

# **DEVICE VIRTUALIZATION ON ARM**



# **VIRTUALIZATION DEVICE CLASSIFICATIONS**

设备虚拟化的分类

# CLASSIFICATIONS

一台虚拟机，它的虚拟设备可能有如下分类：

## 1. 硬件实现的虚拟化设备

- ARM: Generic Timer (CNTVCT\_ELO)
- ARM: GIC Virtual CPU interface (GICV)

## 2. 软件实现的全虚拟化设备

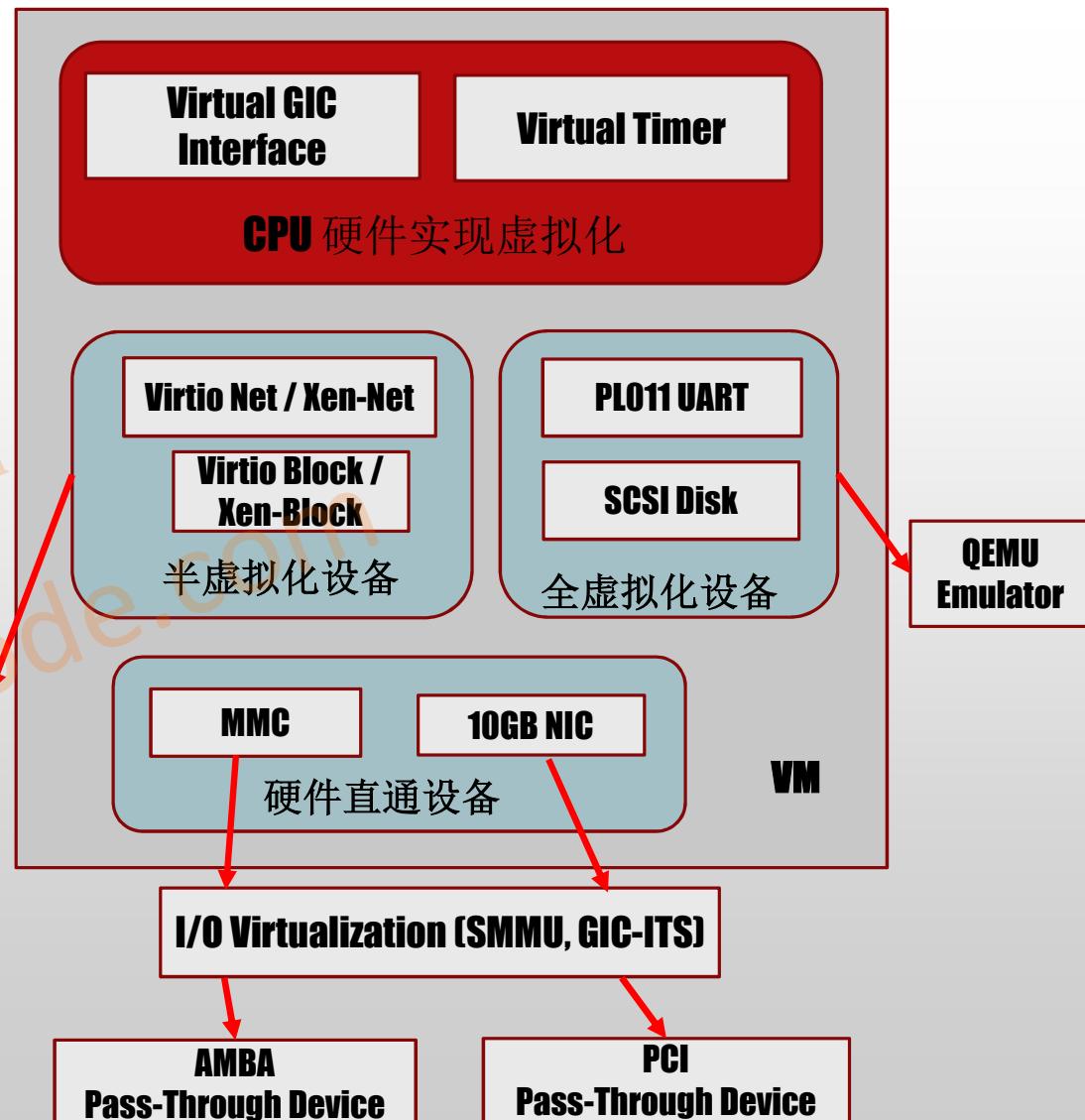
- Linux KVM : QEMU PL011串口设备 (ttyAMA0)
- Linux KVM : QEMU LSI SCSI 磁盘设备 (/dev/sda)

## 3. 软件实现的半虚拟化设备

- Linux KVM: Virtio Block 设备 (/dev/vda)
- Linux KVM: Virtio Net 设备
- XEN: xen-block 设备 (/dev/xvda)
- XEN: xen-net 设备

## 4. 基于硬件 I/O Virtualization 加速的硬件直通设备

- PCI Device Pass-through
- AMBA Device Pass-through



# HARDWARE VIRTUAL DEVICE

硬件实现的虚拟化设备

# HARDWARE VIRTUAL DEVICE

硬件虚拟设备的优势:

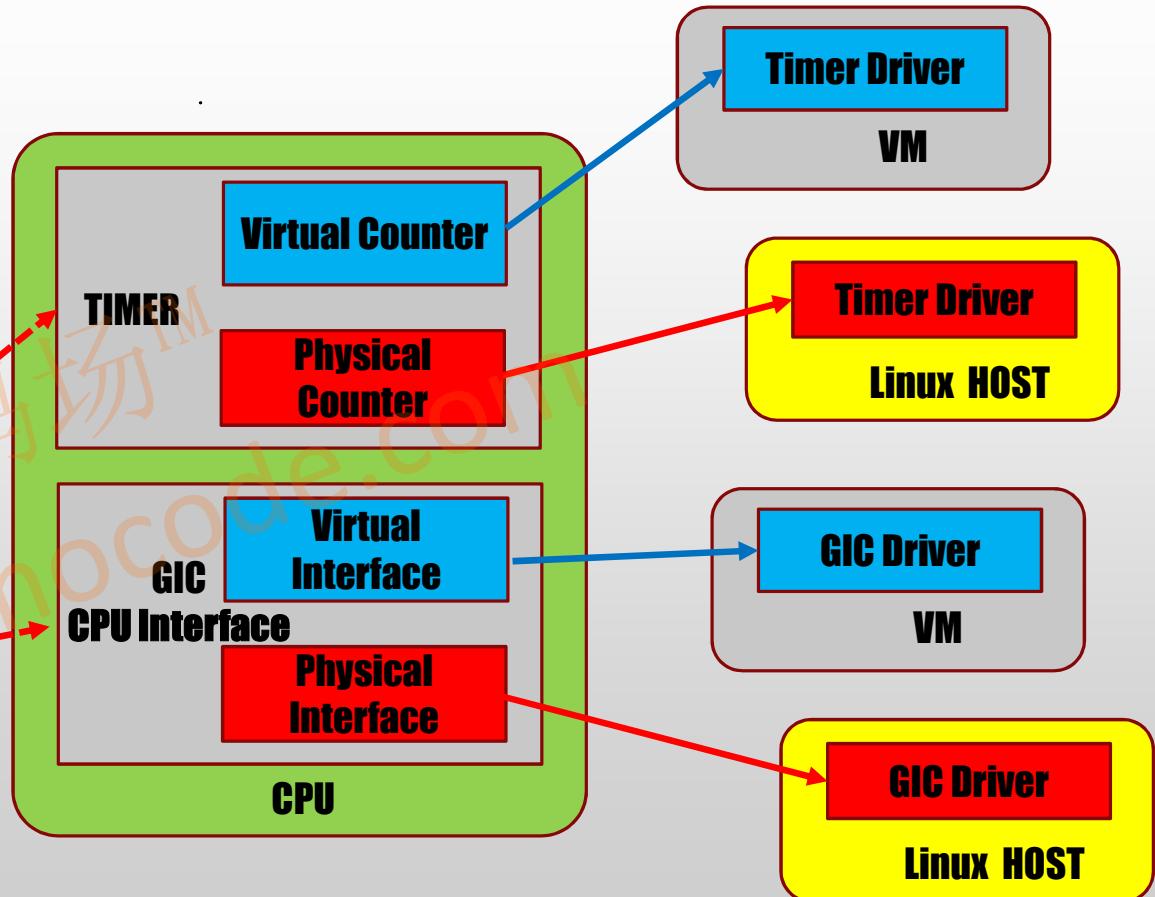
- 虚拟设备直接由**CPU IP**实现
- 虚拟机通过访问系统寄存器的方式访问虚拟设备
- 没有任何性能损失

硬件虚拟设备的缺点:

- 提高硬件的实现成本
- **CPU** 实现直接绑定, 配置难以修改

一般用在对系统性能有重大提升, 使用方式相对固定的场景, 如:

- 操作系统时钟子系统
- 操作系统的中断处理系统



# VIRTUAL TIMER

/linux/drivers/clocksource/arm\_arch\_timer.c

```
static void __init arch_counter_register(unsigned type)
{
    if (type & ARCH_TIMER_TYPE_CP15) {
        .....
        if ((IS_ENABLED(CONFIG_ARM64) && !is_hyp_mode_available()) ||
            arch_timer_uses_ppi == ARCH_TIMER_VIRT_PPI) {
            .....
            rd = arch_counter_get_cntvct;
        } else {
            .....
            rd = arch_counter_get_cntpct;
        }
    }
}
```

当前Linux是作为**HostOS**, 当前操作系统使用**physical timer**作为系统时钟:  
**arch\_counter\_get\_cntpct**

当前Linux是作为**GuestOS**, 操作系统使用**virtual timer**作为系统时钟  
**arch\_counter\_get\_cntvct**

```
... __arch_counter_get_cntvct(void)
{
    .....
    cnt = read_sysreg(cntvct_el0);
    .....
}
```

```
... __arch_counter_get_cntpct(void)
{
    .....
    cnt = read_sysreg(cntpct_el0);
    .....
}
```

# VIRTUAL TIMER

```
... __arch_counter_get_cntvct(void)
{
    .....
    cnt = read_sysreg(cntvct_el0);
    .....
}
```

```
void arch_timer_reg_write_cp15(...)
{
    if (access == ARCH_TIMER_PHYS_ACCESS) {
        .....
        write_sysreg(val, cntp_tval_el0);
        .....
    } else if (access == ARCH_TIMER_VIRT_ACCESS) {
        .....
        write_sysreg(val, cntv_tval_el0);
        .....
    }
}
```

GuestOS直接读取  
cntvct\_el0获取当前的  
系统时钟

GuestOS直接写入cntv\_tval\_el0  
设置时钟中断

整个过程不会陷入Hypervisor,  
Linux-KVM不参与时钟的模拟，由  
硬件将CNTVCT\_ELO的值设置为：  
 $CNTPCT + CNTVOFF\_EL2$

Linux-KVM只在虚拟切换时为不同的虚拟机保存和恢复virtual timer的上下文。  
 $CNTVOFF\_EL2 = 0$ , 表示虚拟机和HOST使用一样的时钟，  
 $CNTVOFF\_EL2 > 0$ , 则表示二者存在时间差

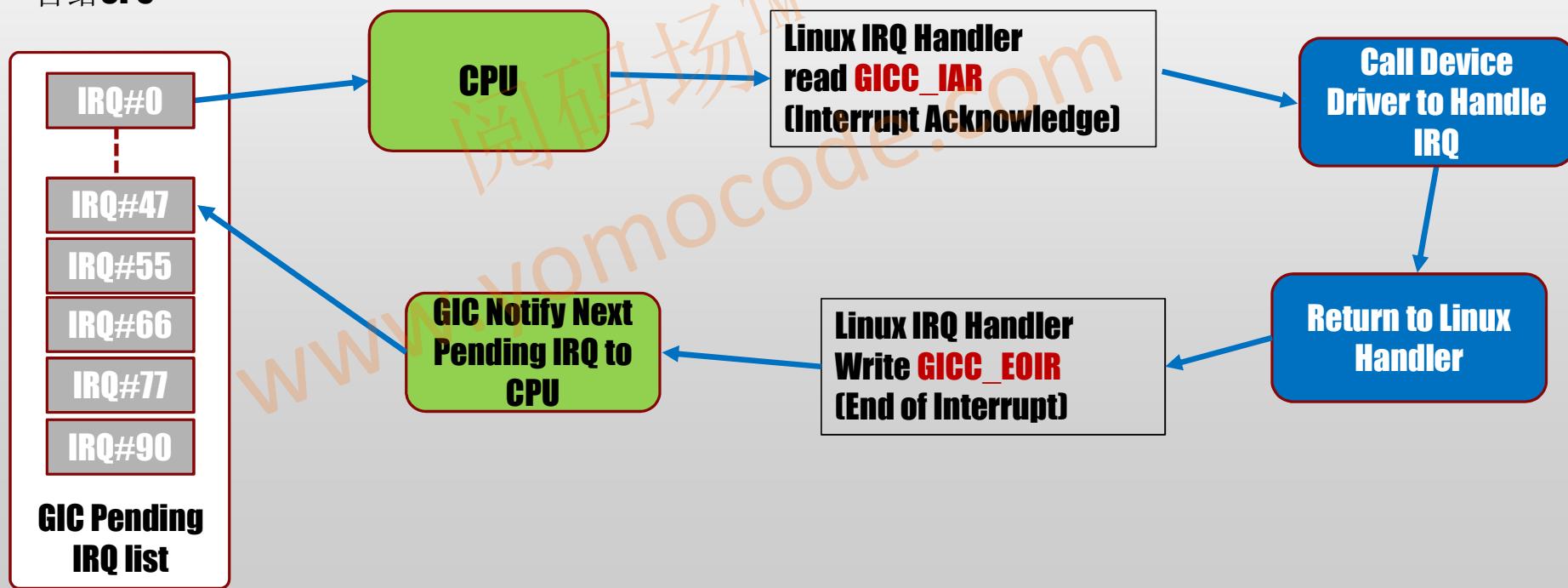
$$\text{CNTVCT\_ELO} = \text{CNTPCT\_ELO} + \text{CNTVOFF\_EL2}$$

