virtio inside-out

CS695

Spring 2024-25

Plan

1. **Introduction to QEMU Execution Model**
    a. qemu_init()
    b. qemu_main_loop_wait()
    c. KVM_IOVENTFD handled in kernel (EPT misconfig -> eventfd_signal)
2. **Introducing VirtIO**
    a. Central idea from spec (managed shm between guest->host)
    b. Connect it to qemu and linux -> modern implementation of the spec
3. **QEMU + Linux implementation of virtio specification**
    a. Devices (via the QOM)
    b. Notifications (via mmio writes to regions with registered callbacks)
    c. Sending and receiving message buffers over a virtqueue
4. **Hands-On VirtIO**
    a. Walkthrough the virtio-demo-pci device. (A minimal virtio device)
        i. QEMU virtio-devices must follow the OOP paradigm.
        ii. Driver-device pair share buffer data structures
    b. Modify the device interface: send in two integers, return the product and the sum
    c. Add a new device --- block number-based in guest, file offset in virtio backend

QEMU Execution Model

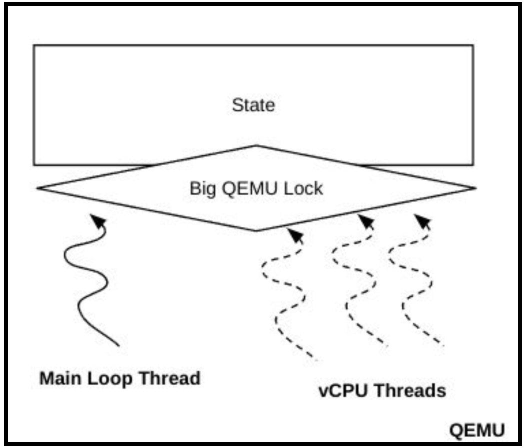
```
$>../qemu/build-0/qemu-system-x86_64  \
 -enable-kvm \
 -smp 8  -m 4096 \
 -nic user,model=virtio,hostfwd=tcp::2222-:22  \
 -drive file=ubuntu.qcow2,media=disk,if=virtio \
 -monitor stdio
```

A minimal QEMU command-line invocation.

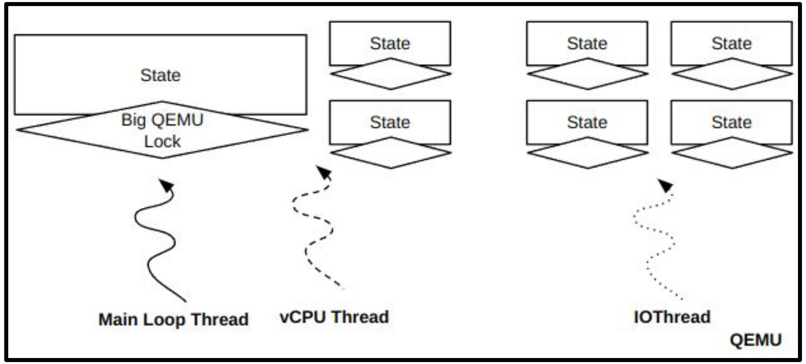
a. `qemu_init()`: parse command line args, create Machine State, register MMIO regions with r/w callbacks, init devices, spawns` vCPU threads that perform ioctl(KVM_RUN).

b. qemu_main_loop_wait(): Single thread, that polls file descriptors for IO events, blocking.

c. IOThreads: Optimization for increased parallelism with main loop.



Components of a QEMU process (2016)



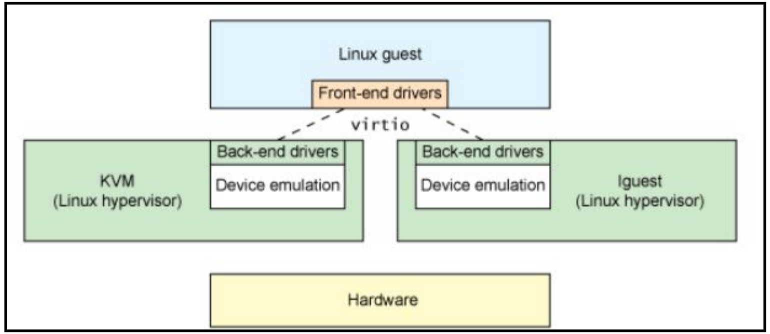
Components of a QEMU process (2024)

Introducing VirtIO

The VirtIO Specification enables the creation of a direct communication channel between a virtual device in a VM and a host-userspace hypervisor such as QEMU, through a shared, managed memory region called a VirtQueue.

