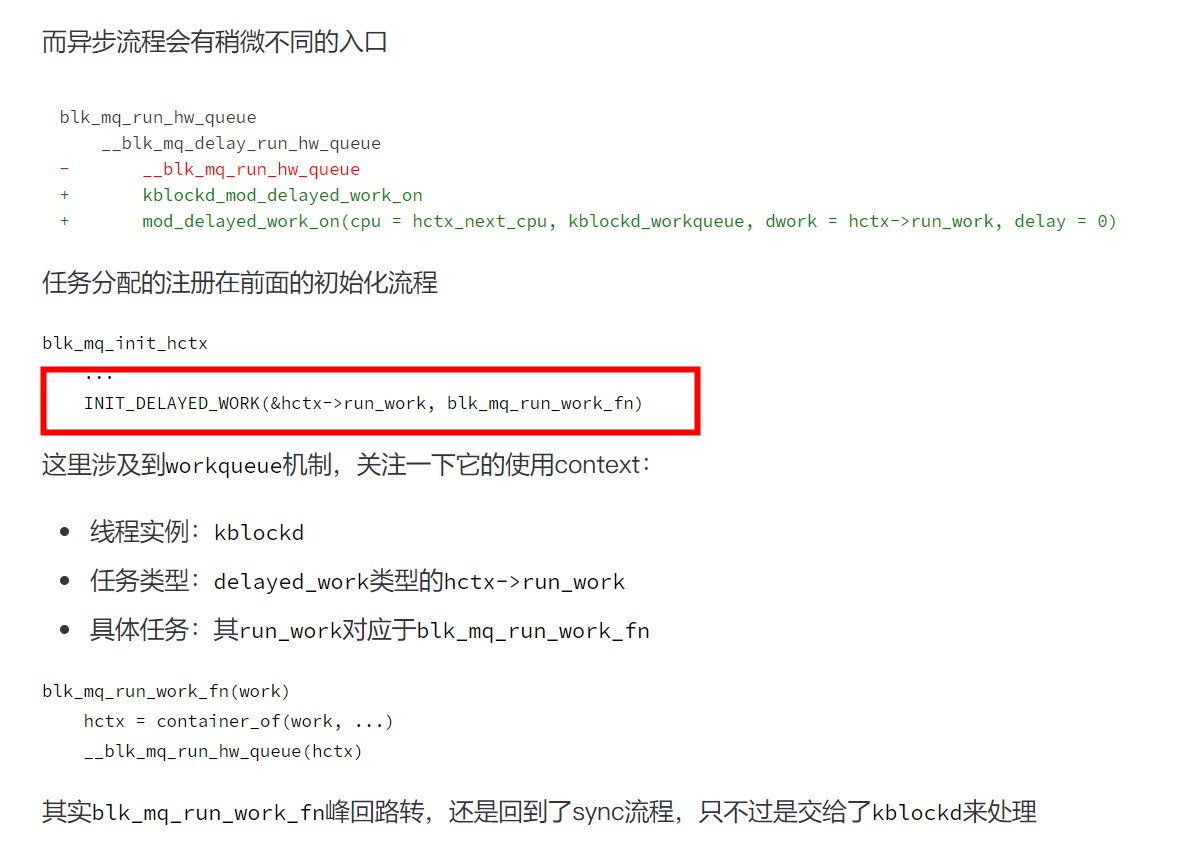
处理IO简而言之就是把bio转换为request结构体，并插入到请求队列中

相较于提交IO是在当前进程的内核栈上进行，处理IO还有可能会在kblockd内核线程中异步执行



* **同步异步**





* 不太重要的细节

1. blk-mq的硬件队列与驱动层的队列无关
2. 虽然软件队列一般认为是per-cpu级别，但是maintainer也指出：如果在NUMA架构中，L3缓存足够大的话，软件队列可以设置为per-socket级别，这样也许能从cache友好和锁竞争中获取一个平衡点
3. 硬件队列个数在不同的场合下是有歧义的，因为kernel里面会把超过CPU个数的硬件队列数目当作看不见（原因是超出部分没有意义），所以并不绝对等于硬件意义上的硬件队列个数
4. tag虽然是给硬件队列使用，但是blk\_mq\_tags实际长度是按CPU个数给的
5. tag对应的request数虽然是set提供的队列深度数，但是每次分配失败的话，会尝试把队列深度数目折半，这也会实际影响到set->queue\_depth
6. 预分配request的每个实例中其实还藏有driver层所需要的payload
7. ns->queue即是request\_queue实例
8. 提交IO过程中，generic\_make\_request已经随着SQ框架的移除也被移除，改为blk\_mq\_submit\_bio，不过本质不变
9. 处理IO过程中，case4走sync运行hctx是因为IO操作本来就是sync类型的
10. “派发到driver层”指的是最终被queue\_rq接口调用，具体实现和[驱动](https://elixir.bootlin.com/linux/v4.18.20/source/drivers/nvme/host/pci.c" \l "L1524)是强相关的

