# 多对列

blk\_mq 的API实现了两级块层设计，该设计使用两组独立的请求队列。

* 软件暂存队列，按CPU分配；
* 硬件调度队列，其数量通常与blcok设备支持的实际硬件队列数量匹配。

软件暂存队列和硬件调度队列之间的映射队列数不同。现在，我们考虑两个队列放置不同的情况。

假设这里有3种情况：

1. 软件暂存队列 > 硬件调度队列

在这种情况下，两个或多个软件暂存队列被分配到一个硬件上下文中。而在硬件上下文将从所有关联的软件队列中拉入请求的同时，进行一次调度。

1. 软件暂存队列 < 硬件调度队列

在这种情况下，软件暂存队列和硬件调度队列之间的映射是有顺序的。

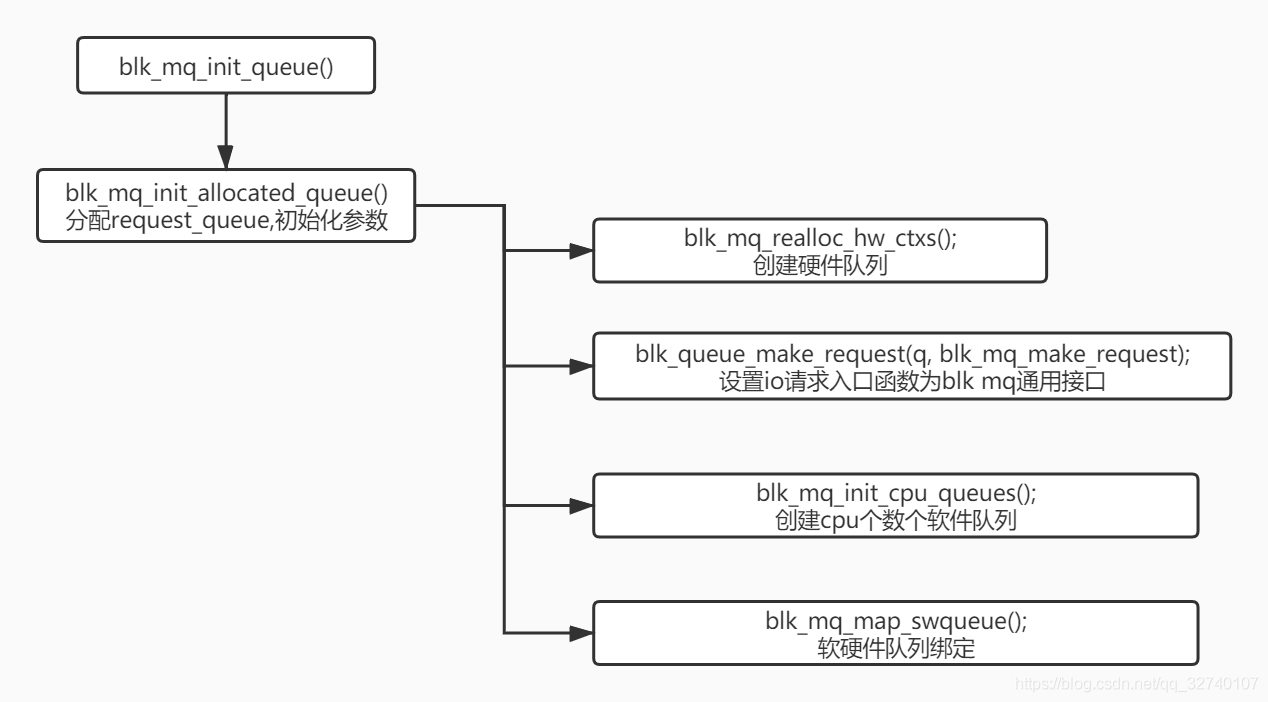
1. 软件暂存队列 == 硬件调度队列

