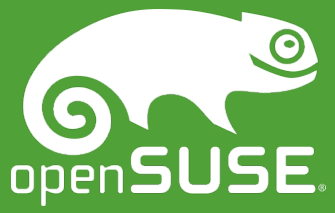
vfio-pci passthrough

Fei Li

fli@suse.com



Summary

- What is VFIO/IOMMU? Why want them?
- VFIO – qemu part
- VFIO – kernel part
- VFIO usage: how to passthrough a pci device

- What is VFIO/IOMMU? Why want them?

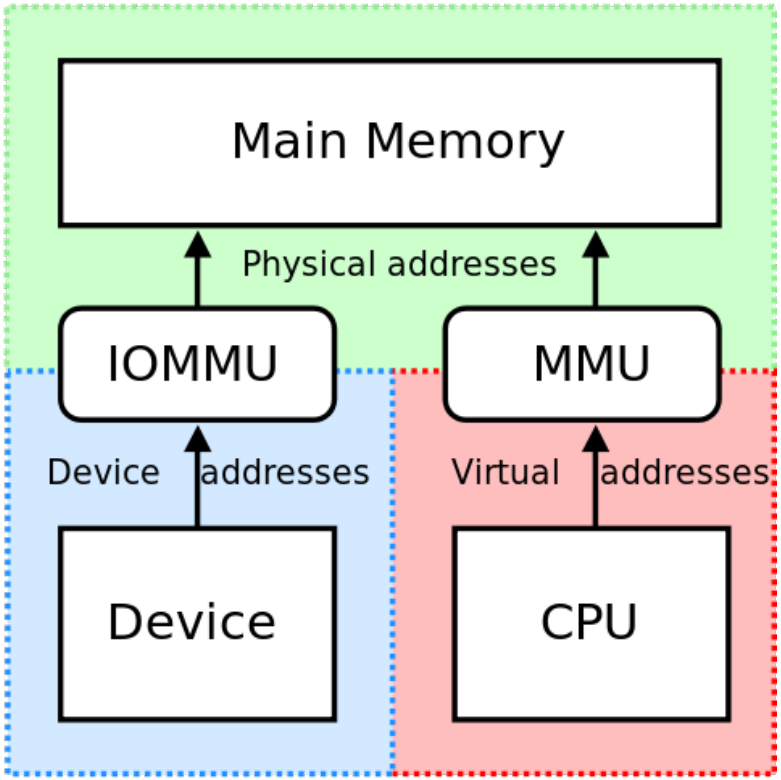
What is VFIO/IOMMU?

- The VFIO (Virtual Function I/O) driver is an IOMMU/device agnostic framework for exposing direct device access to userspace, in a secure, IOMMU protected environment.
- For x86, it needs the I/O MMU hardware support.
- VFIO consists of
- - kernel device driver: vfio_pci_driver, vfio_iommu_driver, vfio_dma
- - QEMU device class: VFIODevice, VFIOPCIDevice
- The guest can operate the pass-throughed PCI device by:
  - accessing the mapped PCI config space and memory space
  - ioctl() on a fd of the VFIO kernel device for control operations

What is VFIO/IOMMU? (Continue.)

- In qemu, use VFIO to configure IOMMU:

  e.g. ioctl(VFIO_SET_IOMMU) && ioctl(VFIO_IOMMU_MAP_DMA)



VFIO: device, group, container

- A group is a set of devices which is isolatable from all other devices, specialized in IOMMU. It is the minimum granularity.
- Within one container, different groups can share a set of page tables to reduce the duplication. The container provides little functionality: version check and extension query.
- The user needs to add a group into the container.
- The VFIO device API includes ioctls for describing the device, the I/O regions and their read/write/mmap offsets on the device descriptor, && mechanisms for describing and registering interrupt notifications.

Why want them?

