# Scalable-IOV简介

传统的IOV(IO Virtualization)是通过PCIE的SR-IOV(Single Root)做设备资源上的切割，intel的vt-d、amd的iommu和arm的smmu（以下统称iommu）做DMA和IRQ重映射，通过VFIO框架直通设备给VM。传统iommu的DMA和IRQ重映射的粒度为BDF即当个PCIE设备（PF/VF）。当前我们只针对dma remapping（phys <--> iova）简单阐述一下当前的主流的IOVA方式。

## SR-IOV

PCIE特性，从物理上将PF0(bdf\_pf0)的资源切割为若干个VFx(bdf\_vf0, bdf\_vf1...bdf\_vfx)，每个VF设备都有自己独立的BDF, 配置空间、以及BAR空间。

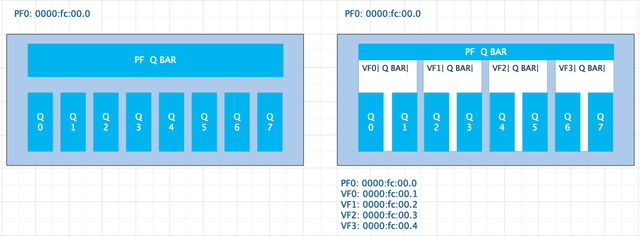
本文以一个假设备PF0为例，该设备有8队queue，物理资源即指队列，及队列对应的配置寄存器。SR-IOV开启之后，切割为了1个PF和4个VF，每个VF内有2对queue。如图，此处略去PCI配置空间的切割等，仅针对Queue资源。

在此我们可以看到：

1. 每个VF都有自己独立的bdf, 所以有自己独立的配置空间

2. 每个VF都会有相应的Q BAR，从PF的角度来说，是将Q的内部地址空间切割为了4份

3. 因此PCI TLP即保证了物理资源的隔离



**图1：左，SR-IOV未开启 右，SR-IOV开启**

既然SR-IOV在物理上保证了设备资源的隔离，为何还需要iommu呢？而iommu在IO虚拟化中的主要作用又是什么呢？

## IOMMU

首先需要澄清的是SR-IOV和IOMMU没有任何必然的联系，两者的维度不同：SR-IOV保证了PCIE层面，设备内部资源（地址空间）的隔离，而IOMMU通过DMA重映射的机制（IOVA），保证了CPU侧设备与设备之间DMA地址空间的隔离。

**假定IOMMU不存在，**如果仍然以直通的方式（passthrough）将PCIE设备给VM独占，那么就可能产生如下问题：

DEVICE0被VM0独占，DEVICE1被VM1独占，VM0的设备驱动和VM1的设备驱动都请求到GPA 4000H的DMA请求，那么DEVICE0和DEVICE1将访问到同一块DDR中的物理地址。无法保证DMA的隔离。

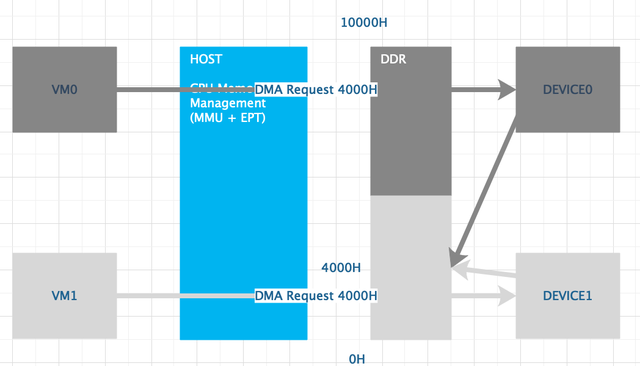


图2：无IOMMU的设备直通（假定场景）

因此在没有IOMMU的情况下，VM想要使用PCIE设备，就需要通过VMM软件模拟设备，VM通过读写模拟设备由VMM代理转发数据到host的内核设备驱动，典型的如tap设备模拟网卡。