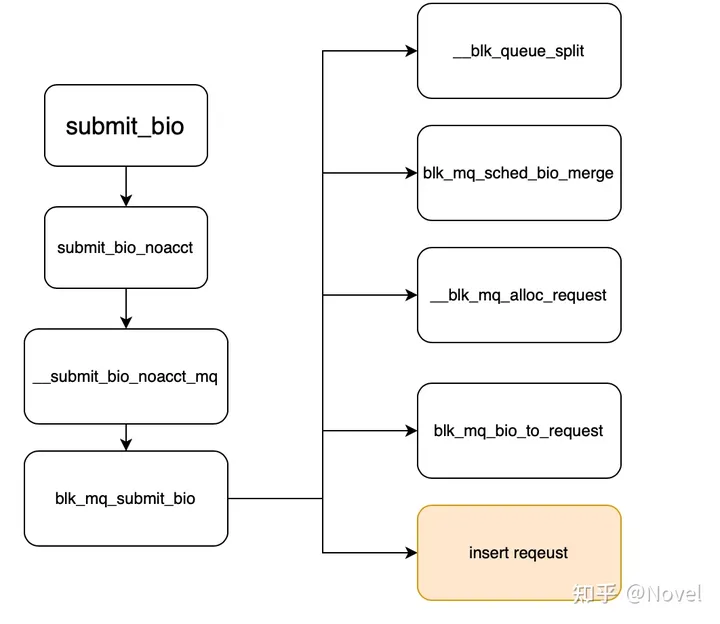
### submit\_bio and insert request

block层提供了submit\_bio的接口， 上层可以调用这个接口来提交请求。其处理流程大致可以分为两步：

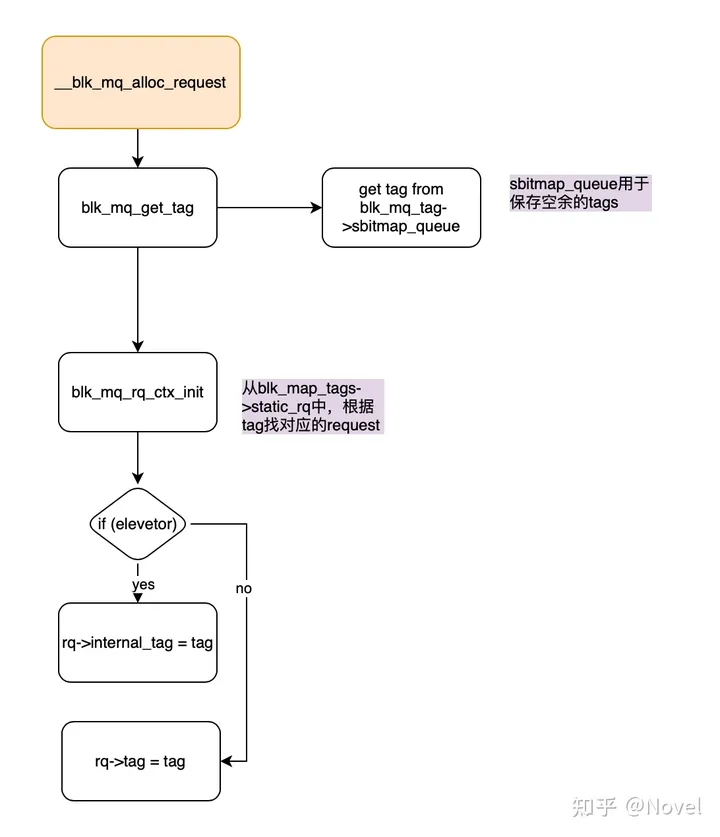
* 将请求插入到某个队列中
* 从队列中取出请求进行处理

#### insert request

请求处理的入口函数是submit\_bio, 它的调用流程大致如下

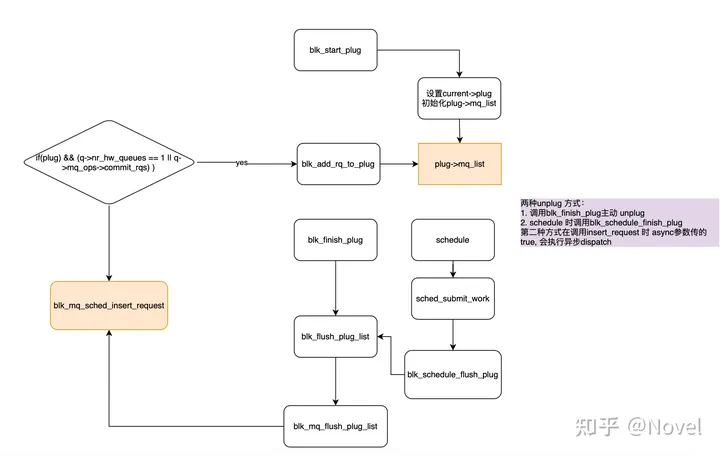


其中， alloc\_request的流程如下：



blk\_mq\_hw\_ctx中有一个blk\_map\_tags用于保存当前的request， 在初始化时， 会根据队列深度初始化一个静态的request数组（blk\_map\_tags->static\_rq）, 申请request时直接从这个数组中取， request结构中有一个internal\_tag（tag)作为在这个数组中的索引。

insert request时， 会先尝试插入到plug队列中， 整体流程大致如下：



* plug队列可以理解为一种“蓄流”的方式， 其大致使用方法如下：

1. blk\_start\_plug()
2. ..submit io...
3. blk\_finish\_plug()

