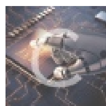
从qemu-virtio到vhost-user

原创

jiang4357291 于 2021-11-13 15:44:16 发布 1013 收藏 4

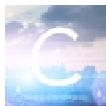
分类专栏： 存储 linux 文章标签： linux 云存储

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/jiang4357291/article/details/121304873
版权

 存储 同时被 2 个专栏收录

1 篇文章 0 订阅

订阅专栏

 linux

4 篇文章 0 订阅

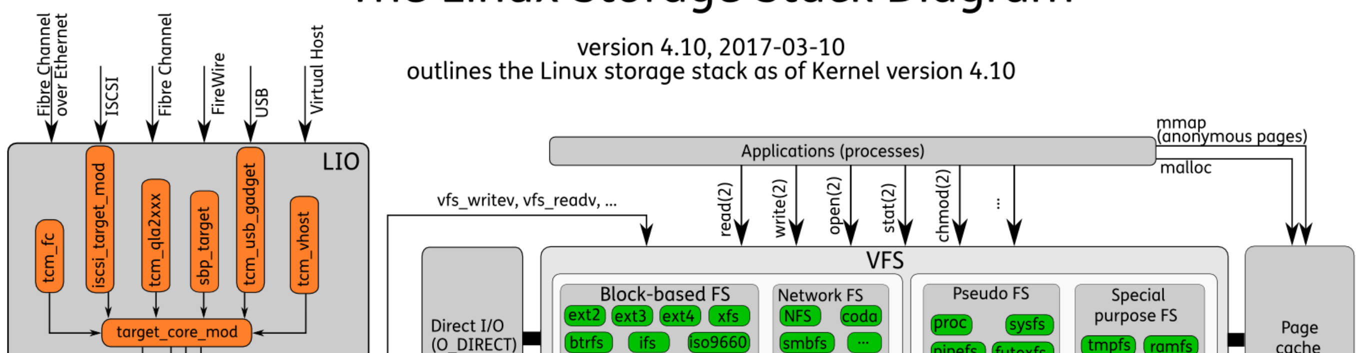
订阅专栏

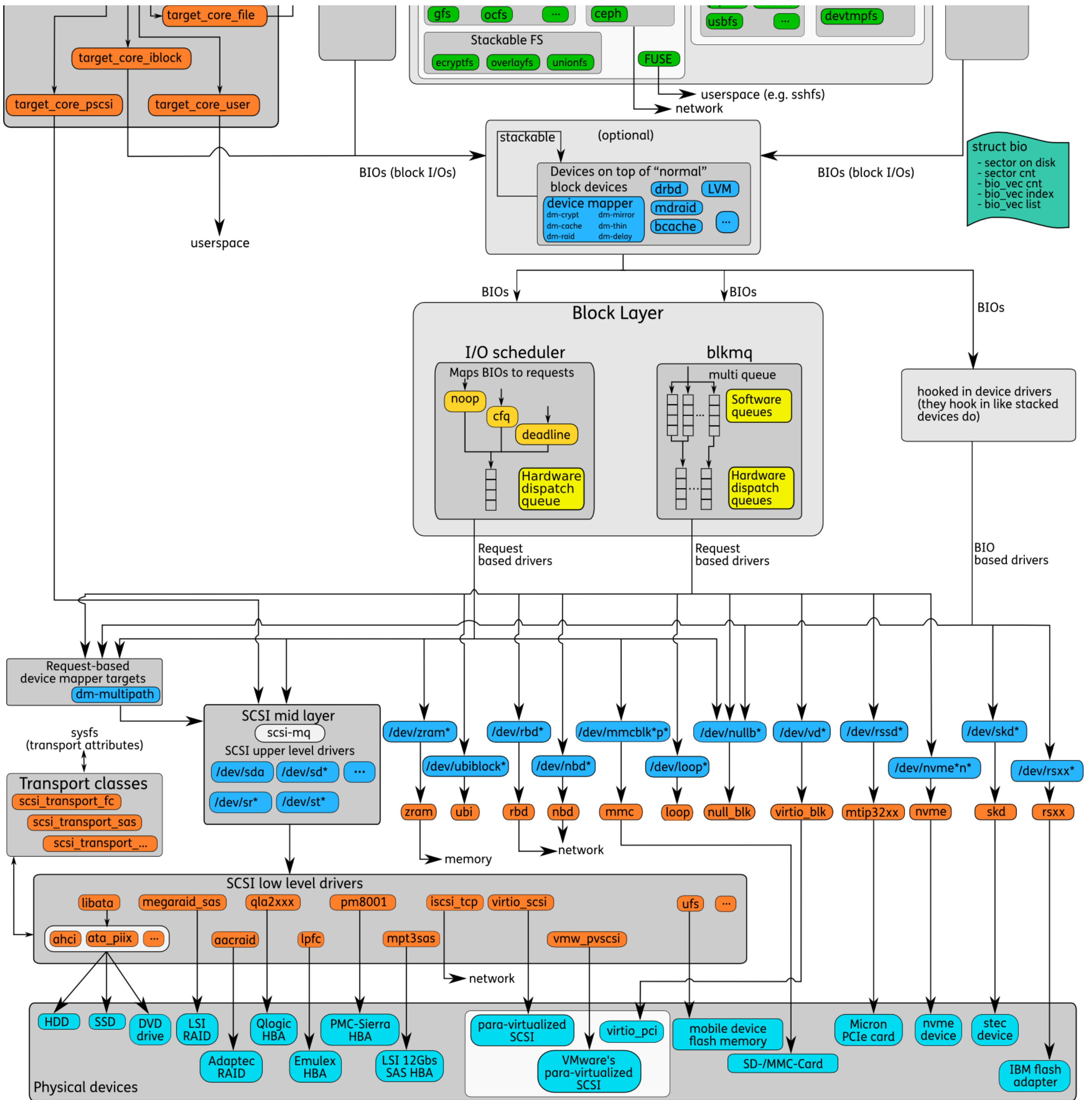
文章目录

一、linux单机存储栈
1.1 概览
1.2 block layer
1.2.1 io scheduler
1.2.2 block mq
二、计算虚拟化
2.1 cpu虚拟化
2.2 内存虚拟化
2.2.1 linux内存管理方案
2.2.2 内存虚拟化
2.3 qemu-kvm
2.3.1 qemu
2.3.2 kvm
2.3.3 qemu-kvm
三、存储虚拟化
3.1 全虚拟化IO
3.2 virtio
3.2.1 概述
3.2.2 架构
3.2.3 virtqueue
3.2.4 virtio-blk/virtio-scsi
3.3 spdk vhost-user
3.3.1 spdk
3.3.2 基于vhost的加速方案
3.3.3 qemu-virtio vs vhost-user
四、参考

一、linux单机存储栈

1.1 概览





The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/

THOMAS KRENN®

CSDN @jiang4357291

- VFS：对用户屏蔽各种文件系统的不同实现，对上层提供统一的接口
- 单机文件系统(ext4/xfs等)：向下管理块设备，向上对接vfs
  - 非direct io：写到page cache，之后由内核定期write back
  - direct io：将用户io提交到通用块层
- page cache：文件系统高速缓存，用于加速读写过程，Page cache由内存中的物理page组成，其内容对应磁盘上的block。
- block layer：处理所有对块设备的请求，核心struct bio，主要是io scheduler和block mq两大模块
- 块设备驱动：驱动程序可以直接管理块设备的硬件读写，驱动程序收到io请求后会触发执行硬件指令，大部分的磁盘驱动程序都采用DMA的方式去进行数据传输，DMA控制器自行在内存和IO设备间进行数据传送，当数据传送完成再通过中断通知CPU

1.2 block layer

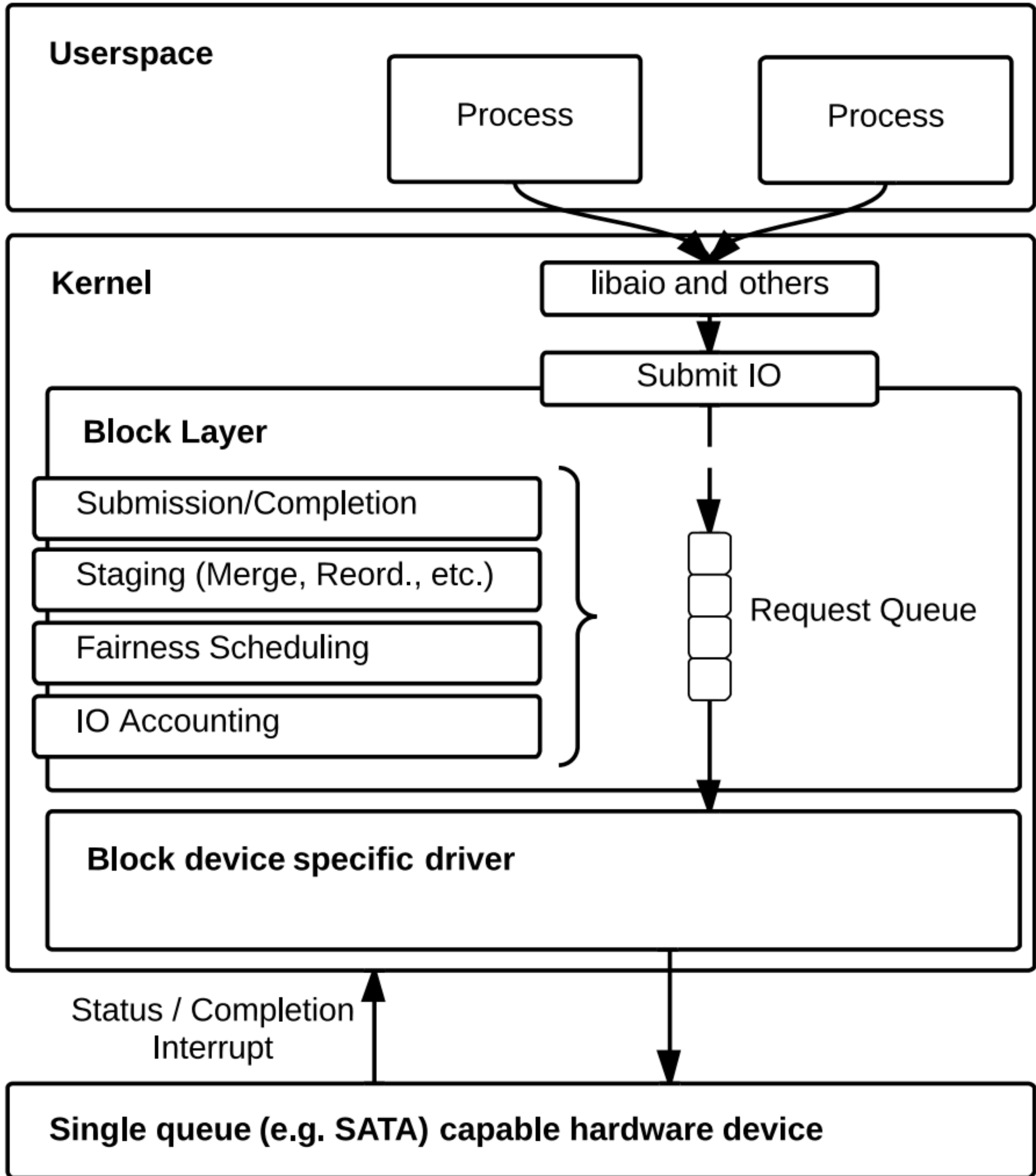
1.2.1 io scheduler

前面说到，block layer提交的io会通过一定的调度算法才会真正写到块设备上，目前linux的io scheduler主要有以下几种：

- noop：不调度，所以io请求进入一个FIFO队列后逐个出队执行，如果对应在磁盘上连续的请求，也会适当做一些合并。
- deadline：改良的电梯算法，每个请求都有默认超时时间(读500ms，写5s)，当部分请求等待太久后，电梯需要掉头处理这些请求。可见其核心在于保证每个IO请求在一定的时间内一定要被服务到，以此来避免某个请求饥饿。
- cfq：类似于进程调度算法里的cfs，为每个请求队列分配一个调度队列和时间片，在调度器分配给进程的时间片内，进程可以将读写请求下发到块设备，时间片消耗完后请求队列将被挂起，等待调度。
  不同的scheduler适用于不同的场景，可以通过/sys/block//queue/scheduler查看和更改某块磁盘的调度算法，实际应用中可以根据应用场景进行分析以配置合适的scheduler。
  值得一提的是，deadline和cfq等调度算法都是针对机械盘的特点进行了设计和优化的，机械盘的随机访问在磁道寻址上要花费大量时间，因此才会出现这些算法，尽可能在寻道的过程中，能把顺序路过的相关磁道的数据请求都“顺便”处理掉，那么就可以在比较小影响响应速度的前提下，提高整体IO的吞吐量。

1.2.2 block mq

上节说到，block layer中的scheduler都是为了hdd设计的，由于hdd的随机读写性能差，IO操作在Block Layer中会经过复杂的操作才会被执行，此时io的性能瓶颈在于硬件，而不是内核，内核通过引入各种调度算法来最大化利用hdd的能力，此时内核采用的还是一个全局共享的单队列(Request Queue)：

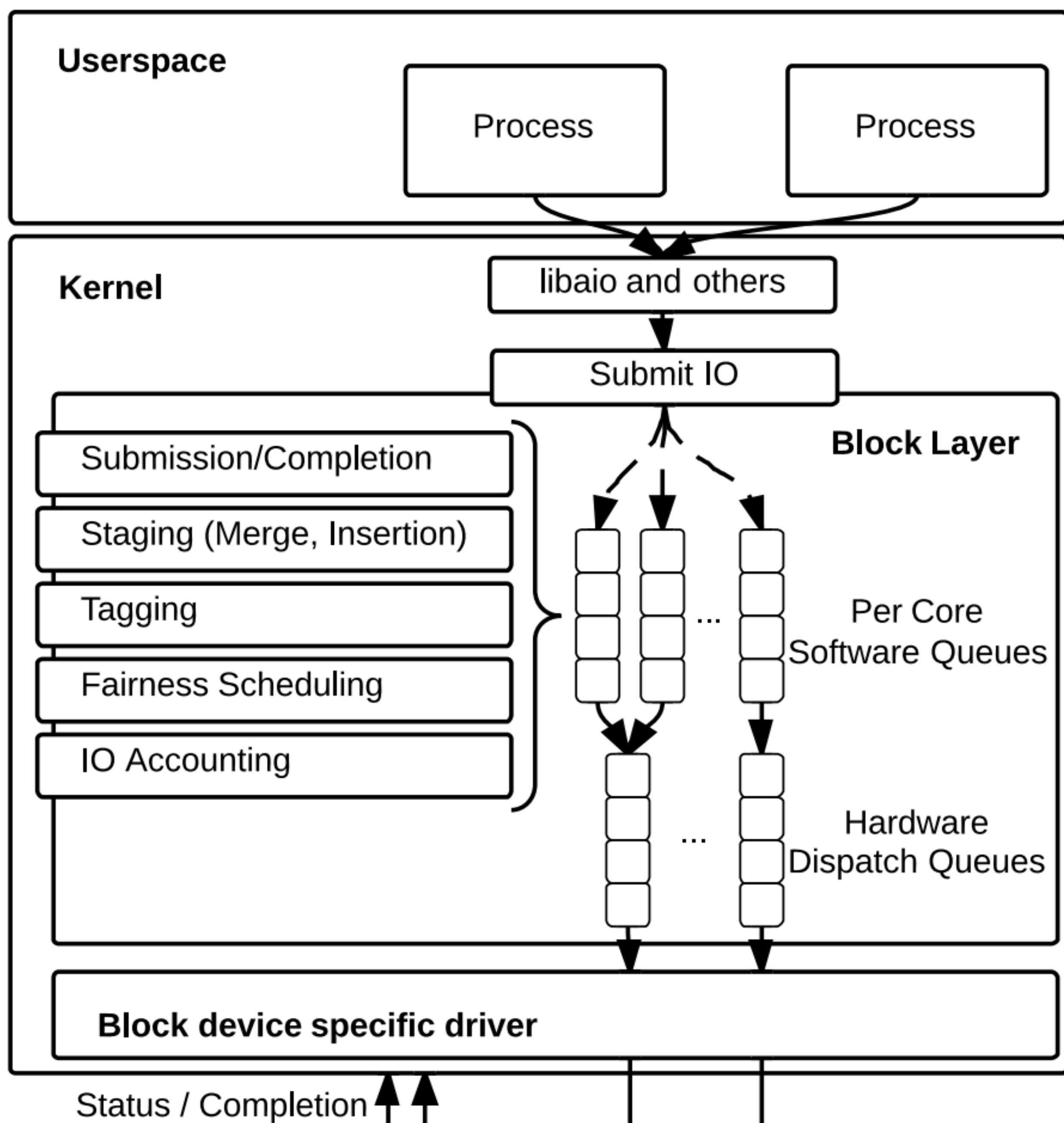


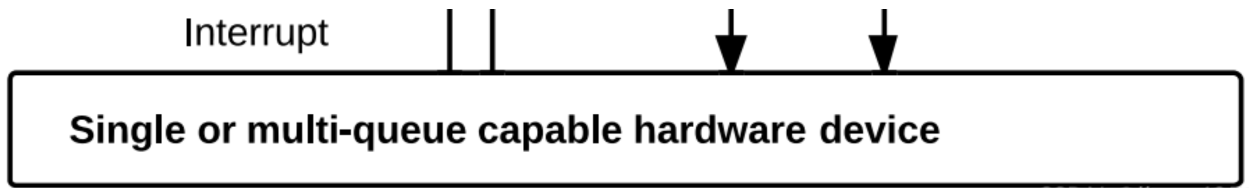
CSDN @jiang4357291

任何io请求都会经过该Request Queue，io的出队、入队、合并、重排等都需要加锁，这个设计在当时并不会成为性能瓶颈，因为：

1. HDD 很慢，内核互斥访问一个全局队列不会成为系统瓶颈。
2. CPU 核数较少，锁竞争的情况不严重。
3. 全局一个队列有利于 I/O 优化（合并、重排序）。
   但是随着ssd的问世和广泛应用，存储硬件的io能力翻了数倍，此时io的主要瓶颈已经从硬件转移到了内核block sq架构，主要在于以下几个方面：