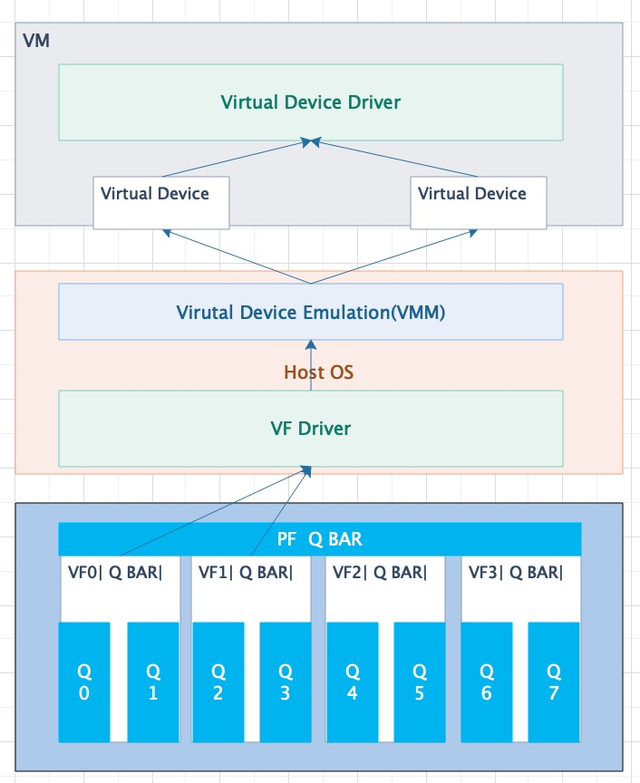


图3：vmm模拟pcie设备数据链路

当IOMMU使能之后，每个设备会有一个属于自己的context table（通过BDF索引），该context table指向的是PA <--> IOVA的IO page table。设备发出的DMA请求的目的地址，此时都被当作IOVA（非iommu=pt模式），如果IOVA未被映射，则DMA请求出错。

回到图3的场景，在IOMMU开启之后，DEVICE0的DMA请求IOVA 4000H会被映射到VM0的物理地址空间，而DEVICE1的DMA请求IOVA 4000H则会被映射到VM1的物理地址空间，从而实现了DMA的隔离。