硬件行为

# GIC VIRTUAL CPU INTERFACE

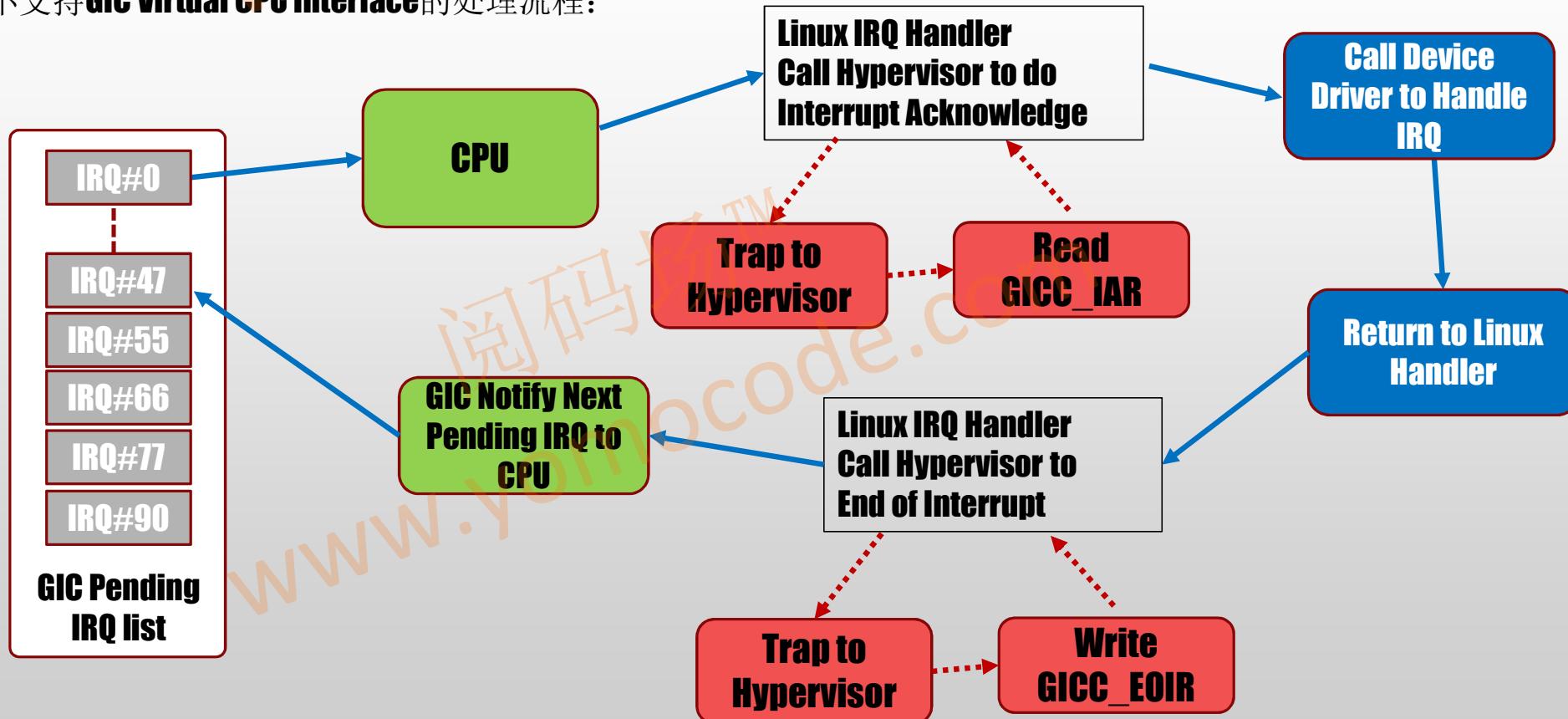
Linux 基于 ARM GIC 的中断处理：

- 当一个中断到达时，操作系统需要调用 GIC 中断控制驱动来响应该中断。**(Interrupt Acknowledge)**
- 当中断处理完成后，还要调用相应的接口来告诉 GIC 中断已经完成处理**(End of Interrupt)**这样 GIC 才会将后续中断继续报告给 CPU



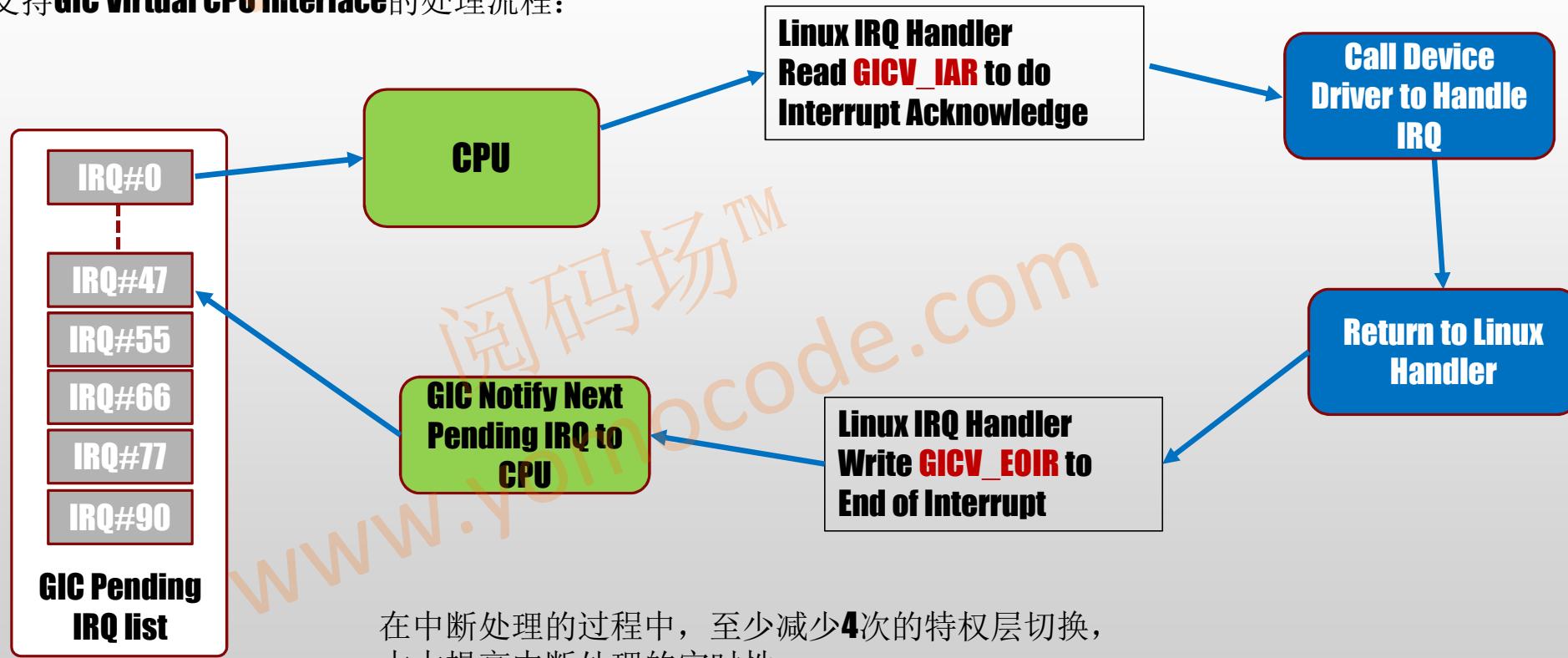
# GIC VIRTUAL CPU INTERFACE

不支持**GIC Virtual CPU Interface**的处理流程:



# GIC VIRTUAL CPU INTERFACE

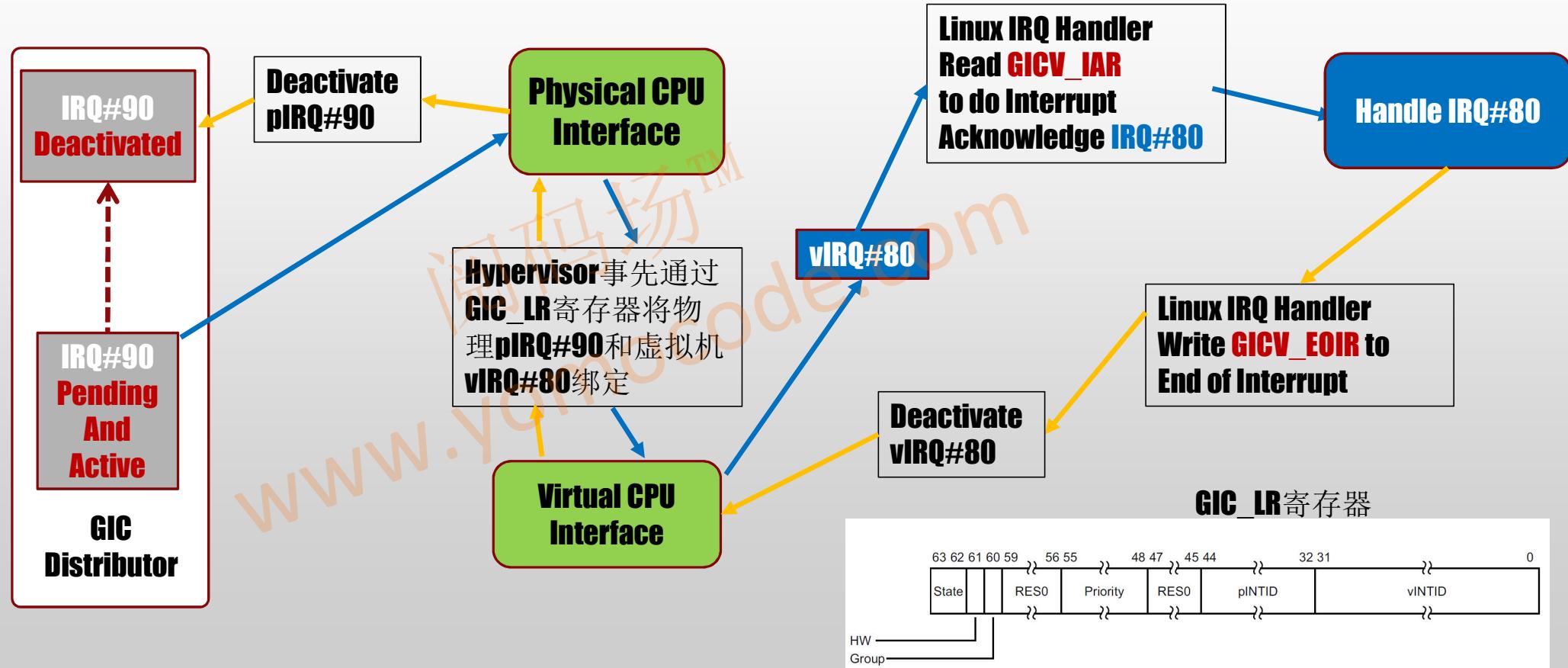
支持**GIC Virtual CPU Interface**的处理流程:



在中断处理的过程中，至少减少**4**次的特权层切换，  
大大提高中断处理的实时性。

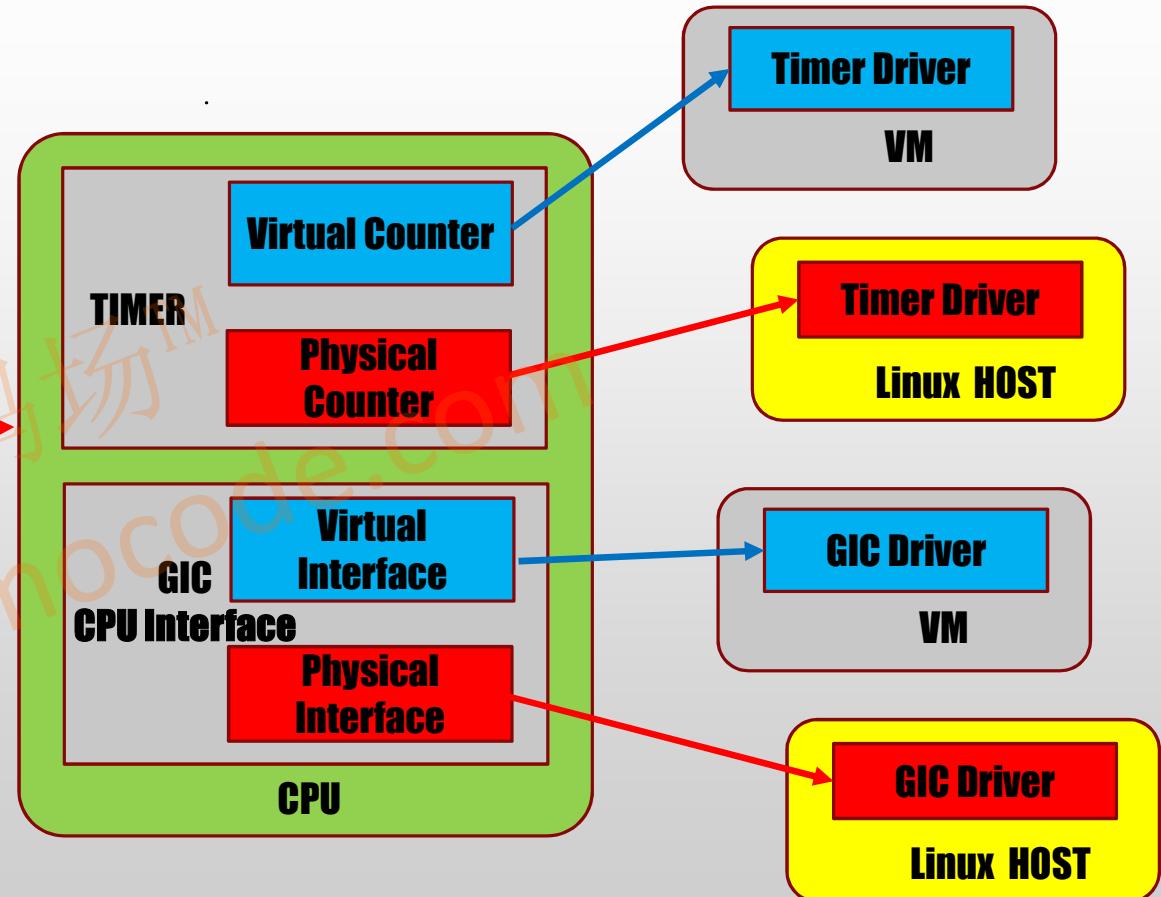
# GIC VIRTUAL CPU INTERFACE

GIC Virtual CPU interface 的硬件逻辑流程:



# HARDWARE VIRTUAL DEVICE

硬件实现，  
性能最好，  
代价最高



# FULL VIRTUALIZATION DEVICE

软件实现的全虚拟化设备

# FULL VIRTUALIZATION DEVICE

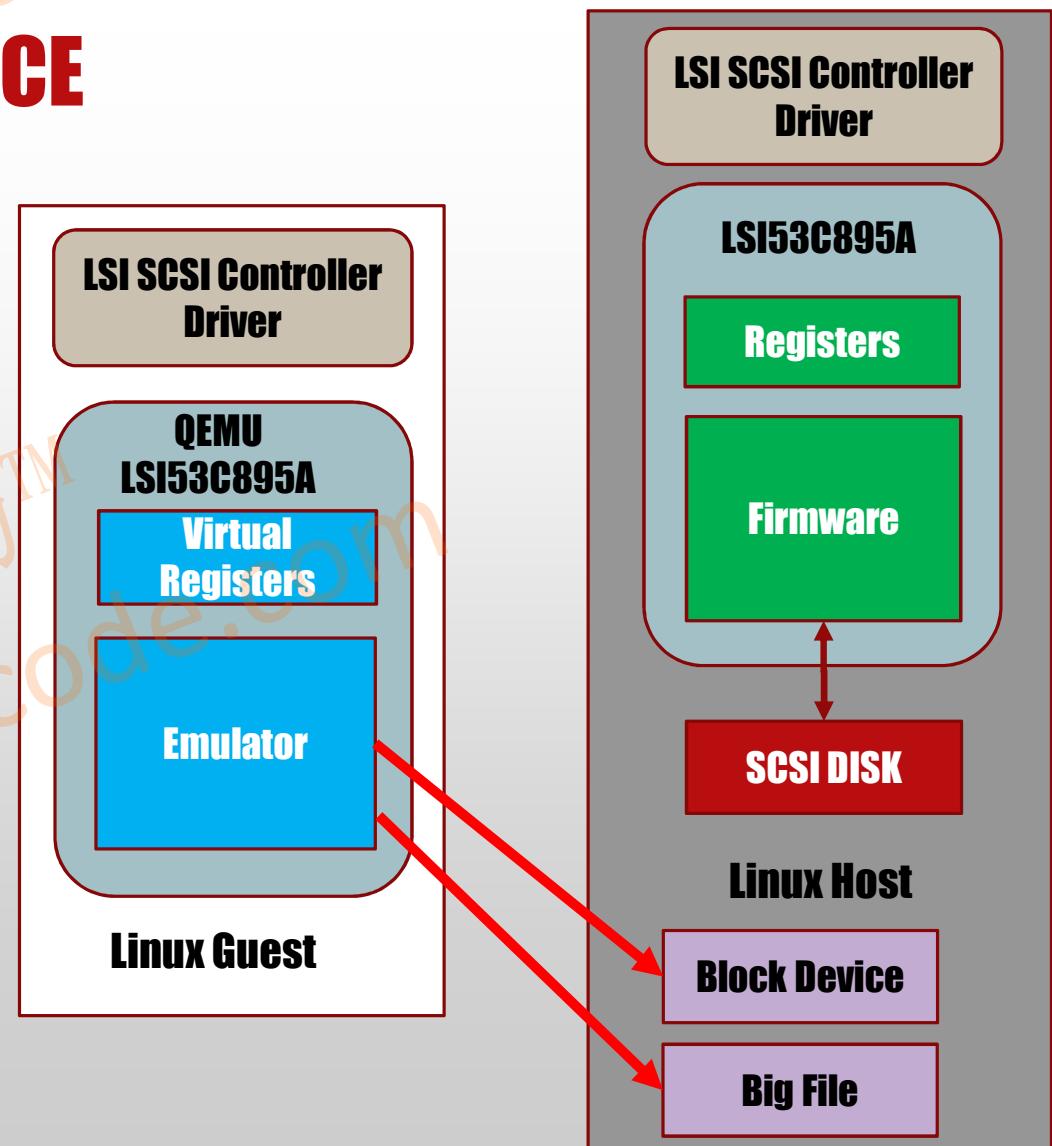
## 全虚拟化设备：

- 完全按照硬件的设计进行设备仿真，对驱动程序提供和真实硬件一样的寄存器接口
- 操作系统不需要修改或者新增驱动程序就可以直接使用。

**Linux-KVM**的全虚拟化设备有用户态模拟器实现。如果我们是**Linux-KVM+QEMU**就是由**QEMU**来实现。当然根据大家的运用场景**QEMU**也可以被**kvmttools, NEMU, Firecracker**等替代。

以**LSI53C895A LSI SCSI HOST controller**为例：

- 1、虚拟机可以完全无感知的使用为真实硬件准备**LSI**驱动程序来驱动**QEMU**提供的**LSI**全虚拟化设备。
- 2、**LSI**虚拟化设备可以从**Linux HOST**上的块设备获取数据，或者利用大文件模拟磁盘



# FULL VIRTUALIZATION DEVICE

定义和真实设备一致的接口：

- 设备的寄存器是设备和驱动程序交互的接口，要兼容真实设备的驱动程序就必须定义和真实设备一致的虚拟寄存器

左图为真实**LSI**设备的寄存器表，右图是**QEMU**用数据结构定义软件寄存器。他们都包含：

**0~0x40**的寄存器空间和**8KBytes**的脚本执行内存

Table 4.2 SCSI Register Address Map

31	16 15	0
SCNTL3	SCNTL2	SCNTL1
GPREG0	SDID	SXFER
SBCL	SSID	SOCL
SSTAT2	SSTAT1	SSTAT0
	DSA	
MBOX1	MBOX0	ISTAT1
CTEST3	CTEST2	CTEST1
	TEMP	
CTEST6	CTEST5	CTEST4
DCMD		DFIFO
	DBC	
	DNAD	
	DSP	
	DSPS	
	SCRATCH A	
DCNTL	SBR	DIEN
	ADDER	DMODE
		0x3C

8KBytes Script RAM

```
typedef struct {  
    .....  
    uint8_t scntl0;  
    uint8_t scntl1;  
    uint8_t scntl2;  
    uint8_t scntl3;  
    .....  
    uint8_t script_ram[8196];  
} LSIState;
```

# FULL VIRTUALIZATION DEVICE

定义和真实设备一致的操作行为：

- 全虚拟化设备的模拟器是一个黑盒
- 驱动程序访问虚拟寄存器得到的结果和驱动访问真实设备的寄存器得到结果要一致

模拟设备寄存器的读行为

```
static uint8_t lsi_reg_readb(LSIState *s, int offset)
{
    .....
    case 0x00: /* SCNTL0 */
        ret = s->scntl0;
        break;
    case 0x01: /* SCNTL1 */
        ret = s->scntl1;
        break;
    case 0x02: /* SCNTL2 */
        ret = s->scntl2;
        break;
    case 0x03: /* SCNTL3 */
        ret = s->scntl3;
        break;
    case 0x04: /* SCID */
        ret = s->scid;
        break;
    .....
}
```

**0x4**是真实硬件的**SCID**寄存器，虚拟设备返回虚拟的**SCID**

往**0x1**写入配置，要求进行复位操作，虚拟设备调用**QEMU**接口进行虚拟设备复位

模拟设备寄存器的写行为

```
static void lsi_reg_writeb(LSIState *s, int offset, uint8_t val)
{
    .....
    switch (offset) {
        .....
        case 0x01: /* SCNTL1 */
            s->scntl1 = val & ~LSI_SCNTL1_SST;
            .....
            if (val & LSI_SCNTL1_RST) {
                if (!(s->sstat0 & LSI_SSTAT0_RST)) {
                    qbus_reset_all(BUS(&s->bus));
                    s->sstat0 |= LSI_SSTAT0_RST;
                    lsi_script_scsi_interrupt(s, LSI_SIST0_RST, 0);
                }
            } else {
                s->sstat0 &= ~LSI_SSTAT0_RST;
            }
            break;
        .....
    }
}
```

# FULL VIRTUALIZATION DEVICE

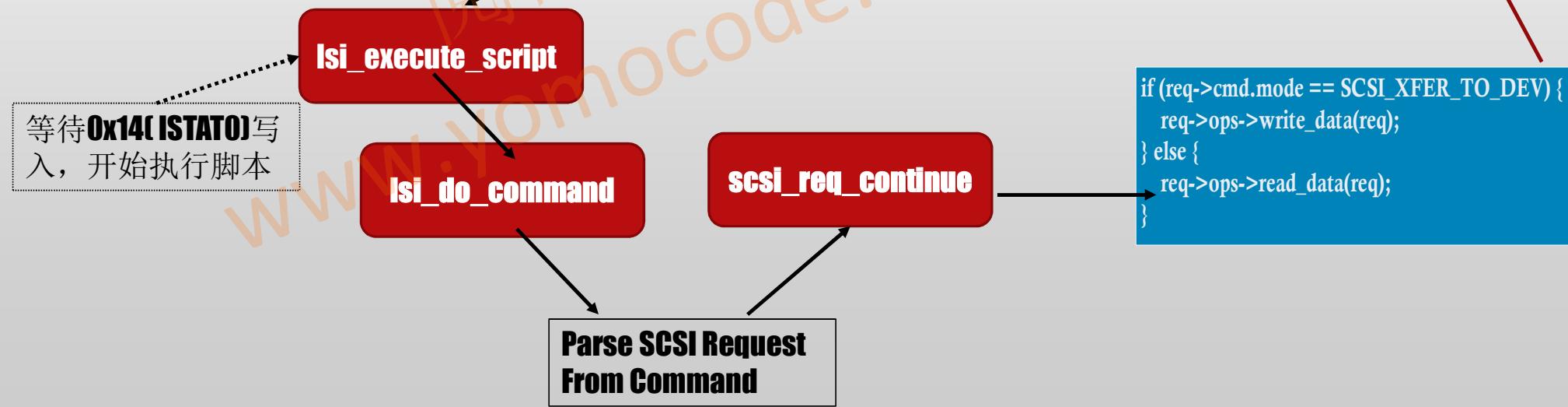
定义和真实设备一致的操作行为:

- 写入读写命令
- 写入访问的磁盘**ID**
- 写入访问的数据块**offset**
- 写入访问的数据块大小
- 写入**DMA**内存的地址
- 写入开始执行的命令
- 等待执行
- 从寄存器读取命令执行的情况

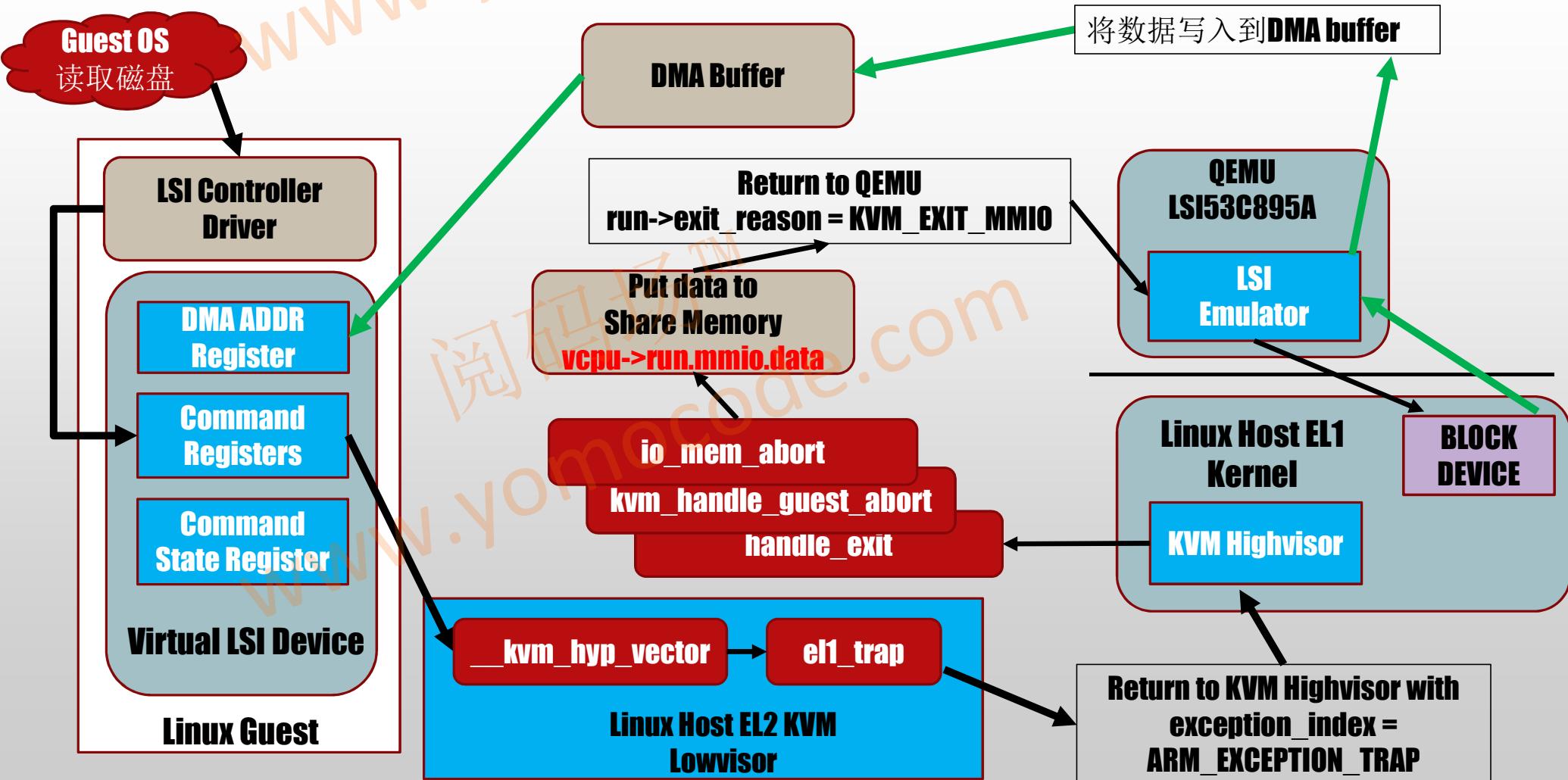


虚拟设备行为:

- 在虚拟寄存器中保存命令
- 在虚拟寄存器中保存磁盘**ID**
- 在虚拟寄存器中保存数据块**offset**
- 在虚拟寄存器中保存数据块大小
- 在虚拟寄存器中保存**DMA**内存的地址
- 从寄存器合成**SCSI**命令，调用回调函数读取数据
- 将数据填充到**DMA**内存
- 设置状态寄存器，注入中断