- Request Queue锁竞争(主要瓶颈)：全局单队列共享的设计，在单核hdd的时代，锁竞争的开销并不明显；但是在动辄几十cpu以及数十万iops的ssd下，每个io都需要经过全局Request Queue，锁竞争已经成为了巨大瓶颈
- 中断：在多数情况下，完成一次IO需要两次中断，一次是存储器件触发的硬件中断，另一次是IPI核间中断用于触发其他cpu上的软中断
- Remote Memory Accesses：如果提交IO请求的cpu不是接收硬件中断的cpu且这两个cpu没有共享缓存，那需要访问远端cpu缓存，这会造成了不小的性能损失，特别是在NUMA架构的机器上
  面对以上暴露的种种问题，linux内核在3.13版本引入了新机制block multi-queue，在3.19版本更抽象化为了multi-queue block layer。multi-queue机制的核心设计是在多核CPU的情况下，引入了两级的多队列，将单队列的锁竞争分散到多队列中，以更好的平衡IO的工作负载，大幅提高SSD等存储设备的IO效率。





CSDN @jiang4357291

两级多队列设计：

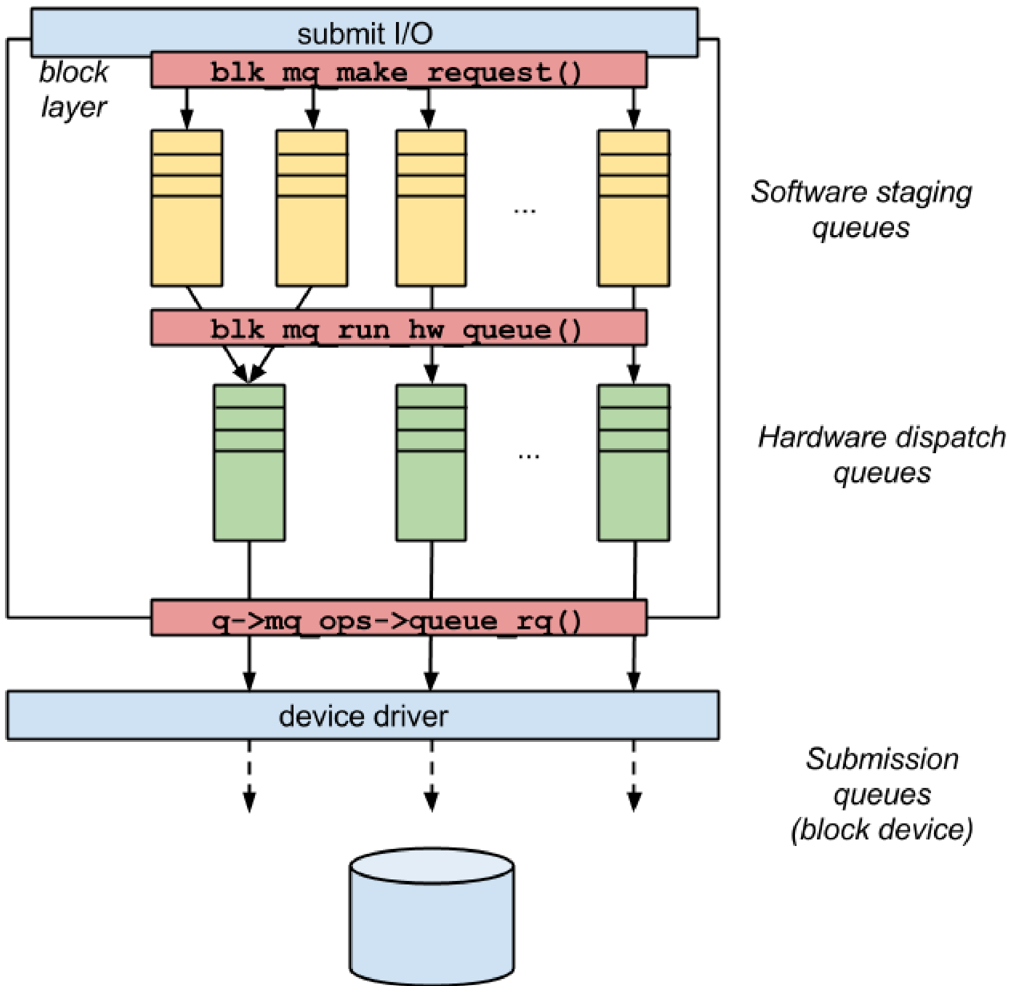
- Software Staging Queue：负责 I/O 的调度和优化，队列的配置可以是per cpu core，也可以是per cpu socket；io的调度优化以queue为单位，不会发生跨queue的调度行为，减少了锁竞争
- Hardware Dispatch Queue：负责将从 Software Staging Queues 过来的 I/O 请求发送给底层硬件，一般和硬件队列的个数相等，每个硬件队列对应一个派发队列

可以看到，block mq的多个staging queue很大程度上减少了锁竞争，同时由于和cpu core绑定的关系，也避免了remote memory access以及节省了核间中断，因此相比于sq架构很大程度上提升了对存储介质的使用效率。

两级多队列设计：

Software Staging Queue：负责 I/O 的调度和优化，队列的配置可以是per cpu core，也可以是per cpu socket；io的调度优化以queue为单位，不会发生跨queue的调度行为，减少了锁竞争

Hardware Dispatch Queue：负责将从 Software Staging Queues 过来的 I/O 请求发送给底层硬件，一般和硬件队列的个数相等，每个硬件队列对应一个派发队列



可以看到，block mq的多个staging queue很大程度上减少了锁竞争，同时由于和cpu core绑定的关系，也避免了remote memory access以及节省了核间中断，因此相比于sq架构很大程度上提升了对存储介质的使用效率。

实例分析二：多队列场景下fio测试时如何测性能最高？
来看下阿里的essd压测脚本：https://help.aliyun.com/document_detail/65077.html

```
cpulist=""
for ((i=1;i<10;i++))
do
    list=`cat /sys/block/your_device/mq/*/cpu_list | awk '{if(i<=NF) print $i;}' i="$i" | tr -d ',' | tr '\n' '
,'`
    if [ -z $list ];then
        break
    fi
    cpulist=${cpulist}${list}
done
spincpu=`echo $cpulist | cut -d ',' -f 2-${nu}`
echo $spincpu
fio --ioengine=libaio --runtime=30s --numjobs=${numjobs} --iodepth=${iodepth} --bs=${bs} --rw=${rw} --filename=
${filename} --time_based=1 --direct=1 --name=test --group_reporting --cpus_allowed=$spincpu --cpus_allowed_polic
y=split
}
```

在一个40 core的机器上测试，以上脚本选择的cpu如下，可以看到通过尽可能选择绑定了不同硬件队列的cpu来减少竞争，提升性能。

```
# cat /sys/block/{dev}/mq/*/cpu_list
0, 8, 16, 20, 28, 36
1, 9, 17, 21, 29, 37
2, 10, 18, 22, 30, 38
3, 11, 19, 23, 31, 39
4, 12, 24, 32
5, 13, 25, 33
6, 14, 26, 34
7, 15, 27, 35

#  echo $spincpu
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
```

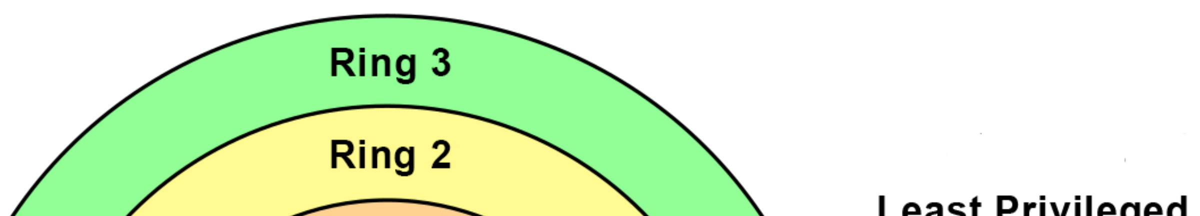
二、计算虚拟化

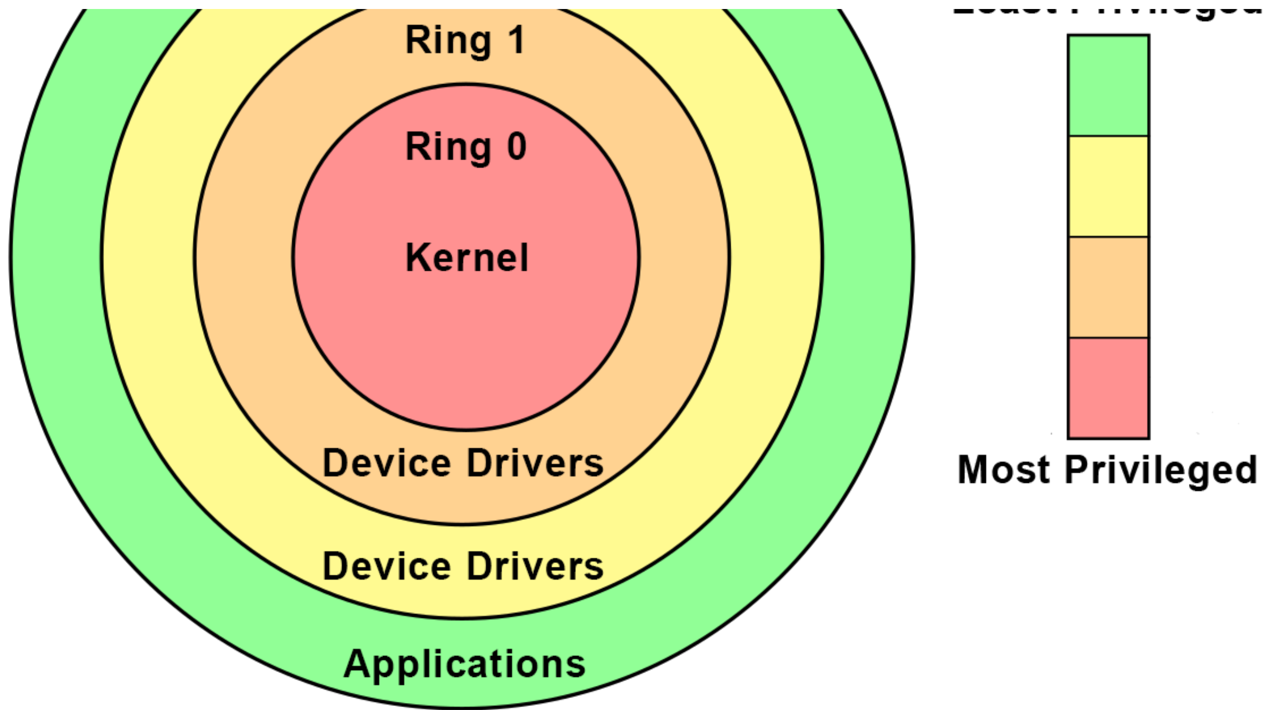
简单介绍下cpu和内存的虚拟化：

2.1 cpu虚拟化

操作系统是设计在直接运行在裸硬件设备上的，因此它们自动认为它们完全占有计算机硬件。x86 架构提供四个特权级别给操作系统和应用程序来访问硬件，从Ring3到Ring0优先级依次升高，但大多数现代操作系统都只用到了Ring0和Ring3。

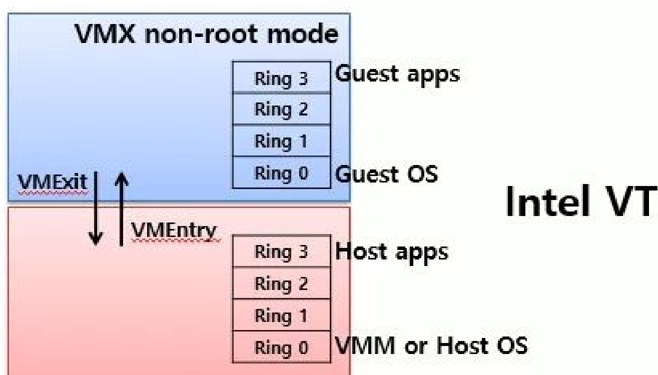
- 操作系统（内核）需要直接访问硬件和内存，因此它的代码需要运行在最高运行级别 Ring0上，这样它可以使用特权指令，控制中断、修改页表、访问设备等等。
- 应用程序的代码运行在最低运行级别上Ring3上，不能做受控操作。如果要做，比如要访问磁盘，写文件，那就要通过执行系统调用（函数），执行系统调用的时候，CPU的运行级别会发生从Ring3到Ring0的切换，并跳转到系统调用对应的内核代码位置执行，这样内核就为你完成了设备访问，完成之后再从Ring0返回Ring3。这个过程也称作用户态和内核态的切换。
  linux on x86只使用了这两个ring，ring0即kernel mode，ring3即user mode。





因为宿主操作系统是工作在 Ring0 的，客户操作系统就不能也在 Ring0 了，但是它不知道这一点，以前执行什么指令，现在还是执行什么指令，但是没有执行权限是会出错的。所以这时候虚拟机管理程序（VMM）需要避免这件事情发生。 虚机怎么通过 VMM 实现 Guest OS 对硬件的访问，根据其原理不同有三种实现技术：