- Device status field
- Feature bits
- Notifications
- Device Configuration space
- One or more virtqueues

Requirements of a VirtIO device, VirtIO spec 1.3. QEMU (and other hypervisors) closely follow the specification in their VirtIO implementations.

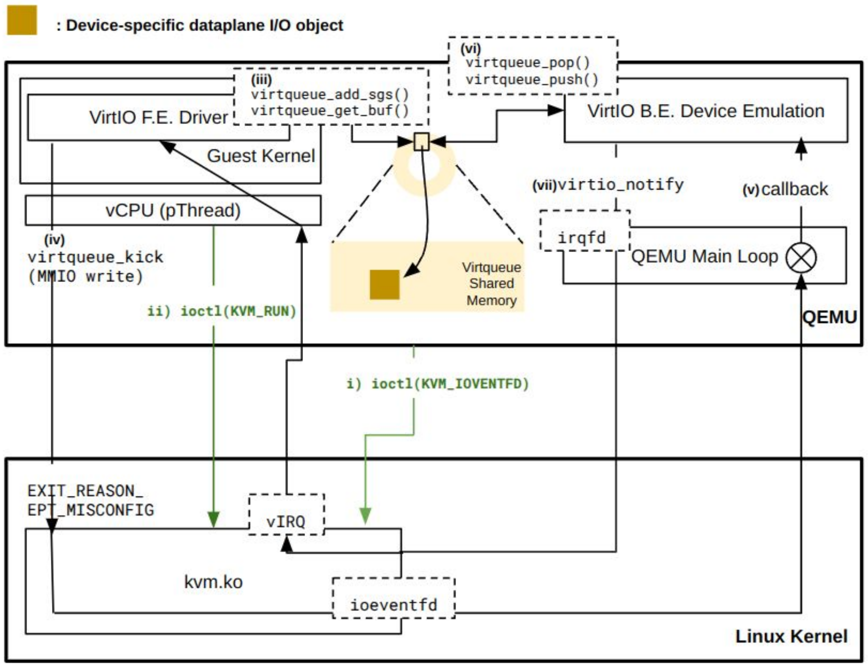


Logical Components of a VirtIO-based IO-virtualization solution. Backend is implemented in VMM (ex. QEMU). Frontend is a kernel module loaded into guest kernel

- The Linux Kernel and QEMU implement APIs to develop VirtIO drivers and devices.

QEMU Execution Model

Mechanics of VirtIO Device Emulation



a. F.E. driver adds request buffers to the `VirtQueue` and “kicks” the queue, sending a notification through KVM to a file-descriptor monitored by the QEMU Main Loop.

b. The main loop dispatches the vq handler (pre-registered at device realize time via `virtio_add_queue()`)

c. `IOEVENTFD/IRQFD` can be enabled for a device by setting the `VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT` as a property.

Hands-on VirtIO

A. virtio-demo-pci

The provided VM image and QEMU (after applying `virtio-demo-patch.patch`) has a complete implementation of a virtio device that exposes an ioctl interface to guest userspace.

This virtio-demo-pci device simply returns the product of two integers passed to it.

a. Attach the virtio-demo-pci device to your QEMU VM.

b. Load the `virtio-demo.ko` driver, run the user-space test program a few times.

c. <u>Modify:</u> For the two numbers passed to the device, return their sum and difference back to userspace.

```c
// test-demo-dev.c
demo_req_t *u_req = safe_malloc(sizeof(demo_req_t));
u_req->m.val_1 = 5;
u_req->m.val_2 = 7;

int ioctl_device_fd = open_device(DEVICE_FILE_PATH);

if (ioctl(ioctl_device_fd, IOCTL_MULTIPLY, u_req) < 0)
{
    error(e_FailedIoctlCommandV2P, errno, "IOCTL 0 failed.\n");
    exit(EXIT_FAILURE);
}
```

