软件层次图

iommu enabled device driver 使用 iommu 的设备驱动，如 gpu/pci 设备驱动

|

DMA mapping API

|

IOMMU 子系统

dma iommu　　drivers/iommu/dma-iommu.c

iova　　drivers/iommu/iova.c

iommu map　　drivers/iommu/iommu.c

|

iommu driver 即操作 iommu 硬件的驱动　drivers/iommu/rockchip-iommu.c

sysfs 接口

/sys/kernel/iommu_groups/0/devices/fdab0000.npu# ls

...

supplier:platform:fdab9000.iommu

...

可以看到，这个 iommu group 0,是由 fdab0000.npu 使用者，和 fdab9000.iommu 组成

dts 举例

```
vpu: video-codec@ff650000 {
	...
	iommus = <&vpu_mmu>;
	...
};
vpu_mmu: iommu@ff650800 {
	compatible = "rockchip,iommu";
	reg = ...
	interrupts = ...
	#iommu-cells = <0>;
}
```

dts 是如何解析的？

通过 drivers/iommu/of_iommu.c

really_probe->platform_dma_configure->of_dma_configure->of_dma_configure_id->of_iommu_configure->of_iommu_configure_device->of_iommu_configure_dev->of_parse_phandle_with_args 解析"iommus"

rockchip iommu 页表结构

```
/*
 * The Rockchip rk3288 iommu uses a 2-level page table.
 * The first level is the "Directory Table" (DT).
 * The DT consists of 1024 4-byte Directory Table Entries (DTEs), each pointing
 * to a "Page Table".
 * The second level is the 1024 Page Tables (PT).
 * Each PT consists of 1024 4-byte Page Table Entries (PTEs), each pointing to
 * a 4 KB page of physical memory.
 *
 * The DT and each PT fits in a single 4 KB page (4-bytes * 1024 entries).
 * Each iommu device has a MMU_DTE_ADDR register that contains the physical
 * address of the start of the DT page.
 *
 * The structure of the page table is as follows:
 *
 *                   DT
 * MMU_DTE_ADDR -> +-----+
 *                 |     |
 *                 +-----+     PT
 *                 | DTE | -> +-----+
 *                 +-----+    |     |     Memory
 *                 |     |    +-----+     Page
 *                 |     |    | PTE | -> +-----+
 *                 +-----+    +-----+    |     |
 *                            |     |    |     |
 *                            |     |    |     |
 *                            +-----+    |     |
 *                                       |     |
 *                                       |     |
 *                                      +-----+
 */

/*
 * Each DTE has a PT address and a valid bit:
 * +---------------------+-----------+-+
 * | PT address          | Reserved  |V|
 * +---------------------+-----------+-+
 *  31:12 - PT address (PTs always starts on a 4 KB boundary)
 *  11: 1 - Reserved
 *      0 - 1 if PT @ PT address is valid
 */
```

Rk 的两级页表分别叫做 dte 和 pte。

```
/*
 * rk3288 iova (IOMMU Virtual Address) format
 *  31       22.21       12.11          0
 * +-----------+-----------+-------------+
 * | DTE index | PTE index | Page offset |
 * +-----------+-----------+-------------+
 *  31:22 - DTE index   - index of DTE in DT
 *  21:12 - PTE index   - index of PTE in PT @ DTE.pt_address
 *  11: 0 - Page offset - offset into page @ PTE.page_address
 */
#define RK_IOVA_DTE_MASK    0xffc00000
#define RK_IOVA_DTE_SHIFT   22
#define RK_IOVA_PTE_MASK    0x003ff000
#define RK_IOVA_PTE_SHIFT   12
#define RK_IOVA_PAGE_MASK   0x00000fff
#define RK_IOVA_PAGE_SHIFT  0

static u32 rk_iova_dte_index(dma_addr_t iova)
{
    return (u32)(iova & RK_IOVA_DTE_MASK) >> RK_IOVA_DTE_SHIFT;
}
```

rk iommu iova 定义

下面均以 rk 的 iommu 驱动为例，解释 iommu 子系统的注册和运行流程。

iommu 的数据结构情况：

见《iommu 数据结构关系.xmind》

iommu_ops 解释

```
static const struct iommu_ops rk_iommu_ops = {
	.domain_alloc = rk_iommu_domain_alloc,//分配 iommu_domain,iova cookie,分配一级页表空间
	.domain_free = rk_iommu_domain_free,
	.attach_dev = rk_iommu_attach_device,//attach 上 iommu_domain,写入一级页表首地址并使能
	.detach_dev = rk_iommu_detach_device,
	.map = rk_iommu_map,//分配二级页表，完成 iova 到 pa 的映射
	.unmap = rk_iommu_unmap,
	.probe_device = rk_iommu_probe_device,//device_link 初始化，将电源管理权限交给 master，由 master 控制 pm runtime get,put。
	.release_device = rk_iommu_release_device,
	.iova_to_phys = rk_iommu_iova_to_phys,//软件方式从 iova 翻译成 pa
	.device_group = rk_iommu_device_group,
	.pgsize_bitmap = RK_IOMMU_PGSIZE_BITMAP,//此 iommu 能支持的页大小的 bitmap，例如 rk 是支持 4kb~4mb
	.of_xlate = rk_iommu_of_xlate,//分配 rk 私有结构体
};
```