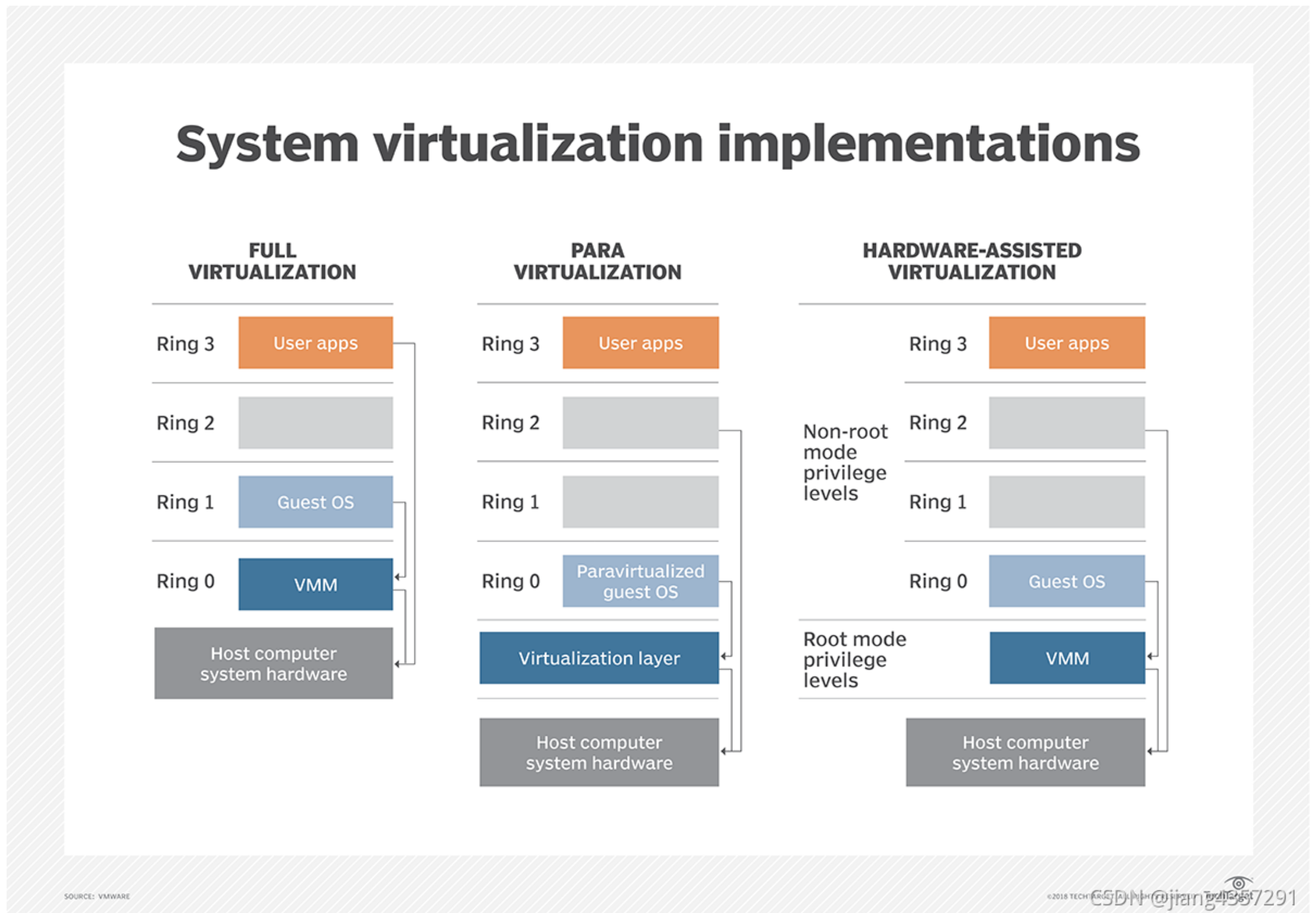
- 全虚拟化
  虚机与硬件完全隔离，虚机的一切指令都由虚拟化软件(也就是Hypervisor或VMM)进行处理，guest os认为自己运行在硬件上。VMM会为GuestOS抽象模拟出它所需要的包括CPU、磁盘、内存、网卡、显卡等抽象硬件资源。
  Guest os运行在Ring1，vmm运行在Ring0，所以guest os在执行特权指令时，会触发异常（CPU的机制，没权限的指令会触发异常）， VMM 捕获这个异常，在异常里面做翻译，模拟，最后返回到guest os内。
  这种方案的缺点是将原本可直接执行的指令变成了复杂的异常捕捉和指令翻译过程，所以速度太慢。
- 半虚拟化
  修改定制guest os，替换掉不能虚拟化的指令，通过超级调用（hypercall）直接和底层的虚拟化层hypervisor来通讯，hypervisor 同时也提供了超级调用接口来满足其他关键内核操作，比如内存管理、中断和时间保持。修改后的guest os直接运行在Ring0。
  这种方案省去了指令的翻译过程，性能接近裸机，缺点是需要对guest os进行修改，guest os知道自己是虚机，而且像windows这种无法改源码的就不支持。
- 硬件辅助的虚拟化
  主要基于Intel的VT和AMD的AMD-V，在硬件层面做了虚拟化的支持。以intel VT为例，为cpu增加了Virtual machine Extensions(即VMX)，开启了VMX后的cpu有root mode和non-root mode两种模式，每种模式都支持Ring 0 ~ Ring 3 共 4 个运行级别。Guest OS运行在非root模式的Ring 0，vmm运行在root模式的Ring0。两种模式可以互相转换，从root模式到非root模式称为VM Entry(像是进入guest)，反之从非root模式到root模式则称为VM Exit(相当于退出vm)。



VMX root mode

CSDN @jiang4357291

VMX root模式和没有VT的cpu的正常模式没有什么区别，只是某些制寄存器的写被限制了，而VMX 非root模式则有明显区别，处理器的某些行为被限制住或者被更改以便实现虚拟化，即某些指令或者事件会引起VM Exit。通常情况下，Guest OS的核心指令可以直接下达到计算机系统硬件执行，而不需要经过VMM。当Guest OS执行到特殊指令的时候，系统会切换到VMM，让VMM来处理特殊指令。
硬件辅助的全虚拟化方案，在性能上接近半虚拟化，且不需要修改定制guest os，是现在主流的方案。
3种虚拟化方案的总结对比：



2.2 内存虚拟化

2.2.1 linux内存管理方案

- 每个进程拥有自己独立的虚拟地址空间，由kernel维护的页表来管理虚拟地址和物理地址的映射
- 当访问到的虚拟地址page还不在物理内存中时，则产生page fault，通过MMU建立映射关系
- 此外还有TLB来进行加速

2.2.2 内存虚拟化

几个基本概念：

```
GVA -> GPA -> HVA -> HPA

GVA - Guest virtual address
GPA - Guest physical address
HVA - Host virtual address
HPA - Host physical address
```

Guest OS需要使用一个从0开始的、连续的物理地址空间，但是真实的物理地址空间是被Host OS所管理的，因此Guest OS是不能直接在物理内存上加载、运行的，唯一可行的办法是为Guest OS提供一个虚拟的物理内存空间，即GPA。Guest内用户可以看到的内存是Guest OS虚拟化出的GVA。
显然，GVA - GPA 的映射由Guest OS负责维护，而 HVA - HPA 由Host OS负责维护，内存虚拟化的核心是GPA - HVA的映射，GPA-HVA的映射主要有两种方案：

1. 影子页表SPT
   纯软件实现，kvm为Guest中的每个页表再额外维护一个影子页表，Guest中原本的页表实际变成了虚拟页表。在Guest中的虚拟内存管理中，Guest的页表基址地址存放在CR3寄存器中，kvm会将 Guest 的页表设置为只读，当 Guest OS 对页表进行修改时就会触发 Page Fault，VM-EXIT 到 kvm，kvm根据GVA 对应的页表项进行访问权限检查，结合错误码进行判断：

   如果是 Guest OS 引起的，则将该异常注入回去，Guest OS 将调用自己的缺页处理函数，申请一个 Page，并将 Page 的 GPA 填充到上级页表项中

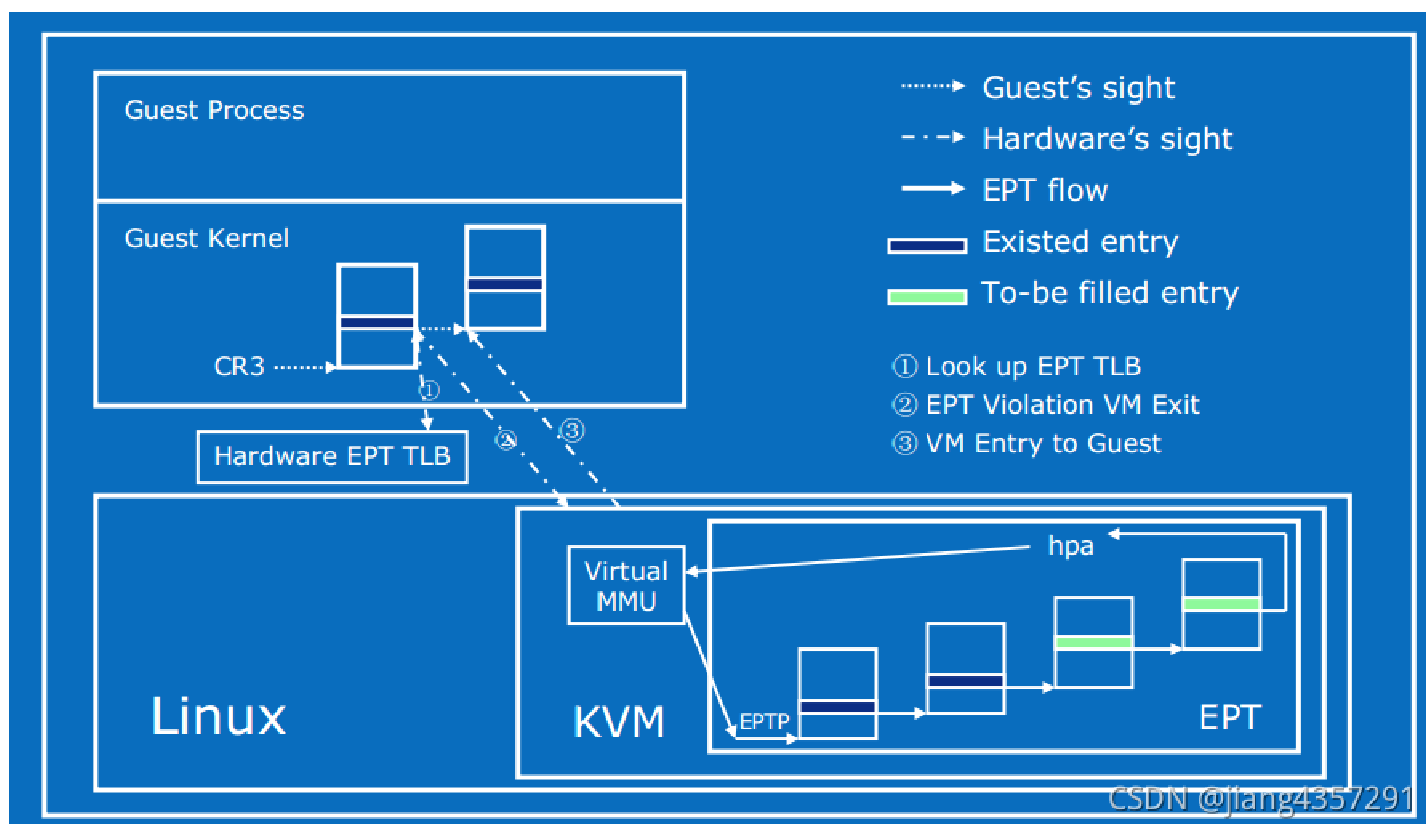
   如果是 Guest OS 的页表和 SPT 不一致引起的，则同步 SPT，根据 Guest 页表和 mmap 映射找到 GPA 到 HVA 的映射关系，然后在 SPT 中增加/更新GVA-HPA表项
   总的来说基于SPT的内存虚拟化方案中，kvm截获了Guest相关的修改操作并更新到SPT，而真正装入物理MMU的是SPT；Guest中GVA和GPA之间的转换实际上变成了GVA与HPA的转换，TLB中缓存的也是GVA和HPA的映射。
   SPT方案的优缺点如下：

   - 优点：Guest内存访问没有额外的地址转换开销。
   - 缺点：但是SPT的引入导致每个页表double了，而且进程过多时本身SPT的建立要花费不少时间，带来不小的内存开销，而且还有频繁的vm exit影响性能。

2. intel EPT
   基于硬件支持的内存虚拟化，intel CPU实现了EPT(Extended Page Table， 扩展页表)，将GVA到HPA的转换由硬件来完成，具体转换流程为：

```
1. 在 Guest OS 运行时，处于非root模式的CPU加载guest进程的gCR3
2. guest 访问gCR3，由传统页表实现GVA到GPA的转换
3. 再通过查询EPT完成GPA到HPA的转换
```

EPT方案下，为每个guest只维护一个EPT，只有cpu处于非root模式下才参与内存地址的转换，guest os的page fault在内部处理，不会exit到vmm，但是如果tlb miss，两级页表的查询会引入大量开销。
综合来看，通过硬件EPT技术，大幅减少了页表更新带来的vm exit，同时也大幅减少了内存虚拟化的难度，虽然也有多级页表查询的开销，但总体来看提升明显，是现在内存虚拟化的主流方案。



2.3 qemu-kvm

上节说到，硬件辅助虚拟化既不用修改guest os保持了很好的兼容性，又有接近半虚拟化的性能，是当前虚拟化领域的大势所趋，而在linux环境下，qemu-kvm则是当前最主流的方案。

2.3.1 qemu

https://www.qemu.org/

What is QEMU?

QEMU is a generic and open source machine emulator and virtualizer.

qemu(Quick Emulator)是一个开源的虚拟化软件，是主机上的vmm，通过动态二进制转换来模拟CPU，并提供一系列的硬件模型，使guest os认为自己和硬件直接打交道，其实是同QEMU模拟出来的硬件打交道，QEMU再将这些指令翻译给真正硬件进行操作。
qemu自身就是一个完整的虚拟化方案，不需要其他任何组件，但纯qemu的方案效率太低，因此需要加速方案，cpu、内存的虚拟化通过硬件辅助的方式实现，网络、存储的加速在下文中会提到。

2.3.2 kvm

https://www.linux-kvm.org/page/Main_Page

Kernel Virtual Machine

KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko.

Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc.

KVM is open source software. The kernel component of KVM is included in mainline Linux, as of 2.6.20. The userspace component of KVM is included in mainline QEMU, as of 1.3.

Blogs from people active in KVM-related virtualization development are syndicated at http://planet.virt-tools.org/

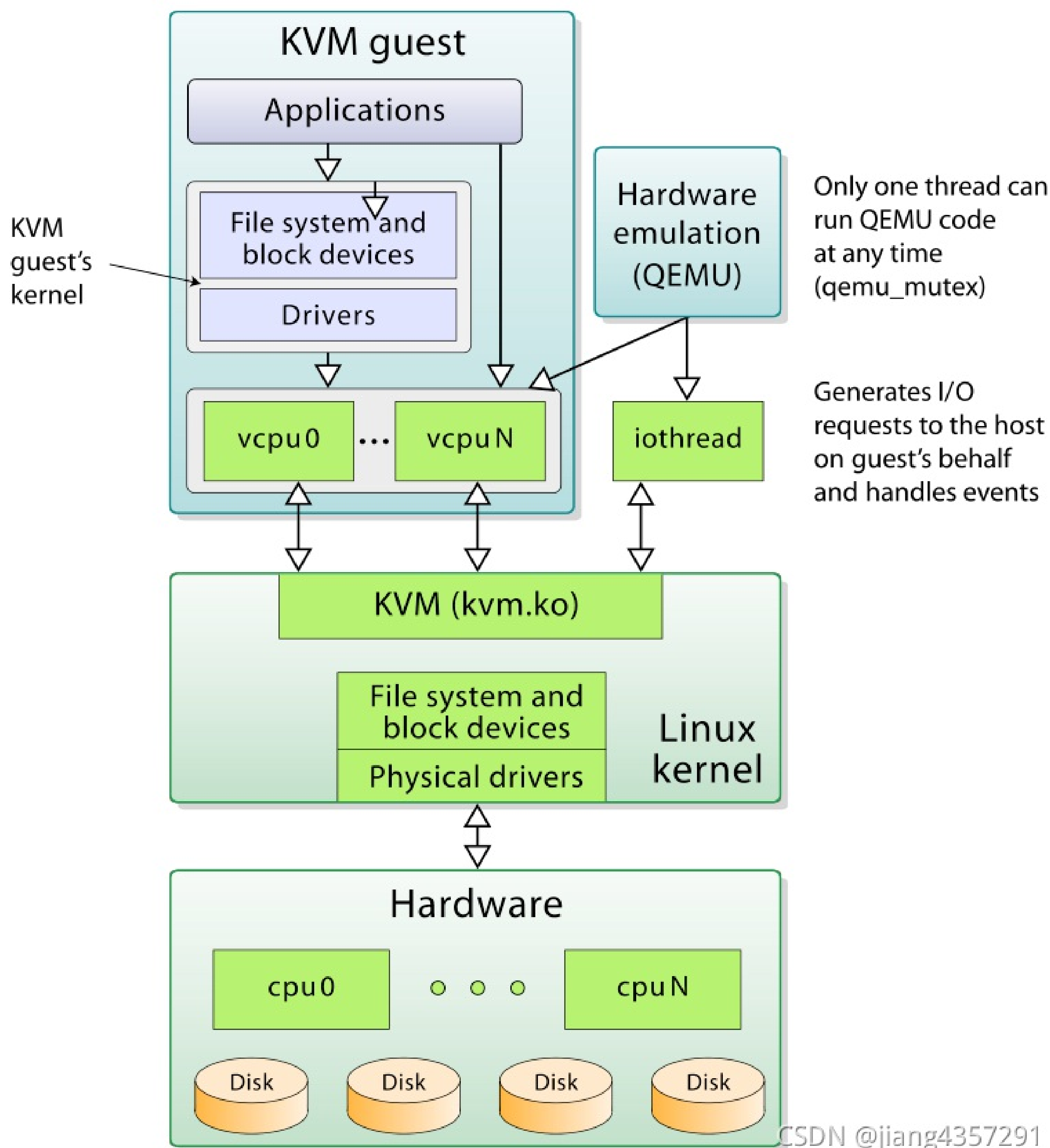
CSDN @jiang4357291

kvm(Kernel Virtual Machine)是Linux on x86上的一个全虚拟化解决方案，主要由两个内核模块组成，kvm.ko提供核心虚拟化功能，kvm-intel.ko或kvm-amd.ko提供硬件虚拟化能力。kvm能够让Linux主机成为一个Hypervisor，kvm只实现cpu和内存的虚拟化，但是需要cpu硬件本身支持虚拟化扩展，也即Intel VT和AMD-V。本质上，KVM是管理虚拟硬件设备的驱动，该驱动使用字符设备/dev/kvm（由KVM本身创建）作为管理接口，主要负责vCPU的创建，虚拟内存的分配，vCPU寄存器的读写以及vCPU的运行。(有关kvm的cpu虚拟化和内存虚拟化会在qemu-kvm中介绍。)
每一个kvm客户机对应一个linux进程，由标准 Linux 调度程序进行调度，每一个vCPU是该进程下的一个子线程，这使得kvm可以使用linux内核的已有功能。
kvm通过硬件辅助虚拟化可以接近物理机的性能，但其本身并不是一个完整的虚拟化方案，只能虚拟化cpu和内存。