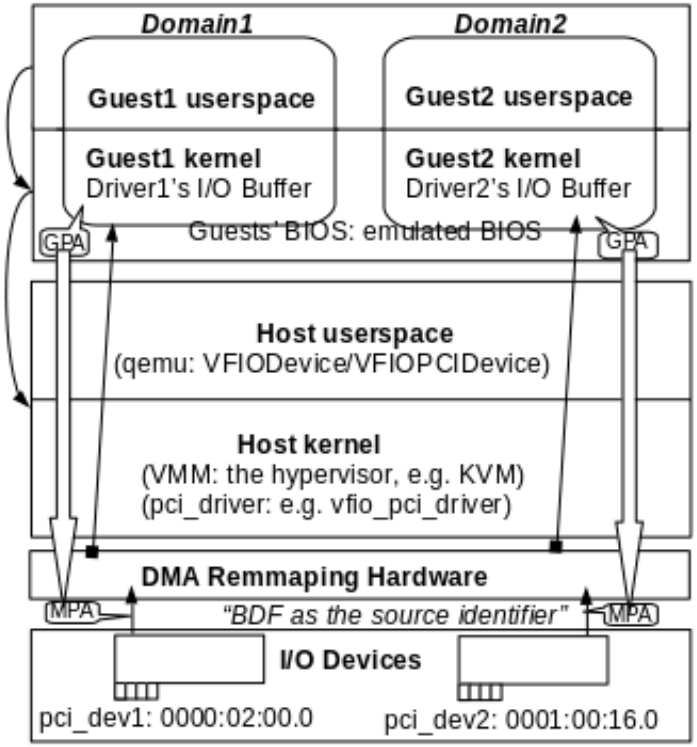
- In short, for higher I/O performance by lessening the times of VM-EXIT/VM-ENTRY when accessing PCI BAR and doing DMA.

1. How does the guest visit the pth-ed pci's BAR

1.1 When the BAR is firstly accessed via MMIO, the emulated GPA can not be visited.

1.3 When later accessed, no VM-EXIT is needed.

1.2 VM-EXIT. VMM: update the EPT page table to establish the address mapping between GPA and HPA.



2.1 qemu:
=>vfio_realize()
=> vfio_region_mmap()
to get the HVA
(RAMBlock→ host) via
mmap() for each bar-region.
=> qemu_vfio_do_mapping()
{
ioctl(s->container,
VFIO_IOMMU_MAP_DMA,
&dma_map)
}
to do the real memory
mapping between VFIO iova
and HPA

||

2.2 kernel:
=> vfio_dma_do_map()
=> vfio_pin_map_dma()
=> vfio_pin_pages_remote()
&& vfio_iommu_map
==> intel_iommu_map()

2. DMA:
- guest driver 'tells' GPA
- host pci needs MPA

Why want them? (literal)

- - when accessing PCI BAR. The emulated guest BIOS emulate the BAR address for the guest (On the contrary, if expose the host real BAR to the guest, there may be conflicts between host real PCI BAR and the other emulated PCI device's BAR in the guest). For the first time when BAR is accessed and can not be visited, the VM exists and does the address mapping between GPA and HPA using EPT, and records the mapping. When later access, no VM-exit is needed.
- - when PCI device communicates with GPA via DMA. When initializing the vfio: vfio_realize() in qemu, a memory mapping is established via vfio_region_mmap(). Then a (ioctl(s->container, VFIO_IOMMU_MAP_DMA, &dma_map) in qemu_vfio_do_mapping will do the real mapping between VFIO iova and HPA.

VFIO – qemu part

- How to define vfio-pci in qemu-cli
- Code: how to initialize the vfio-pci device

VFIO – qemu part

1. How to define vfio-pci in qemu command line:

   `-device vfio-pci, host=0000:02:00.0, bus=pci.1, id=vpci1`

2. Code: how to initialize the vfio-pci device

   2.1 Deploy the common device_init_func() to initialize each qdev in vl.c:

   main(argc, **argv, **envp) {

```
qemu_opts_foreach(qemu_find_opts("device"),
                  device_init_func, NULL, &error_fatal);
```

   }

   2.2 device_init_func() => qdev_device_add() to parse the qdev's every attribute (next page)

   2.3 object_property_set_bool() => … => prop→set() => property_set_bool => device_set_realized() => pci_qdev_realize() => vfio_realize()