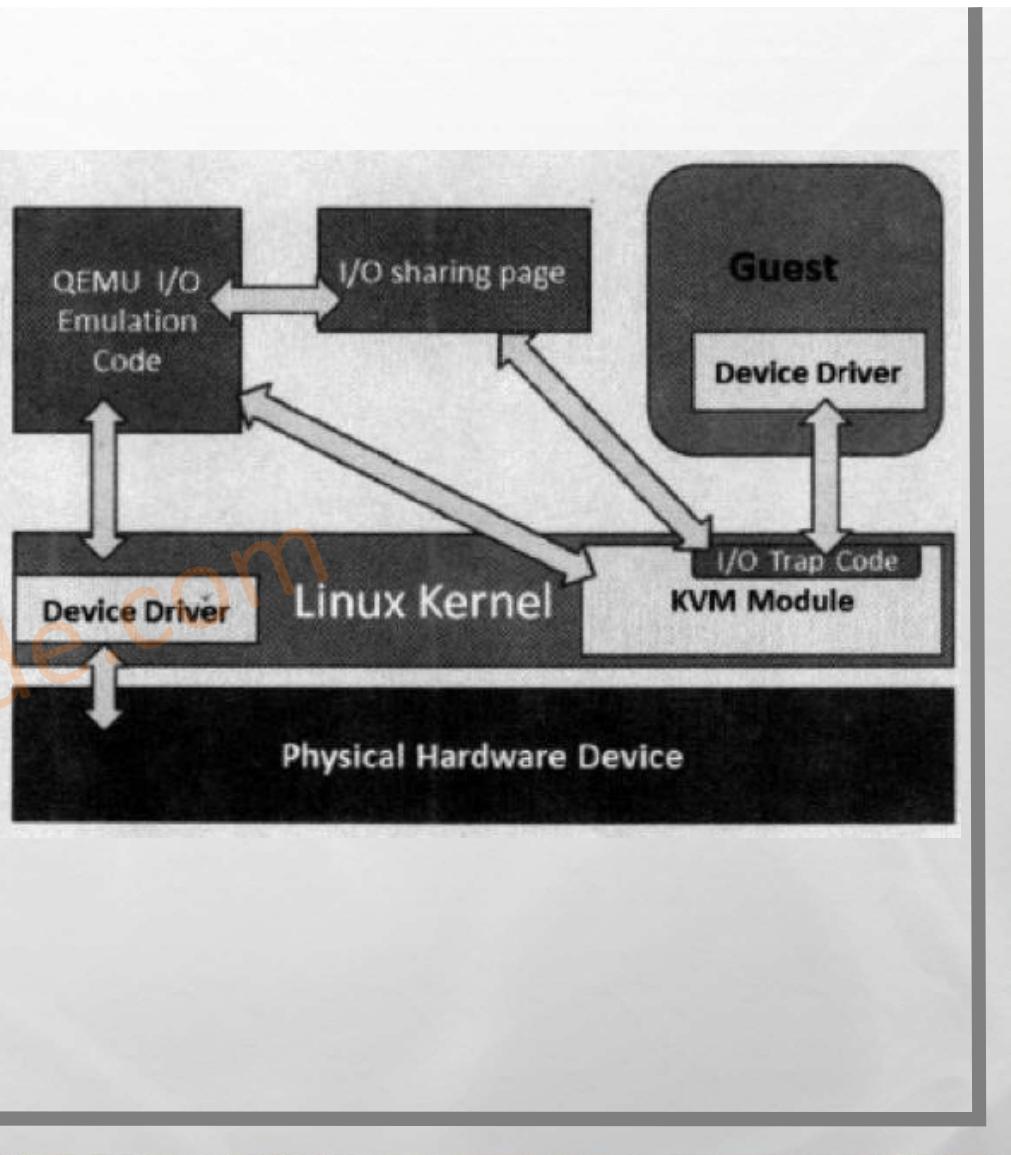


# I/O PROGRESS IN LINUX-KVM+QEMU



## 简单描述：

- **GUEST OS**的设备驱动程序配置寄存器发起**I/O**请求操作请求，
- 写入虚拟寄存器产生**STAGE 2**地址转换异常，
- **KVM EL2 HIGHVISOR** 异常向量表截获异常，返回**KVM EL1 LOWVISOR**，
- **LOWVISOR** 发现时**GUEST**的**I/O**操作，
- 将**I/O**操作的信息放到和**QEMU**的共享内存：
- **VCPU->RUN.MMIO.DATA**，并通知**QEMU**，
- **QEMU**程序获得**I/O**操作的具体信息之后，交由**LSI**模拟代码来完成本次**I/O**操作，
- **LSI**模拟代码将读取的数据放入**DMA**寄存器描述的内存空间内，并修改虚拟**STATE REGISTER**寄存器的状态，
- **QEMU**返回**KVM EL1 LOWVISOR**，**KVM EL1 LOWVISOR**注入中断通知**GUEST**，该**I/O**操作完成。
- **GUEST LSI**驱动读取**STATE REGISTER**寄存器的状态，并从**DMA BUFFER**取得数据。



# **GUEST ENTER/EXIT OF LINUX-KVM + QEMU**

**LINUX KVM + QEMU** 中虚拟机的进入和退出

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
【**kvm\_run**数据结构体的大小】

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行(**KVM\_RUN**)，进去**GUEST OS**

```
static int kvm_init(MachineState *ms)
```

**QEMU**示例

```
{  
    .....  
    ret = kvm_ioctl(s, KVM_GET_API_VERSION, 0);  
    s->nr_slots = kvm_check_extension(s, KVM_CAP_NR_MEMSLOTS);  
    .....  
    do {  
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);  
    } while (ret == -EINTR);  
}
```

检查**hypervisor**支持多少内存分段，是否支持**IRQFD**，是否支持**MSI**，是否支持中断注入等特性。

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行(**KVM\_RUN**)，进去**GUEST OS**

```
int kvm_init_vcpu(CPUState *cpu)
{
    .....
    mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);
    .....
    cpu->kvm_run = mmap(..., mmap_size, ..., MAP_SHARED, ...);
    .....
}
```

**QEMU示例**

该结构体用于内核中**KVM EL1 Lowvisor**和**QEMU**共享**VCPU**的状态信息：比如**I/O**操作信息存放在**VCPU->RUN.MMIO.DATA**

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器，配置中断
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行(**KVM\_RUN**)，进去**GUEST OS**

```
int kvm_init_vcpu(CPUState *cpu)
{
    ret = kvm_get_vcpu(s, kvm_arch_vcpu_id(cpu)); QEMU示例
    cpu->kvm_fd = ret;
    .....
    mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);
    .....
    cpu->kvm_run = mmap(..., mmap_size, ..., MAP_SHARED, ...);
    .....
}
```

```
static int kvm_get_vcpu(KVMState *s, unsigned long vcpu_id)
{
    .....
    return kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)vcpu_id); QEMU示例
}
```

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行(**KVM\_RUN**)，进去**GUEST OS**

```
static int kvm_init(MachineState *ms)
{
    .....
    do {
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);
    } while (ret == -EINTR);
    .....
    if (machine_kernel_irqchip_allowed(ms)) {
        kvm_irqchip_create(ms, s);
    }
    .....
}
```

```
static void kvm_irqchip_create(.....)
{
    .....
    kvm_check_extension(s, KVM_CAP_IRQCHIP);
    .....
    kvm_vm_ioctl(s, KVM_CREATE_IRQCHIP);
    .....
}
```

ARM中**vGIC**的实现是在内核，属于**KVM EL1 Lowvisor**的一部分

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3个等级的IOCTL**

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
【**kvm\_run**数据结构体的大小】

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器
- **给虚拟机分配内存**
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行【**KVM\_RUN**】，进去**GUEST OS**

```
static int kvm_init(MachineState *ms)
{
    .....
    do {
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);
    } while (ret == -EINTR);
    .....
    kvm_memory_listener_register(s, &s->memory_listener,
                                &address_space_memory, 0);
    .....
}
```

```
void kvm_memory_listener_register(.....)
{
    .....
    kml->listener.region_add = kvm_region_add;
    kml->listener.region_del = kvm_region_del;
    .....
}
```

**kvm\_set\_user\_memory\_region**

**kvm\_vm\_ioctl {s, KVM\_SET\_USER\_MEMORY\_REGION, &mem}**

在虚拟机地址空间上加入地址段映射或者删除映射都会调用该接口【虚拟设备添加删除】

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器，配置中断
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行(**KVM\_RUN**)，进去**GUEST OS**

```
int kvm_create_device(KVMState *s, uint64_t type, ...)  
{  
    .....  
    kvm_vm_ioctl(s, KVM_CREATE_DEVICE, ...);  
    .....  
}  
  
int kvm_arm_vgic_probe(void)  
{  
    .....  
    if (kvm_create_device(kvm_state,  
        KVM_DEV_TYPE_ARM_VGIC_V3, true) == 0) {  
        return 3;  
    }  
    .....  
}
```

在内核创建**KVM\_DEVICE**，比如**VGIC**。  
**LSI**全虚拟化设备，属于**QEMU**设备，调用**qdev**接口在用户态创建即可。

在虚拟机地址段 **0x101F1000** 创建一个  
**PL011** 虚拟设备

```
pl011_create(0x101f1000, pic[12], serial_hd(0));
pl011_create(0x101f2000, pic[13], serial_hd(1));
pl011_create(0x101f3000, pic[14], serial_hd(2));
pl011_create(0x10009000, sic[6], serial_hd(3));
```

```
... pl011_create(hwaddr addr, qemu_irq irq, ...)
{
    DeviceState *dev;
    SysBusDevice *s;

    dev = qdev_create(NULL, "pl011");
    s = SYS_BUS_DEVICE(dev);
    qdev_prop_set_chr(dev, "chardev", chr);
    qdev_init_nofail(dev);
    sysbus_mmio_map(s, 0, addr); ——————→ 向虚拟机地址空间注册该地
    sysbus_connect_irq(s, 0, irq);

    return dev;
}
```

```
static const MemoryRegionOps pl011_ops = {
    .read = pl011_read,
    .write = pl011_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};
```

↑  
创建一个类型为 “**PL011**” 的  
**QEMU** 设备，该设备对应的虚  
拟寄存器读写回调函数

向虚拟机地址空间注册该地  
址段

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器，配置中断
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- **初始化VCPUs状态**
- 配置**VCPUs**特定的寄存器
- 让**VCPUs**运行(**KVM\_RUN**)，进去**GUEST OS**

```
int kvm_reset_vcpu(struct kvm_vcpu *vcpu)
{
    .....
    cpu_reset = &default_regs_reset;           内核示例

    /* Reset core registers */
    memcpy(vcpu_gp_regs(vcpu), cpu_reset, sizeof(*cpu_reset));

    /* Reset system registers */
    kvm_reset_sys_regs(vcpu);
    .....
}
```

初始化通用寄存器和系统寄存器

```
int kvm_arm_vcpu_init(CPUState *cs)           QEMU示例
{
    .....
    return kvm_vcpu_ioctl(cs, KVM_ARM_VCPU_INIT, &init);
}
```

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPUs**
- 在虚拟机中创建中断控制器，配置中断
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPUs**

- 初始化**VCPUs**状态
- 配置**VCPUs**特定的寄存器
- 让**VCPUs**运行(**KVM\_RUN**)，进入**GUEST OS**

```
reg.id = AARCH64_CORE_REG(elr_el1);  
reg.addr = (uintptr_t) &env->elr_el1;    QEMU示例  
ret = kvm_vcpu_ioctl(cs, KVM_SET_ONE_REG, &reg);  
if (ret) {  
    return ret;  
}
```

如果我们像修改虚拟机运行的入口地址，我们可以单独使用**KVM\_SET\_ONE\_REG**修改**VCPUs**的**elr\_el1**虚拟寄存器，让**VCPUs**返回时从我们指定的地址执行。

# KVM USERSPACE APIs

根据操作的对象，**KVM**提供了**3**个等级的**IOCTL**接口：

➤ **KVM\_IOCTL** → 操作对象为**hypervisor**:

- 请求**hypervisor**的版本信息和特性
- 请求**hypervisor**创建虚拟机
- 获取**hypervisor**中和**QEMU**共享页面的大小  
[**kvm\_run**数据结构体的大小]