2.3.3 qemu-kvm

通过前面的介绍我们可以看到：

- 从qemu的角度来看：qemu是完整的虚拟化方案，但由于所以指令都要经过qemu转译，导致性能太低
- 从kvm的角度来看：kvm是借助硬件辅助方案实现虚拟化，性能损失极低，但其只实现了cpu和内存的虚拟化，且其运行在内核空间，用户无法直接和其进行交互，需要依赖用户态的管理工具
  可以看到qemu和kvm天然存在着互补关系，因此也就衍生出了一个对两者互相取长补短的方案：qemu-kvm，其主要架构如下图：
  - 一个虚拟机对应一个qemu进程
  - vcpu线程用于运行guest代码
  - 单独的io线程用于管理模拟的设备
  - 此外还有其他如处理 event loop，offloaded tasks 等的线程



在qemu-kvm架构下，虚拟机的配置和创建、虚拟设备的模拟、虚机运行时的用户环境和交互等都是由qemu完成的，而在虚机运行状态下，QEMU 会通过 KVM 模块提供的系统调用进入内核，由 KVM 负责将虚拟机置于处理的特殊模式运行。当虚机进行 I/O 操作时，KVM 会从上次系统调用出口处返回 QEMU，由 QEMU 来负责解析和模拟这些设备。
下面通过一段伪代码来说明qemu-kvm的启动和工作流程：

```
// 第一步，获取到 KVM 句柄
kvmfd = open("/dev/kvm", O_RDWR);
// 第二步，创建虚拟机，获取到虚拟机句柄。
vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);
// 第三步，为虚拟机映射内存，还有其他的 PCI，信号处理的初始化。
ioctl(kvmfd, KVM_SET_USER_MEMORY_REGION, &mem);
// 第四步，将虚拟机镜像映射到内存，相当于物理机的 boot 过程，把镜像映射到内存。
// 第五步，创建 vCPU，并为 vCPU 分配内存空间。
ioctl(kvmfd, KVM_CREATE_VCPU, vcpuid);
vcpu->kvm_run_mmap_size = ioctl(kvm->dev_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
// 第五步，创建 vCPU 个数的线程并运行虚拟机。
ioctl(kvm->vcpus->vcpu_fd, KVM_RUN, 0);
// 第六步，线程进入循环，并捕获虚拟机退出原因，做相应的处理。
for (;;) {
    ioctl(KVM_RUN)
    switch (exit_reason) {
        case KVM_EXIT_IO:  /* ... */
        case KVM_EXIT_HLT: /* ... */
    }
}
// 这里的退出并不一定是虚拟机关机，
// 虚拟机如果遇到 I/O 操作，访问硬件设备，缺页中断等都会退出执行，
// 退出执行可以理解为将 CPU 执行上下文返回到 Qemu。
```

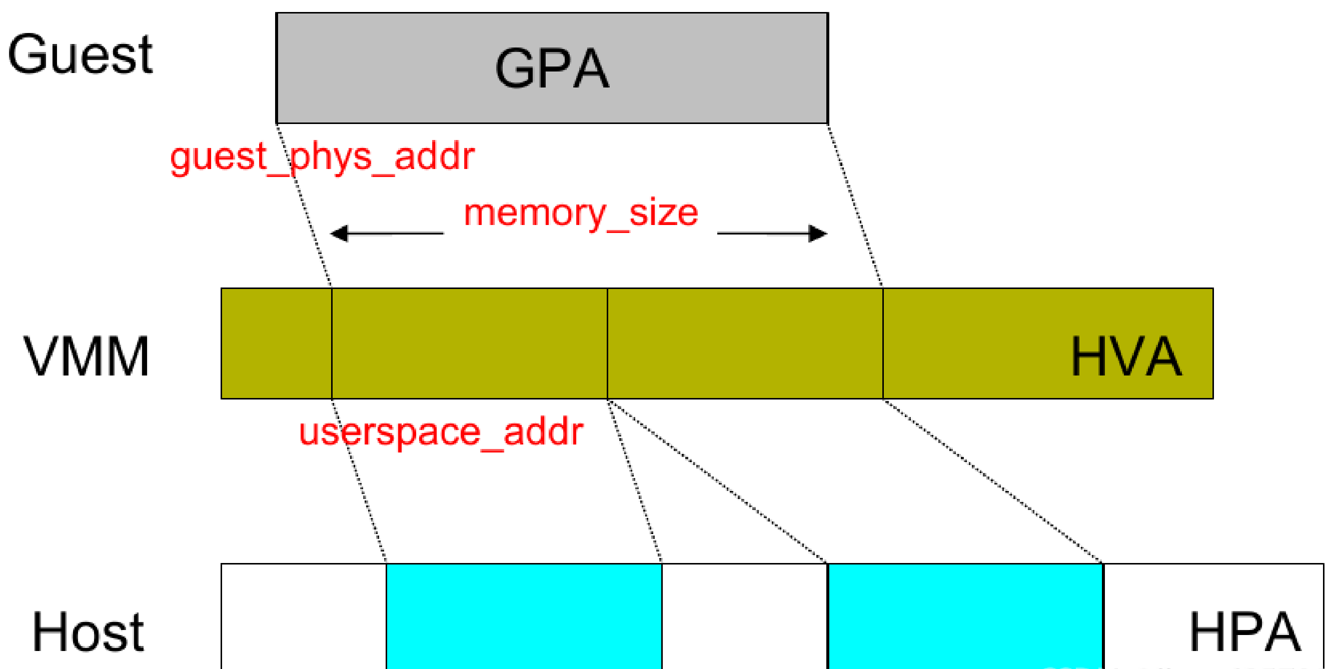
cpu虚拟化

虚机在 KVM 的支持下，被置于VMX的非根模式下执行二进制指令。在非root模式下，所有敏感的二进制指令都被CPU捕捉到，CPU 在保存现场之后自动切换到根模式，由 KVM 决定如何处理(或直接由kvm处理或交由用户态的qemu处理)。

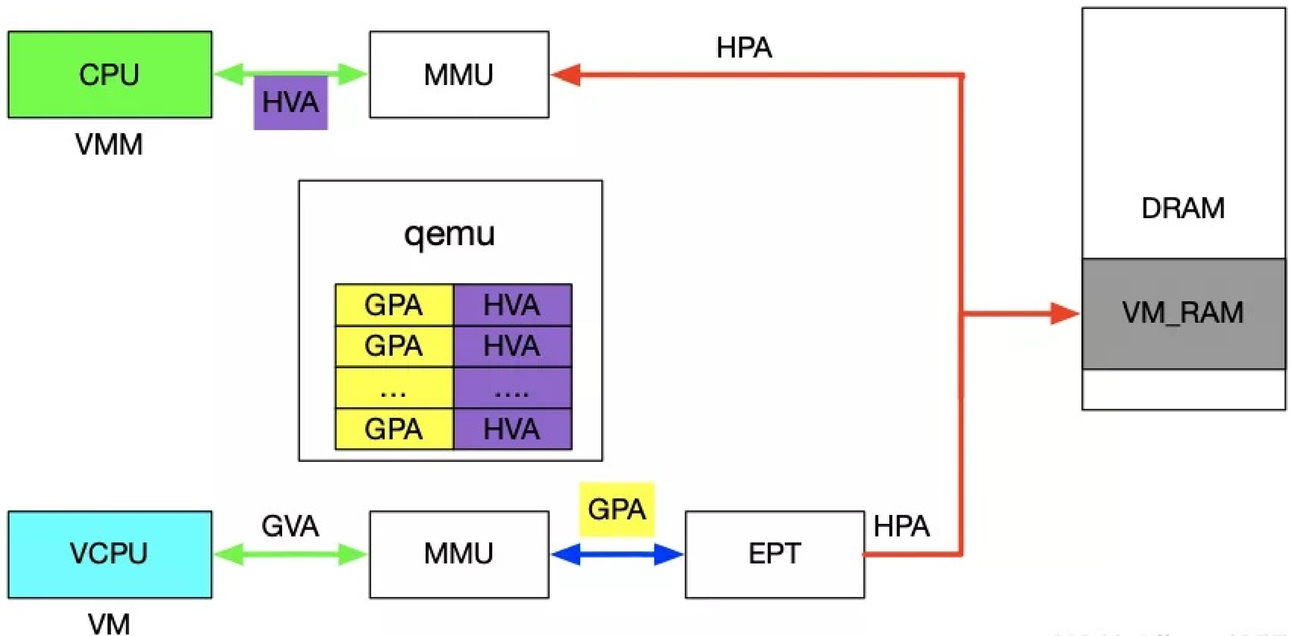
内存虚拟化

在qemu-kvm架构下，GPA是由qemu进行申请，由kvm进行管理的，具体来看：

qemu根据guest的内存大小通过mmap系统调用在本进程的虚拟地址空间中申请对应大小的连续内存块(只是HVA连续)，再通过ioctl的KVM_SET_USER_MEMORY_REGION接口将该内存地址注册到kvm中，由kvm进行维护。其中ioctl传入的参数主要有两个：guest_phys_addr为虚机GPA起始地址，userspace_addr为mmap得到的HVA起始地址。kvm拿到GPA和GVA的起始地址后，就会为当前虚机建立EPT，实现GPA->HPA的映射，同时会为VMM建立HVA->HPA映射。



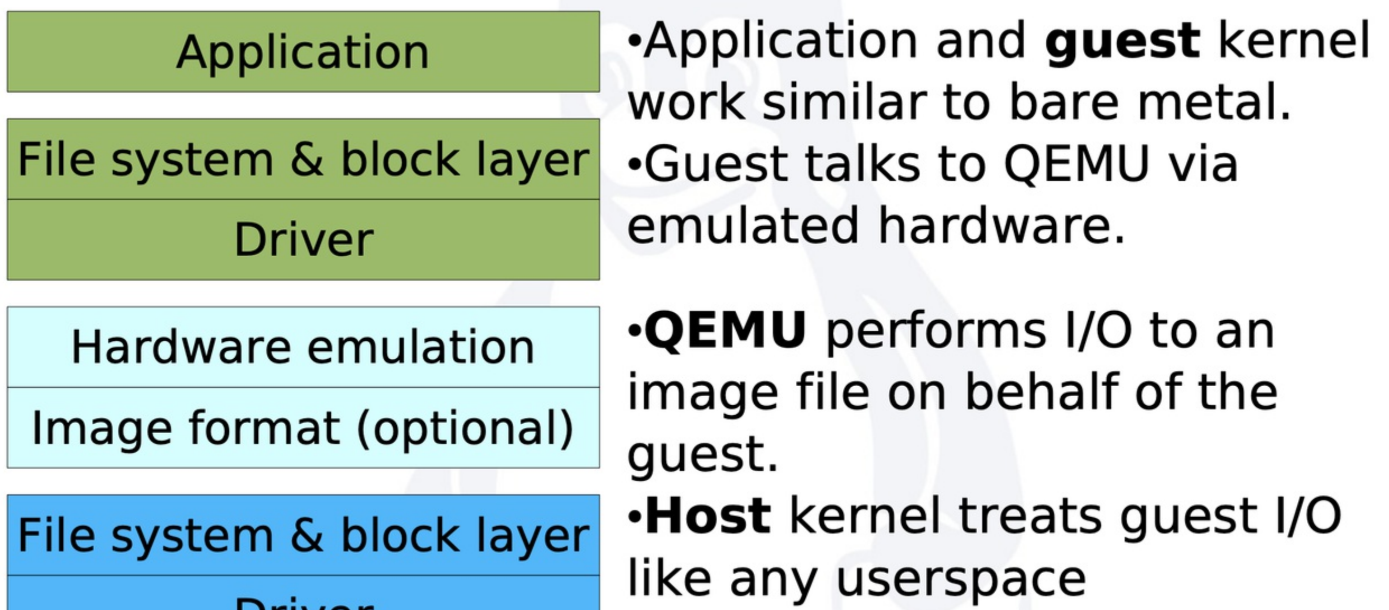
vm exit发生时，vmm需要能够处理异常，此时vmm获取的是GPA，需要能转换到HPA，由于GPA和HVA的映射关系是qemu维护的，且已经传给了kvm，所以kvm可以通过GPA查询到对应的HVA，再转换到HPA。下图清晰展示了vm和vmm的内存映射关系：



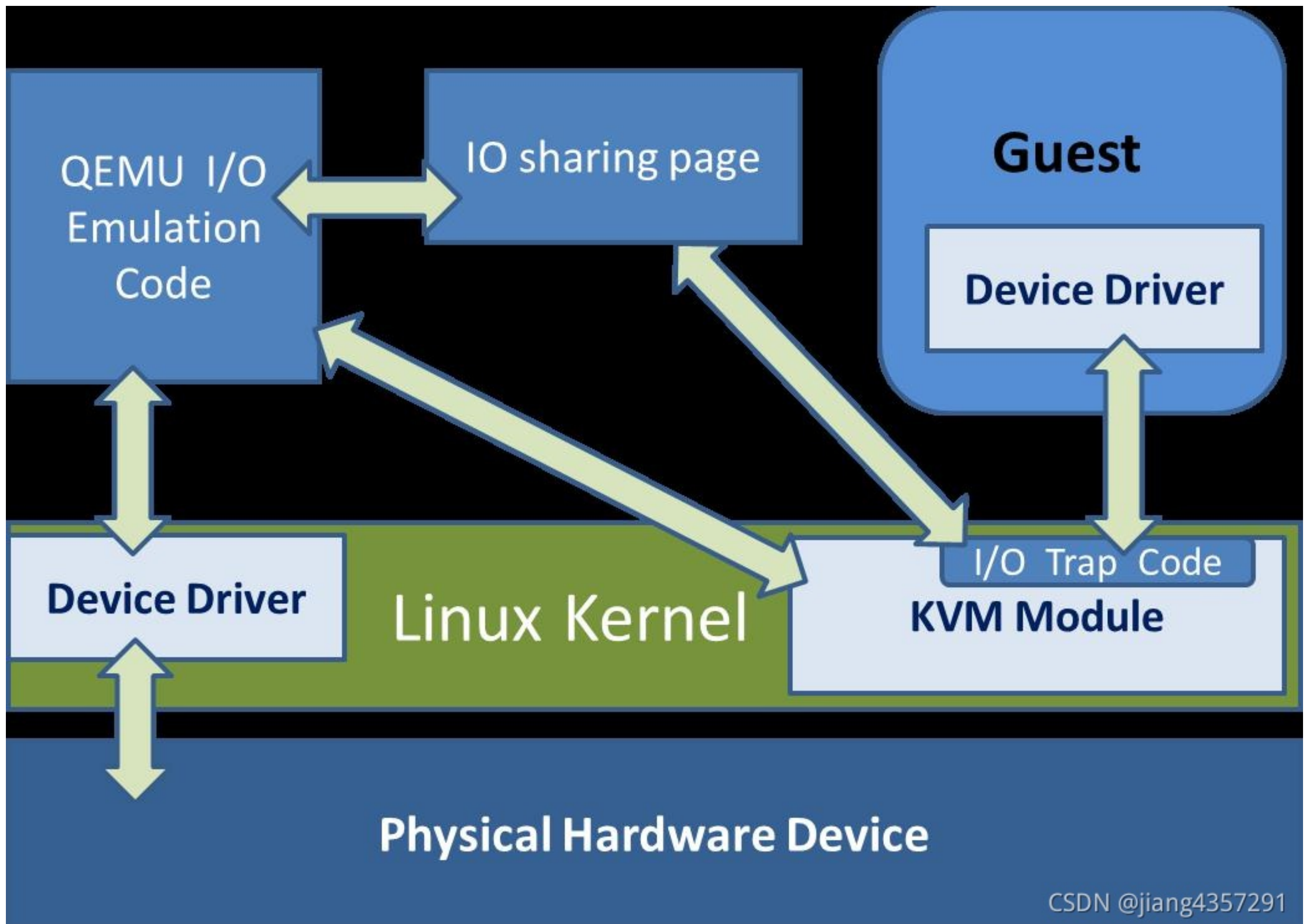
三、存储虚拟化

3.1 全虚拟化IO

qemu是软件实现的全虚拟化方案，在全虚拟化io的架构下，qemu通过本地的镜像文件向guest模拟出硬盘设备，所有guest io对host来说就和其他应用写本地文件一样。



具体来看下完整的io流程：




```
1. guest 发起io
2. 对PCI空间的读写是特权指令，会触发VM Exit，被kvm的i/o trap code捕获，kvm将io信息放到sharing page，并通知用户态的qemu
3. qemu从共享页中取出io请求，交由硬件模拟代码去处理：io需要经过host文件系统->page cache->block device这套完整的链路
4. qemu完成此次io后，再将结果放回共享页，并通知kvm
5. kvm中的i/o trap code读取sharing page中的操作结果，并将结果返回到客户机中
6. 触发VM Entry，guest再次获取cpu控制权，根据io返回结果进行处理
```

当 Guest 通过 DMA 访问大块内存时，QEMU 模拟程序不会把操作结果放到 I/O 共享页中，而是通过内存映射的方式将结果直接写到 Guest 的内存中去，然后通过 KVM 告诉Guest 的 DMA 操作完成。
全虚拟化的io方案简单通用，可以模拟各种硬件设备，但性能很差，具体原因有：

1. io路径过长，且存在多处数据复制
2. 频繁的VMEntry、VMExit，多次上下文切换
   由于全虚拟化io的以上缺点，演进出了在性能方面更具优势的半虚拟化io。

3.2 virtio

3.2.1 概述

Virtio

So-called "full virtualization" is a nice feature because it allows you to run any operating system virtualized. However, it's slow because the hypervisor has to emulate actual physical devices such as RTL8139 network cards . This emulation is both complicated and inefficient.