VFIO – qemu part: qdev_device_add()

```
DeviceState *qdev_device_add(QemuOpts *opts, Error **errp)
{
    BusState *bus = NULL;
    char *driver = qemu_opt_get(opts, "driver");

    /* find driver */
    DeviceClass *dc = qdev_get_device_class(&driver, errp);

    /* find bus */
    char *path = qemu_opt_get(opts, "bus");
    if (path != NULL) {
        bus = qbus_find(path, errp);
    } else if (dc->bus_type != NULL) {
        bus = qbus_find_recursive(sysbus_get_default(), NULL, dc->bus_type);
    }

    /* create device */
    DeviceState *dev = DEVICE(object_new(driver));
    /* add the dev as one of the bus's child && set bus as dev->parent_bus */
    qdev_set_parent_bus(dev, bus);

    qdev_set_id(dev, qemu_opts_id(opts));

    /* set properties */
    if (qemu_opt_foreach(opts, set_property, dev, &err)) {
        goto err_del_dev;
    }

    dev->opts = opts;

    /* call each device/driver's realize(), e.g. vfio_realize() */
    object_property_set_bool(OBJECT(dev), true, "realized", &err);
}
```

VFIO – qemu part: vfio_realize()

```
1. Get the container.
    /* Create a new container by open "/dev/vfio/vfio" each time. */
    container = open("/dev/vfio/vfio", O_RDWR);

    if (ioctl(container, VFIO_GET_API_VERSION) != VFIO_API_VERSION)
                /* Unknown API version handling*/
    if (!ioctl(container, VFIO_CHECK_EXTENSION, VFIO_X86_IOMMU))
                /* Doesn't support the IOMMU driver we want. */

2. Get the group, and add the group to the container.
    /* Get the sysfdev via `- device vfio_pci, host=xxxx:xx:xx.x` */
    vdev->vbasedev.sysfsdev =
          g_strdup_printf("/sys/bus/pci/devices/%04x:%02x:%02x.%01x",
          vdev->host.domain, vdev->host.bus, vdev->host.slot, vdev->host.function);

    /* Find the group_id: readlink(the above sysfsdev/iommu_group), e.i.
     * "/sys/kernel/iommu_groups/14" and basename() to get 14. Then open the group */
    group = open("/dev/vfio/14", O_RDWR);
    /* Test the group is viable and available, and some other check */
    ioctl(group, VFIO_GROUP_GET_STATUS, &group_status);

    /* Add the group to the container */
    ioctl(group, VFIO_GROUP_SET_CONTAINER, &container);

3. Set the IOMMU via container.
    /* Enable the IOMMU model we want */
    ioctl(container, VFIO_SET_IOMMU, VFIO_X86_IOMMU)
    /* Get addition IOMMU info */
    ioctl(container, VFIO_IOMMU_GET_INFO, &iommu_info);
```

VFIO – qemu part: vfio_realize() (Continue.)

```
4. Get the pci_dev fd, set the config space and 6 region bars, and the irq info.
    /* Get a file descriptor for the device */
    device = ioctl(group, VFIO_GROUP_GET_DEVICE_FD, "0000:02:00.0");
    /* Test and setup the device */
    ioctl(device, VFIO_DEVICE_GET_INFO, &device_info);

    for (i = 0; i < device_info.num_regions; i++) {
        struct vfio_region_info reg = { .argsz = sizeof(reg) };
        reg.index = i;
        ioctl(device, VFIO_DEVICE_GET_REGION_INFO, &reg);
        /* Setup mappings... read/write offsets, mmaps. */
        /* For PCI devices, config space is a region */
    }
    for (i = 0; i < device_info.num_irqs; i++) {
        struct vfio_irq_info irq = { .argsz = sizeof(irq) };
        irq.index = i;
        ioctl(device, VFIO_DEVICE_GET_IRQ_INFO, &reg);
        /* Setup IRQs... eventfds, VFIO_DEVICE_SET_IRQS */
    }

5. Map the device iova (e.g. [0, 1MB] as follows) with HPA
    /* In qemu, GPA is described by each root MemoryRegion's RAMBlock.
     * For each BAR region, its MR's RAMBlock->host is dma_map.vaddr (HVA) */
    dma_map.vaddr = mmap(0, 1024 * 1024, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    dma_map.size = 1024 * 1024;
    dma_map.iova = 0; /* 1MB starting at 0x0 from device view */
    dma_map.flags = VFIO_DMA_MAP_FLAG_READ | VFIO_DMA_MAP_FLAG_WRITE;
    ioctl(container, VFIO_IOMMU_MAP_DMA, &dma_map);

6. /* Gratuitous device reset and go... */
    ioctl(device, VFIO_DEVICE_RESET);
```