➤ **KVM\_VM\_IOCTL** → 操作对象为虚拟机

- 给虚拟机中创建**VCPU**
- 在虚拟机中创建中断控制器，配置中断
- 给虚拟机分配内存
- 给虚拟机中创建虚拟设备

➤ **KVM\_VCPU\_IOCTL** → 操作对象为**VCPU**

- 初始化**VCPU**状态
- 配置**VCPU**特定的寄存器
- 让**VCPU**运行**[KVM\_RUN]**，进去**GUEST OS**

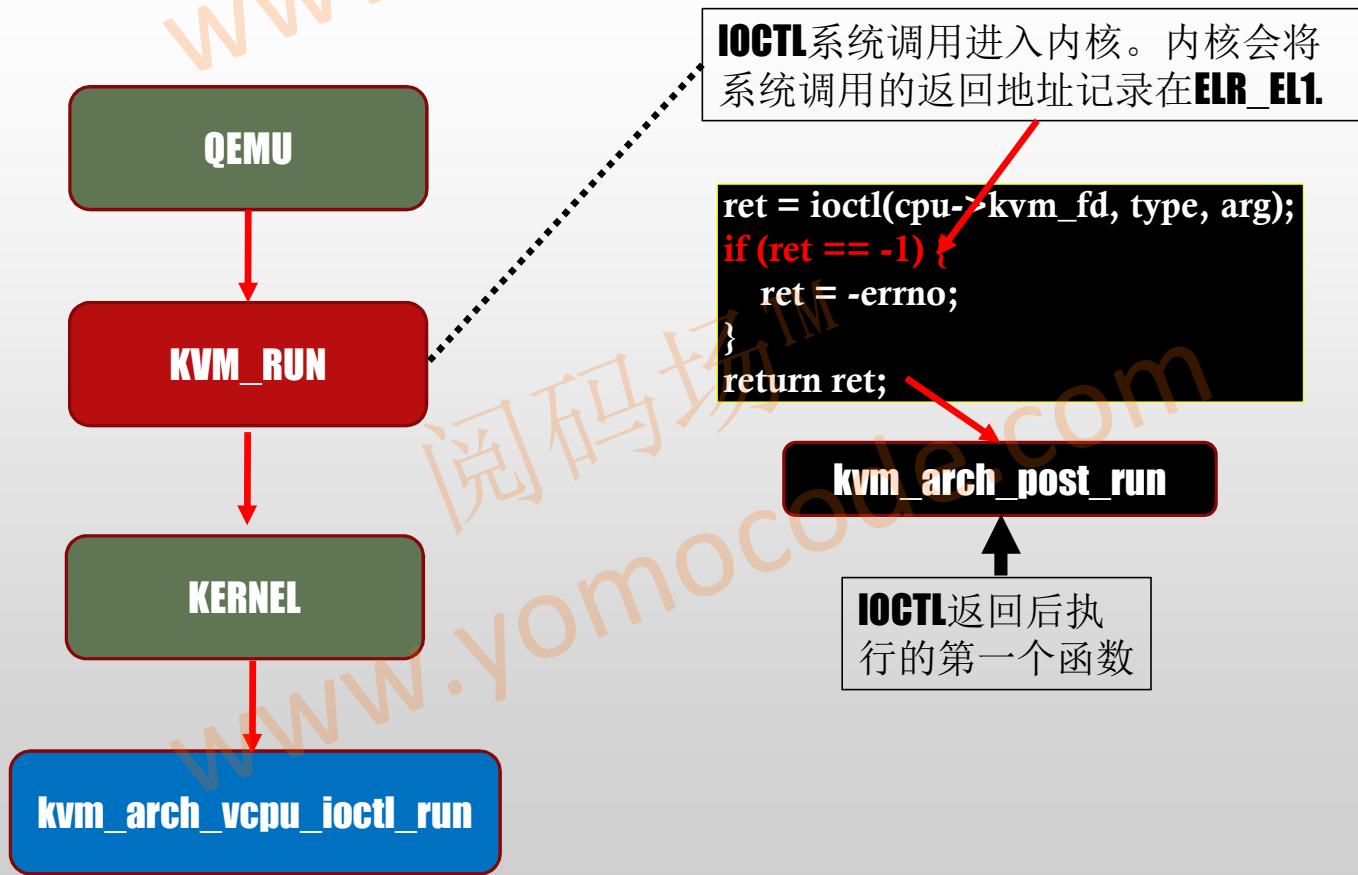
```
int kvm_cpu_exec(CPUState *cpu)
```

QEMU示例

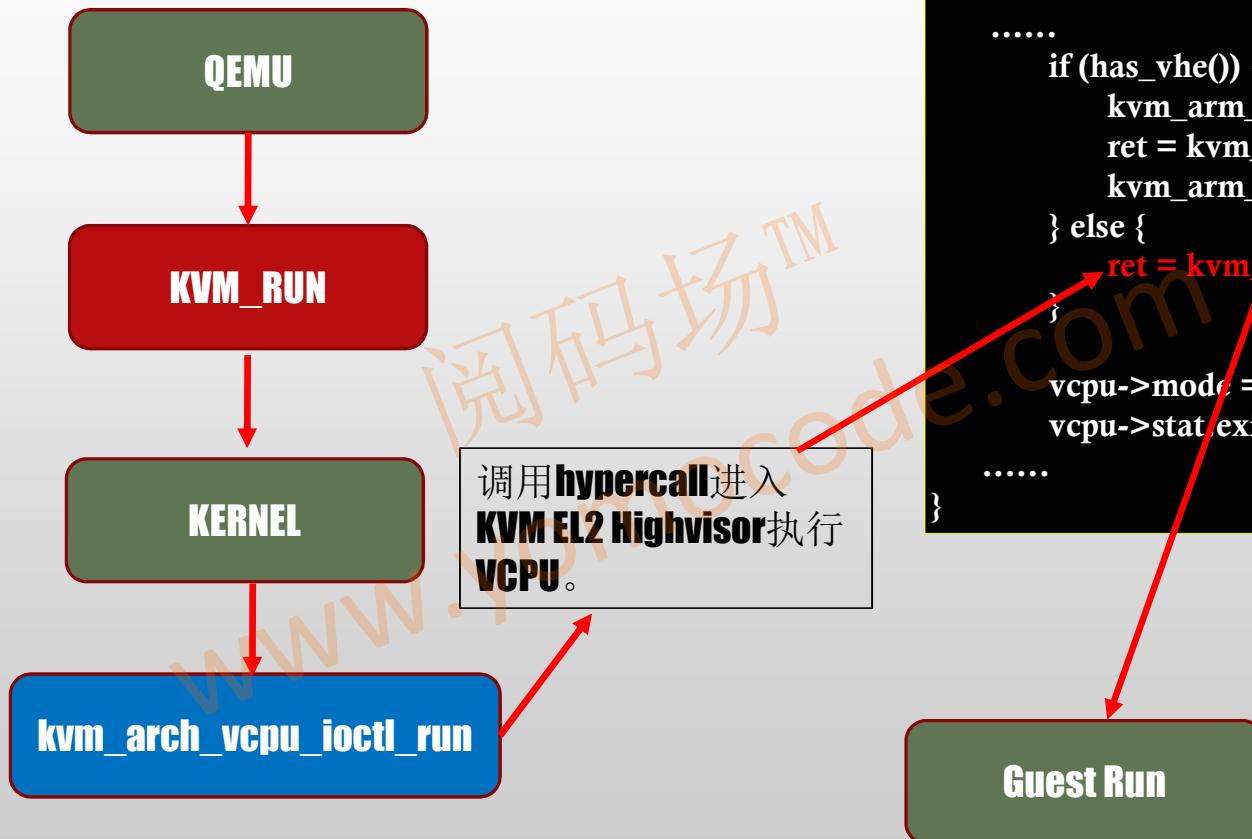
```
{  
    .....  
    do {  
        .....  
        run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);  
  
        attrs = kvm_arch_post_run(cpu, run);  
        switch (run->exit_reason) {  
            .....  
            case KVM_EXIT_MMIO:  
                DPRINTF("handle_mmio\n");  
                /* Called outside BQL */  
                address_space_rw(&address_space_memory,  
                    run->mmio.phys_addr, attrs,  
                    run->mmio.data,  
                    run->mmio.len,  
                    run->mmio.is_write);  
            .....  
        } while (ret == 0);  
    .....  
}
```

虚拟机退出时的返回地址。判断返回原因，在我们这个例子里就是**LSI I/O**操作**KVM\_EXIT\_MMIO**，然后调用相应的回调函数处理。

# CONTEXT SWITCHES



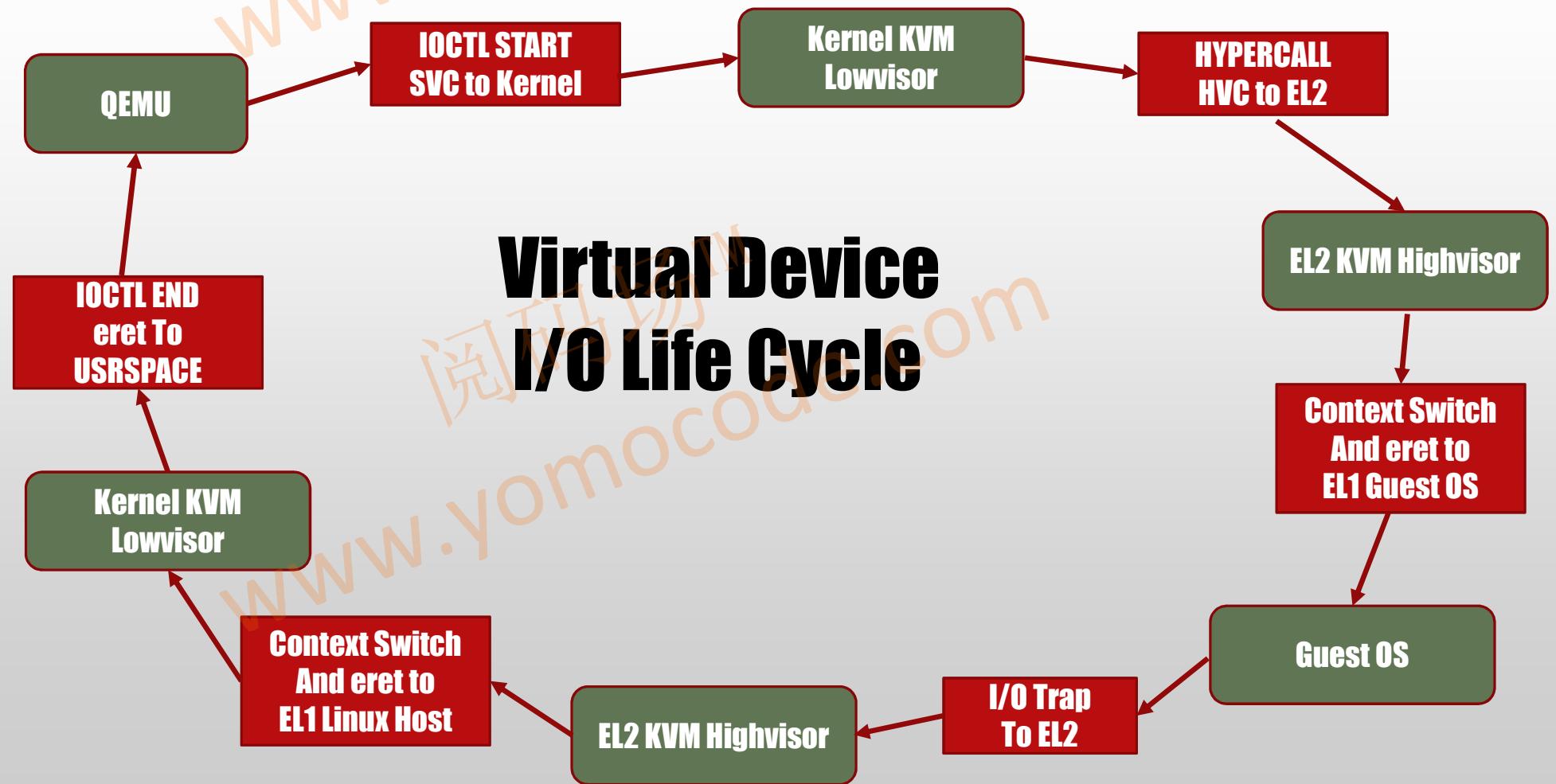
# CONTEXT SWITCHES



```
int kvm_arch_vcpu_ioctl_run(...)  
{  
    .....  
    while (ret > 0) {  
        .....  
        if (has_vhe()) {  
            kvm_arm_vhe_guest_enter();  
            ret = kvm_vcpu_run_vhe(vcpu);  
            kvm_arm_vhe_guest_exit();  
        } else {  
            ret = kvm_call_hyp_ret(__kvm_vcpu_run_nvhe, vcpu);  
        }  
        vcpu->mode = OUTSIDE_GUEST_MODE;  
        vcpu->stat.exits++;  
    }  
    .....  
}
```

返回地址保存在 **ELR\_EL2**

# CONTEXT SWITCHES



# FULL VIRTUALIZATION DEVICE

定义和真实设备一致的操作行为:

- 写入读写命令
- 写入访问的磁盘**ID**
- 写入访问的数据块**offset**
- 写入访问的数据块大小
- 写入**DMA**内存的地址
- **写入开始执行的命令**
- 等待执行
- 从寄存器读取命令执行的情况



虚拟设备行为:

- 在虚拟寄存器中保存命令
- 在虚拟寄存器中保存磁盘**ID**
- 在虚拟寄存器中保存数据块**offset**
- 在虚拟寄存器中保存数据块大小
- 在虚拟寄存器中保存**DMA**内存的地址
- **从寄存器合成**SCSI**命令，调用回调函数读取数据**
- 将数据填充到**DMA**内存
- 设置状态寄存器，注入中断

全虚拟化的劣势:

每个虚拟寄存器的操作都要涉及大量的特权层切换，系统开销巨大。

# **PARAVIRTUALIZATION DEVICE**

半虚拟化设备

# PARA-VIRTUALIZATION DEVICE

## 什么是半虚拟化设备？

- 半虚拟化设备它还是一种软件虚拟化设备
- 它通过在**GuestOS**和**Host**之间设计的半虚拟化接口来简化**I/O**操作，减少**I/O TRAP**，提高效率。
- **Virtio/XEN-PV Device**

## 全虚拟设备行为：

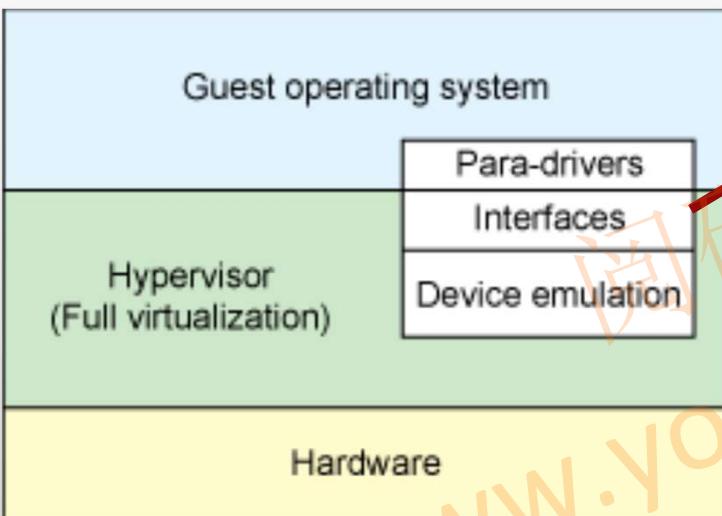
- 在虚拟寄存器中保存命令
- 在虚拟寄存器中保存磁盘**ID**
- 在虚拟寄存器中保存数据块**offset**
- 在虚拟寄存器中保存数据块大小
- 在虚拟寄存器中保存**DMA**内存的地址
- 从寄存器合成**SCSI**命令，调用回调函数读取数据
- 将数据填充到**DMA**内存
- 设置状态寄存器，注入中断

简化

## 半虚拟设备行为：

- 将**IO Request**信息放入共享内存**[IO类型，磁盘偏移，数据块大小，数据存放地址]**
- 调用半虚拟化接口通知**HOST**从共享内存读取**IO Request**
- **HOST**处理**IO Request**，调用半虚拟化接口通知**Guest OS**

# VIRTIO DEVICE

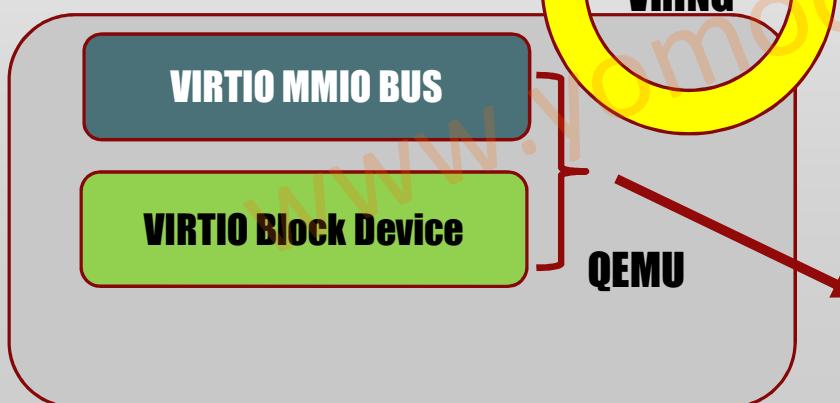
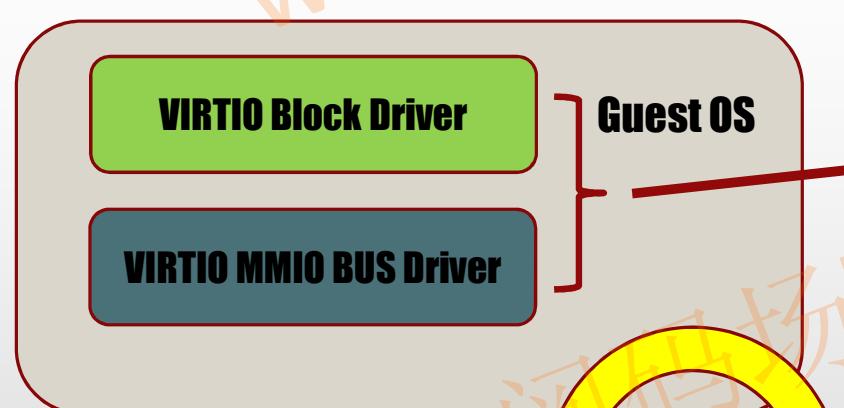


**VIRTIO API**封装了对**VRING**存取数据的操作  
和对**VIRTIO DEVICE**虚拟寄存器读写的操作

## VirtIO MMIO Device Register Layout

0x00	<b>VIRTIO_MMIO_MAGIC_VALUE</b>
	<b>VIRTIO_MMIO_VERSION</b>
	<b>VIRTIO_MMIO_DEVICE_ID</b>
	.....
	<b>VIRTIO_MMIO_GUEST_PAGE_SIZE</b>
	<b>VIRTIO_MMIO_QUEUE_SEL</b>
	<b>VIRTIO_MMIO_QUEUE_NUM_MAX</b>
	<b>VIRTIO_MMIO_QUEUE_NUM</b>
	<b>VIRTIO_MMIO_QUEUE_ALIGN</b>
	<b>VIRTIO_MMIO_QUEUE_PFN</b>
	<b>VIRTIO_MMIO_QUEUE_READY</b>
	<b>VIRTIO_MMIO_QUEUE_NOTIFY</b>
	<b>VIRTIO_MMIO_INTERRUPT_STATUS</b>
	<b>VIRTIO_MMIO_INTERRUPT_ACK</b>
	<b>VIRTIO_MMIO_STATUS</b>
	.....
0x100	<b>VIRTIO_MMIO_CONFIG</b>

# VIRTIO ARCHITECTURE



2

## Virtio Block Device Frontend Drivers:

1. 从 **Virtio MMIO BUS** 中枚举出 **Virtio Block** 设备 [通过 **Device ID, Vendor ID** 等进行关联]
2. 创建 **virtqueue** 和 **vring**