调用blk\_start\_plug之后， 提交的IO请求会被缓存起来， 然后在适当的时机一并提交。

plug->mq\_list会保存当前的请求链表

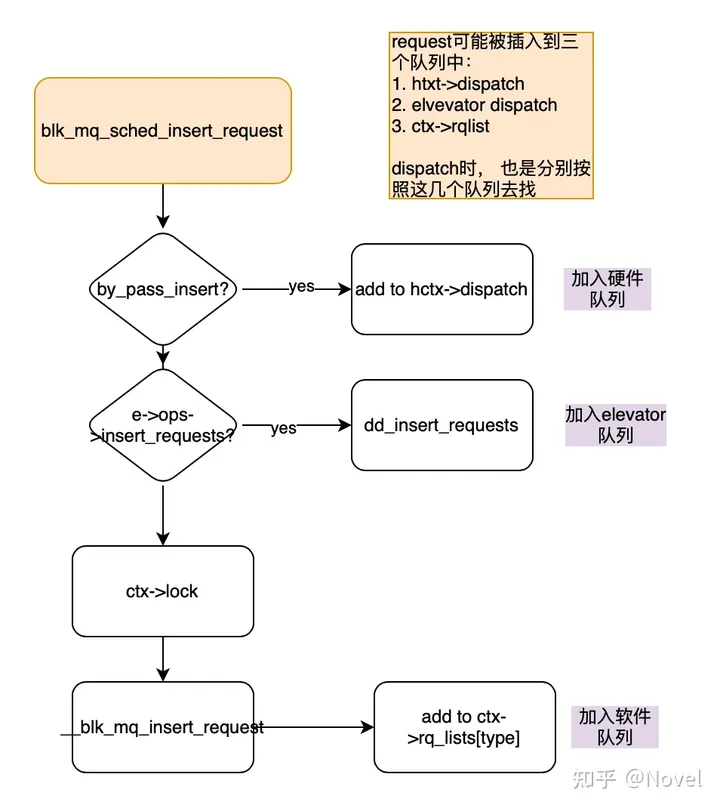
* "泄流"的时机有两种：

调用blk\_finish\_plug主动 unplug

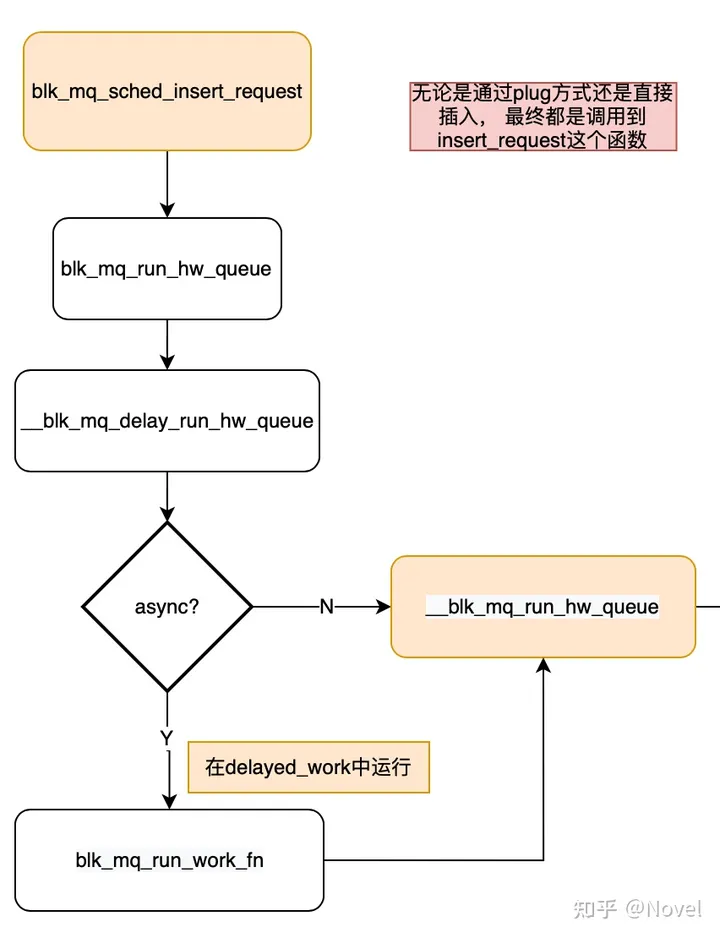
schedule 时调用blk\_schedule\_finish\_plug

**无论是通过plug方式， 还是直接提交， 最终都会到blk\_mq\_sched\_insert\_request这个函数**

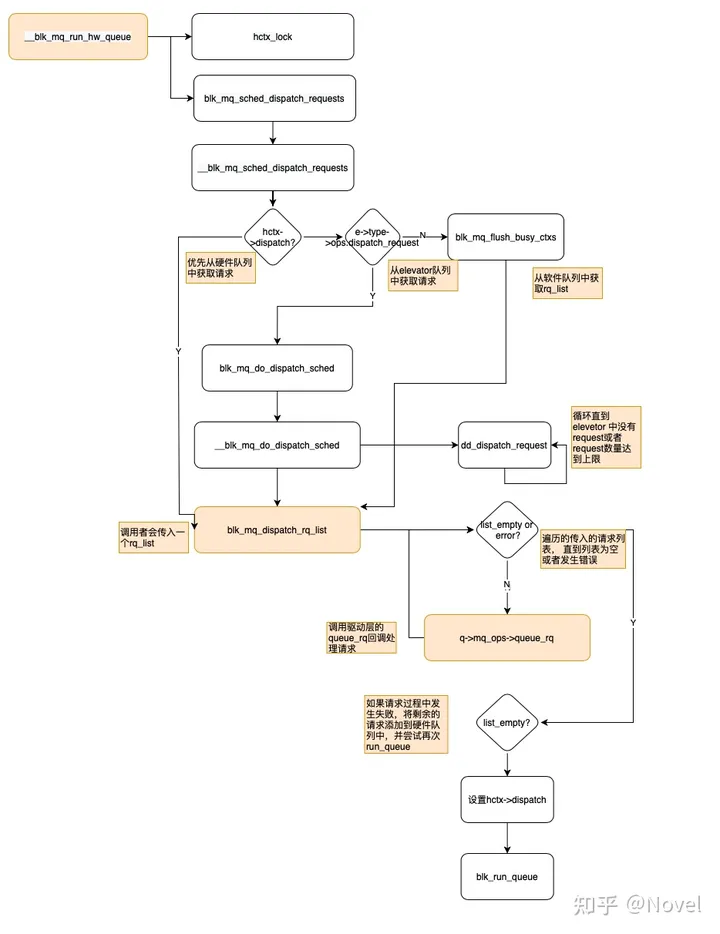