VFIO – qemu part: qemu_vfio_dma_map()

```
43 struct QEMUVFIOState {
44     QemuMutex lock;
45     /* These fields are protected by BQL */
46     int container, group, device;
47     struct vfio_region_info config_region_info, bar_region_info[6];
48
49     /* These fields are protected by @lock */
50     /* VFIO's IO virtual address space is managed by splitting into a few
51      * sections:
52      *
53      * ---------------       <= 0
54      * |xxxxxxxxxxxx|
55      * |-------------|       <= QEMU_VFIO_IOVA_MIN
56      * |             |
57      * |    Fixed    |
58      * |             |
59      * |-------------|       <= low_water_mark
60      * |             |
61      * |    Free     |
62      * |             |
63      * |-------------|       <= high_water_mark
64      * |             |
65      * |    Temp     |
66      * |             |
67      * |-------------|       <= QEMU_VFIO_IOVA_MAX
68      * |xxxxxxxxxxxx|
69      * |xxxxxxxxxxxx|
70      * ---------------
71      *
72      * - Addresses lower than QEMU_VFIO_IOVA_MIN are reserved as invalid;
73      *
74      * - Fixed mappings of HVAs are assigned "low" IOVAs in the range of
75      *   [QEMU_VFIO_IOVA_MIN, low_water_mark).  Once allocated they will not be
76      *   reclaimed - low_water_mark never shrinks;
77      *
78      * - IOVAs in range [low_water_mark, high_water_mark) are free;
79      *
80      * - IOVAs in range [high_water_mark, QEMU_VFIO_IOVA_MAX) are volatile
81      *   mappings. At each qemu_vfio_dma_reset_temporary() call, the whole area
82      *   is recycled. The caller should make sure I/O's depending on these
83      *   mappings are completed before calling.
84      **/
85     uint64_t low_water_mark;
86     uint64_t high_water_mark;
87     IOVAMapping *mappings;
88     int nr_mappings;
89 };
```

VFIO – kernel part

- related drivers
- container/group/device’s fops
- connects with pci: vfio_pci_driver
- DMA map via iommu_type1
- get io: config space + 6 bar
- msix interrupt: eventfd/irqfd

VFIO – kernel part: related drivers

1. vfio interface:

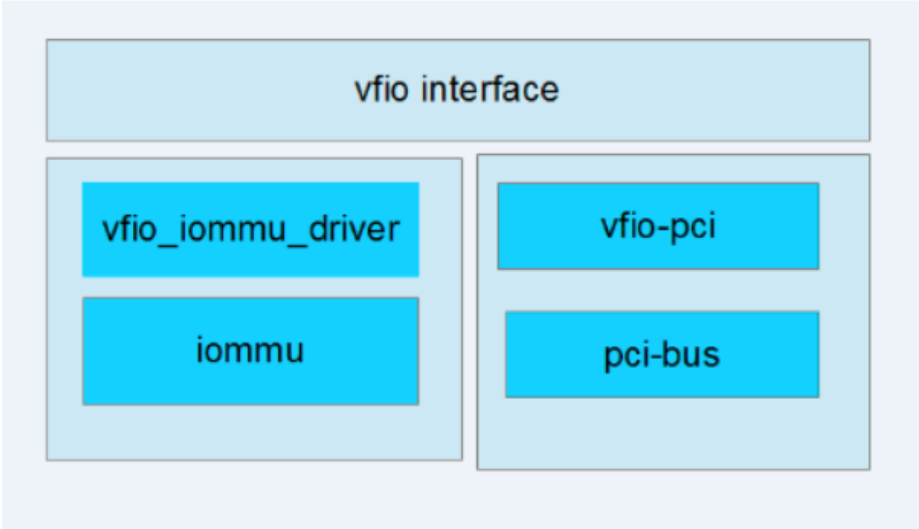
   Provide the common ioctl() for userspace, including container, group, device.