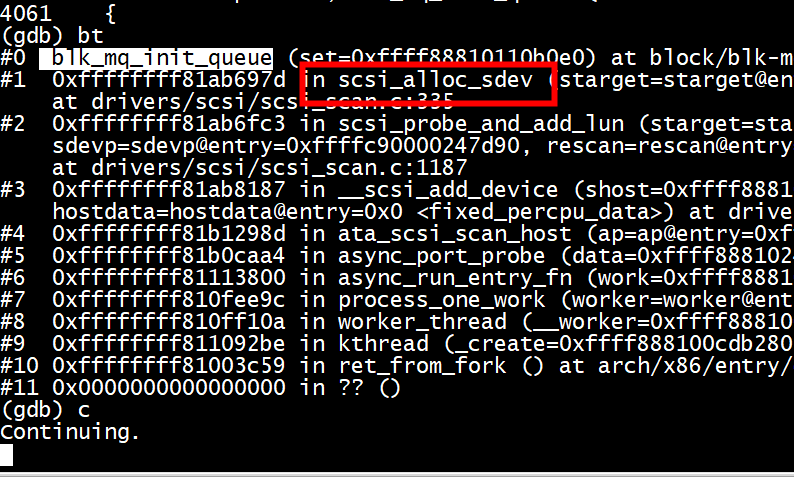
在这种情况下，这是最简单的情况，即执行直接1：1映射。

## blk\_mq\_init\_queue

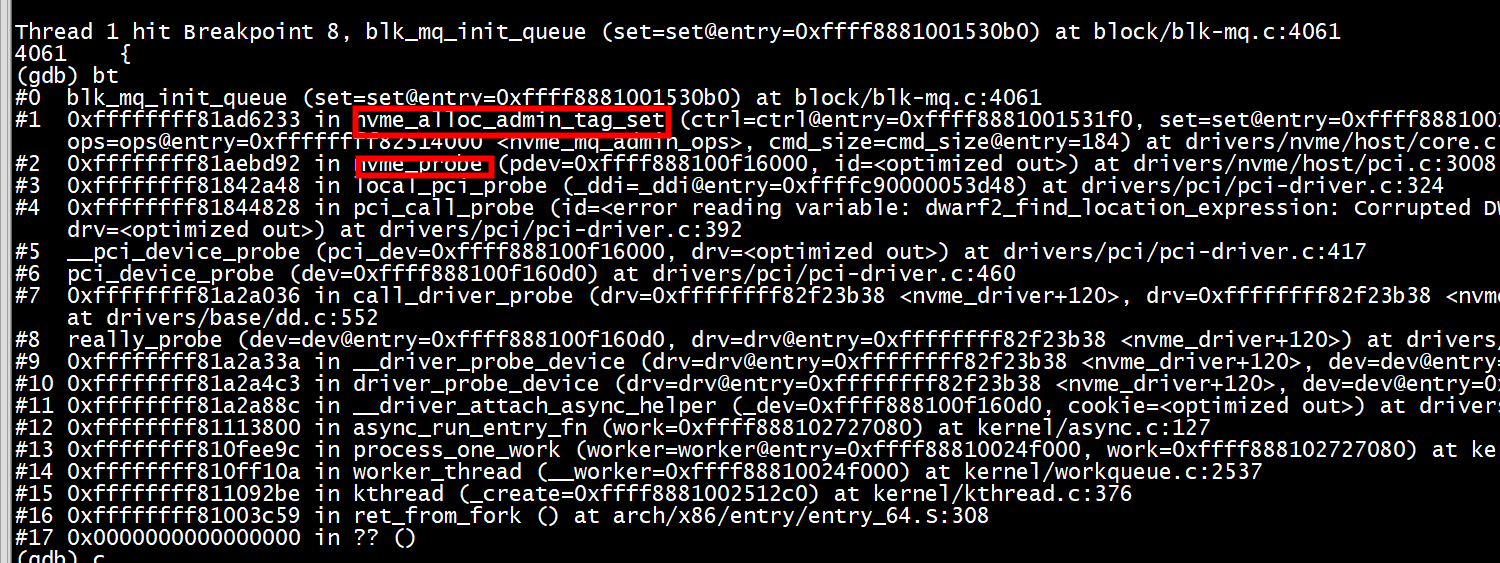
块设备初始化时通过blk\_mq\_init\_queue()创建request\_queue并初始化,主要功能包含：

* request\_queue与块设备的blk\_mq\_tag\_set相互绑定，根据blk\_mq\_tag\_set设置一些参数。
* 创建软硬件队列及进行绑定。
* 设置io请求入口函数make\_request\_fn 为 blk\_mq\_make\_request()。