Hands-on VirtIO

Relevant Files

```
# git apply --stat virtio-demo-patch.patch
hw/virtio/Kconfig                            |    5 +
hw/virtio/meson.build                        |    2
hw/virtio/virtio-demo-pci.c                  |  152 +++++++++++++++++++++
hw/virtio/virtio-demo.c                      |  189 ++++++++++++++++++++++++++
hw/virtio/virtio-pci.c                       |    6 +
hw/virtio/virtio-qmp.c                       |   19 +++
hw/virtio/virtio.c                           |    3
include/hw/pci/pci.h                         |    1
include/hw/virtio/virtio-demo.h              |   44 ++++++
include/standard-headers/linux/virtio_demo.h |   12 ++
include/standard-headers/linux/virtio_ids.h  |    1
12 files changed, 435 insertions(+), 2 deletions(-)
```

Summary of changes to QEMU source to add a new VirtIO device

Writing a Frontend Driver for a VirtIO device

PCI Enumeration

```
bus: pci.0
      type PCI
      dev: virtio-xblk-pci, id "v0"
        disable-legacy = "on"
        disable-modern = false
        ioeventfd = true
        vectors = 2 (0x2)
        [...]
        class Class 00ff, addr 00:05.0, pci id 1af4:106b (sub 1af4:1100)
        bar 1: mem at 0xc0000000 [0xc0000fff]
        bar 4: mem at 0xc000008000 [0xc00000bfff]
        bus: virtio-bus
          type virtio-pci-bus
          dev: virtio-xblk-device, id ""
            in-order-buf = true
            test-feature-0 = true
            test-feature-1 = false
            indirect_desc = true
            event_idx = true
            notify_on_empty = true
            any_layout = true
            iommu_platform = false
            packed = false
            queue_reset = true
            in_order = true
            use-started = true
            use-disabled-flag = true
            x-disable-legacy-check = false
```

`(qemu) info qtree` after successful device realize. Above is how QEMU represents an attached PCI device. Observe that PCI BDF numbers and BAR regions are assigned by QEMU

a. On successful attachment of the new virtio device to the pci bus, QEMU sends an interrupt to the kernel.

b. This triggers kernel-space code to allocate necessary structures to represent the new PCI device.

c. The Kernel virtio subsystem initiates a virtio handshake with the virtio-device.

d. If a driver is found that matches the device <vendor id, device id> pair, they are linked and *feature negotiation* occurs.

e. Finally, control enters the F.E. driver and the driver's `.probe()` method is called.

For details see README in virtio-demo VM

Writing a Frontend Driver for a VirtIO device

The `.probe` method

```c
static struct virtio_driver virtio_demo_driver = {
  .feature_table = features, // desired guest features
  .feature_table_size = ARRAY_SIZE(features),
  .driver.name = KBUILD_MODNAME,
  .id_table = id_table, // to bind with device
  .probe = virtio_demo_probe,
  .remove = virtio_demo_remove,
};
```

```c
static int virtio_demo_probe(struct virtio_device *vdev)
{
  pr_info("virtio-demo FE probing...\n");

  // @priv: private pointer for the driver's use. (virtio.h)
  v695 = kzalloc(sizeof(*v695), GFP_KERNEL);
  if (!v695)
    return -ENOMEM;

  // point to each other.
  // v695 contains vdev
  vdev->priv = v695;
  v695->vdev = vdev;

  v695->vq = virtio_find_single_vq(vdev, virtio_demo_inbufs_cb, "demo-bidirectional");

  /* from this point on, the device can notify and get callbacks */
  // performs mmio write to device , setting the status to VIRTIO_CONFIG_S_DRIVER_OK
  // i.e. informing the device that the driver is ready to handle interrupts.
  virtio_device_ready(vdev);

  u64 features = vdev->config->get_features(vdev);

  // (44) 0001 0000 0001 0011 0000 0000 0000 0000 0000 0000 0010
  // (44) 0001 0000 1001 0011 0000 0000 0000 0000 0000 0000 0010 #35 set -> IN_ORDER
  pr_info("Virtio features: 0x%llx\n", features);

  [...]
```

Simplest possible virtio device probe. `find_vq()` maps virtqueue interrupt lines to MSI-X vectors exposed by the device. `virtio_device_ready()` completes the VirtIO handshake.