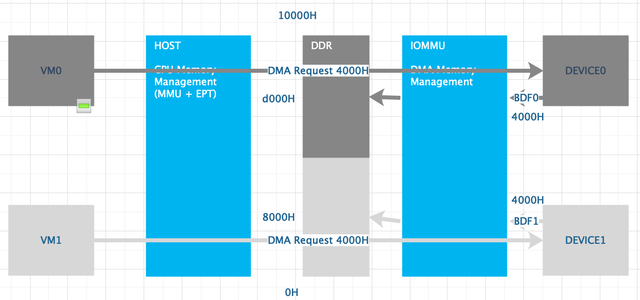


图4: iommu开启设备直通

目前x86的kvm的IO虚拟化中，主要通过vfio-pci软件框架，将设备直通（passthough）给VM。

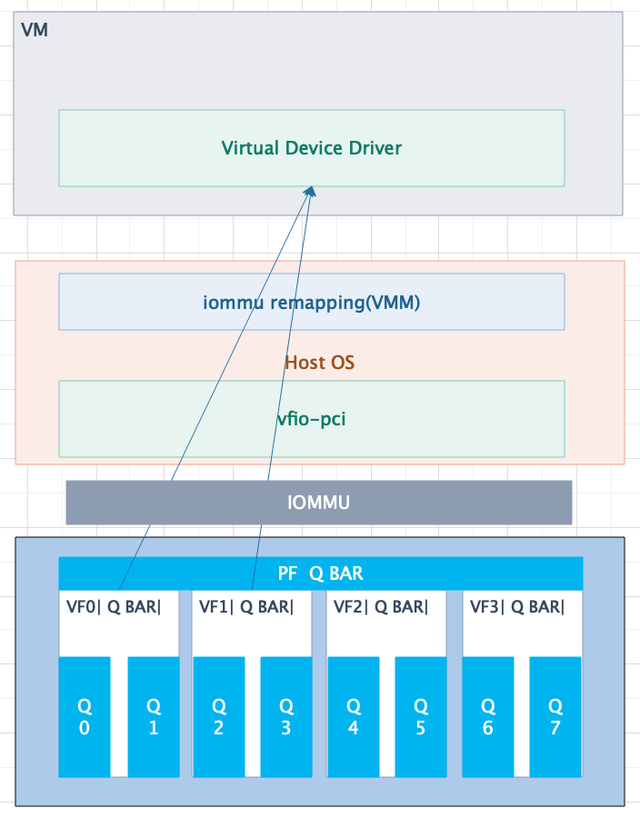


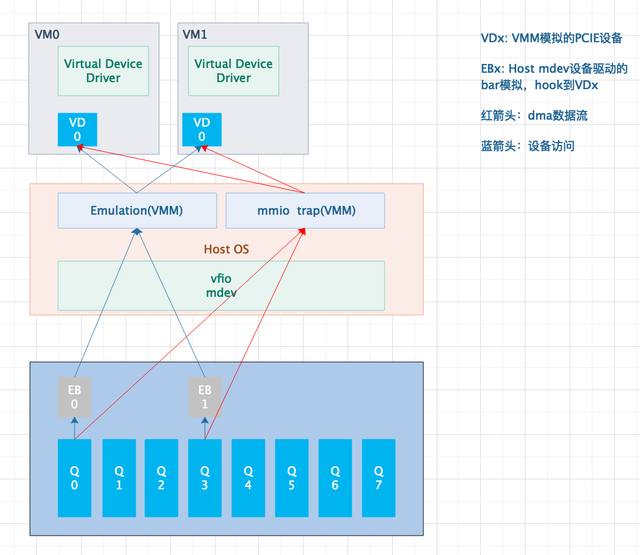
图5: vfio-pci设备直通

## VFIO-MDEV

了解了SR-IOV和IOMMU在IO虚拟化中做的事情之后，IOMMU的DMA隔离粒度为BDF，思考另一个问题：如果PCIE没有SR-IOV的能力，我们该如何将同一个PCIE设备的资源分配给不同的VM使用呢？同时SR-IOV的资源分割是在PCIE设备在设计时就决定的，无法动态更改和分配，那么设备资源的弹性和伸缩怎么解决呢？

例如，PF0只有两个VF，每个VF有2对queue，面对需求1对queue的客户和3对queue的客户无法精准地满足其需求。

为了解决这个问题，基于vfio框架的vfio mediated device框架被提出，通过该框架，每一对queue都可以作为一个mdev设备（host角度为mdev设备，guest角度为模拟的PCIE设备）单独直通给VM，综合了VMM软件模拟设备 -- 软件模拟mdev设备的PCIE配置空间读写、MMIO读写等由VMM trap并代理读写；以及设备直通 -- VM直接下发descriptor给mdev，由硬件自己完成DMA操作（而无需从VM拷贝数据到host后由host代为发送）。这里不再花过多篇幅描述vfio和mdev的软件细节。



**图6: vfio-mdev设备直通**

看图6后，我们可能会有一点疑问：

1. mdev设备的粒度是什么，由什么决定的？
2. 为什么dma的访问还是要经过mmio的trap？
3. IOMMU在哪？
4. mdev又是如何保证DMA的隔离性的呢？

对此一一解答：

1. mdev设备的粒度有host 设备驱动软件决定，一般最细粒度为硬件可操作资源的最细粒度 -- queue
2. dma本身无需trap，但是VM进行enqueue操作时（写MMIO)，会被trap到host设备驱动，host设备驱动需要更改VM的descriptor（将descriptor内GPA替换为HPA或者host IOVA)。
3. mdev不依赖于IOMMU
4. 如回答2中所讲，由host驱动将descriptor内的GPA替换为HPA或者host IOVA，再由host驱动代为enqueue。