Virtio is a virtualization standard for network and disk device drivers where just the guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor. This enables guests to get high performance network and disk operations, and gives most of the performance benefits of paravirtualization.

Note that virtio is different, but architecturally similar to, Xen paravirtualized device drivers (such as the ones that you can install in a Windows guest to make it go faster under Xen). Also similar is VMWare's Guest Tools.

CSDN @jiang4357291

virtio是一套通用的半虚拟化io框架，提供了在hypervisor之上通用模拟设备IO的抽象，它基于hypervisor导出一组通用的io模拟设备，并基于一组通用api使得这些设备可以在虚机内使用。在virtio的设计中，客户机意识到自己运行在虚拟化环境中，通过virtio标准与hypervisor进行配合，进而达到更好的性能。
Guest 使用 VirtIO devices 最典型的方式是通过 PCI/PCIe 协议，PCI/PCIe 是 QEMU 和 Linux 中成熟且支持良好的总线协议。在物理环境中，PCI/PCIe 硬件设备会使用特定的物理内存地址范围，设备的驱动程序可以通过访问该内存范围来读取或写入设备的寄存器，也可以通过特殊的处理器指令来暴露其配置空间（Configuration Space）。基于这个原理，在虚拟化环境中，Hypevisor 可以通过捕获对该内存范围的访问并执行设备仿真。VirtIO 规范还定义了 PCI 配置空间的布局，因此实现起来非常简单。
virtio起初只是Rusty Russell针对自己的虚拟化方案lguest提出的，如今已经成为半虚拟化io的事实标准。virtio的意义有两个：

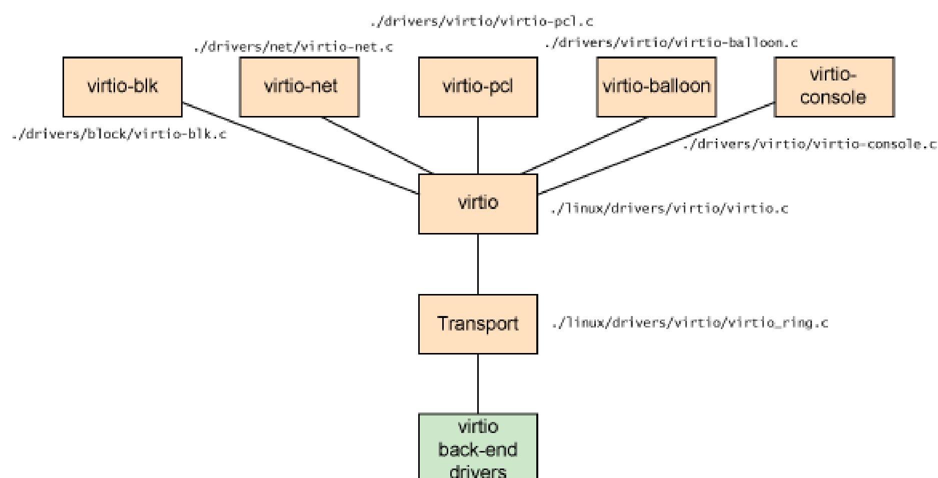
- 为众多虚拟化平台提供了一个统一的io模型，KVM、XEN、VMWare等均可以利用virtio进行io虚拟化
- 相对于全虚拟化io方案，提升了io性能

3.2.2 架构

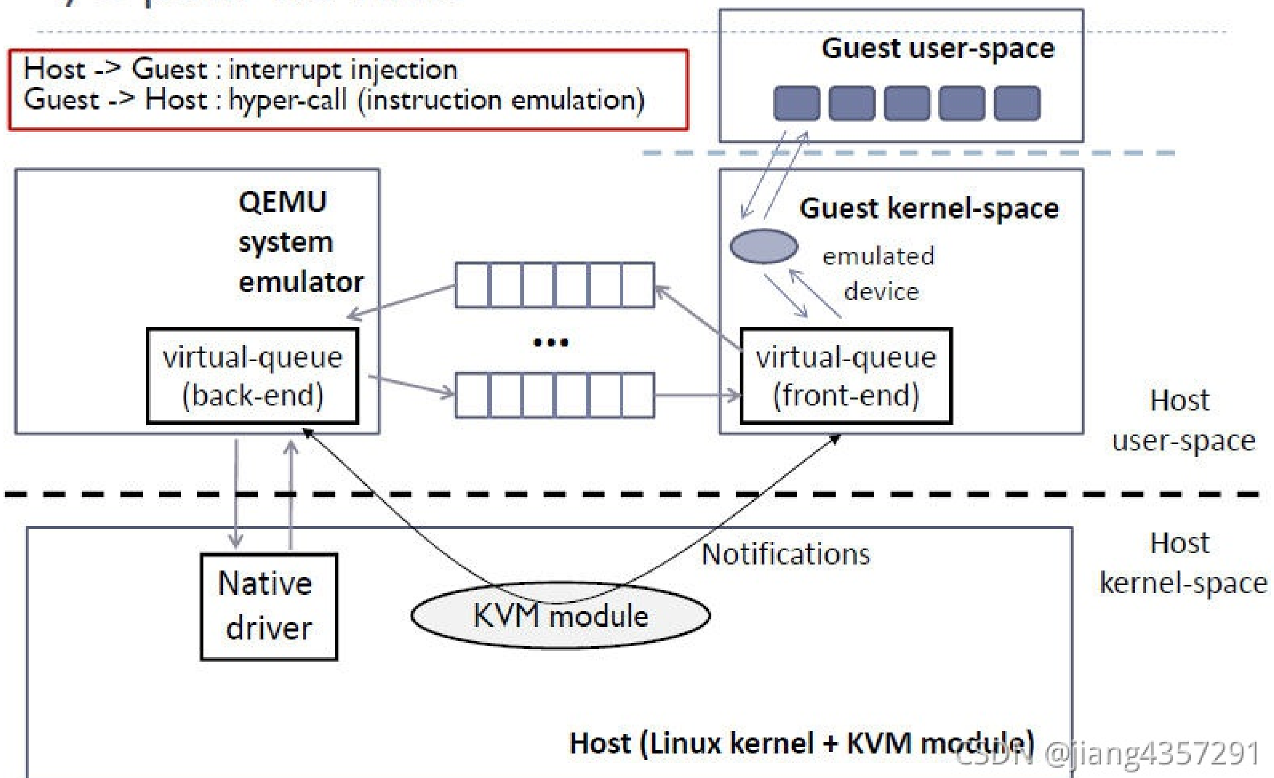
virtio是前后端的架构，以qemu-kvm+virtio为例，前端是位于guest os中的kernel module，后端是qemu中的驱动代码。前后端之间通过一个ring buffer进行交互，前端将I/O 请求放到buffer中，后端取出后再进行处理，处理完成后再放回buffer中，一次交互过程可以有多个io。具体ring buffer的组织方式也就是virtqueue。
virtio提供io设备的统一抽象，所以在前端中可以实现各种基于virtio的io设备驱动，如网络virtio_net，硬盘virtio_blk和virtio_scsi。

virtio_ blk
virtio_ net
virtio_ pci
virtio_ balloon
virtio_ scsi
virtio_ console
virtio
transport (virtio-ring)
virtio backend
CSDN @jiang4357291

virtio在linux kernel中的实现：




```
1. guest 发起io
2. io到达guest os，由kernel中的virtio前端驱动进行处理，将io放到virtio-ring中并通知virtio后端
3. qemu作为virtio后端从virtio-ring中取出io请求并进行处理，可以一次性取出多个io并处理
4. qemu完成此次io后，再将结果放回virtio-ring，并通知virtio前端
5. 客户机virtio前端获取io结果并最终返回给应用
```



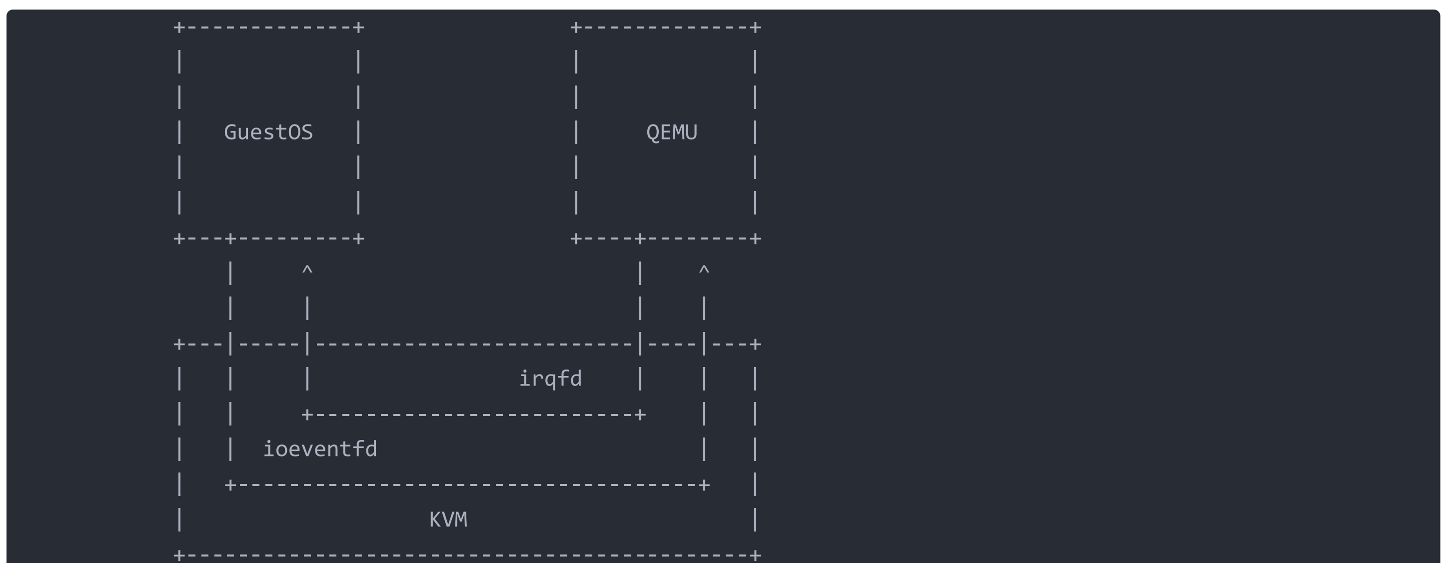
关于virtio-ring在qemu-kvm场景下：

内存虚拟化的时候介绍过，guest的GPA内存空间是由qemu通过mmap进行申请的，virtio-ring便是由前端驱动在GPA空间上申请的，所以当qemu去从中取io请求时，可以直接将GPA转换到对应的HVA；在io完成后又可以将io结果直接写到GPA上，整个virtio-ring的交互过程无需拷贝。

前后端的通知机制：

guest通知qemu通过ioeventfd，qemu通知guest通过irqfd，两者都是通过eventfd实现的。

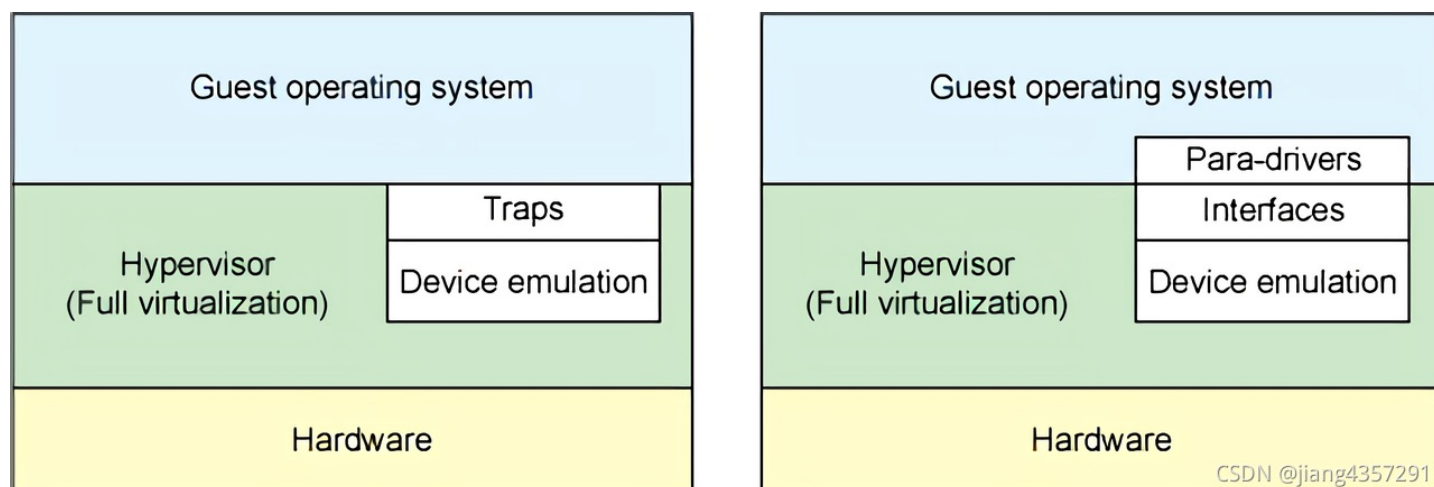
- ioeventfd: 将一个eventfd绑定到一段客户机的地址空间，当guest写这段地址空间时，会触发EPT_MISCONFIGURATION缺页异常，KVM处理时如果发现这段地址落在了已注册的ioeventfd地址区间里，会通过写关联eventfd通知qemu
- irqfd: kvm为host通知guest提供的机制，将一个eventfd与一个全局中断号联系起来，当qemu写该eventfd时，kvm作为另一侧被唤醒，接着调用irqfd_inject将对应的中断注入到虚拟机中



问题思考:

https://stackoverflow.com/questions/46418131/in-virtio-why-does-guest-notifier-and-host-notifier-use-ioeventfd-and-irqfd-res

总结：基于virtio的半虚拟化io方案，一方面减少了VM Exit和VM Entry(主要优化，VM Exit对性能的影响巨大)，一方面基于vrtio协议，一次可以并行处理多个io，在性能上较之全虚拟化io有明显提升，但要注意其并未缩短io路径，io还是需要经过qemu好host kernel。
最后再回过来看全虚拟化io和半虚拟化io的区别：