`virtio_find_vq_*`():

a. Create a virtqueue.

b. Compute MSI-X addresses for interrupts and write to device MSI-X capability struct. (qemu<->vm vq interrupt mapping)

c. *Registers a user-provided callback as the interrupt handler for that virtqueue's interrupt vector.*

A device may have several vqs, each is initialized in .probe

Writing a Frontend Driver for a VirtIO device

Reading and writing to VirtQueues

Virtio drivers use the `scatter-gather kernel API` to create descriptors that reference arbitrary structs on kernel heap or stack.

```
static void
do_multiply_vq(demo_req_t *req) {

  struct scatterlist sg_out, sg_in, *sgs[2]; // outbuf, inbuf

  sg_init_one(&sg_out, &(req->m), sizeof(req->m));
  sgs[0] = &sg_out;

  sg_init_one(&sg_in, &(req->res), sizeof(req->res));
  sgs[1] = &sg_in;

  // Key: out_sgs must be before the in_sgs in the sgs list.
  // see virtqueue_add_sgs() -> virtqueue_add_split() in linux kernel.
  virtqueue_add_sgs(v695->vq, sgs, 1, 1, req, GFP_ATOMIC);

  // only kick the outbuf vq
  virtqueue_kick(v695->vq); // calls demo_read_outbuf() in qemu
```

Passing an outgoing request to the device (`sg_out`), and registering a location to store the response (`sg_in`).

a. **To write a request to a virtqueue:**
Use `sg_init_one()` to convert your request outbuf to a scatter-gather entry (s-g entry). Similarly for a response inbuf. Finally, `virtqueue_add_sgs(*, token)` will create virtio descriptor entries and update the management metadata.

b. **To retrieve a response from a virtqueue**
Use `virtqueue_get_buf()`

```
// virtio-demo.c (driver)
static void virtio_demo_inbufs_cb(struct virtqueue *vq)
{
  pr_info("cb reached\n");
  demo_req_t *req;
  int len;
  if ((req = virtqueue_get_buf(vq, &len)) != NULL)
  {
    pr_info("Response received in callback");
    v695->mult_res = req->res;
    complete(&(v695->req_done));
  }
}
```

Writing a VirtIO Backend for QEMU

Build-system Changes

Step 0: Add a config option for your new device and add new meson build rules

```
// qemu/hw/virtio/Kconfig
config VIRTIO_XBLK
    bool
    default y
    depends on VIRTIO
```

```
// qemu/hw/virtio/Kconfig
config VIRTIO_PCI
    bool
    default y if PCI_DEVICES
    depends on PCI
    select VIRTIO
    select VIRTIO_MD_SUPPORTED
```

```
// qemu/hw/virtio/meson.build
[...]
specific_virtio_ss.add(when: 'CONFIG_VIRTIO_XBLK', if_true: files('virtio-xblk.c'))
[...]
virtio_pci_ss.add(when: 'CONFIG_VIRTIO_XBLK', if_true: files('virtio-xblk-pci.c'))
[..]
```

For details read the virtio inside-out doc

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 1: Get QEMU to recognize the name of your PCI device.

```
// in qemu/virtio/virtio-xblk-pci.c
#define TYPE_VIRTIO_XBLK_PCI "virtio-xblk-pci-base"
static const VirtioPCIDeviceTypeInfo virtio_demo_pci_info = {
    .parent = TYPE_VIRTIO_PCI, // see virtio-pci.h. Allows reg on the pci bus.
    .base_name = TYPE_VIRTIO_XBLK_PCI,
    .generic_name = "virtio-xblk-pci",
    .transitional_name = "virtio-xblk-pci-transitional",
    .non_transitional_name = "virtio-xblk-pci-non-transitional",
};

// Boilerplate to register a qemu type
static void virtio_demo_pci_register(void) {
    virtio_pci_types_register(&virtio_demo_pci_info);
}

type_init(virtio_demo_pci_register);
```

Creating a TypeInfo struct for your new virtio device.
`type_init()` adds to a type-table. `$QEMU -device help` queries this table.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 2: Setup a PCI-bindings struct to allow your device to attach to the Virtio-PCI bus

```c
// qemu/hw/virtio/virtio-xblk-pci
typedef struct VirtIOXBlkPCI VirtIOXBlkPCI;

struct VirtIOXBlkPCI {
    VirtIOPCIProxy parent_obj; // required acc. to qemu docs
    VirtIOXBlk vdev;           // link to an instance of your backend device
};


#define TYPE_VIRTIO_XBLK_PCI "virtio-xblk-pci-base"
```

Linkage between the PCI bindings and the virtio device backend.