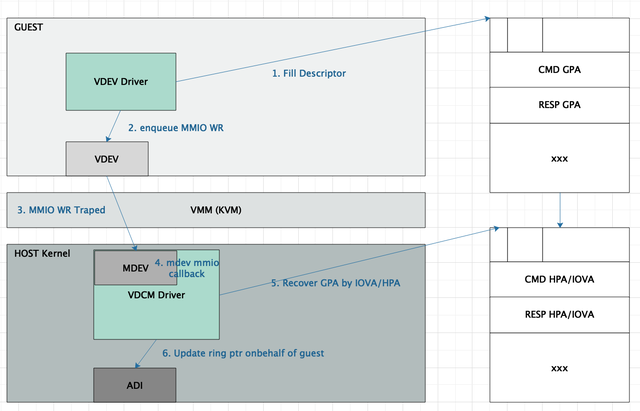


图7: MDEV enqueue流程

简单总结一下SR-IOV+IOMMU和vfio mdev的特点。

* SR-IOV:

1. 每个VF有独立的配置空间，资源预先切割，无非动态分配

2. 每个PF最大支持256个VF （PCIE ARI特性支持）

3. 编程简单，host驱动无需额外模拟

4. 可以直接配合IOMMU硬件做DMA隔离

* vfio mdev:

1. 资源可动态切割和分配

2. 软件做DMA隔离（mmio trap），host mdev驱动需要额外编程模拟

是否可以综合SR-IOV和vfio mdev各自的优点，让IOMMU在硬件上支持更细粒度的DMA重映射，增强弹性与伸缩性的同时，又可以省去mdev软件做DMA隔离的开销，减小host mdev驱动的复杂度呢？

## Scalable-IOV

### Scalable-IOV概述

针对上一节最后的问题，Scalable-IOV被提出。在BDF的基础上，增加PASID作为IO page table的索引，从而支持更细粒度的DMA重映射。

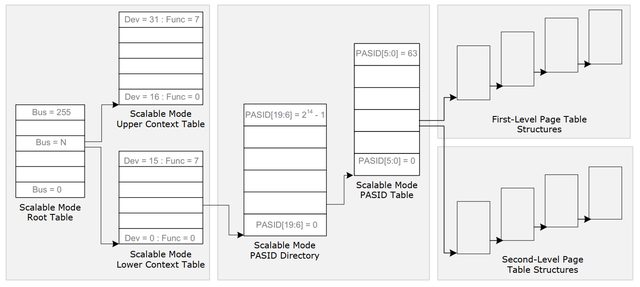


图8: Scalable-IOV DMA重映射

在Scalable-IOV中，最小粒度的可分配的资源叫做ADI（Assignable Device Interface，硬件设计决定），在我们的例子里，每一对独立的queue就是一个ADI。而管理这些ADI的host mdev驱动，在S-IOV中又叫做VDCM(Virtual Device Composition Module)。

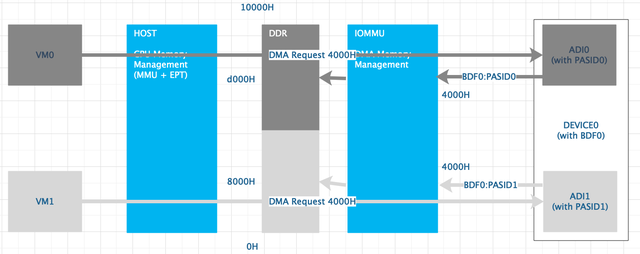


图9: Scalable-IOV 设备直通DMA重映射

在上一章节中，host mdev设备需要trap VM的enqueue操作，替换GPA后代理enqueue，在Scalable-IOV中，由于PASID的存在无需再trap了，由硬件自动完成GPA到HPA的翻译。