## Virtqueue + Vring:

负责 **FrontEnd** 和 **backend** 间的数据传递

1

## Virtio Block Device Backend Drivers:

1. 在 **Guest** 启动之前, 在创建虚拟机的时候将 **Virtio MMIO Device** 的 **Device ID, Vendor ID**, 设备类型设置到虚拟寄存器中。
2. 关联好回调函数和后端真实设备或者文件

# VIRTIO DEVICE DISCOVERY

虚拟机初始化:

1. 创建**virtio-mmio** 设备，分配相应的资源，并插入到虚拟总线中
2. 修改**Guest DTB**加入**virtio MMIO**的寄存器地址段和中断信息

**QEMU**

后端设备初始化:

1. 进行**virtio block device**后端初始化，比如多少个**virtqueue**, **virtqueue**支持多少个元素 **SEG\_MAX**是多少等

**QEMU**

**VIRTIO MMIO BUS**枚举:

1. 扫描**DTB**, 发现“**virtio, mmio**”设备
2. 访问寄存器读取**DEVICE\_ID**, **VENDOR\_ID**
3. 向系统注册设备

**GuestOS**

发现后端设备:

1. 系统事先注册的**virtio block driver**发现了**virtio block**设备
2. 调用**virtio block probe**

**GuestOS**

前端驱动:

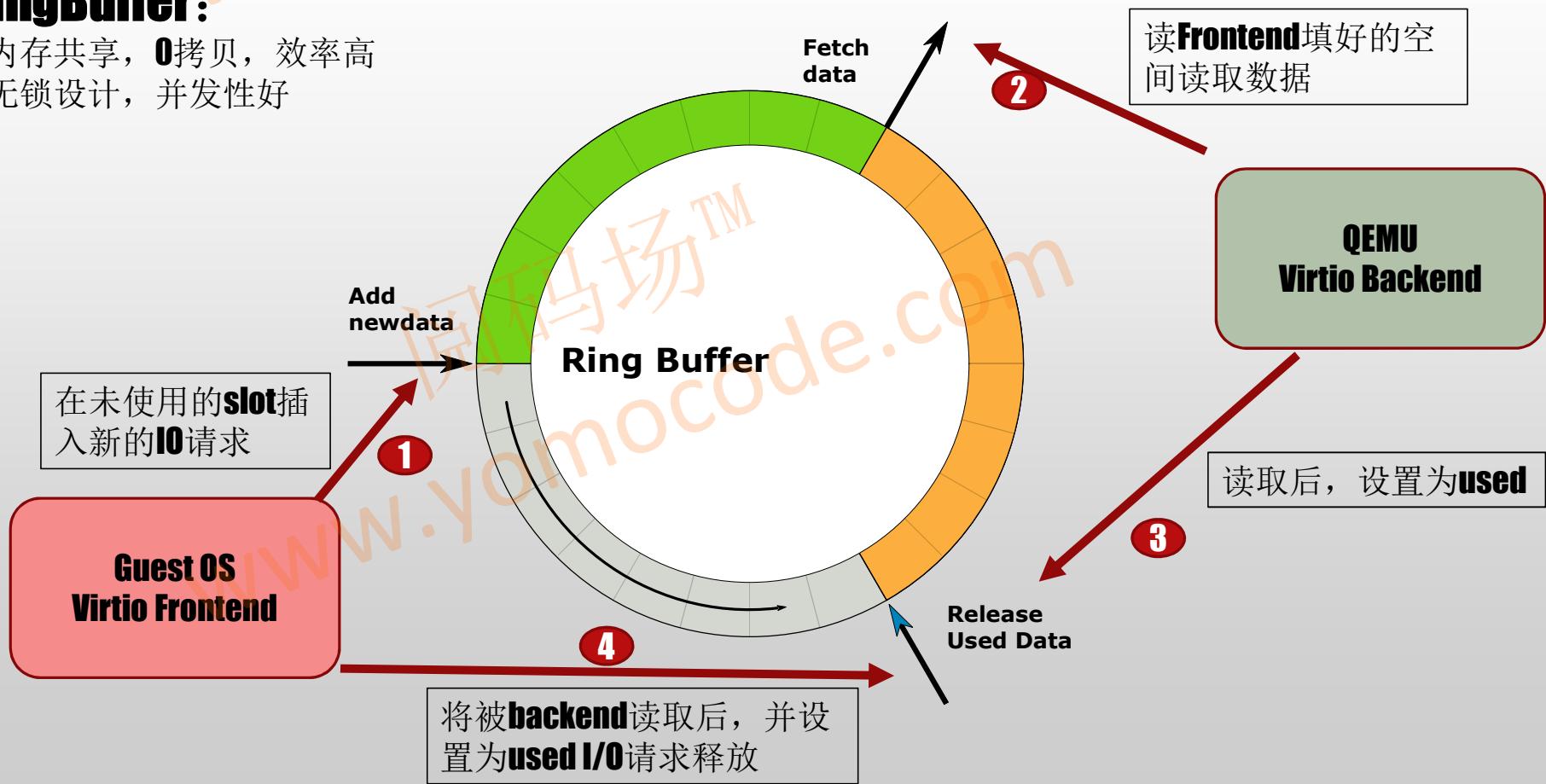
1. 读取设备信息，创建**virqueue**
2. 向系统注册新的块设备/**/dev/vda**

**GuestOS**

# VIRTIO RINGBUFFER (VRING)

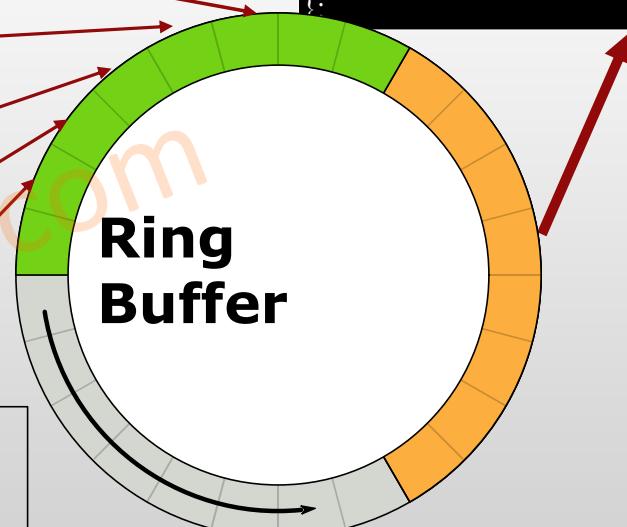
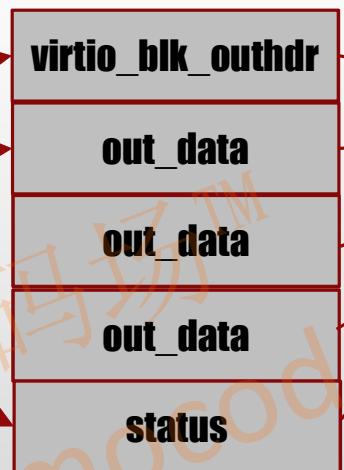
## VringBuffer:

- 内存共享，0拷贝，效率高
- 无锁设计，并发性好



# VIRTIO BLOCK VRING DESC FOR WRITING

```
struct virtblk_req {  
    struct virtio_blk_outhdr out_hdr;  
    u8 status;  
    struct scatterlist sg[];  
};
```

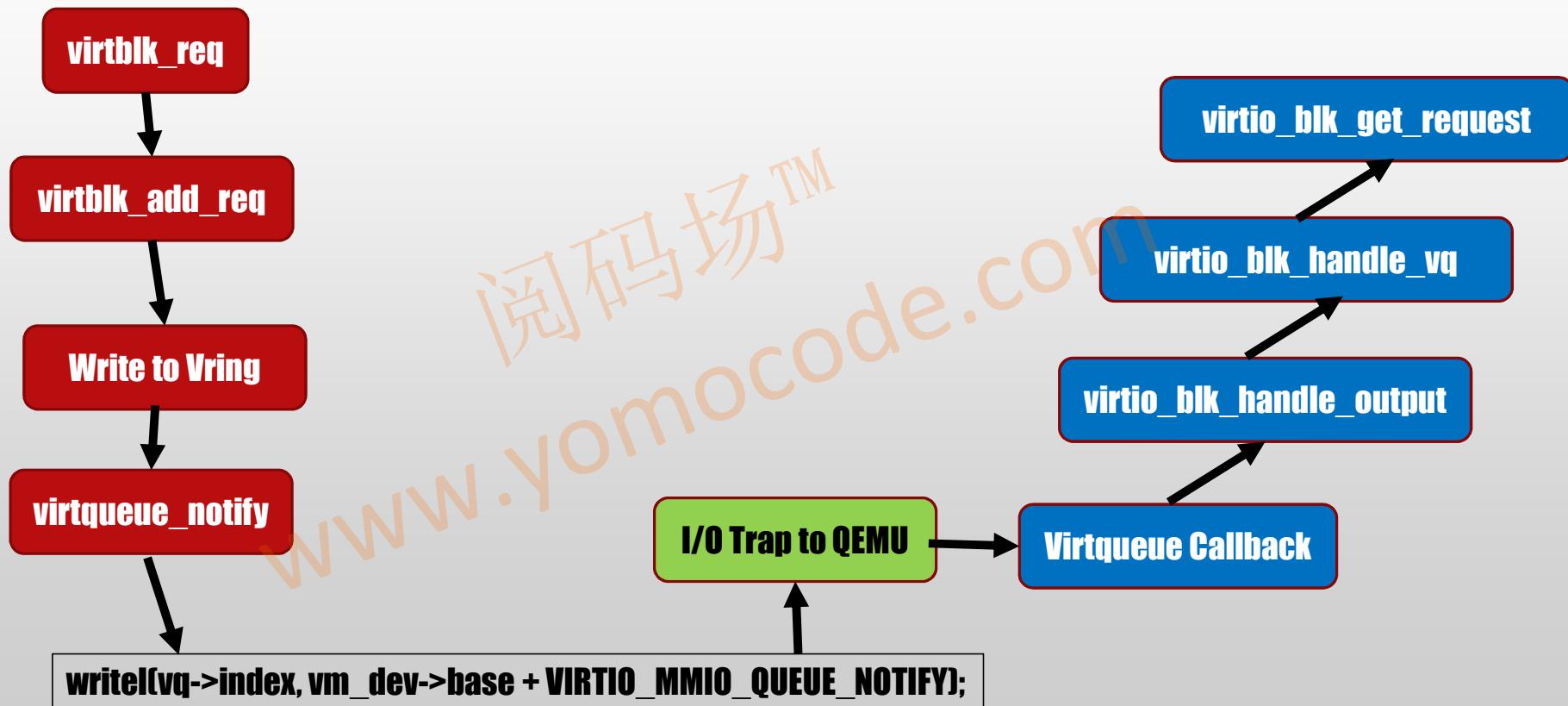


```
struct vring_desc {  
    /* Address (guest-physical). */  
    __virtio64 addr;  
    /* Length. */  
    __virtio32 len;  
    /* The flags as indicated above. */  
    __virtio16 flags;  
    /* We chain unused descriptors via this, too */  
    __virtio16 next;  
};
```

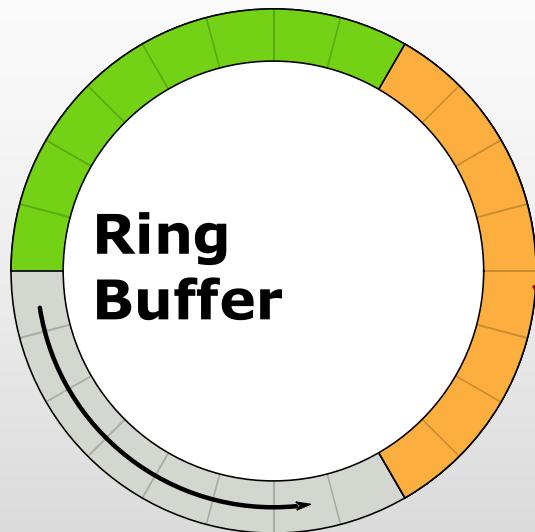
## Virtio blk frontend:

1. 向vring中写入**virtio\_blk\_outhdr**的**address**和**size**
2. 如果是写磁盘操作，还需要在**vring**中写入数据对应的**address**和**size**写入。磁盘操作往往使用**sg\_list**，所以可能有多个数据页面要写入
3. 向**vring**中写入**virtio\_blk\_inhdr**的**address**和**size**，它只有一个成员**status**，用于让**QEMU**写入这个I/O操作的状态

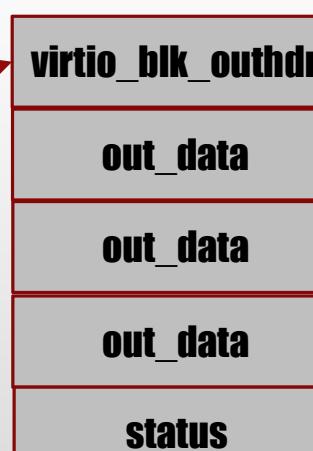
# VIRTIO BLOCK VRING DESC FOR WRITING



# VIRTIO BLOCK VRING DESC FOR WRITING



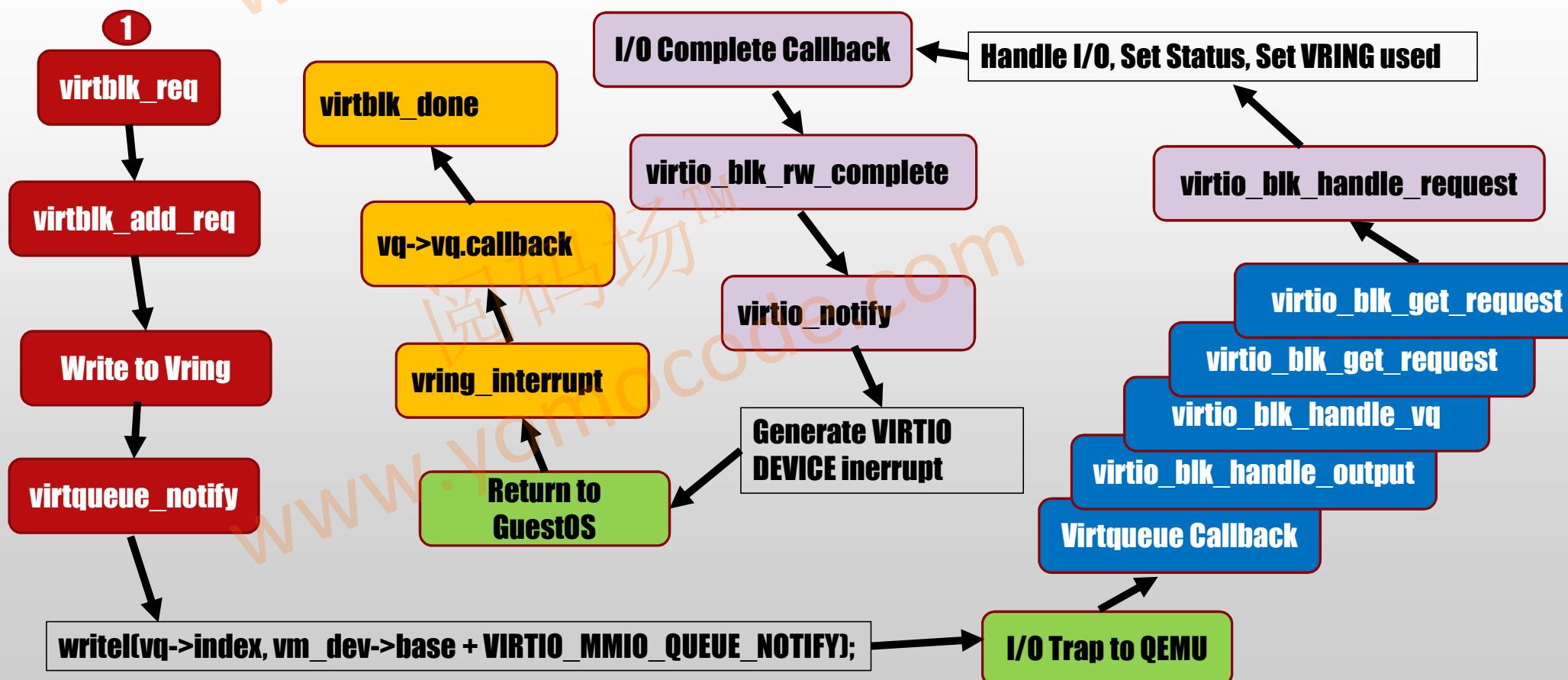
```
do {  
    .....  
    while ((req = virtio_blk_get_request(s, vq))) {  
        progress = true;  
        if (virtio_blk_handle_request(req, &mrb)) {  
            .....  
        }  
        .....  
    } while (!virtio_queue_empty(vq));
```