- `VirtIOXBlk` is an instance of the TYPE_VIRTIO_XBLK_DEVICE class.
- `VirtIOXBlkPCI` is an instance of the TYPE_VIRTIO_XBLK_PCI class.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 3. Define a set of properties for your device

```
// qemu/hw/virtio/virtio-blk-pci.c
static Property virtio_xblk_pci_properties[] = {
    DEFINE_PROP_BIT("ioeventfd", VirtIOPCIProxy, flags,
                    VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT, true), // enable the ioventfd mechanism
    DEFINE_PROP_UINT32("vectors", VirtIOPCIProxy, nvectors,
                       DEV_NVECTORS_UNSPECIFIED),
    DEFINE_PROP_END_OF_LIST(),
};
```

Defining properties for the virtio bindings class. Every `VirtIOPCIProxy` object has these properties. Here we set them to desired values.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 4: Define a struct to represent your device backend

```
// qemu/include/hw/virtio/virtio-xblk.h
#define TYPE_VIRTIO_XBLK "virtio-xblk-device" // name of backend

// create the VIRTIO_XBLK() macro that can obtain reference to a
// struct VirtIOXblk from an `Object`.
OBJECT_DECLARE_SIMPLE_TYPE(VirtIOXBlk, VIRTIO_XBLK)

// Define your device-backend struct
struct VirtIOXBlk {
    // Check the hierarchy in the doc,
    // each custom virtio device must inherit from the
    // TYPE_VIRTIO_DEVICE i.e. VirtIODevice class.
    VirtIODevice parent_obj;
    VirtQueue *vq;            // declare a single virtqueue
    uint64_t host_features;   // necessary by virtio spec
};
```

Device backend struct definition. The one shown is the simplest possible QEMU-compliant backend.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 5: Define #.class_init() method to create a useable OOP class from TypeInfo struct

```c
// qemu/hw/virtio/virtio-xblk-pci.c
static void virtio_xblk_pci_class_init(ObjectClass *klass, void *data) {
    // create direct references to all the
    // heirarchy classes to avoid an
    // obj->parent->parent chain.
    // when assigning properties
    DeviceClass *dc = DEVICE_CLASS(klass);
    PCIDeviceClass *pcidev_k = PCI_DEVICE_CLASS(klass);
    VirtioPCIClass *k = VIRTIO_PCI_CLASS(klass);

    // Register the properties
    device_class_set_props(dc, virtio_xblk_pci_properties);

    k->realize = virtio_xblk_pci_realize;
    [...]
```

The class_init() defines the attributes and methods that this upcoming OOP class has. In effect, it is a C++ class definition.

```c
// qemu/hw/virtio/virtio-blk-pci.c
static Property virtio_xblk_pci_properties[] = {
    DEFINE_PROP_BIT("ioeventfd", VirtIOPCIProxy, flags,
                    VIRTIO_PCI_FLAG_USE_IOEVENTFD_BIT, true), // enable the ioventfd mechanism
    DEFINE_PROP_UINT32("vectors", VirtIOPCIProxy, nvectors,
                       DEV_NVECTORS_UNSPECIFIED),
    DEFINE_PROP_END_OF_LIST(),
};
```

```c
// This function is called by QEMU main code, starting from qdev_device_add()
// and triggers the realization of the device backend,
// - Realization could involve several virtio-specific actions such
// as initializing virtqueues an their notification handlers.
static void virtio_xblk_pci_realize(VirtIOPCIProxy *vpci_dev, Error **errp) {
    VirtIOXBlkPCI *dev = VIRTIO_XBLK_PCI(vpci_dev);

    // not sure why cast to devicestate is valid
    DeviceState *vdev = DEVICE(&dev->vdev);

    vpci_dev->class_code = PCI_CLASS_OTHERS;

    // nvectors sets the number of MSI-X vectors supported
    // by this device. Subsequent virtqeueue <-> vector
    // mappings are done through a driver device handshake
    if (vpci_dev->nvectors == DEV_NVECTORS_UNSPECIFIED) {
        vpci_dev->nvectors = 2;
    }

    // @imp: eventually calls the realize of the backend device
    qdev_realize(vdev, BUS(&vpci_dev->bus), errp);
}
```