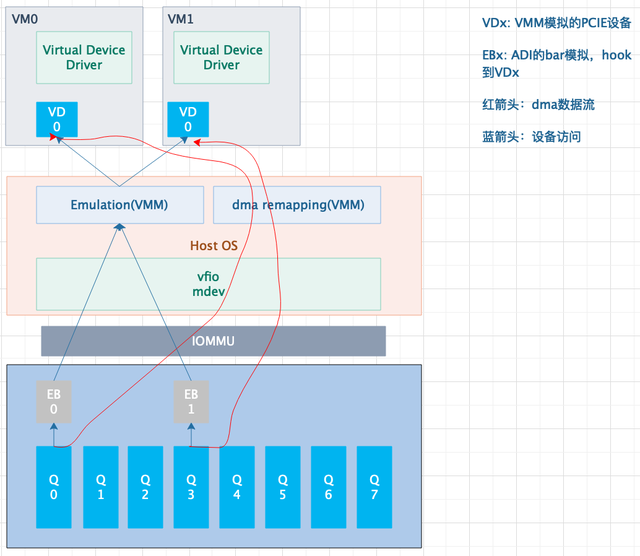


图10: Scalable-IOV设备直通

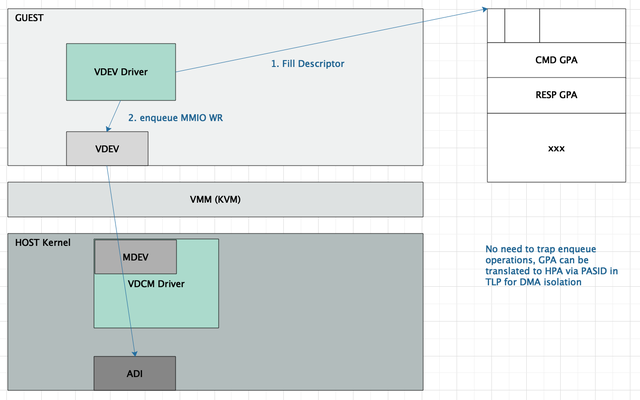


图11: Scalable-IOV vfio-mdev软件框架

IOMMU通过BDF:PASID索引IO pagetable, 将ADI的DMA请求中的GPA，翻译为HPA，从而硬件上直接实现DMA的隔离。

BDF是在PCIE设备枚举阶段由系统软件或者Firmware(BIOS)分配的，那么每个ADI的PASID又是如何分配的呢？硬件上ADI又是如何使用他们的PASID的呢？IOMMU又是如何获取ADI的PASID进行IO page table索引的呢？

## PASID

### PASID在哪

在PCI的TLP中，包含累了header -- 可获取BDF，和prefix -- 可获取PASID。

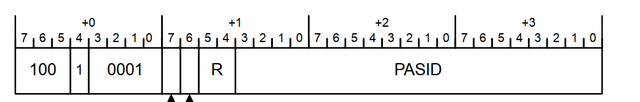


图12：PCE TLP Prefix格式

PCI的TLP都是由硬件填充和传输的，软件无法更改。因此，在ADI的硬件设计中，如果每个ADI需要绑定不同的PASID，那么就必须提供PASID的配置接口，ADI在发起DMA请求时将该PASID填入TLP的prefix中。

### ADI的PASID配置接口

在我们上面的例子中，每一对queue是一个ADI，并且只能被一个进程/VM独占，这种情况下硬件只需要给这对queue提供一个PASID\_CSR由host VDCM配置PASID即可。

在intel dsa中，每对queue都可以配置为dedicated work queue （DWQ)模式和shared work queue (SWQ）模式。

在DWQ模式下面，每对queue只能被一个进程/VM独占，类似我们的例子此时每个ADI有对应唯一的PASID，即每对queue对应唯一PASID即可。

在SWQ模式下，每对queue可被多个进程/VM共享（取决于硬件设计），比如如果每对queue可以被256个进程或者VM共享，那么1/256 queue即为一个ADI，由于每个ADI同样需要对应唯一的PASID，每对queue则需要对应256个PASID。

硬件上是如何处理这两种模式的呢？我们仍然以Intel dsa为例。

### DWQ模式的PASID配置

DWQ模式下，硬件通常的做法是每对queue的配置寄存器中，会有一个可编程的PASID\_CSR，该寄存器仅可由host VDCM配置。但queue发起DMA请求时，硬件会从该queue的PASID\_CSR中将PASID填入PCI的TLP中，从而实现DMA隔离。如果硬件的descriptor中，有PASID字段，那么此时的PASID字段应该被硬件忽略。