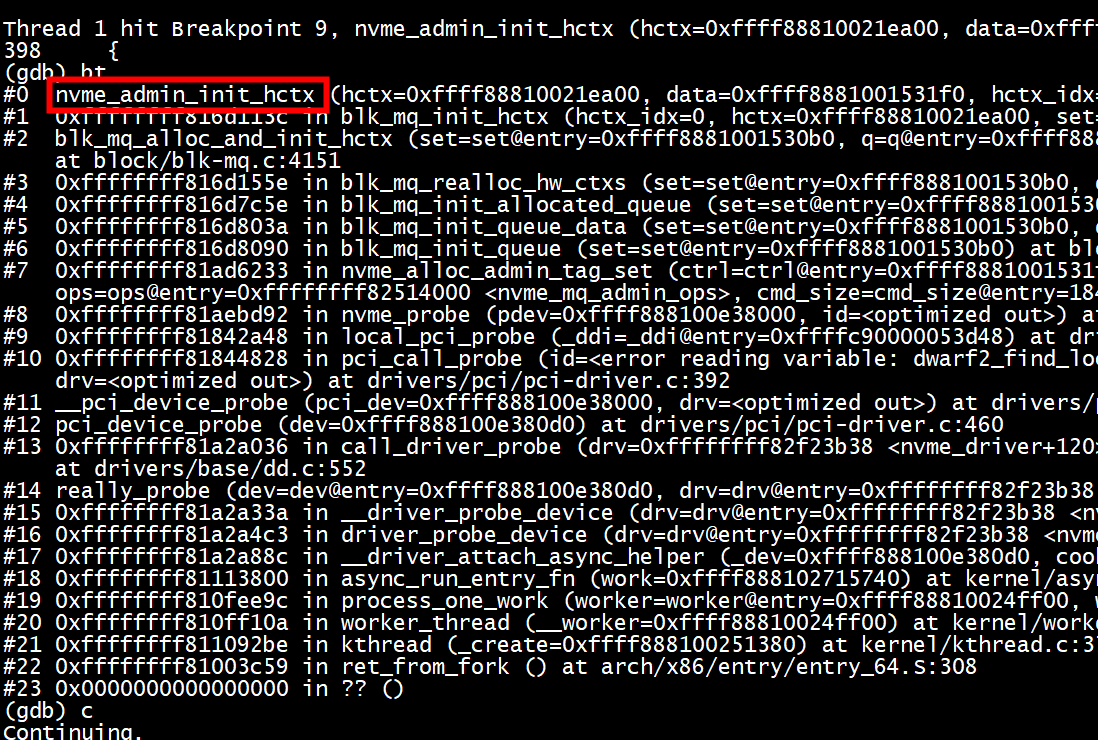




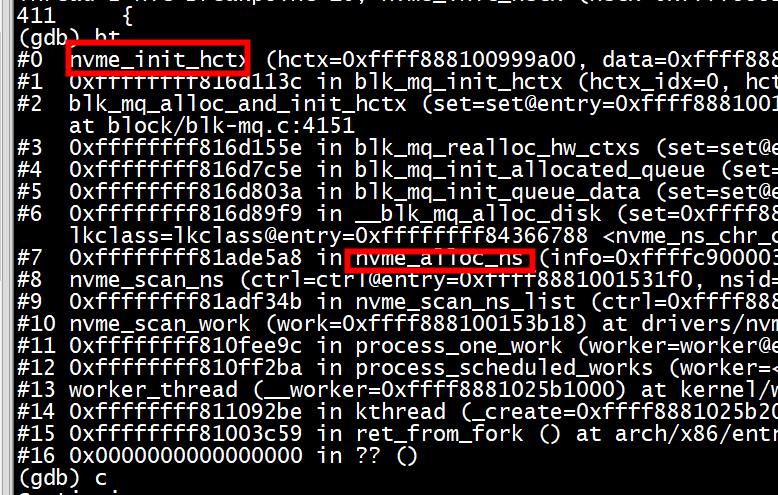
### Nvme admin



#### nvme\_admin\_init\_hctx

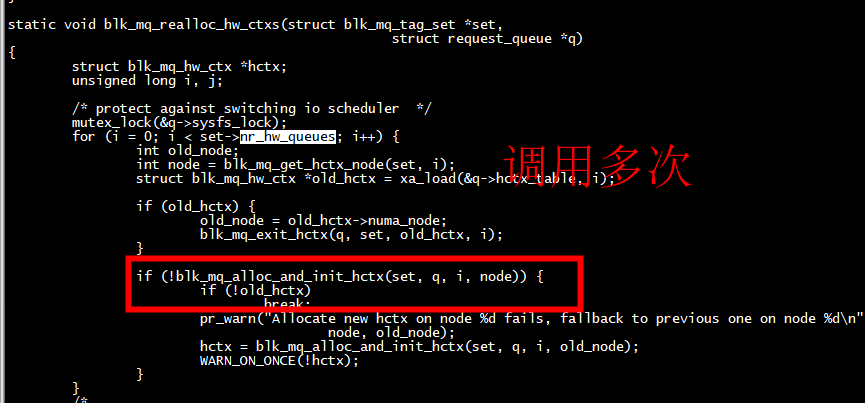


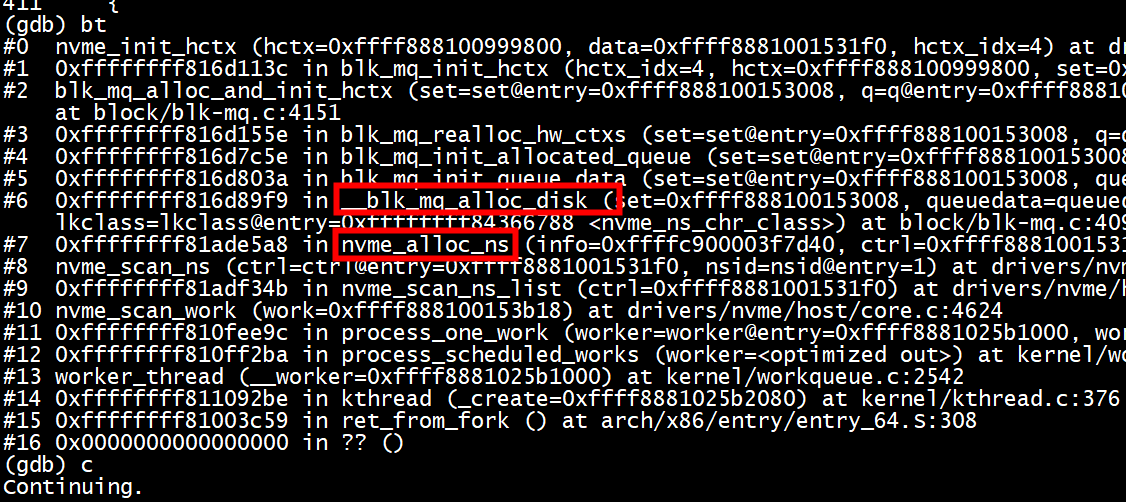
### Nvme IO

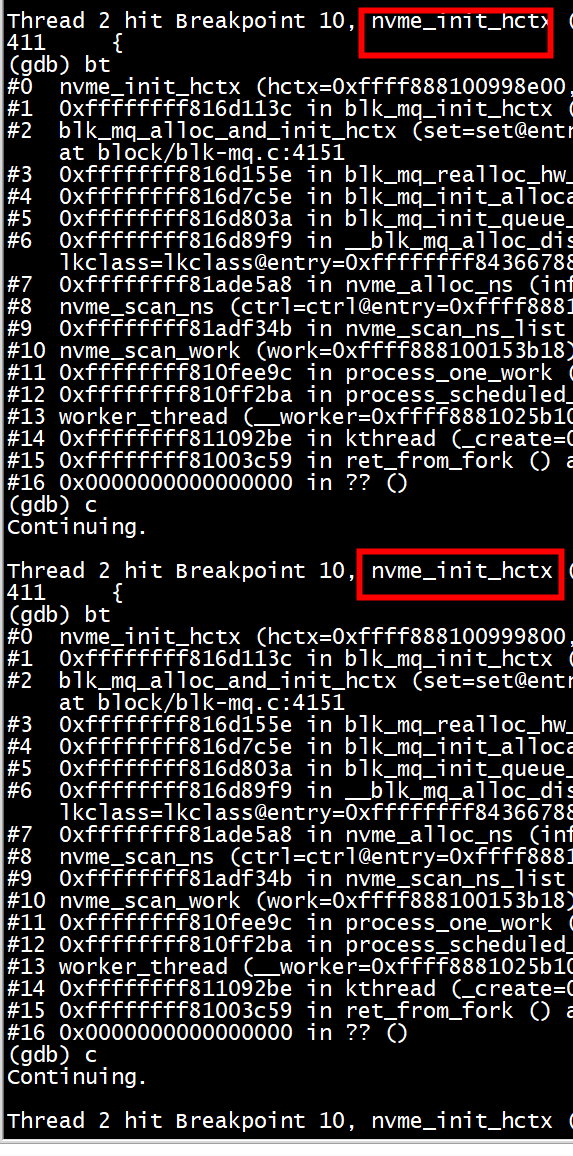