这些 iommu_ops 的调用先后顺序：

of_xlate
probe_device
domain_alloc
attach_dev
map

以上函数的调用栈，见《iommu_ops 调用栈.xmind》

不同 ip 间，传递非连续地址的 buffer

假设 IP a 和 IP b 各自有专属自己的 iommu 实例。
a 是生产者，b 是消费者。
a 通过 scatterlist 组织了一堆 buffer，物理不连续。
首先 a 通过 dma mapping api，自己编程 a 的 dma 和 iomma，作为生产者生产好内容。
然后 a 通过 dma-buf 把 fd 传递给 b（进程 A 到进程 B 间传递）
进程 B 拿到 dma-buf 后，通过 dma_buf_get 拿到 dma_buf 结构体
然后通过 dma_buf_attach 和 dma_buf_map_attachment 拿到 sg

执行 dma_buf->dma_buf_ops->map_dma_buf->dma_map_sg 来映射给 dma，建立 b 自己的 iova-pa 的映射并使用

各家 iommu 概念横向比对

vmm: virtual machine manager

	vmm 区分 deivce	vmm 区分 vm	所在数据结构	guest 区分 process	所在数据结构
smmuv3	streamID	VMID	stream table entry	subStreamID	Context Descriptor
risc-v	device_id	SDCID	device-directory-table	process_id	Process-directory-table

绿色是嵌套翻译的 stage 2，由 vmm 维护。黄色是嵌套翻译的 stage 1，由 guest 维护。

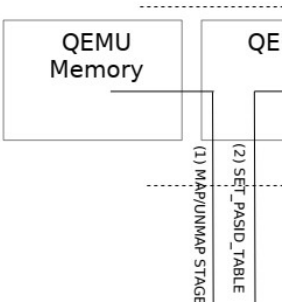
arm smmuv3 嵌套翻译设计

硬件翻译顺序：stage 1 -> stage 2

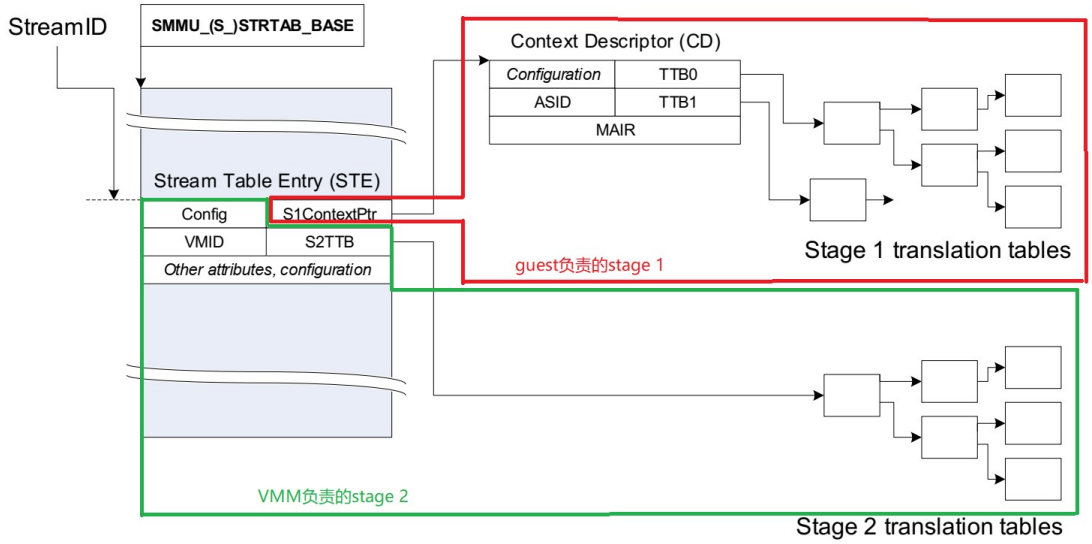
软件配置顺序：stage 2 -> stage 1

Nested Stage Control Flow

1	A Guest RAM region is added	Build stage2 mapping. Force HW stage2 to be used.
2	Guest config invalidation commands	Propagate stage 1 guest config to the host
3	Guest sends TLB/PASID cache invalidation commans	Propagate invalidations to Host
4	MSI Enable	Propagate stage1 MSI binding from guest to host
5	Stage 1 related fault	Propagate stage 1 faults from host to guest



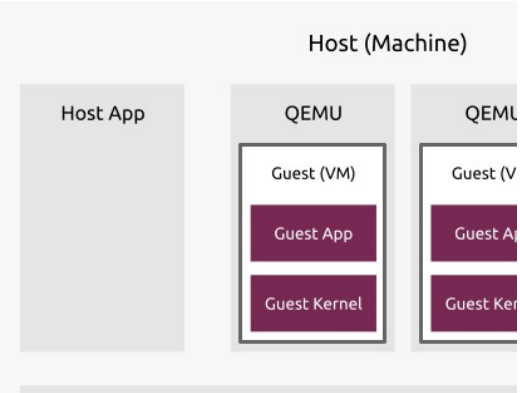
guest 和 host 一样都有 smmuv3 内核驱动，只是 guest 配置的是 stage 1 only，host 配置的是 nested 方式。