2. vfio_iommu_driver (we now use vfio_iommu_type1):

   Expose DMA for userspace, e.g. VFIO_IOMMU_MAP_DMA

3. vfio_pci_driver:

   Deploy the existed pci standards: to be a pci driver, attach to a pci bus, realize I/O and interrupts, etc.



VFIO – kernel part: container

1. container: via vfio_fops

```
1287 static const struct file_operations vfio_fops = {
1288         .owner          = THIS_MODULE,
1289         .open           = vfio_fops_open,
1290         .release        = vfio_fops_release,
1291         .read           = vfio_fops_read,
1292         .write          = vfio_fops_write,
1293         .unlocked_ioctl = vfio_fops_unl_ioctl,
1294 #ifdef CONFIG_COMPAT
1295         .compat_ioctl   = vfio_fops_compat_ioctl,
1296 #endif
1297         .mmap           = vfio_fops_mmap,
1298 };
```

```
2188 static struct miscdevice vfio_dev = {
2189         .minor = VFIO_MINOR,
2190         .name = "vfio",
2191         .fops = &vfio_fops,
2192         .nodename = "vfio/vfio",
2193         .mode = S_IRUGO | S_IWUGO,
2194 };
2195
2196 static int __init vfio_init(void)
2197 {
2198         int ret;
2199
2200         idr_init(&vfio.group_idr);
2201         mutex_init(&vfio.group_lock);
2202         mutex_init(&vfio.iommu_drivers_lock);
2203         INIT_LIST_HEAD(&vfio.group_list);
2204         INIT_LIST_HEAD(&vfio.iommu_drivers_list);
2205         init_waitqueue_head(&vfio.release_q);
2206
2207         ret = misc_register(&vfio_dev);
```

VFIO – kernel part: group

1. group: via vfio_group_fops

```
1636 static const struct file_operations vfio_group_fops = {
1637         .owner          = THIS_MODULE,
1638         .unlocked_ioctl = vfio_group_fops_unl_ioctl,
1639 #ifdef CONFIG_COMPAT
1640         .compat_ioctl   = vfio_group_fops_compat_ioctl,
1641 #endif
1642         .open           = vfio_group_fops_open,
1643         .release        = vfio_group_fops_release,
1644 };
```

```
2196 static int __init vfio_init(void)
2197 {
2198         ret = misc_register(&vfio_dev);
2199
2200         /* /dev/vfio/$GROUP */
2201         vfio.class = class_create(THIS_MODULE, "vfio");
2202
2203         vfio.class->devnode = vfio_devnode;
2204
2205         ret = alloc_chrdev_region(&vfio.group_devt, 0, MINORMASK, "vfio");
2206
2207         cdev_init(&vfio.group_cdev, &vfio_group_fops);
2208         ret = cdev_add(&vfio.group_cdev, vfio.group_devt, MINORMASK);
```

VFIO – kernel part: device

1. device: indirectly via group fd

   To allocate the fd and the struct file which has a anon inode.

```
1558        case VFIO_GROUP_GET_DEVICE_FD:
1559        {
1560                char *buf;
1561
1562                buf = strndup_user((const char __user *)arg, PAGE_SIZE);
1563                if (IS_ERR(buf))
1564                        return PTR_ERR(buf);
1565
1566                ret = vfio_group_get_device_fd(group, buf);
1567                kfree(buf);
1568                break;
1569        }
```

```
1480        filep = anon_inode_getfile("[vfio-device]", &vfio_device_fops,
1481                                   device, O_RDWR);
```

```
1715 static const struct file_operations vfio_device_fops = {
1716         .owner          = THIS_MODULE,
1717         .release        = vfio_device_fops_release,
1718         .read           = vfio_device_fops_read,
1719         .write          = vfio_device_fops_write,
1720         .unlocked_ioctl = vfio_device_fops_unl_ioctl,
1721 #ifdef CONFIG_COMPAT
1722         .compat_ioctl   = vfio_device_fops_compat_ioctl,
1723 #endif
1724         .mmap           = vfio_device_fops_mmap,
1725 };
```