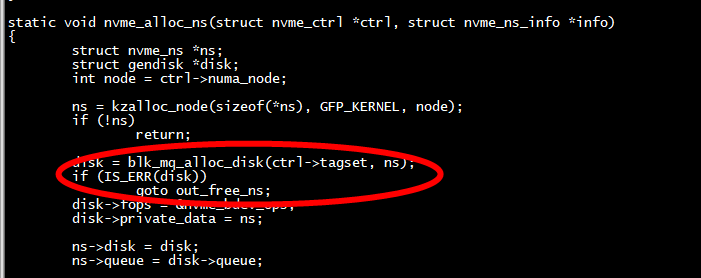
#### 创建多个hw\_queues

nvme\_init\_hctx被调用多次，

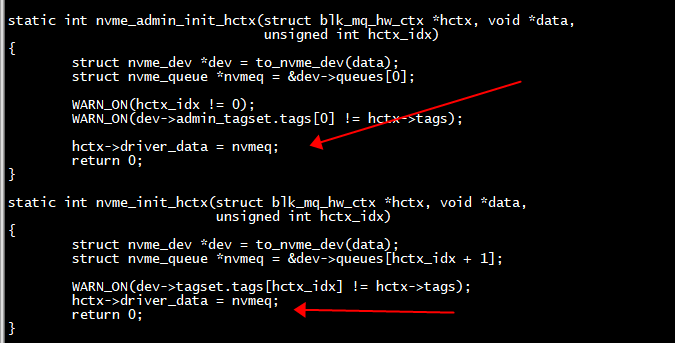








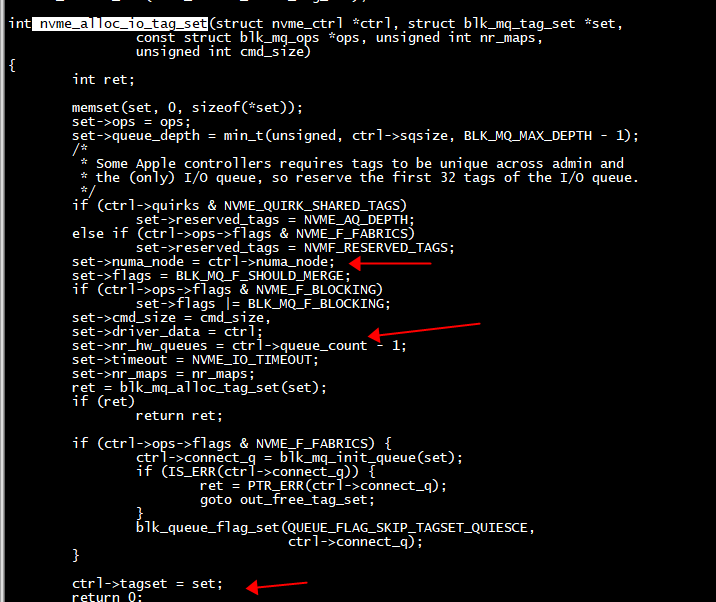
#### blk\_mq\_hw\_ctx关联 struct nvme\_queue