## Virtio blk backend:

1. 从vring中还原出**VirtIOBlockReq**
2. 判断outhdr中的描述是写操作。重组IO使用的**sg\_list**
3. 调用host接口完成磁盘写操作
4. 将读取完的**vring slot**设置为**used**。

# VIRTIO BLOCK I/O LOOP



# DEVICE PASSTHROUGH

设备直通

# WHAT AND WHY

## 全虚拟化设备：

- 兼容性好，性能差

## 半虚拟化设备：

- 性能比较好，兼容性上需要新增半虚拟化驱动

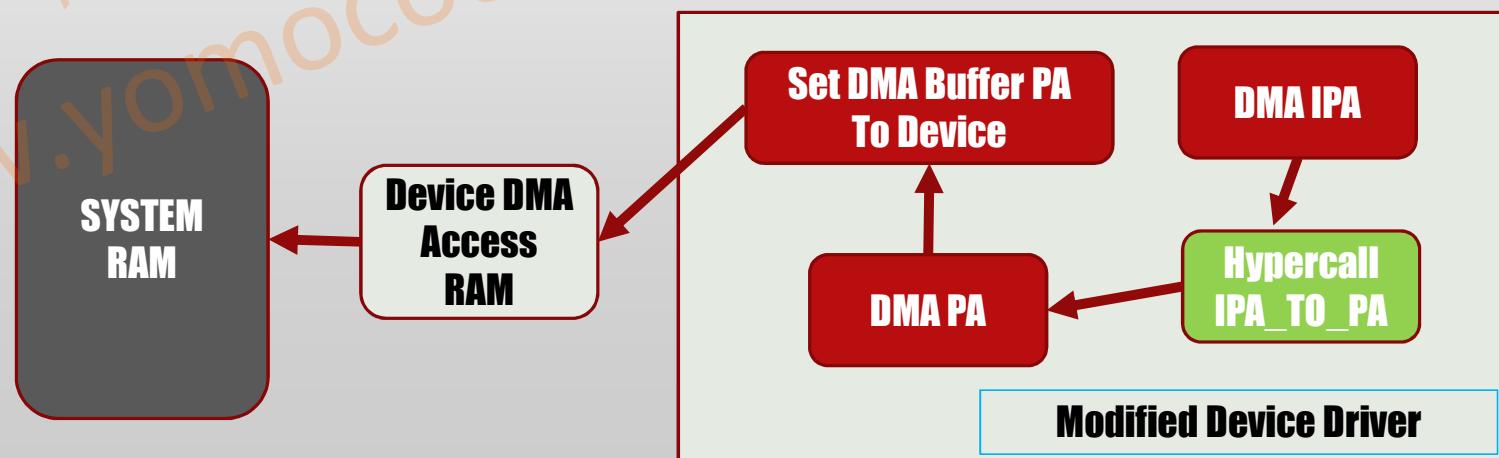
## 直通设备：

- 设备直通[**Device Passthrough**]是虚拟化场景中的一种高性能解决方案
- **GuestOS**在虚拟机中直接访问主机**host**上的物理外设
- **GuestOS**可以使用原有的硬件驱动，不要增加新驱动
- **GuestOS**直接访问设备可以有效减少特权层切换的开销
- 性能和在**Host**主机上直接使用该物理硬件非常接近

# HOW PHYSICAL DEVICE WORK IN GUESTOS

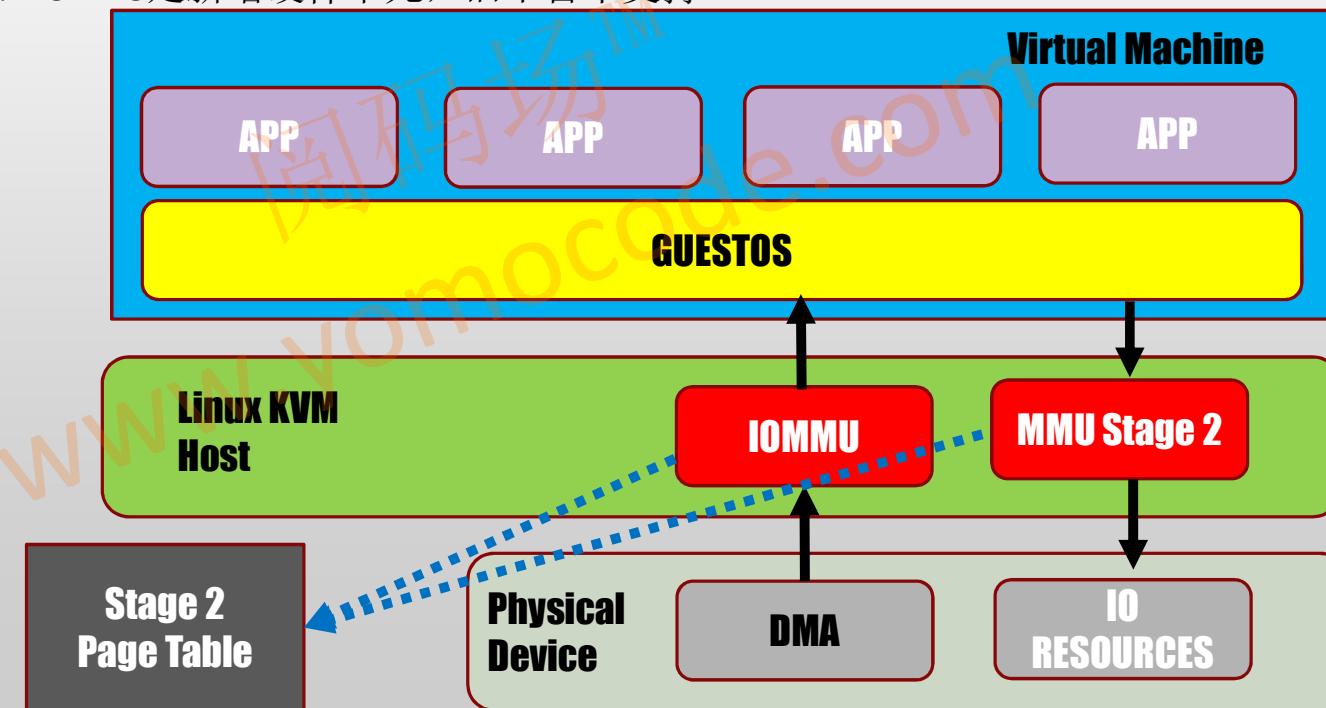
让物理设备在虚拟机中能够正常工作需要做到：

- GuestOS能够直接访问物理设备上的寄存器和I/O内存(比如GPU RAM)
  - 通过MMU的STAGE2地址转换，将物理设备的I/O资源都映射到虚拟机的IPA地址空间
- 物理设备在虚拟机环境中能够正常的进行DMA操作
  - 方法1：通过半虚拟化接口，在hypervisor的帮助下将DMA buffer的IPA转换成PA
    - 优点：不需要新增额外的地址转换硬件单元
    - 缺点：
      - 1. 需要修改设备驱动，IPA->PA转换会造成特权层切换。
      - 2. 安全性差，直通设备的DMA访问权限不受控。
      - 3. GuestOS给的任意有效物理地址它都可以访问。



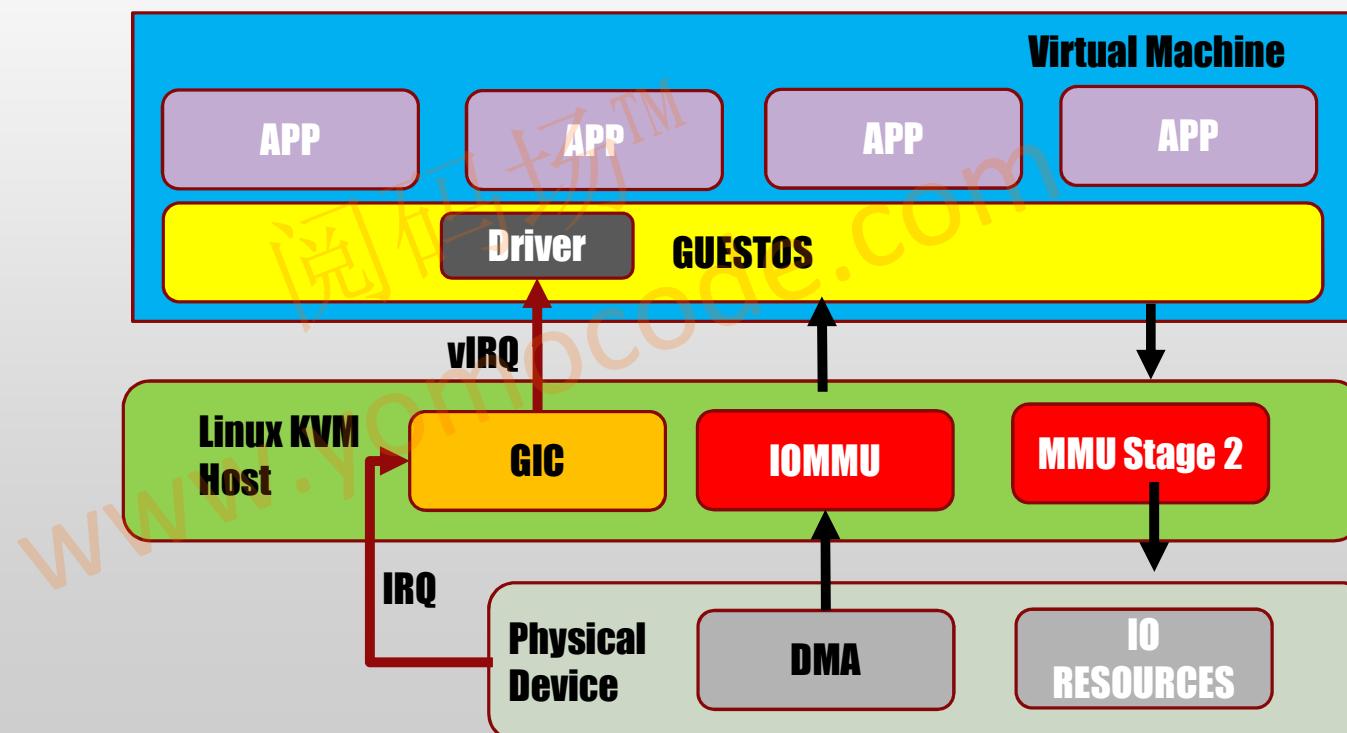
# HOW PHYSICAL DEVICE WORK IN GUESTOS

- 物理设备在虚拟机环境中能够正常的进行**DMA**操作
  - 方法**2**: 通过在系统总线上新增**IOMMU**帮助设备完成**IPA->PA**的转换，同时限制设备访问的**PA**空间
    - 优点: 1. 不需要修改驱动，**GuestOS**使用该设备完全透明和在物理主机上一样
    - 2. 设备访问的**PA**地址范围受控。**IOMMU**页表可以锁定转**IPA->PA**的转换和**MMU**一致，仅限于虚拟机被授予的**PA**空间
  - 缺点: **IOMMU**是新增硬件单元，旧平台不支持



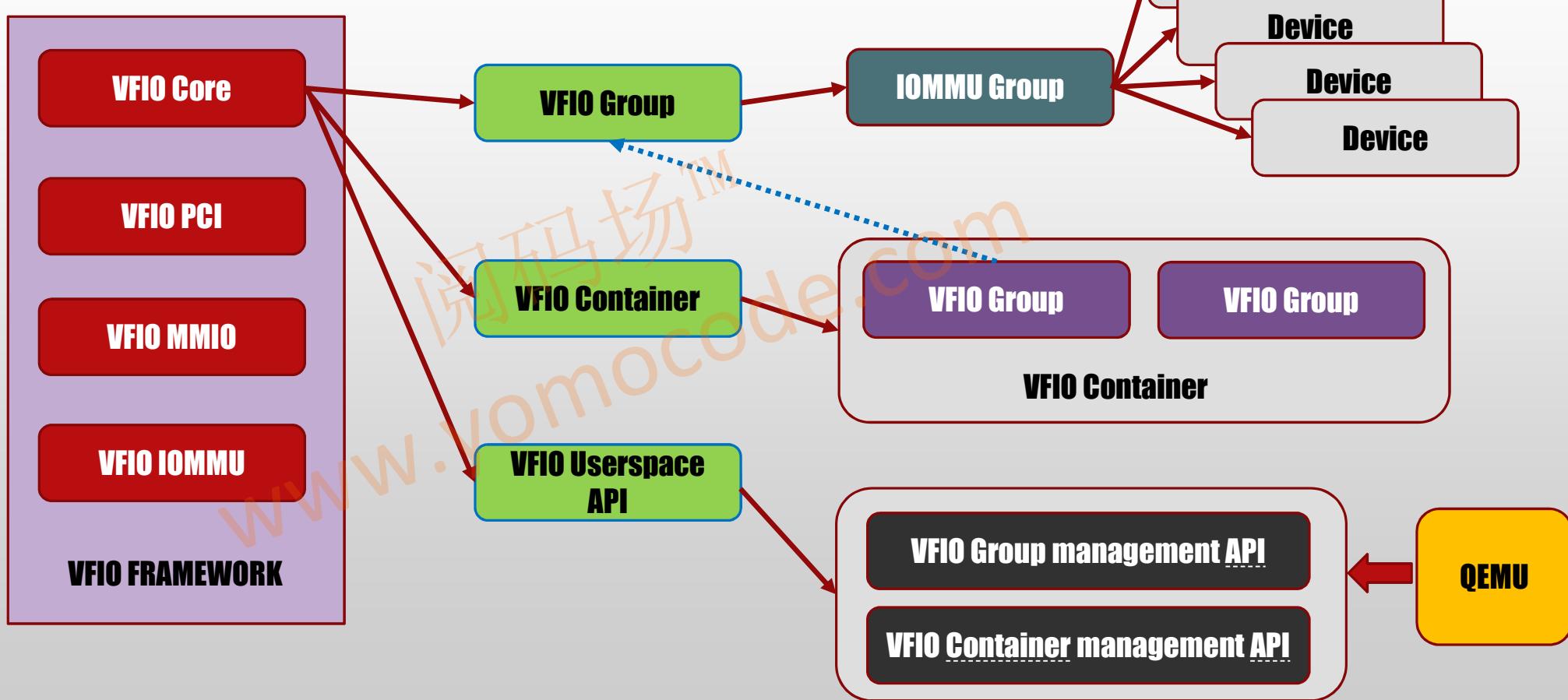
# HOW PHYSICAL DEVICE WORK IN GUESTOS

- 物理设备的中断能偶在虚拟机环境中能够正常处理
  - 物理设备的中断由**Linux-KVM host**先接受，然后再注入到虚拟机
  - 虚拟内部的设备驱动处理虚拟中断