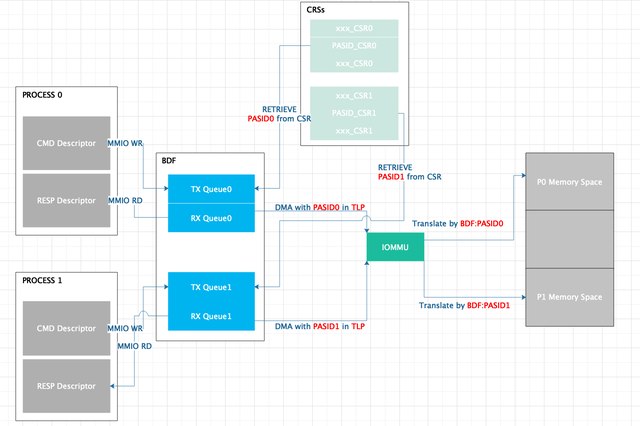


图13: DWQ PASID配置

### SWQ模式的PASID配置

SWQ模式下，queue实际上是被分时复用的，但是ADI可以同时缓存n（queue深度）个descriptor，而这些descriptor可能属于不同的进程/VM，如果只通过一个PASID\_CSR就无法满足实际需求。因此这里有两条路径可供选择：

1. 直接从descriptor中获取PASID (intel dsa/qat的做法）

2. queue提供额外的寄存器，为每个adi配置pasid（创新做法，正在思考）

仅以intel dsa/qat的方法，简单讲讲SWQ的工作方式。

在硬件设计中，如果要支持第一种方式的SWQ就必须考虑以下几个问题：

1. descriptor中必须支持“pasid”字段

2. descriptor中pasid必须由硬件填写/覆盖

3. 每个使用SWQ的进程必须有个PASID且该PASID可被硬件自动retrieve

4. 不能使用head/tail指针方式enqueue，必须由硬件保证enqueue的原子性

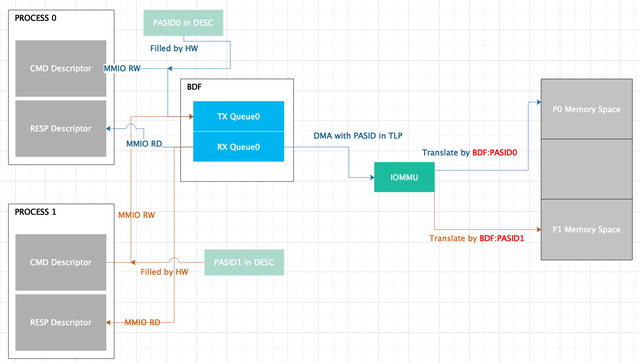


图14：SWQ PASID配置假设

关于问题2，如果descriptor中的PASID可以被软件填写，那么VM可随意更改descriptor中的PASID导致无法实现DMA隔离。

关于问题4，如果Queue在不同的进程/VM之间共享，head/tail指针可同时被修改，无法保证enqueue/dequeue的原子性，导致硬件工作异常。

问题2有另一个解决思路，在图6和图7中，我们提到过引入Scalable-IOV之前的mdev是通过host mdev驱动trap MMIO读写替换实现DMA隔离的，SWQ的PASID同样可以通过trap的方式有host VDCM填写，这也就是为什么intel除了引入ENQCMD指令之外还引入了ENQCMDS指令（ENQCMD的特权模式，由host kernel执行，ENQCMD/ENQCMDS马上就说）。

关于问题3，Intel从spr开始，引入了一个IA32\_PASID\_MSR的MSR用于存放当前进程的PASID，所以从系统软件角度，每当发生进程切换时，必须同时更新IA32\_PASID\_MSR。