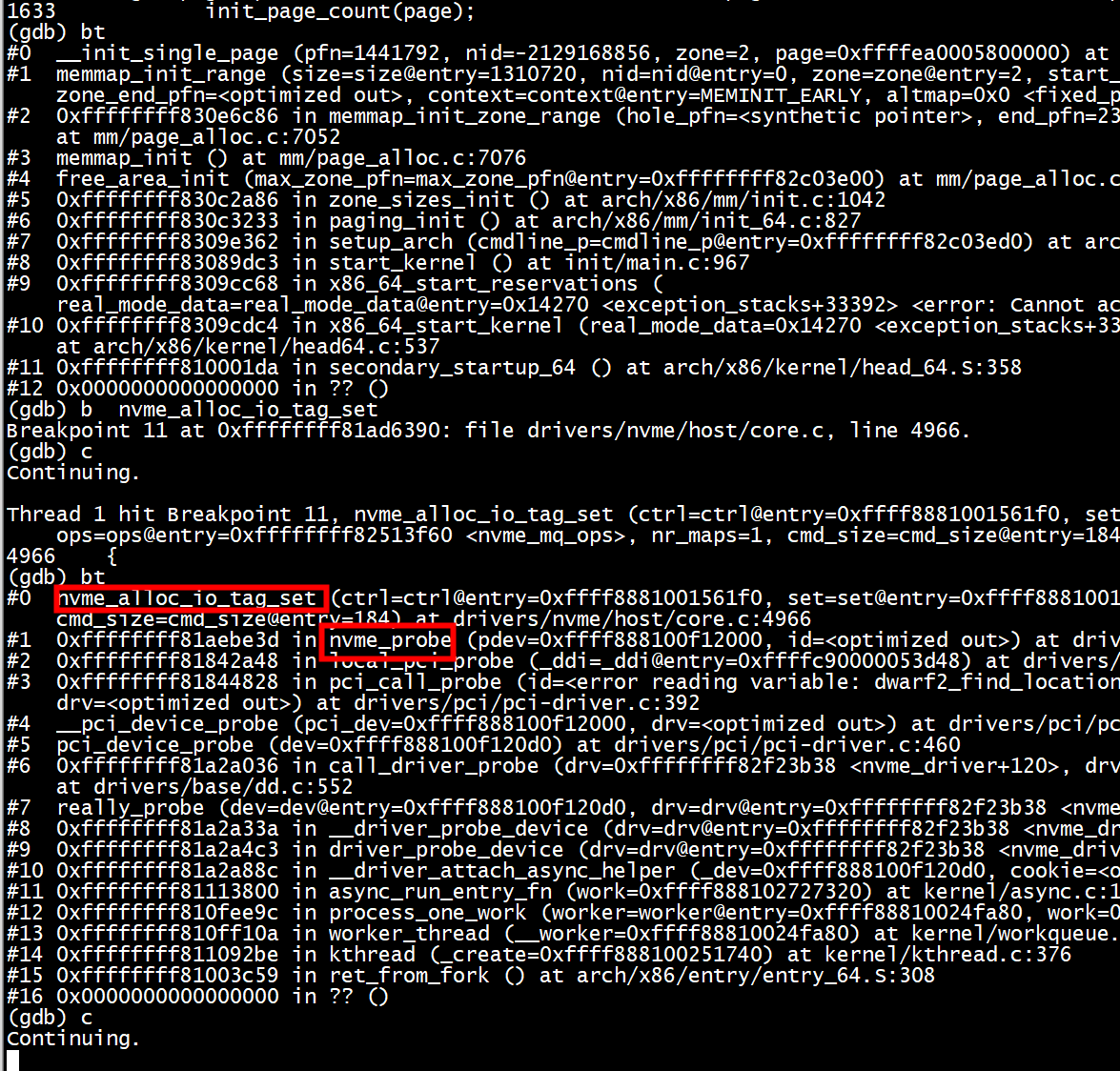


#### Tagset

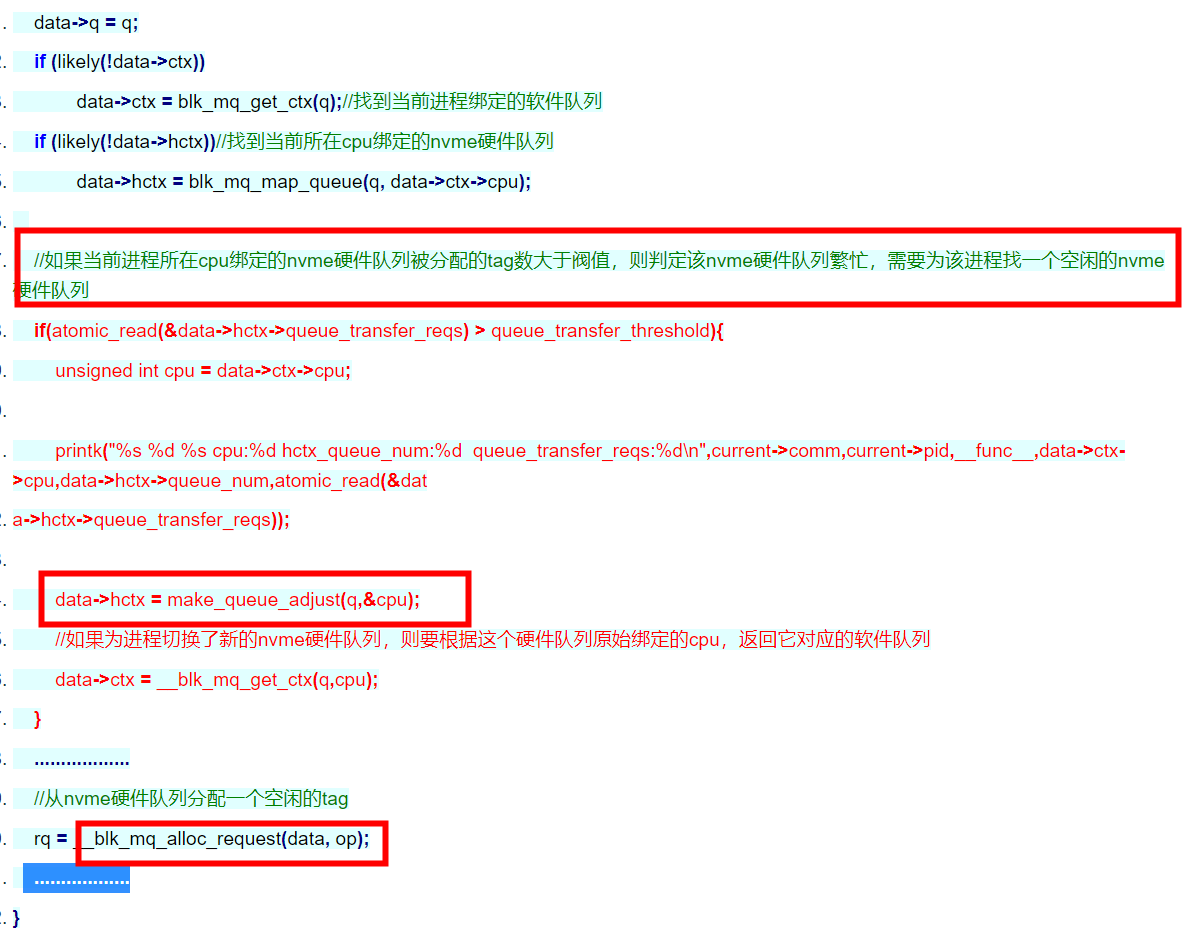
set->numa\_node = ctrl->numa\_node;

set->nr\_hw\_queues = ctrl->queue\_count - 1;





### nvme硬件队列分配

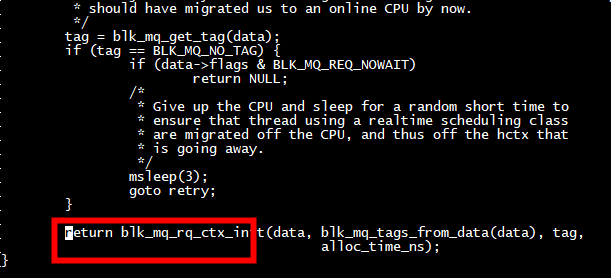


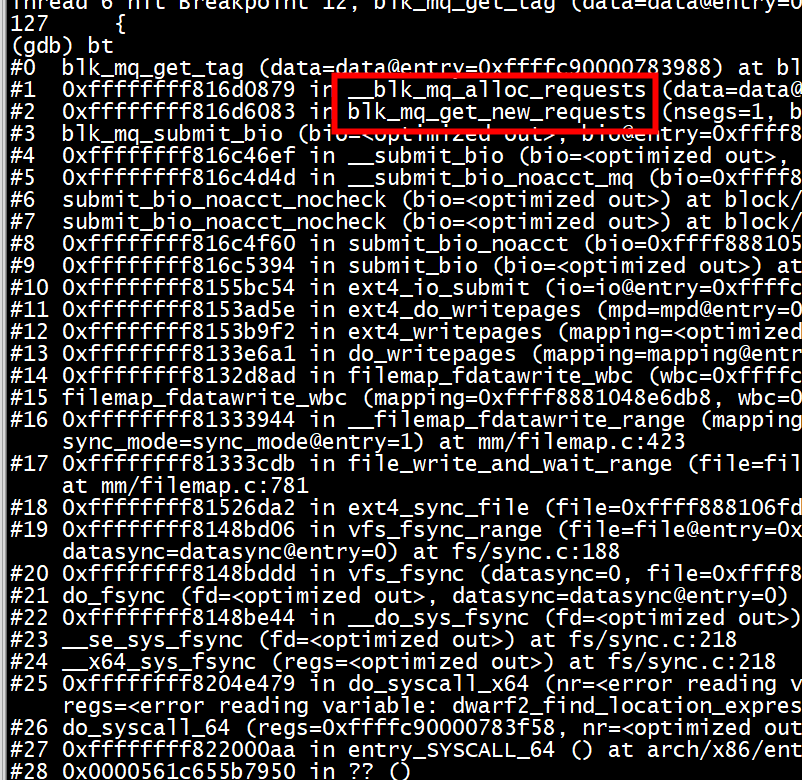
submit\_bio…->blk\_mq\_make\_request发送IO请求开始。按照内核block层多队列的要求，需要执行blk\_mq\_sched\_get\_request->\_\_blk\_mq\_alloc\_request->blk\_mq\_get\_tag为bio分配一个tag。具体细节是，找到当前进程所在cpu，继而找到该cpu绑定的nvme硬件队列，nvme硬件队列用blk\_mq\_hw\_ctx结构体表示。然后，从blk\_mq\_hw\_ctx的成员struct blk\_mq\_tags \*tags分配一个tag，测试的机器发现最多可以分配1023个tag。如果进程已经从nvme硬件队列分配了1023个tag，没有剩余tag，只能休眠等待谁释放tag。