VFIO – kernel part: vfio_pci_driver

1. __init vfio_pci_init() => pci_register_driver(&vfio_pci_driver)

=> bus_add_driver(drv) => __driver_attach() => really_probe()
=> dev→bus→probe()
[pci_bus_type.probe = pci_device_probe: pci_drv->probe()]
=> vfio_pci_probe

```
1190 static int vfio_pci_probe(struct pci_dev *pdev,
1191                            const struct pci_device_id *id)
1192 {
1193         group = vfio_iommu_group_get(&pdev->dev);
1194
1195         ret = vfio_add_group_dev(&pdev->dev, &vfio_pci_ops, vdev);
```

```
1315 static struct pci_driver vfio_pci_driver = {
1316         .name           = "vfio-pci",
1317         .id_table       = NULL, /* only dynamic ids */
1318         .probe          = vfio_pci_probe,
1319         .remove         = vfio_pci_remove,
1320         .err_handler    = &vfio_err_handlers,
1321 };
```

VFIO – kernel part: vfio_pci_driver (Continue)

```
805 int vfio_add_group_dev(struct device *dev,
806                         const struct vfio_device_ops *ops, void *device_data)
807 {
808         struct iommu_group *iommu_group;
809         struct vfio_group *group;
810         struct vfio_device *device;
811
812         iommu_group = iommu_group_get(dev);
813
814         group = vfio_group_get_from_iommu(iommu_group);
815         if (!group) {
816                 group = vfio_create_group(iommu_group);
817         } else {
818                 iommu_group_put(iommu_group);
819         }
820
821         device = vfio_group_get_device(group, dev);
822         if (device) {
823                 WARN(1, "Device %s already exists on group %d\n",
824                      dev_name(dev), iommu_group_id(iommu_group));
825                 ...
826                 return -EBUSY;
827         }
828
829         device = vfio_group_create_device(group, dev, ops, device_data);
```

VFIO – kernel part: DMA MAP via iommu_type1

1. vfio_fops_unl_ioctl => vfio_iommu_type1_ioctl
=> vfio_dma_do_map
[record (iova, vaddr) in rb_root dma_list of struct vfio_iommu]
=> vfio_pin_map_dma =>
{1.1 vfio_pin_pages_remote => vaddr_get_pfn => get_user_pages
=> get_user_pages_unlocked => __get_user_pages
=> handle_mm_fault
1.2 vfio_iommu_map(iova, pfn) => iommu_map()
=> domain→ops→map() [e.g. intel_iommu_map()]
=> domain_pfn_mapping() => __domain_mapping()
=> pfn_to_dma_pte()
}

VFIO – kernel part: get io e.g. config space + 6 bar

```
	} else if (cmd == VFIO_DEVICE_GET_REGION_INFO) {
		struct pci_dev *pdev = vdev->pdev;
		struct vfio_region_info info;
		struct vfio_info_cap caps = { .buf = NULL, .size = 0 };
		int i, ret;

		minsz = offsetofend(struct vfio_region_info, offset);

		if (copy_from_user(&info, (void __user *)arg, minsz))
			return -EFAULT;

		if (info.argsz < minsz)
			return -EINVAL;

		switch (info.index) {
		case VFIO_PCI_CONFIG_REGION_INDEX:
			info.offset = VFIO_PCI_INDEX_TO_OFFSET(info.index);
			info.size = pdev->cfg_size;
			info.flags = VFIO_REGION_INFO_FLAG_READ |
				     VFIO_REGION_INFO_FLAG_WRITE;
			break;
		case VFIO_PCI_BAR0_REGION_INDEX ... VFIO_PCI_BAR5_REGION_INDEX:
			info.offset = VFIO_PCI_INDEX_TO_OFFSET(info.index);
			info.size = pci_resource_len(pdev, info.index);
			if (!info.size) {
				info.flags = 0;
				break;
			}

			info.flags = VFIO_REGION_INFO_FLAG_READ |
				     VFIO_REGION_INFO_FLAG_WRITE;
			if (vdev->bar_mmap_supported[info.index]) {
				info.flags |= VFIO_REGION_INFO_FLAG_MMAP;
				if (info.index == vdev->msix_bar) {
					ret = msix_mmappable_cap(vdev, &caps);
					if (ret)
						return ret;
				}
			}

			break;
		case VFIO_PCI_ROM_REGION_INDEX:
```