综合问题2、3、4，Intel从spr开始，同时引入了ENQCMD/ENQCMDS指令。ENQCMD指令从硬件角度是non-post指令（即指令执行后会等结果返回，不会发生进程切换），进程执行该指令时，可自动从IA32\_PASID\_MSR中获取PASID并将其填入ENQCMD的descriptor中。ENQCMDS是ENQCMD的特权模式，只能由host kernel执行，该指令不会自动获取PASID而需要软件填写，以此解决上述问题2中提到的另一种思路。

从虚拟化软件的角度，VM中如果要支持ENQCMD就必须支持vIOMMU，以此获得全局唯一的PASID。如果hypervisor不支持vIOMMU，就必须有hypervisor trap ENQCMD之后由host VDCM通过ENQCMDS代理VM enqueue。

关于问题1，descriptor中必须支持“PASID”字段，由于不同的PCI设备自己的descriptor格式都不尽相同，为了解决这个问题，ENQCMD/ENQCMDS使用固定64字节大小的descriptor（一下叫enq desc），enq desc的前4字节格式固定，存放PASID(可由硬件直接修改/覆盖)，其它字段由支持SWQ的设备自己定义，因此一般设备内部必须有一个解析enq desc的硬件解码单元。

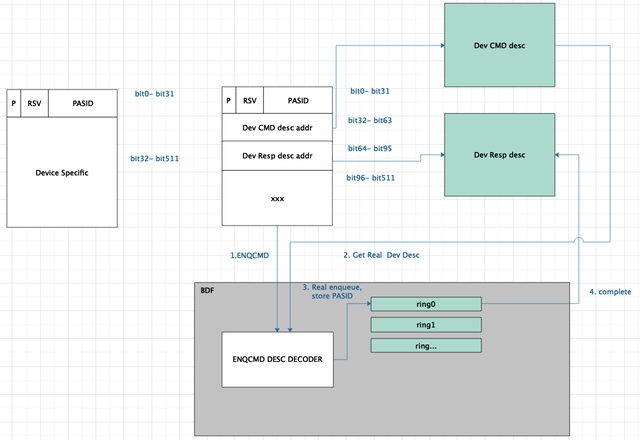


图15: SWQ的ENQCMD enqueue硬件流程

一般的ENQCMD的enqueue流程如上图：

1. 进程构建enq desc，其中Dev CMD/RESP desc addr分别指向设备specific的descriptor地址（类似indirect模式），执行ENQCMD指令，此时CPU将从IA32\_PASID\_MSR中获取PASID并将其填入enq desc的PASID字段中。ENQCMD最终会将enq desc写入到queue划分的特定的MMIO地址空间内。

2. queue的MMIO地址被ENQCMD写之后，设备内部的ENQCMD DESC DECODER会解析enq desc，获取硬件所必需的meta data，比如PASID、Dev CMD/RESP desc addr等。

3. 入队方式取决于硬件设计。Intel qat在enq desc被decode之后，会将Dev CMD desc（通过Dev CMD desc addr获取）DMA到queue desc buffer中，并更新head/tail指针（注意该指针不能暴露给软件）。而Intel DSA则直接使用enq desc内的地址做DSA spec中的各种操作。

4. 业务逻辑做完之后，将数据DMA回DDR（TLP中带着该请求的PASID）

# vfio-mdev逻辑空间分析

VFIO是[Linux](https://so.csdn.net/so/search?q=Linux&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/runshui27/article/details/_blank) Kernel UIO特性的升级版本。UIO的作用是把一个设备的IO和中断能力暴露给用户态，从而实现在用户态对硬件的直接访问。它的基本实现方法是，当我们probe一个设备的时候，通过uio\_register\_device()注册为一个字符设备/dev/uioN，用户程序通过对这个设备mmap访问它的IO空间，通过read/select等接口等待中断。