接着，将bio转成request，执行blk\_mq\_try\_issue\_directly-> \_\_blk\_mq\_try\_issue\_directly-> \_\_blk\_mq\_issue\_directly，把IO请求的相关数据发送给nvme磁盘驱动。等nvme磁盘驱动把该IO请求的数据传输完，产生中断，在中断服务函数里释放刚才从nvme硬件队列blk\_mq\_hw\_ctx分配的tag，函数流程是

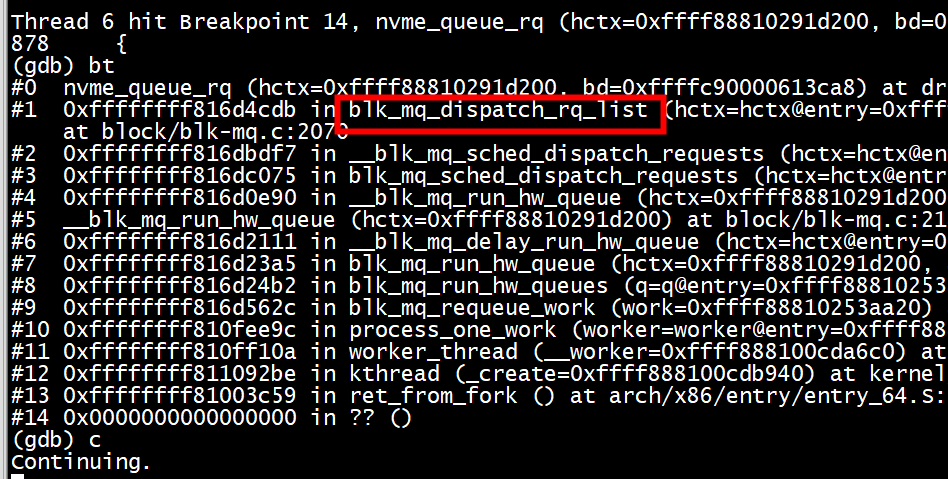
blk\_mq\_complete\_request->\_\_blk\_mq\_complete\_request->blk\_mq\_end\_request->\_\_blk\_mq\_end\_request-> \_\_blk\_mq\_end\_request->blk\_mq\_free\_request->blk\_mq\_sched\_put\_request->blk\_mq\_finish\_request->blk\_mq\_finish\_hctx\_request->\_\_blk\_mq\_finish\_request->blk\_mq\_put\_tag。