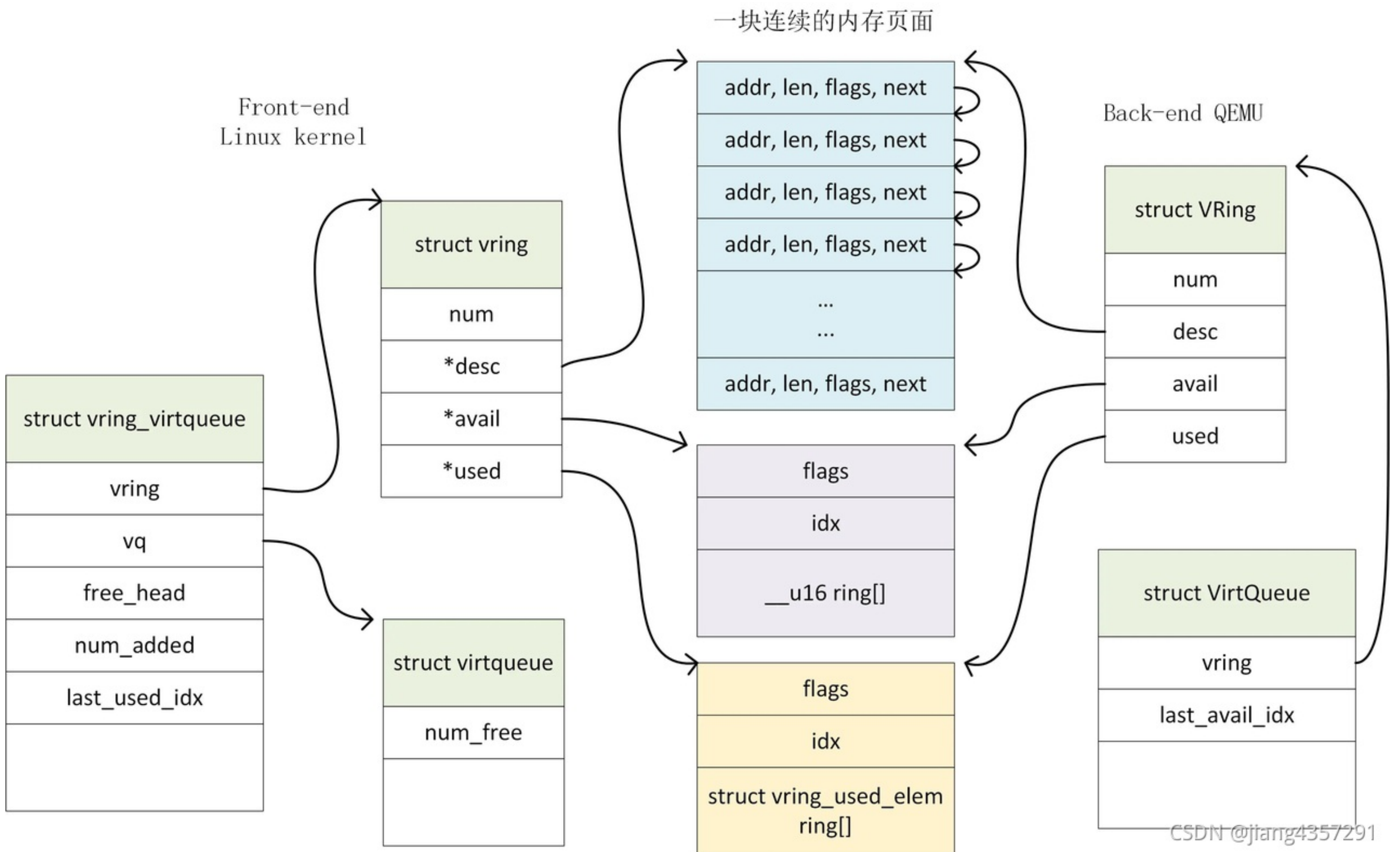


3.2.3 virtqueue

virtqueue就是virtio-ring的具体组织形式，virtio的前后端基于virtqueue来实现io传输，每种设备可以有0个或多个virtqueue，每个virtqueue由三部分组成：

```
+------------------+-------------------------------+-----------------------+
    | Descriptor Table |   Available Ring  (padding)    |       Used Ring    |
+------------------+-------------------------------+-----------------------+
```

其总体结构如下：



- Descriptor Table：存放描述符，每个描述符指向一块buffer
- Available Ring：guest driver发起请求时，将在descriptor table中的索引放到avail ring中，后端device不断从中取出并进行消费
- Used Ring ：后端device从avail中取出的descriptor并处理完成后，将desc索引放到used ring中
  virtio 1.0之前要求这三部分在一块连续内存上，1.0之后则无此要求，只需要各自连续即可。virtio 1.1之后对这三部分的名称也改了，且引入了packed virtqueue，具体变更可参见：
  https://www.dpdk.org/wp-content/uploads/sites/35/2018/09/virtio-1.1_v4.pdf

Each virtqueue can consist of up to 3 parts:

- Descriptor Area - used for describing buffers
- Driver Area - extra data supplied by driver to the device
- Device Area - extra data supplied by device to driver

Note: Note that previous versions of this spec used different names for these parts (following 2.6):

- Descriptor Table - for the Descriptor Area
- Available Ring - for the Driver Area
- Used Ring - for the Device Area

CSDN @jiang4357291

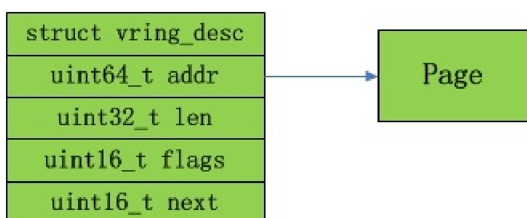
下面介绍主要针对1.0版本的split virtqueue:

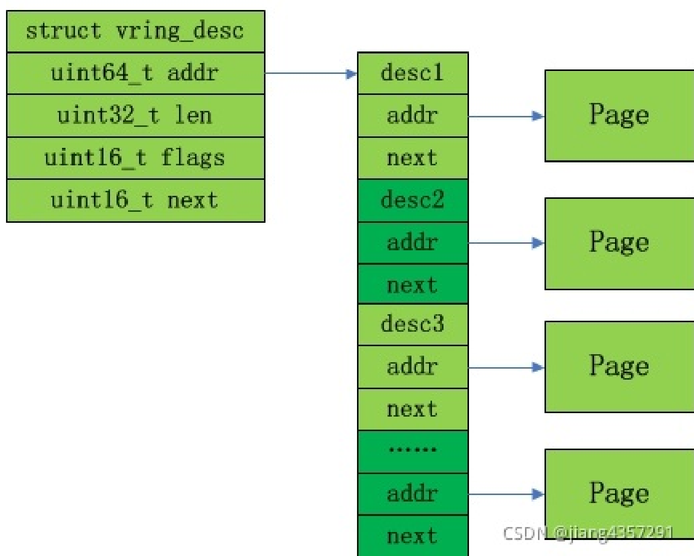
1. Descriptor Table

```
struct virtq_desc {
        /* Address (guest-physical). */
        le64 addr;
        /* Length. */
        le32 len;

/* This marks a buffer as continuing via the next field. */
#define VIRTQ_DESC_F_NEXT   1
/* This marks a buffer as device write-only (otherwise device read-only). */
#define VIRTQ_DESC_F_WRITE     2
/* This means the buffer contains a list of buffer descriptors. */
#define VIRTQ_DESC_F_INDIRECT   4
        /* The flags as indicated above. */
        le16 flags;
        /* Next field if flags & NEXT */
        le16 next;
};
```

- addr：每个desc，都会对应一个buffer，addr即为此desc对应buffer的地址，地址为GPA
- len：buffer的总长度
- flags：
  - VIRTQ_DESC_F_WRITE表示buffer是write-only的，否则是read-only的；write-only的buffer表示virtio前端希望后端填充的，此desc称为in类型，反之read-only的buffer则是希望后端读取的，称为out类型
  - 一次交互不一定只有一个descriptor，可以由多个desc组成一组descriptor chain，后端在读取desc的时候，如果有VIRTQ_DESC_F_NEXT的flag，则表示后面还有descriptor，需要继续读取，此时next就是下一个descriptor；否则当前desc就是descriptor chain中的最后一个
  - 通常情况下addr为buffer page，这种descriptor称谓direct desc；但如果有VIRTQ_DESC_F_INDIRECT的flag则该descriptor对应的buffer是一组descriptor list，此为indirect desc
- next：配合flag进行使用，要注意的是next并不是GPA，而是下一个descriptor在descriptor table中的索引





2. Available Ring

Guest driver通过avail ring向device提供buffer，每次将io request转换为一组descriptor chain，并向avail ring中添加一个元素，即avail ring的每一个entry指向一组descriptor chain的头部(Decriptor Table索引)，avail ring只会被driver填写，device读取。

当Guest Driver向Vring中添加buffer时，可以一次添加一个或多个buffer，所有buffer组成一个Descriptor chain，Guest Driver添加buffer成功后，需要将Descriptor chain头部的地址记录到Avail Ring中，让Host端能够知道新的可用的buffer是从VRing的哪个地方开始的。Host查找Descriptor chain头部地址，需要经过两次索引Buffer Adress = Descriptor Table[Avail Ring[last_avail_idx]]，last_avail_idx是Host端记录的Guest上一次增加的buffer在Avail Ring中的位置。Guest Driver每添加一次buffer，就将Avail Ring的idx加1，以表示自己工作在Avail Ring中的哪个位置。当host被通知并取出desc后，此时[last_avail_idx,avial->idx)区间则是要处理的请求。

```
struct virtq_avail {
#define VIRTQ_AVAIL_F_NO_INTERRUPT      1
        le16 flags;
        le16 idx;
        le16 ring[ /* Queue Size */ ];
        le16 used_event; /* Only if VIRTIO_F_EVENT_IDX */
};
```

- flags：限制是否向guest注入中断
- idx：表示driver下一个descriptor将要放在avail ring的位置，默认从0开始(单调递增，需要%descriptor table的长度)
- ring：一个索引数组，每一个成员对应在descriptor table中表项的下标，代表一个buffer的head。

3. Used Ring

```
struct virtq_used {
#define VIRTQ_USED_F_NO_NOTIFY  1
        le16 flags;
        le16 idx;
        struct virtq_used_elem ring[ /* Queue Size */];
        le16 avail_event; /* Only if VIRTIO_F_EVENT_IDX */
};

/* le32 is used here for ids for padding reasons. */
struct virtq_used_elem {
        /* Index of start of used descriptor chain. */
        le32 id;
        /* Total length of the descriptor chain which was used (written to) */
        le32 len;
};
```

Host device通过used ring归还buffer，其只会被device填写，diver读取。used ring的主体也是一个数组，但不同于avail ring只需要记录索引，used ring由于是存放处理后的结果，所以还需要记录写回的数据长度。

- flags：用于限制客户机是否增加buffer后是否通知host
- idx：device下次往used ring中添加元素的下标
- id：相当于avail ring中的ring成员的value，表示一个used descriptor chain的头部的下标
- len：写到该descriptor chain对应的buffer中数据的总长度

3.2.4 virtio-blk/virtio-scsi

基于virtio实现的块设备驱动有两种，virtio-blk和virtio-scsi：

- virtio-blk 是作为 pci 设备挂在 qemu 里面，所以最多只能有16块 virtio-blk 盘；virtio-scsi 作为 scsi 子系统，挂在 scsi 总线上，数量上可以多得多
- virtio-scsi 实现了 scsi 的协议，支持scsi命令，复杂度更高一些
- virtio-blk的io路径更短，所以性能上略好于virtio-scsi，两者io路径对比：

```
【virtio-blk】
guest: app -> Block Layer -> virtio-blk
host: QEMU -> Block Layer -> Block Device Driver -> Hardware

【virtio-scsi】
guest: app -> Block Layer -> SCSI Layer -> scsi_mod
host: QEMU -> Block Layer -> SCSI Layer -> Block Device Driver -> Hardware
```

下面介绍一下virtio-blk的协议细节：
一个virtio-blk的请求格式如下，注意只是逻辑上的表示，实际上并不是有一个virtio_blk_req的结构体定义。

```
struct virtio_blk_req {
        // out header
        le32 type;
        le32 reserved;
        le64 sector;
        // buffer
        u8 data[][512];
        // in header
        u8 status;
};
```

一个virtio_blk_req实际上分为3个部分：

- virtio_blk_outhdr
  每次请求的前16个字节一定是一个virtio_blk_outhdr，描述了io的类型，优先级，offset等信息；它由一个read-only的descriptor描述，

```
struct virtio_blk_outhdr
{
    __u32 type; // io的类型
    __u32 ioprio; // io优先级
    __u64 sector; // io offset，以512 bytes的sector为单位，通常后端收到后需要<<9转到以byte为单位
};
```

type的常用类型有：

```
enum {
    /* These two define direction. */
    VIRTIO_BLK_T_IN = 0, // 读
    VIRTIO_BLK_T_OUT = 1, // 写

    /* This bit says it's a scsi command, not an actual read or write. */
    VIRTIO_BLK_T_SCSI_CMD = 2,

    /* Cache flush command */
    VIRTIO_BLK_T_FLUSH = 4,

    /* Get device ID command */
    VIRTIO_BLK_T_GET_ID = 8,

    /* Discard command */
    VIRTIO_BLK_T_DISCARD = 11
};
```

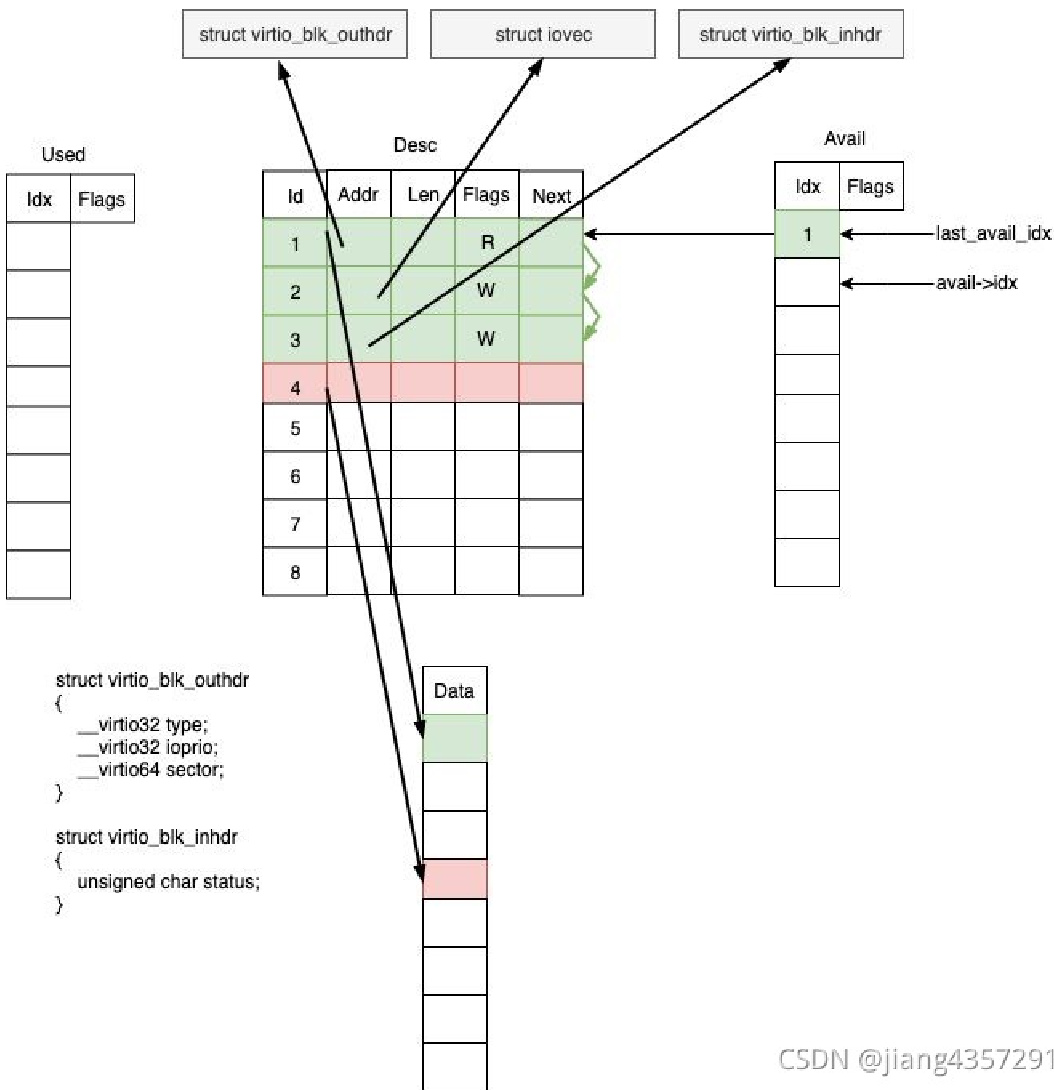
buffer
请求的中间是一个或多个buffer，这些buffer可能是read-only的也可能是write-only的，它们由descriptor chain中间的desc描述。

virtio_blk_inhdr
请求的最后一个字节是virtio_blk_inhdr，用于表示io结果，它由一个write-only的descriptor描述，由device进行填写。