VFIO – kernel part: vfio_pci_mmap for each bar

```
1106 static int vfio_pci_mmap(void *device_data, struct vm_area_struct *vma)
1107 {
1108         struct vfio_pci_device *vdev = device_data;
1109         struct pci_dev *pdev = vdev->pdev;
1110         unsigned int index;
1111         u64 phys_len, req_len, pgoff, req_start;
1112         int ret;
1113
1114         index = vma->vm_pgoff >> (VFIO_PCI_OFFSET_SHIFT - PAGE_SHIFT);
1115
1116         if (vma->vm_end < vma->vm_start)
1117                 return -EINVAL;
1118         if ((vma->vm_flags & VM_SHARED) == 0)
1119                 return -EINVAL;
1120         if (index >= VFIO_PCI_ROM_REGION_INDEX)
1121                 return -EINVAL;
1122         if (!vdev->bar_mmap_supported[index])
1123                 return -EINVAL;
1124
1125         phys_len = PAGE_ALIGN(pci_resource_len(pdev, index));
1126         req_len = vma->vm_end - vma->vm_start;
1127         pgoff = vma->vm_pgoff &
1128                 ((1U << (VFIO_PCI_OFFSET_SHIFT - PAGE_SHIFT)) - 1);
1129         req_start = pgoff << PAGE_SHIFT;
1130
1131         /* Even though we don't make use of the barmap for the mmap,
1132          * we need to request the region and the barmap tracks that. */
1133         if (!vdev->barmap[index]) {
1134                 ret = pci_request_selected_regions(pdev,
1135                                                    1 << index, "vfio-pci");
1136                 if (ret)
1137                         return ret;
1138
1139                 vdev->barmap[index] = pci_iomap(pdev, index, 0);
1140                 if (!vdev->barmap[index]) {
1141                         pci_release_selected_regions(pdev, 1 << index);
1142                         return -ENOMEM;
1143                 }
1144         }
1145
1146         vma->vm_private_data = vdev;
1147         vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
1148         vma->vm_pgoff = (pci_resource_start(pdev, index) >> PAGE_SHIFT) + pgoff;
1149
```

VFIO – kernel part: interrupt via eventfd/irqfd

If uses MSIX to do the interrupt:

=> vfio_pci_ioctl() [parse the struct vfio_irq_set assigned in qemu earlier: vfio_msix_vector_do_use()]

=> vfio_pci_set_irqs_ioctl() [to check use which irq: intx/msi/msix]

=> vfio_pci_set_msi_trigger() [if msi or msix]

=> vfio_msi_set_block() [if uses VFIO_IRQ_SET_DATA_EVENTFD]

=> vfio_msi_set_vector_signal()

[1. //get the eventfd_ctx via the event_fd registered in qemu
eventfd_ctx *trigger = eventfd_ctx_fdget(fd); //fd: next page

2. //allocate an interrupt line: irq for the vfio pci device

```
336         ret = request_irq(irq, vfio_msihandler, 0,
337                           vdev->ctx[vector].name, trigger);
```

3. //wait for the IRQ occurs, then call vfio_msihandler to handle.]

```
242 static irqreturn_t vfio_msihandler(int irq, void *arg)
243 {
244         struct eventfd_ctx *trigger = arg;
245         //Notify the userspace which waits on the epoll.
246         //Ref: qemu_set_fd_handler() to check the qemu part.
247         eventfd_signal(trigger, 1);
248         return IRQ_HANDLED;
249 }
```