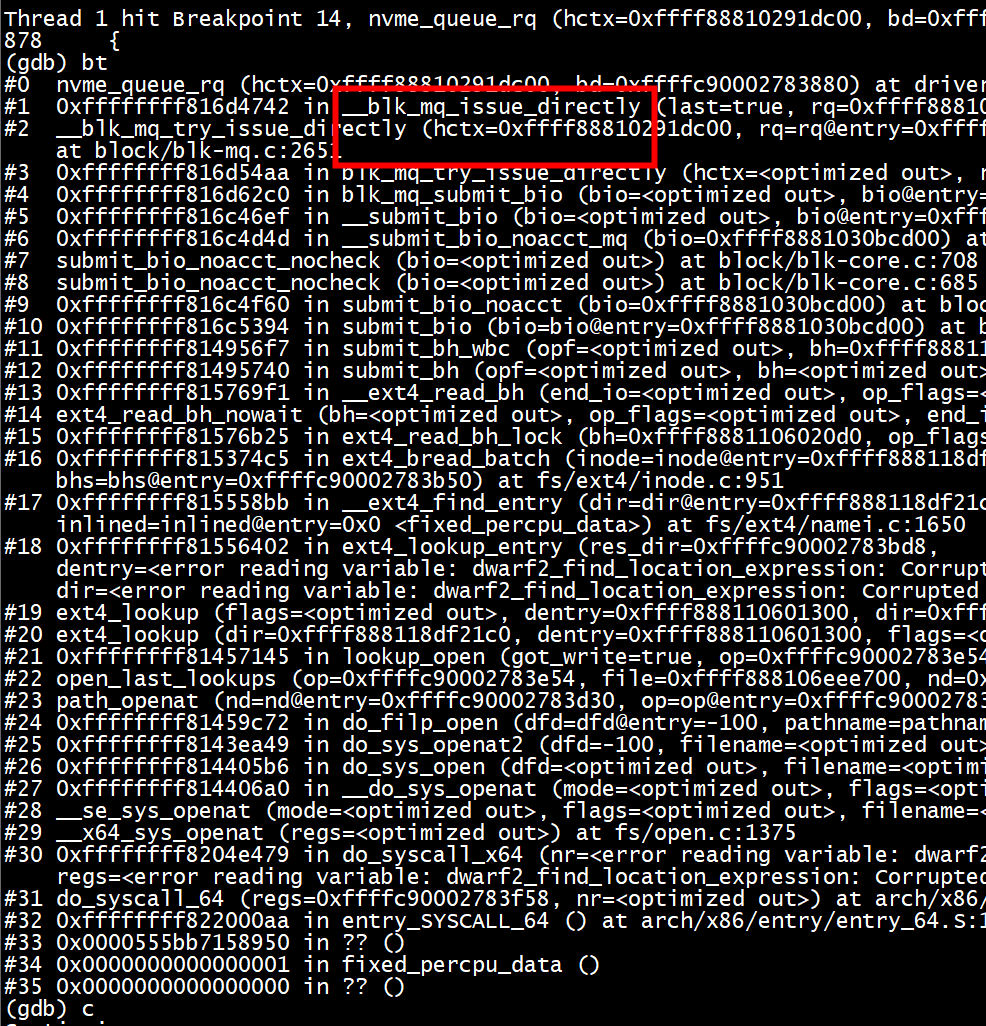
##### blk\_mq\_get\_tag





### 从blk\_mq\_hw\_ctx到 nvme\_queue\_rq



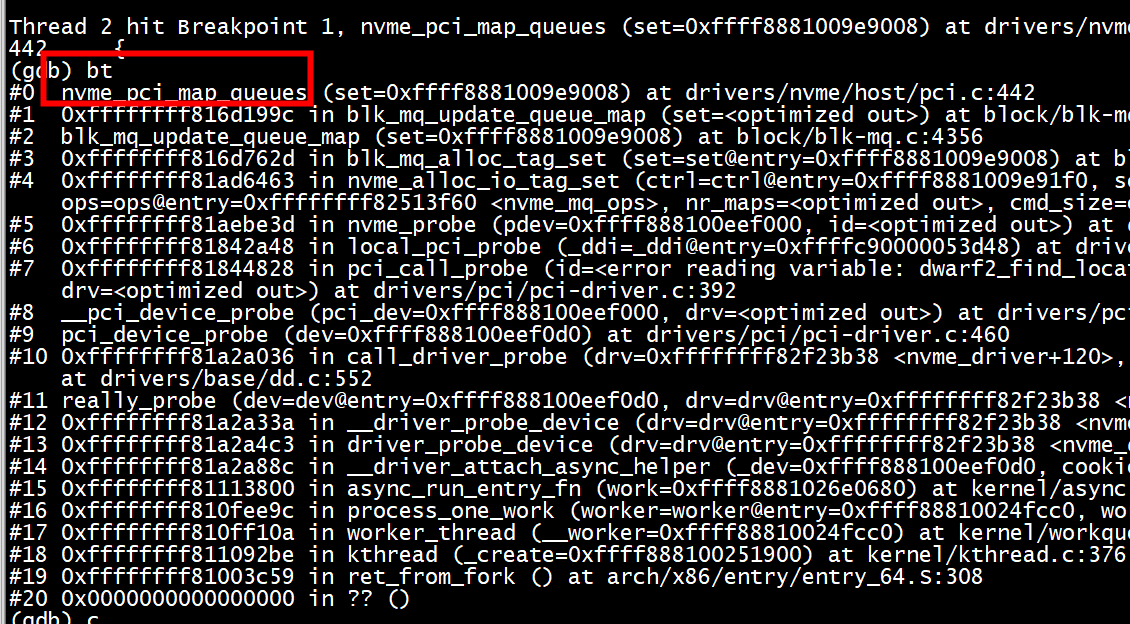


## blk\_mq\_hw\_ctx和软件ctx 映射

  blkdev driver 可以通过 mq\_ops->map\_queues() 回调函数自己指定 software queue 与 hardware dispatch queue 之间的映射，通常基于 PCI 总线的块设备会走这一路径。

这些基于 PCI 总线的块设备一般使用 MSI-X 作为中断机制，nvme/virtio-blk 设备在 probe 过程中都会调用 pci\_alloc\_irq\_vectors\_affinity() 来完成 MSI-X 中断的分配、初始化操作，一般优先为每个硬件队列分配单独的一个 MSI-X 中断，其中会决定分配的 MSI-X 中断的 CPU affinity，其算法也是类似于之前介绍的按照 CPU sibling 实现映射的算法之后在设备驱动的 mq\_ops->map\_queues() 回调函数中，通常是根据硬件队列绑定的 MSI-X 中断的 CPU affinity，来决定该硬件队列映射到哪些 software queue，例如假设一个硬件队列绑定的 MSI-X 中断的 CPU affinity 是绑定到 CPU A/B/C 的，那么这个硬件队列就会映射到 CPU A/B/C 的 software queue；而如果设备不支持 MSI-X 中断，或者设备支持的 MSI-X 中断数量有限导致多个硬件队列需要共用一个 MSI-X 中断，那么此时就会回退到之前介绍的 default CPU mapping 算法来实现 hardware queue 与 software queue 之间的映射。

### nvme\_pci\_map\_queues



## 硬件队列无锁化



1. \_\_blk\_mq\_alloc\_requests调用 blk\_mq\_map\_queue获取blk\_mq\_hw\_ctx

映射算法可以查看https://lostjeffle.bitcron.com/post/blog/mweb/docs/16006780007525

1. \_\_blk\_mq\_alloc\_requests调用blk\_mq\_get\_tag获取tag, 可能因当前无可用tag进入iowait（io\_schedule）状态
2. nvme\_queue\_rq函数不需要**加锁**