UIO的缺点在于，用户态的虚拟地址无法直接用于做设备的DMA地址（因为在用户态无法知道DMA内存的物理地址），这样限制了UIO的使用范围。基本上UIO现在只能用于做工控卡这种IO量不大，可以直接把内存地址拷贝到IO空间的场景（相当于不做DMA）。我们有人通过UIO设备自己的ioctl来提供求物理地址的机制，从而实现DMA，但这种方案是有风险的，因为你做ioctl求得的物理地址，可能因为swap而被放弃，就算你做gup，但gup只保证物理内存不被释放，不能保证vma还指向这个物理页，要保证后者需要vm\_pin这样的解决方案，但vm\_pin根本就没有能够上传主线。

这里提到的UIO的缺点，基本上拒绝了大流量IO设备使用该机制提供用户空间访问的能力了。

VFIO通过IOMMU的能力来解决这个问题。IOMMU可以为设备直接翻译虚拟地址，这样我们在提供虚拟地址给设备前，把地址映射提供给VFIO，VFIO就可以为这个设备提供页表映射，从而实现用户程序的DMA操作。

背负提供DMA操作这个使命，VFIO要解决一个更大的问题，就是要把设备隔离掉。在Linux的概念中，内核是可信任的，用户程序是不可信任的，如果我们允许用户程序对设备做DMA，那么设备也是不可信任的，我们不能允许设备访问程序的全部地址空间（这会包括内核），所以，每个设备，针对每个应用，必须有独立的页表。这个页表，通过iommu\_group承载（iommu\_group.domain)，和进程的页表相互独立。进程必须主动做DMA映射，才能把对应的地址映射写进去。

所以VFIO的概念空间是container和group，前者代表设备iommu的格式，后者代表一个独立的iommu\_group(vfio中用vfio\_group代表），我们先创建container，然后把物理的iommu\_group绑定到container上，让container解释group，之后我们基于group访问设备（IO，中断，DMA等等）即可。

这个逻辑空间其实是有破绽的，iommu\_group是基于设备来创建的，一个设备有一个iommu\_group（或者如果这个设备和其他设备共享同一个IOMMU硬件，是几个设备才有一个iommu\_group），那如果我两个进程要一起使用同一个设备呢？基于现在的架构，你只能通过比如VF（Virtual Function，虚拟设备），在物理上先把一个设备拆成多个，然后还是一个进程使用一个设备。这用于虚拟机还可以，但如果用于其他功能，基本上是没戏了。

再说，VF功能基本都依赖SR-IOV这样的实现，也不是你想用就能用的。

这我们就要引出VFIO-mdev（以下简称mdev）了。mdev本质上是在VFIO层面实现VF功能。在mdev的模型中，通过mdev\_register\_device()注册到mdev中的设备称为父设备（parent\_dev），但你用的时候不使用父设备，而是通过父设备提供的机制（在sysfs中，后面会详细谈这个）创建一个mdev，这个mdev自带一个iommu\_group，这样，你有多个进程要访问这个父设备的功能，每个都可以有独立的设备页表，而且互相不受影响。

所以，整个mdev框架包括两个基本概念，一个是pdev（父设备），一个是mdev（注意， 我们这里mdev有时指整个vfio-mdev的框架，有时指基于一个pdev的device，请注意区分上下文）。前者提供设备硬件支持，后者支持针对一个独立地址空间的请求。

两者都是device(struct device)，前者的总线是真实的物理总线，后者属于虚拟总线mdev，mdev上只有一个驱动vfio\_mdev，当你通过pdev创建一个mdev的时候，这个mdev和vfio\_mdev驱动匹配，从而给用户态暴露一个普通vfio设备的接口（比如platform\_device或者pci\_device）的接口。

换句话说，如果一个设备需要给多个进程提供用户态驱动的访问能力，这个设备在[probe](https://so.csdn.net/so/search?q=probe&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/runshui27/article/details/_blank)的时候可以注册到mdev框架中，成为一个mdev框架的pdev。之后，用户程序可以通过sysfs创建这个pdev的mdev。

## mdev出现的背景

mdev主要解决的问题是，不支持sr-iov的设备，如何做到切分，也就是说，虽然物理上

不支持vf的切分，他通过mdev，可以时分复用的方式来实现虚拟化，这样就实现了多个用户态驱动“同时“ 操作一块物理设备的能力。