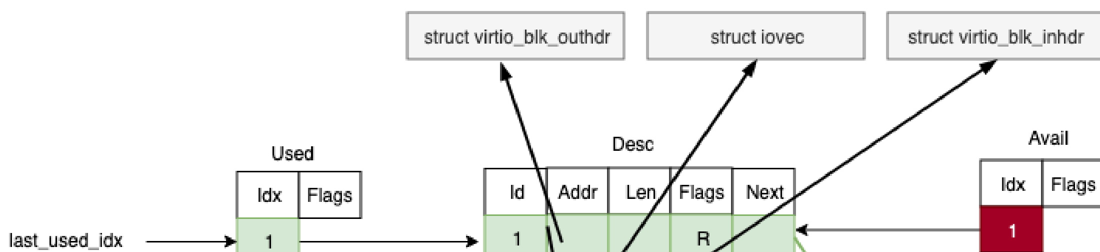
```
struct virtio_blk_inhdr {
    unsigned char status;
};
```

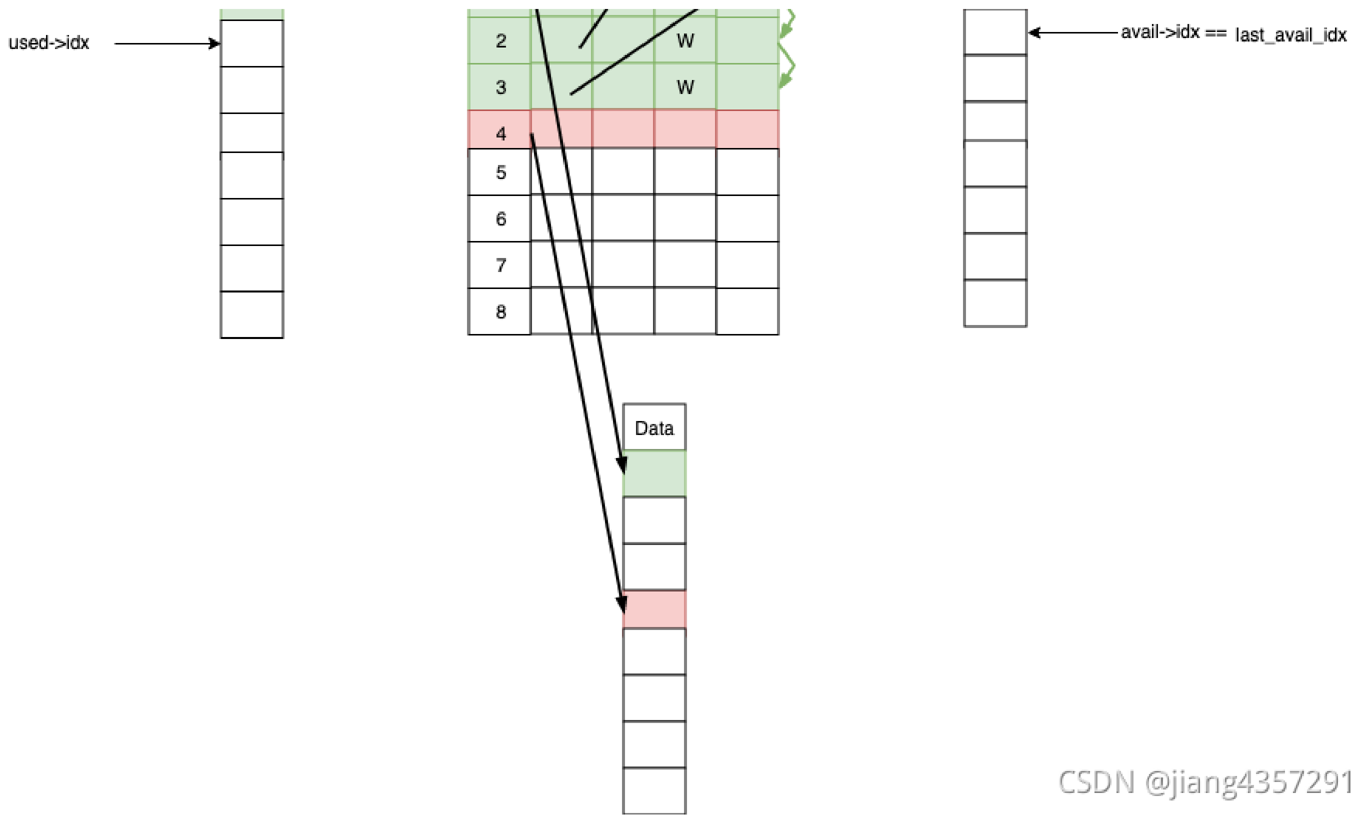
下面以两张图来看一次io过程中virtqueue的具体组织形式

host device接收请求



host device完成请求





3.3 spdk vhost-user

virtio半虚拟化io方案解决了频繁vm exit的问题，但是仍未缩短io路径，有待进一步优化，其性能上的瓶颈主要有两个：

- guest提交请求到virqueue后，还需要通知qemu
- qemu收到io请求并处理时还需要经过一次host上的完整io栈，其中还存在用户态到内核态的拷贝(写本地盘场景)
  于是为了进一步优化io性能，spdk vhost方案出现了。

3.3.1 spdk

https://spdk.io/doc/about.html

SPDK是由Intel发起的，用于加速NVMe SSD作为后端存储使用的应用软件加速库。这个软件库的核心是用户态、异步、轮询方式的NVMe驱动。相比内核的NVMe驱动，SPDK可以大幅降低NVMe command的延迟，提高单CPU核的IOps，形成一套高性价比的解决方案。

从目前来讲，SPDK并不是一个通用的适配解决方案。把内核驱动放到用户态，导致需要在用户态实施一套基于用户态软件驱动的完整I/O栈。文件系统毫无疑问是其中一个重要的话题，显而易见内核的文件系统，如ext4、Btrfs等都不能直接使用了。虽然目前SPDK提供了非常简单的文件系统blobfs/blostore，但是并不支持posix接口，为此使用文件系统的应用需要将其直接迁移到SPDK的用户态“文件系统”上，同时需要做一些代码移植的工作，如不使用posix接口，而采用类似AIO的异步读/写方式。

What is SPDK

The Storage Performance Development Kit (SPDK) provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications. It achieves high performance through the use of a number of key techniques:

- Moving all of the necessary drivers into userspace, which avoids syscalls and enables zero-copy access from the application.
- Polling hardware for completions instead of relying on interrupts, which lowers both total latency and latency variance.
- Avoiding all locks in the I/O path, instead relying on message passing.

The bedrock of SPDK is a user space, polled-mode, asynchronous, lockless NVMe driver. This provides zero-copy, highly parallel access directly to an SSD from a user space application. The driver is written as a C library with a single public header. See NVMe Driver for more details.

SPDK further provides a full block stack as a user space library that performs many of the same operations as a block stack in an operating system. This includes unifying the interface between disparate storage devices, queueing to handle conditions such as out of memory or I/O hangs, and logical volume management. See Block Device User Guide for more information.

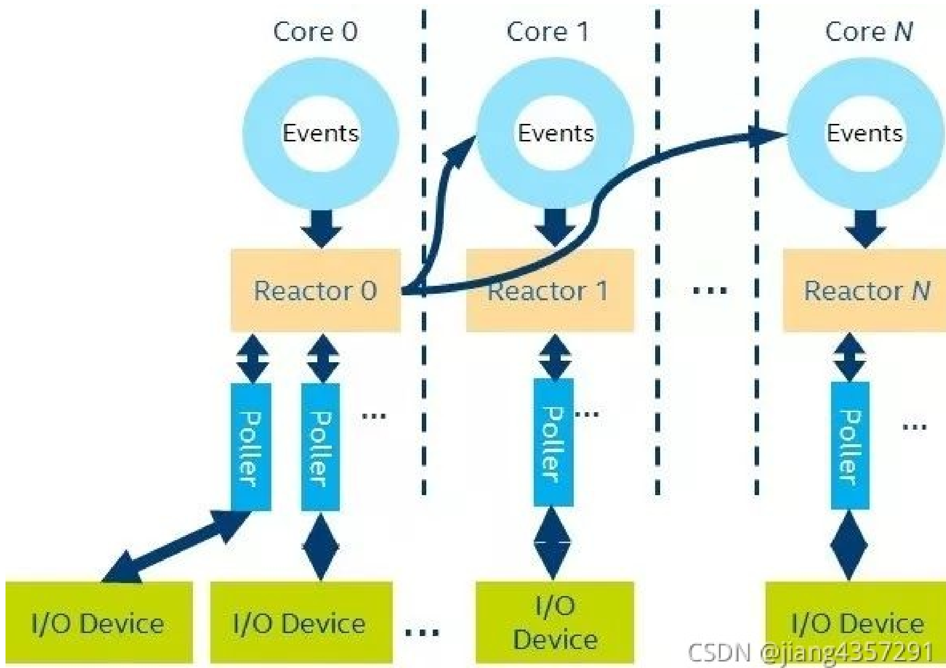
Finally, SPDK provides NVMe-oF, iSCSI, and vhost servers built on top of these components that are capable of serving disks over the network or to other processes. The standard Linux kernel initiators for NVMe-oF and iSCSI interoperate with these targets, as well as QEMU with vhost. These servers can be up to an order of magnitude more CPU efficient than other implementations. These targets can be used as examples of how to implement a high performance storage target, or used as the basis for production deployments.

CSDN @jiang4357291

spdk目前有主要以下几种应用场景：

- 提供块设备接口的后端存储应用，如iSCSI Target、NVMe-oF Target。
- 对虚拟机中I/O的加速，主要是指在Linux系统下QEMU/KVM作为Hypervisor管理虚拟机的场景，使用vhost交互协议，实现基于共享内存通道的高效vhost用户态Target。如vhost SCSI/blk/NVMe Target，从而加速虚拟机中virtio SCSI/blk及Kernel Native NVMe协议的I/O驱动。其主要原理是减少了VM中断等事件的数目（如interrupt、VM_EXIT），并且缩短了host OS中的I/O栈。
- SPDK加速数据库存储引擎，通过实现RocksDB中的抽象文件类，SPDK的blobfs/blobstore目前可以和RocksDB集成，用于加速在NVMe SSD上使用RocksDB引擎，其实质是bypass kernel文件系统，完全使用基于SPDK的用户态I/O栈。此外，参照SPDK对RocksDB的支持，亦可以用SPDK的blobfs/blobstore整合其他的数据库存储引擎。

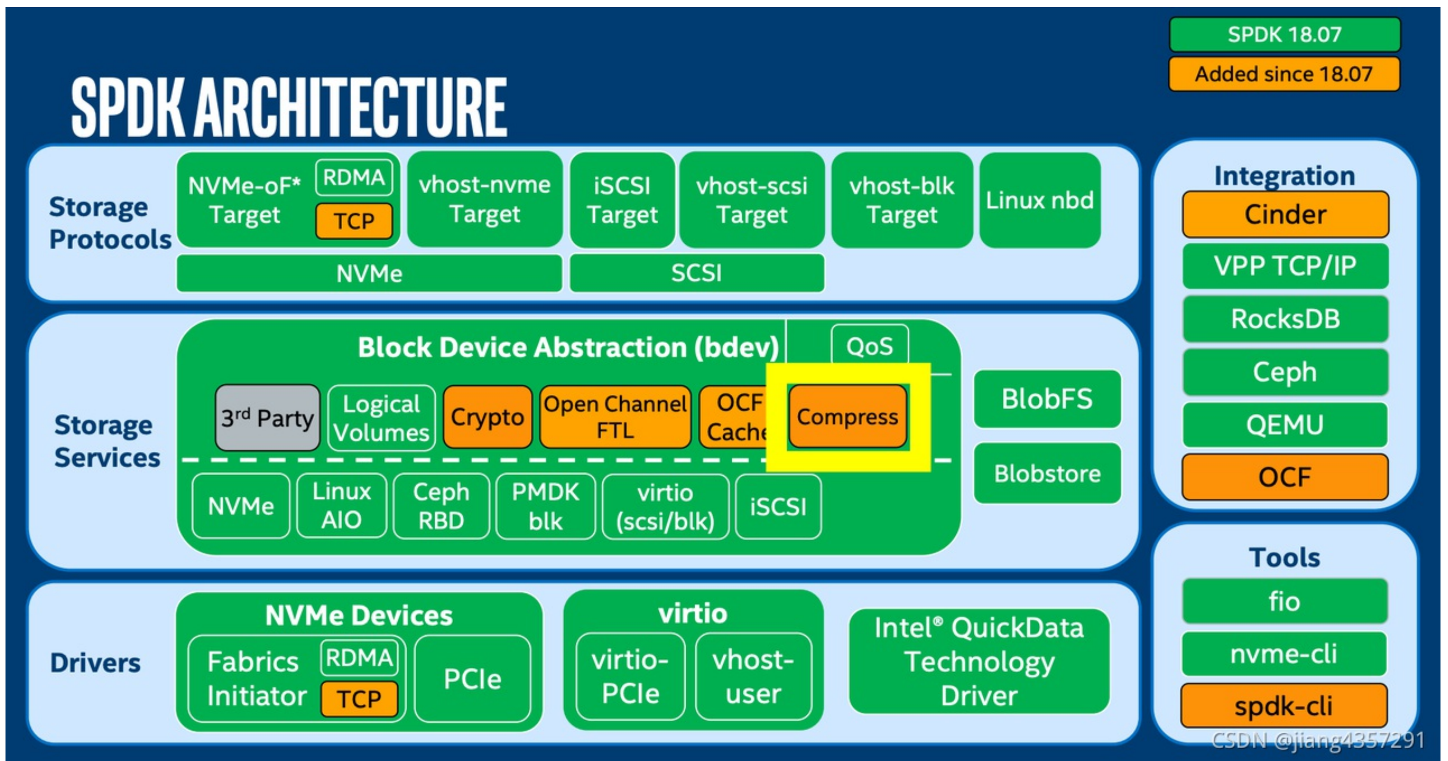
spdk应用框架：



1. 对CPU core和线程的管理
   SPDK的原则是使用最少的CPU核和线程来完成最多的任务。为此SPDK在初始化程序的时候限定使用绑定CPU的哪些核。通过CPU核绑定函数的亲和性，可以限制对CPU的使用，并且在每个核上运行一个thread，这个thread在SPDK中叫作Reactor。
   此外，SPDK提供了一个Poller机制。所谓Poller，其实就是用户定义函数的封装。SPDK的Reactor thread对应的数据结构由相应的列表来维护Poller的机制，并且提供Poller的注册及销毁函数。在Reactor的while循环中，会不停地检查这些Poller的状态，并且进行相应的调用。由于单个CPU核上，只有一个Reactor thread，所以同一个Reactor thread中不需要一些锁的机制来保护资源。当然位于不同CPU核上的thread还是有通信的必要的。为此，SPDK封装了线程间异步传递消息（Async Messaging Passing）的功能。
2. 线程间的高效通信
   SPDK提供了事件调用（Event）的机制用于线程间进行通信，这个机制的本质是每个Reactor对应的数据结构维护了一个Event事件的环，这个环是多生产者和单消费者（Multiple Producer Single Consumer，MPSC）的模型，意思是每个Reactor thread可以接收来自任何其他Reactor thread（包括当前的Reactor thread）的事件消息进行处理。
   目前SPDK中这个Event环的默认实现依赖于DPDK的机制，这个环应该有线性的锁的机制，但是相比较于线程间采用锁的机制进行同步，要高效得多。毫无疑问的是，这个Event环其实也在Reactor的函数_spdk_reactor_run中进行处理。每个Event事件的数据结构包括了需要执行的函数和相应的参数，以及要执行的core。
3. I/O的处理模型及数据路径的无锁化机制
   SPDK主要的I/O处理模型是run to completion，其原则是让一个线程最好执行完所有的任务。

- spdk架构
  整体的SPDK架构分为三层：

```
- 最下层为驱动层，管理物理和虚拟设备，还管理本地和远端设备。
- 中间层为通用块层，实现对不同后端设备的支持，提供对上层的统一接口，包括逻辑卷的支持、流量控制的支持等存储服务。这一层也提供了对Blob（Binary Larger Object）及简单用户态文件系统BlobFS的支持。
- 最上层为协议层，包括NVMe协议、SCSI协议等，可以更好地和上层应用相结合。
```