### blk\_mq\_sched\_insert\_request



blk\_mq\_sched\_insert\_request最终会尝试run queue, 走到dispatch request流程



#### dispatch request



dispatch的整体流程如上。

在insert request时， request可能被插入到三个队列中：硬件队列， 软件队列， elevator\_queue。

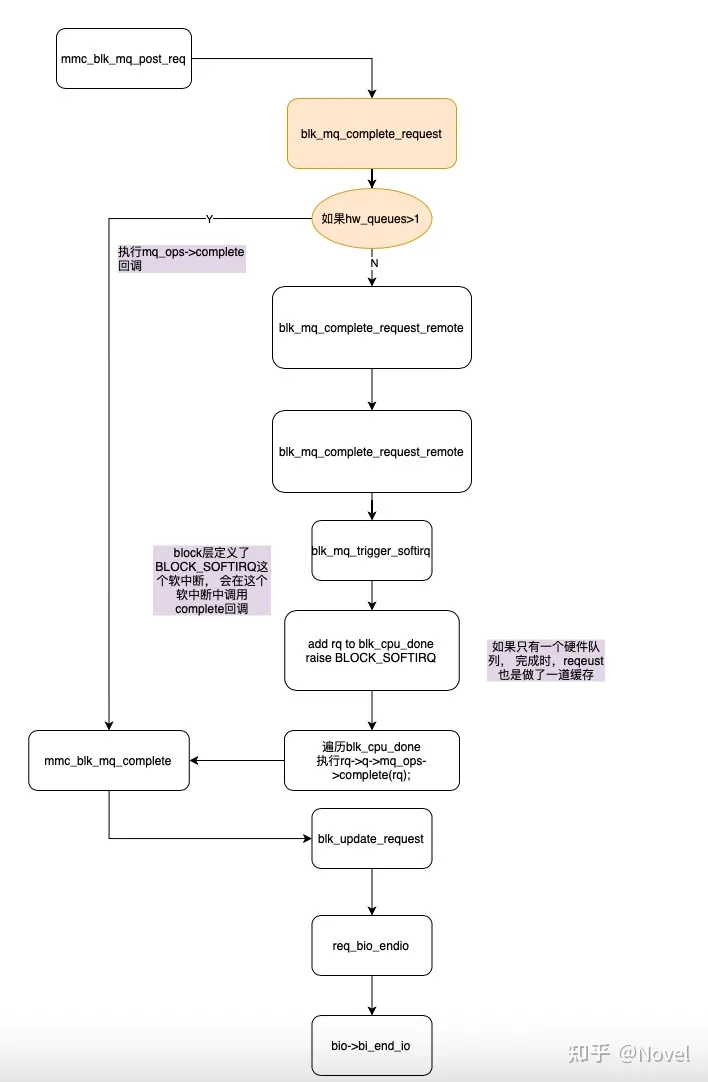
dispatch时， 也是分别尝试从这三个队列中取出请求。然后会将请求组成一个链表rq\_list， 作为参数传入blk\_mq\_dispatch\_rq\_list

这个函数中主要是调用了驱动层的queue\_rq回调来处理具体的请求。

## 请求完成流程

请求处理是一个异步的流程， 当请求完成后， 驱动层也需要通过完成回调通知block层，这样才是完成了请求。

总的来说，每个bio会设置一个完成回调， 当请求完成时，驱动层会调用到block层的回调函数， 进而调用到bio的完成回调。请求完成的处理大致如下：



block层会根据hw\_queues的数量决定调用回调的方式：

1. 如果有多个hw\_queue， 直接调用complete回调
2. 否则， 会触发BLOCK\_SOFTIRQ这个软中断， 在软中断中调用complete回调。

在blk\_update\_request中， 会最终调用到bio->bi\_end\_io, 通知上层请求完成。