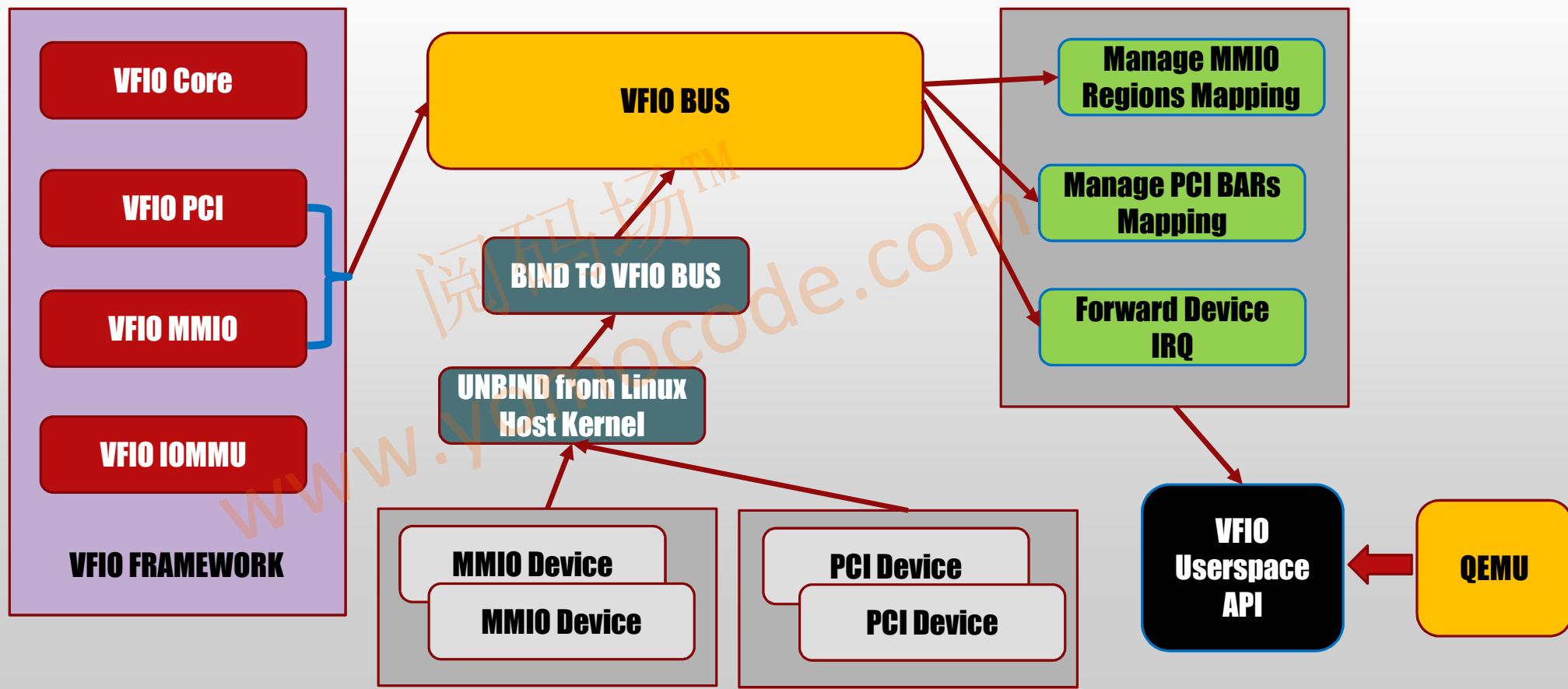
# LINUX-KVM + QEMU'S METHOD

- Linux-KVM 提供了一个VFIO(Virtual Function I/O)框架



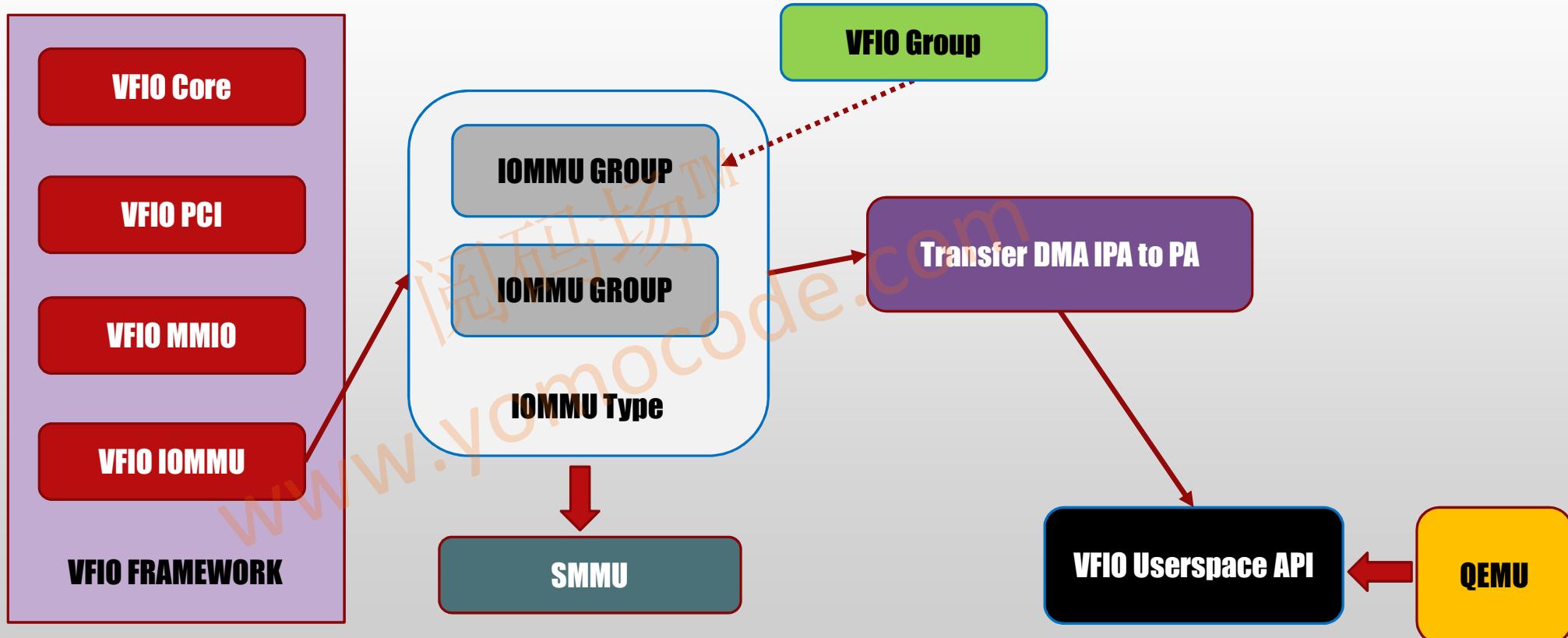
# LINUX-KVM + QEMU'S METHOD

- Linux-KVM 提供了一个VFIO(Virtual Function I/O)框架



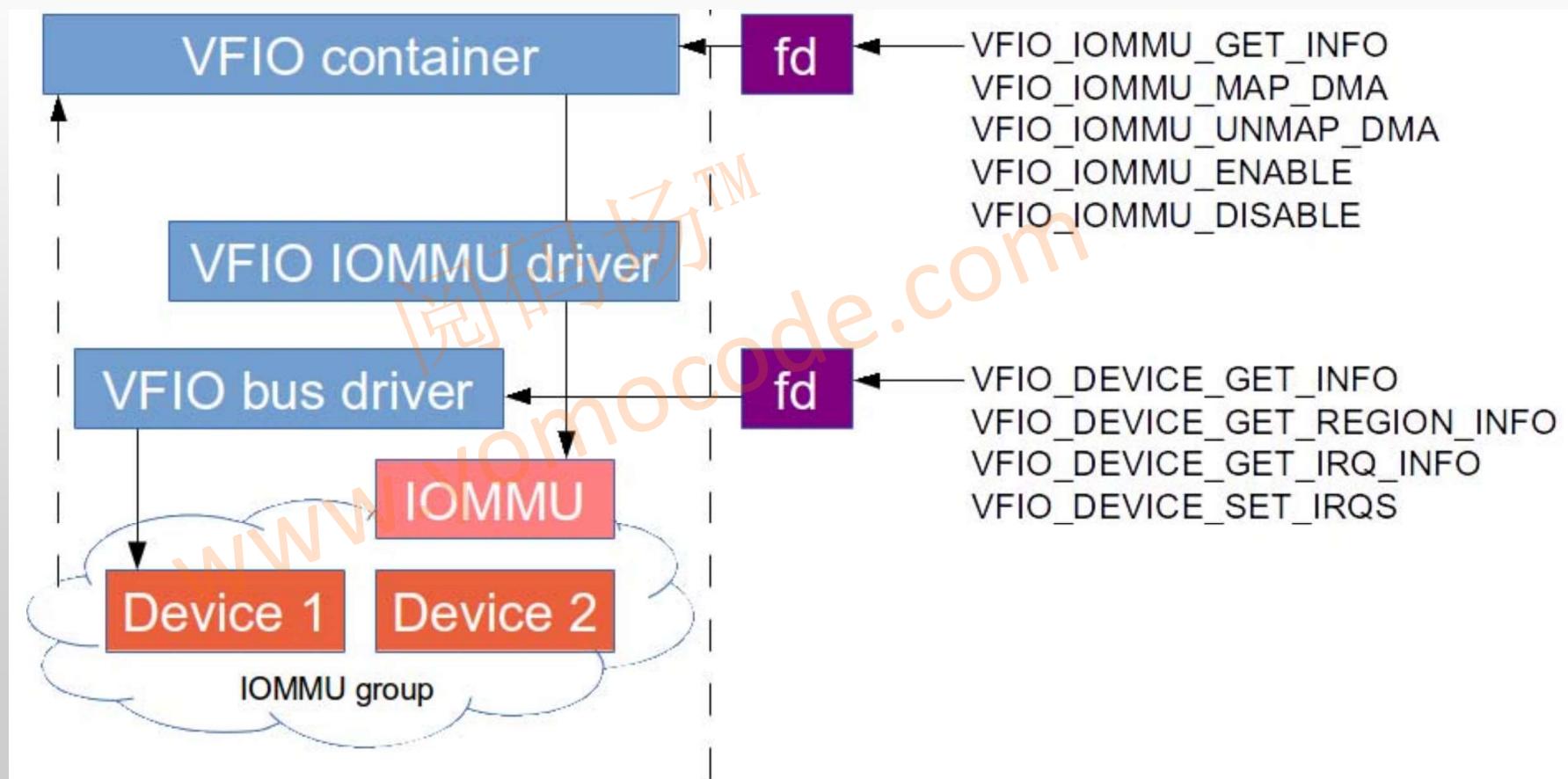
# LINUX-KVM + QEMU'S METHOD

- Linux-KVM 提供了一个VFIO(Virtual Function I/O)框架



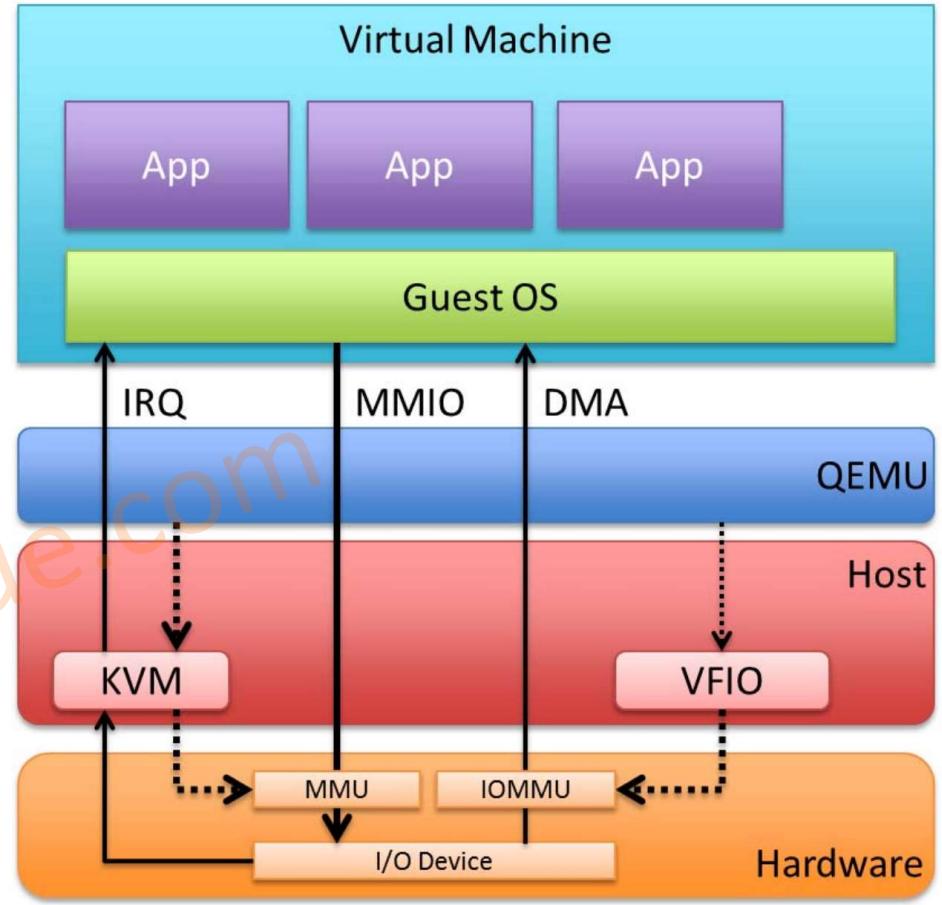
# LINUX-KVM + QEMU'S METHOD

- Linux-KVM 提供了一个**VFIO(Virtual Function I/O)**框架



# 利用**VFIO**完成的设备直通

- 利用**VFIO API**获取设备的**I/O**资源，然后利用**KVM API**映射到虚拟机**IPA**地址空间
- 使用**VFIO API**关联**IOMMU**完成设备**IPA->PA**页表的创建和使能
- 利用**VFIO**中断**API**完成中断的注册与转发



谢谢!  
**THANK YOU!**