QEMU

qemu 支持全/半虚拟化。

qemu 的每个实例就是一个虚拟机，作为 host linux 里面的一个进程，qemu 加载 guest dtb 和 kernel 运行 guest os。



VFIO

类似 iommu 这样的设备，想要给 guest 用，但原先设计是暴露给 dma mapping api 的。

通过 VFIO，本质上就是用字符设备封装了一层，产生了一个 iommu 的字符设备，这样 qemu 的用户态逻辑里面，添加相应的 vfio 逻辑后就可以直接操作 host 的 iommu 驱动了。

方案设计

https://github.com/eauger/linux/tree/v5.0-rc1-2stage-rfc-v3

https://github.com/eauger/qemu/tree/v3.1.0-rc5-2stage-v3-for-rfc3-test-only

代码现状：

虚拟化场景下，现有代码（kernel+qemu）不支持通过 guest 通过 vfio 使用 host 的 iommu，只支持 qemu 作为 vmm 去配置 iommu（完成 guest physical addr->host physical addr 翻译）

改动点：

a. smmuv3 驱动：
   iommu_ops 增加
   arm_smmu_set_pasid_table　　让 guest 单独设置 stage 1 table
   arm_smmu_cache_invalidate　　让 guest 单独 inv stage 1 cache
   arm_smmu_bind_guest_msi
b. vfio 驱动：
   对 smmuv3 新增 iommu_ops 增加的对应的 ioctl。
   硬件 iommu stage 1 传输错误，注册给 iommu 子系统，出错的时候回调并通知 guest
c. iommu 子系统
   对 smmuv3 新增 iommu_ops 增加的对应的 iommu_set_pasid_table 等函数
d. qemu
   vfio/smmu 层：增加对 guest 做配置 stage 1 table/map/unmap 出发 mmio range 回调函数的支持

Stage 2 config + mapping 流程

stage 2 是 qemu 做 guest physical addr->host physical addr 翻译的过程。

vfio_platform_realize->vfio_base_device_init->vfio_get_group->vfio_connect_container->vfio_init_container->VFIO_SET_IOMMU 完成 stage 2 配置。

qemu 发起主要步骤包括：

1. 通过 vifo_ioctl_set_iommu，设置 host 的 smmu_domain->stage = ARM_SMMU_DOMAIN_NESTED。完成 host 的 iommu_attach_group，分配 stage 2 的 pgd 页表，生成 vmid，填充 stage 2 的配置即 smmu_domain->s2_cfg
2. 通过 vfio_dma_map，完成 stage 2 的 map 映射操作。

这里 stream table entry 的 s1_cfg 是 NULL，即 stage 1 留给 guest 来配置。

Stage 1 config 流程

qemu 通过 mmio 映射区域的方式，监视 guest 内部 smmuv3 的硬件寄存器区域。当 guest 写 smmuv3 的 command queue 时，触发

MemoryRegionOps.write_with_attrs->smmu_write_mmio->smmu_writel->smmuv3_cmdq_consume->smmuv3_notify_config_change->smmuv3_get_config->smmuv3_decode_config->decode_ste-> cfg->s1ctxptr = STE_CTXPTR(ste)拿到 stage 1 的 S1ContextPtr

并且

MemoryRegionOps.write_with_attrs->smmu_write_mmio->smmu_writel->smmuv3_cmdq_consume->smmuv3_notify_config_change->IOMMU_NOTIFIER_PASID_CFG->vfio_iommu_nested_notify->VFIO_IOMMU_SET_PASID_TABLE->arm_smmu_set_pasid_table 实际配置 host 的 stage 1

即：

1. qemu 通过 mmio 区域，获取 guest 配置的 stream table entry 的 S1ContextPtr，即指向 stage 1 context pointer 的入口地址
2. qemu 通过 IOMMU_NOTIFIER_PASID_CFG，通过 ioctl VFIO_IOMMU_SET_PASID_TABLE 最终设置 arm_smmu_set_pasid_table 配置 stage 1。

guest 想做 cache inv 也是类似的流程

Stage 1 mapping 流程

由 guest os 自行更新 stage 1 页表，入口地址是 S1ContextPtr

stage 1 iommu 错误上报到 guest

iommu 硬件出现错误中断

->arm_smmu_evtq_thread->arm_smmu_report_event->iommu_report_device_fault->vfio_pci_iommu_dev_fault_handler->vfio_pci_device.dma_fault_trigger->eventfd_signal(dev->dma_fault_trigger, 1)用 event fd 接口从内核反向通知用户态

用户态收到通知后：在 qemu 内部，

vfio_dma_fault_notifier_handler->memory_region_inject_fault 通知 guest os mmio 空间，从而在 guest os 产生一个硬件“错误”