(plus) VFIO – qemu part: register the vfio_irq_set

```
int argsz;
struct vfio_irq_set *irq_set;
int32_t *pfd;

argsz = sizeof(*irq_set) + sizeof(*pfd);

irq_set = g_malloc0(argsz);
irq_set->argsz = argsz;
irq_set->flags = VFIO_IRQ_SET_DATA_EVENTFD |
                 VFIO_IRQ_SET_ACTION_TRIGGER;
irq_set->index = VFIO_PCI_MSIX_IRQ_INDEX;
irq_set->start = nr;
irq_set->count = 1;
pfd = (int32_t *)&irq_set->data;

if (vector->virq >= 0) {
    *pfd = event_notifier_get_fd(&vector->kvm_interrupt);
} else {
    *pfd = event_notifier_get_fd(&vector->interrupt);
}

ret = ioctl(vdev->vbasedev.fd, VFIO_DEVICE_SET_IRQS, irq_set);
```

VFIO usage: how to passthrough a pci device

VFIO usage: how to passthrough a pci device

1. Enable VT-d in BIOS, and add `intel_iommu=on` in host kernel
2. Build vfio/iommu in kernel or as module
3. Insmod the vfio-pci.ko: `modprobe vfio-pci`
4. If the pci needs to be passthroughed has been bound to other host driver, unbind it firstly:

```
echo 0000:02:00.0 > /sys/bus/pci/devices/0000\:02\:00.0/driver/unbind
```

5. echo “vendor_id device_id” > /sys/bus/pci/drivers/vfio-pci/new_id

   PS: `lspci -n -s BDF` to check the vendor_id and device_id

```
linux-50ts:/build/gitcode # lspci -n -s 00:02.0
00:02.0 0300: 8086:191b (rev 06)
```

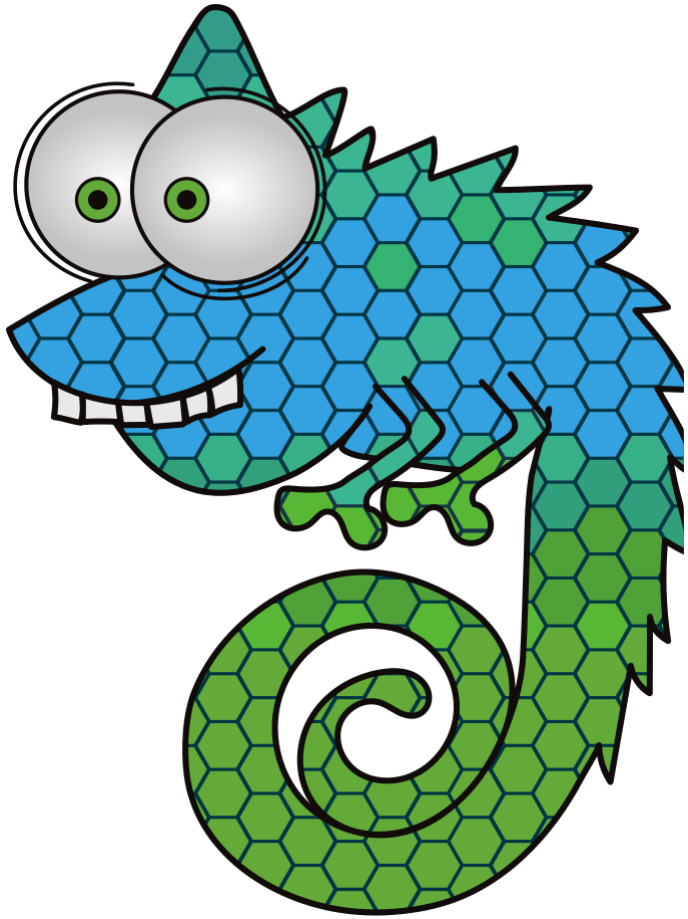
Reference

- https://www.kernel.org/doc/Documentation/vfio.txt
- https://lwn.net/Articles/509153/
- kernel source code
- qemu source code

Questions?

Join the conversation,
contribute & have a lot of fun!
www.opensuse.org

Thank you.



Have a Lot of Fun, and Join Us At:

www.opensuse.org

License

General Disclaimer

Credits

Template
Richard Brown
rbrown@opensuse.org

Design & Inspiration
openSUSE Design Team
http://opensuse.github.io/branding-guidelines/