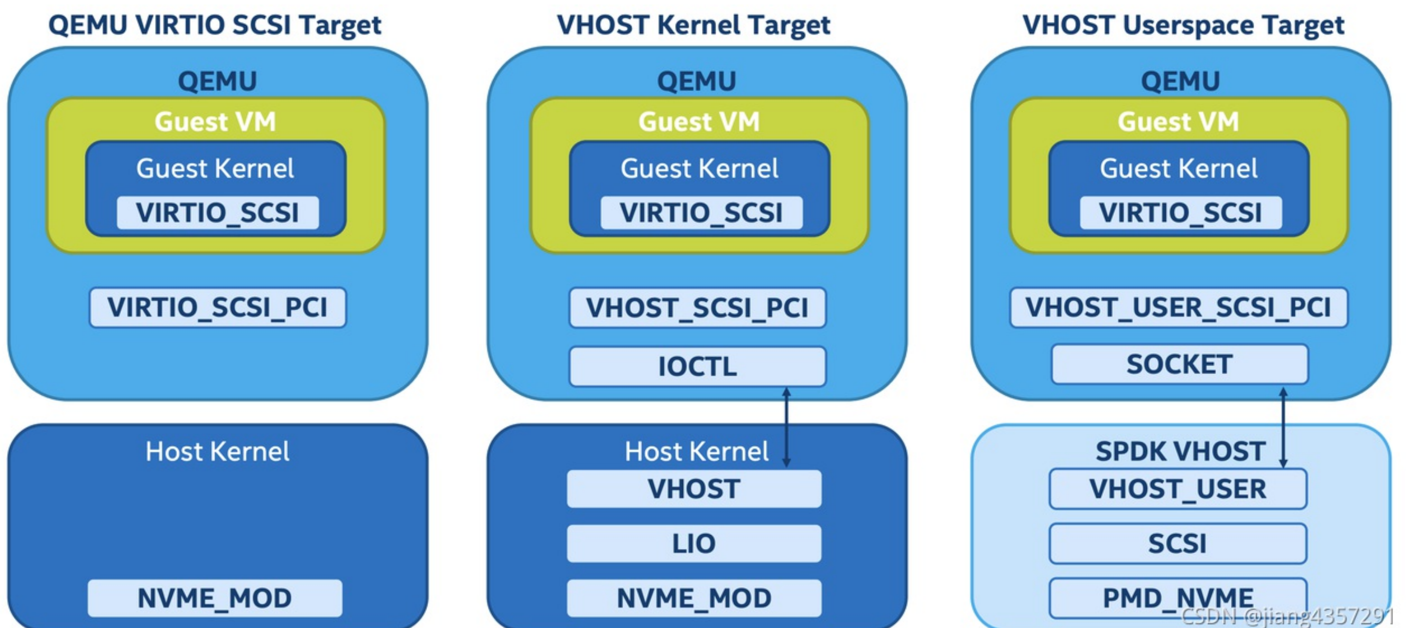


CSDN @jiang4357291

spdk目前主要的应用场景就是块存储，其通过bdev接口层，统一了块设备的调用方法，使用者只要调用不同的rpc将不同的块设备加到spdk进程中，就可以使用各种bdev，而不用修改代码。
一个很常见的使用spdk的方式是，用户定义自己的bdev，用以访问自己的分布式存储集群。

3.3.2 基于vhost的加速方案

把virtio backend在qemu外实现即为vhost，spdk target对外暴露指定协议的存储服务。下面以virtio-scsi为例，看一下vhost是如何实现加速的。



CSDN @jiang4357291

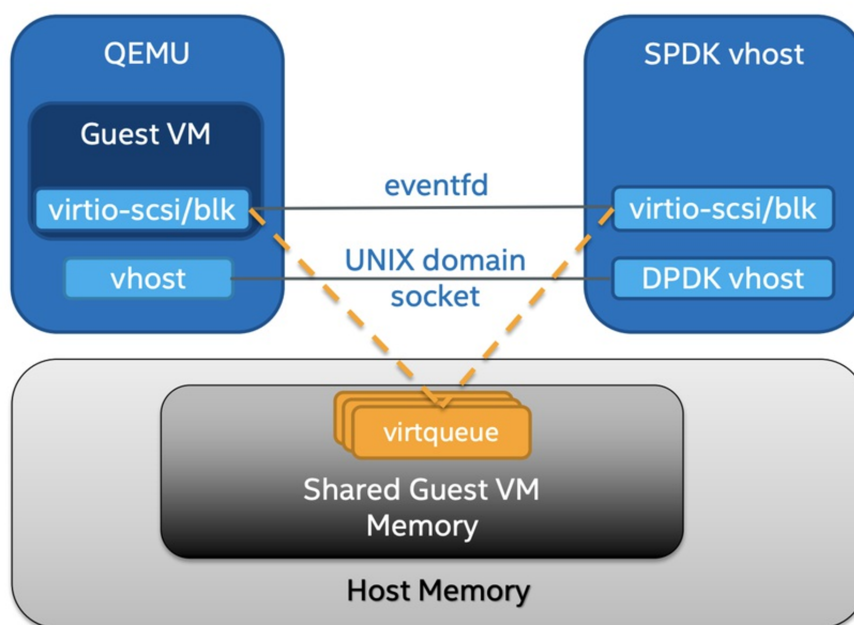
1. qemu virtio-scsi
   基于virtio的半虚拟化原始方案，guest和qemu之间通过virtqueue实现数据共享和传输，通过ioeventfd和irqfd实现通知。
   该方案的缺点前面已经介绍过：

- 每次io都需要双向通知
- io路径仍然过长，如果backend读写的是本地设备，还需要经历host上的完整io栈，需要从用户态拷贝到内核态

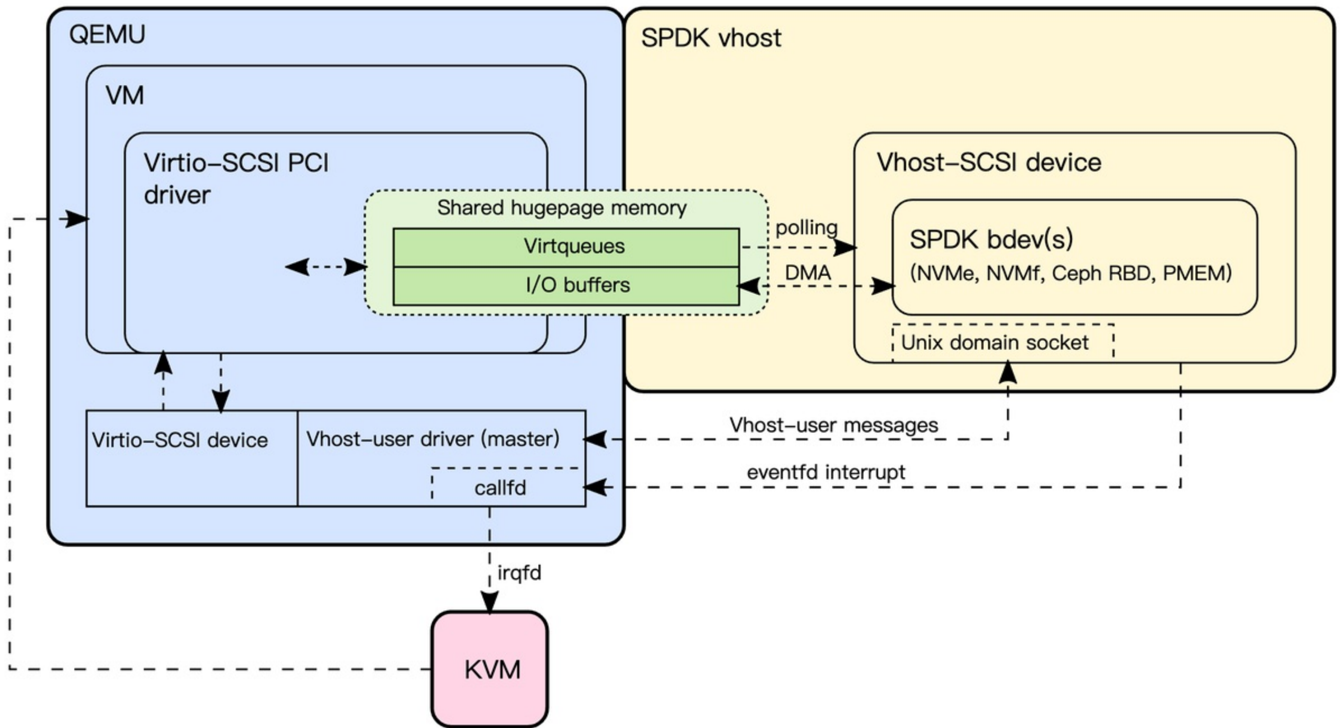
spdk vhost-kernel-scsi
qemu virtio-scsi方案的演进，块设备模拟仍然是由qemu来做，只是把virtio backend放到了host kernel中，由kernel去处理virtqueue。
host kernel要处理virtqueue需要知道地址，因此qemu会把virtqueue的内存信息和guest的GPA-HVA的映射告知内核vhost-scsi模块，host kernel直接接收virtqueue中的请求并下发到后端，缩短了io路径，省去了host上用户态到内核态的拷贝。
注：这种方案只有在本地nvme场景下才有优化，针对云盘的bedv做后端和virtio-scsi没有区别。

spdk vhost-user-scsi
vhost-kernel方案相较于virtio-scsi优化了host上的io，但是仍然存在通知的开销，guest需要通知qemu，qemu需要通知host kernel vhost-scsi，于是进一步演进出了vhost-user方案：

SPDK VHOST ARCHITECTURE



CSDN @jiang4357291



QEMU/SPDK vhost data flow

CSDN @jiang4357291

整体架构如上图所示，virtio backend仍然在host用户态，但是放到了qemu外部，vhost作为独立进程运行在host用户态，通过hugepage的共享内存和qemu共享vierqueue的地址空间，并通过轮询的方式不断从中取出io请求，再交由bdev进行处理。
这里同样再说一下本地nvme设备和Bytedrive bdev的区别：

- 本地nvme设备：vhost poll到请求后直接通过用户态的nvme驱动直接将io下发到硬盘了，无需再经过kernel nvme驱动，所以仍然缩短了io路径。且vhost会轮询nvme设备的queue pair，有io完成后也会立刻得知，相较于内核驱动的中断通知更为高效
- bytedrive bdev：vhost将请求交由bytedrive bdev，会调用bytedrive sdk从网络发出请求，这点上qemu+virtio也是一样的
  当io请求完成后，vhost将io response放到virtqueue中并通过写eventfd通知qemu，之后通过irqfd由kvm注入中断通知guest请求已完成。可以看到响应部分仍然存在中断通知，这部分和qemu-virtio是一样的，但是也可以通过一个poll-mode Virtio driver优化掉。
  补充一点，其中的unix socket连接是用于控制面消息传输的，如virtqueue共享内存的建立。

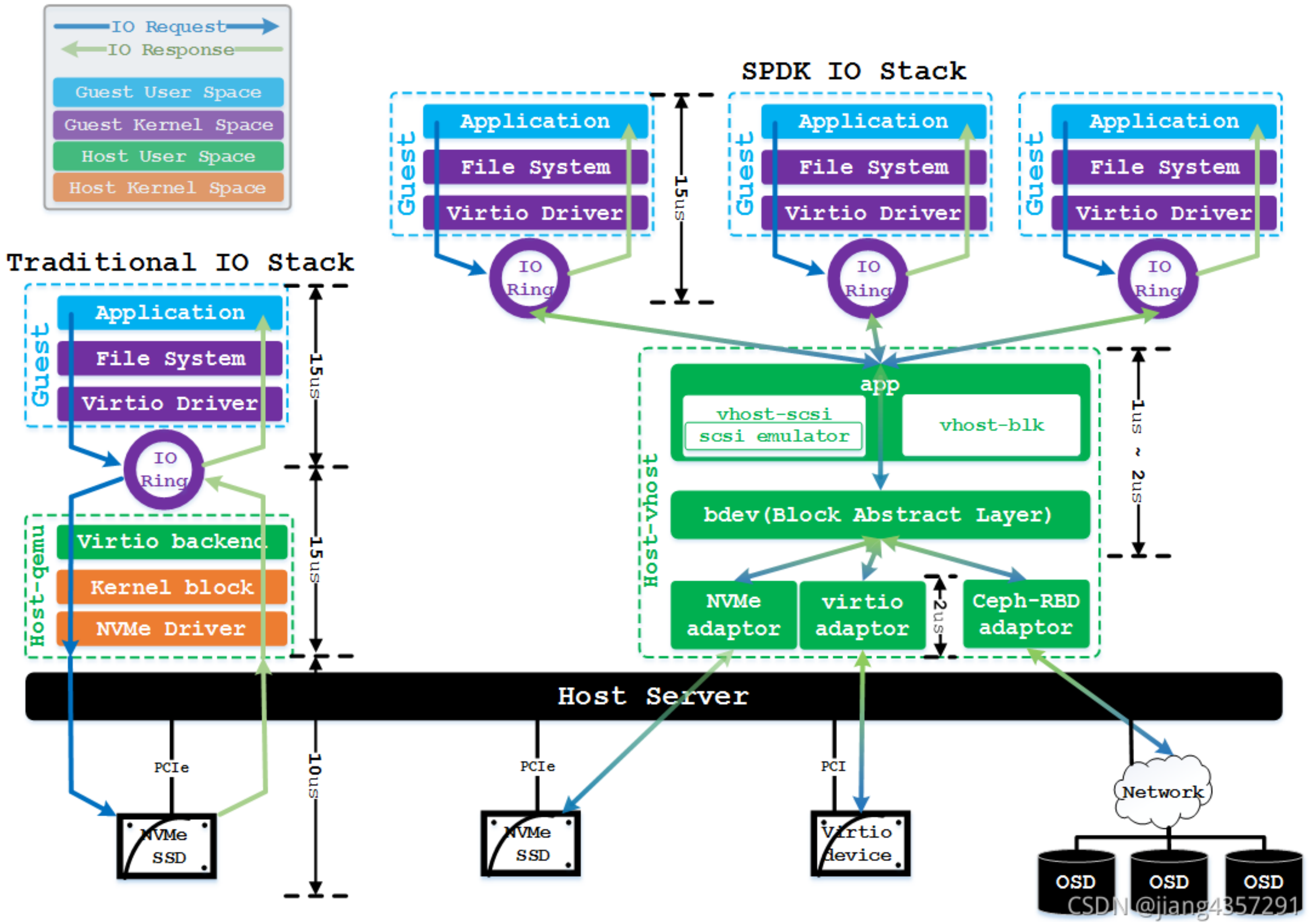
3.3.3 qemu-virtio vs vhost-user

两种方案下guest os内的io路径完全相同，从guest放到virtqueue中之后开始有区别：

```
 - qemu-virtio：通过ioeventfd通知qemu处理io，io在qemu内部io thread进行处理
 - vhost-user：vhost不断去poll virtqueue，省去了通知的开销
```

如果是写云盘，之后两者也没有区别，但是vhost-user在线程模型上更具优势；如果是写本地盘，vhost-user的另一个优势：

```
 - qemu-virtio：在qemu中写本地nvme盘，数据会拷贝到内核，再由内核的nvme驱动写盘，写完后由中断通知内核
 - vhost-user：通过vhost实现的高速nvme驱动，无需拷贝到内核，直接在用户态写盘，同时busy polling nvme盘的queue pair，io完成也不需要中断通知
```



四、参考

https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
https://cloud.tencent.com/developer/article/1052883
https://kernel.dk/blk-mq.pdf
https://cloud.tencent.com/developer/article/1425141
https://www.cnblogs.com/sammyliu/p/4543597.html
https://searchservervirtualization.techtarget.com/definition/hardware-assisted-virtualization
https://abelsu7.top/2019/09/02/virtio-in-kvm/
https://blog.linuxplumbersconf.org/2010/ocw/system/presentations/651/original/Optimizing_the_QEMU_Storage_Stack.pdf
https://www.static.linuxfound.org/jp_uploads/JLS2009/jls09_hellwig.pdf
http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html
https://www.cs.cmu.edu/~412/lectures/Virtio_2015-10-14.pdf
https://kernelgo.org/virtio-overview.html
https://www.ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
https://abelsu7.top/2019/07/07/kvm-memory-virtualization/
https://www.cnblogs.com/yi-mu-xi/p/12544695.html
https://mp.weixin.qq.com/s/wuQ8-pwqb9qXfOt4w3Zviw
https://zhuanlan.zhihu.com/p/68154666
https://www.linux-kvm.org/images/a/a7/02x04-MultithreadedDevices.pdf
http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html
https://www.cnblogs.com/qxxnxxFight/p/11050159.html
http://bos.itdks.com/506e078a39b84f8cb06300cff8e00bbc.pdf
https://rootw.github.io/2018/05/SPDK-ioanalyze/
https://rootw.github.io/2018/05/SPDK-iostack/
https://vmsplice.net/~stefan/VHPC%202021%20-%20Bring%20your%20own%20virtual%20devices.pdf