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 6: Enable successful device “attach” to the Virtio-PCI bus

```
// qemu/hw/virtio/virtio-xblk-pci.c

// QEMU cli_create_devices() triggers a instance creation
// for the device type passed. Here also, the idea is to create a
// object of .vdev type linked to the pci bindings object.
static void virtio_xblk_pci_instance_init(Object *obj) {
    VirtIOXBlkPCI *dev = VIRTIO_XBLK_PCI(obj);

    // creates the virtio backend for the virtio-bus
    virtio_instance_init_common(obj, &dev->vdev, sizeof(dev->vdev),
                                TYPE_VIRTIO_XBLK);

}
```

Step 1a of enabling device attach - `#.instance_init()`. Any additional properties that were assigned during class_init will be a part of the `dev->vdev` instance.

```
(qemu) device_add
virtio-xblk-pci,id=v0,disable-legacy=on
```

```
error:
virtio_instance_init_common():
unknown type 'virtio-xblk-device'
```

- `instance_init()` for the `.vdev` fails since device backend is not yet a registered QEMU Type.
- Need to create file virtio-xblk.c with the backend TypeInfo

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 6: Enable successful device "attach" to the Virtio-PCI bus

```
//qemu/hw/virtio/virtio-xblk.c
static const TypeInfo virtio_xblk_info = {
    .name = TYPE_VIRTIO_XBLK,

    // The device backend object will only inherit attributes
    // and methods of the DEVICE_CLASS. (no pci stuff)
    // and VIRTIO_DEVICE class.
    .parent = TYPE_VIRTIO_DEVICE,
    .instance_size = sizeof(VirtIOXBlk),
    .instance_init = virtio_xblk_instance_init,
    .class_init = virtio_xblk_class_init,
};

// QEMU device boilerplate to register type
static void virtio_register_types(void)
{
    type_register_static(&virtio_xblk_info);
}

type_init(virtio_register_types)
```

```
(qemu) device_add
virtio-xblk-pci,id=v0,disable-legacy=on
```

Detour: New backend file added to build. We now have `virtio-xblk-pci.c + virtio-xblk.c`

Step 1b of enabling device attach: Defining a new TypeInfo struct for the "virtio-xblk-device" i.e. the device backend.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 6: Enable successful device “attach” to the Virtio-PCI bus

```
(qemu) device_add
virtio-xblk-pci,id=v0,disable-legacy=on
```

```c
// qemu/hw/virtio/virtio-xblk-pci.c
// This function is called by QEMU main code, starting from qdev_device_add()
// and triggers the realization of the device backend,
// - Realization could involve several virtio-specific actions such
// as initializing virtqueues an their notification handlers.
static void virtio_xblk_pci_realize(VirtIOPCIProxy *vpci_dev, Error **errp) {
    VirtIOXBlkPCI *dev = VIRTIO_XBLK_PCI(vpci_dev);

    // not sure why cast to devicestate is valid
    DeviceState *vdev = DEVICE(&dev->vdev);

    vpci_dev->class_code = PCI_CLASS_OTHERS;

    // nvectors sets the number of MSI-X vectors supported
    // by this device. Subsequent virtqeueue <-> vector
    // mappings are done through a driver device handshake
    if (vpci_dev->nvectors == DEV_NVECTORS_UNSPECIFIED) {
        vpci_dev->nvectors = 2;
    }

    // @imp: eventually calls the realize of the backend device
    qdev_realize(vdev, BUS(&vpci_dev->bus), errp);
}
```

reduces to

```c
// qemu/hw/virtio/virtio-xblk.c
// minimal backend device realize
static void virtio_xblk_device_realize(DeviceState *dev, Error **errp)
{
    VirtIODevice *vdev = VIRTIO_DEVICE(dev);
    VirtIOXBlk *s = VIRTIO_XBLK(dev);
    virtio_init(vdev, VIRTIO_ID_XBLK, 0); // can reserve non-zero config-size
    s->vq = virtio_add_queue(vdev, 64, NULL);
}
```

Step 2 of enabling device attach - #.realize() -> qdev_realize().

- The `vpci_dev->bus` was set after `#.instance_init()` but before `#.realize()` during qemu init.

Writing a VirtIO Backend for QEMU

Adding core device functionality

Step 6: Enable successful device “attach” to the Virtio-PCI bus

```
// GOAL: Setup the methods of the parent (VirtioDeviceClass)
// of this virtio-demo-device class. All these methods will
// be inherited by the child.
static void virtio_xblk_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    VirtioDeviceClass *vdc = VIRTIO_DEVICE_CLASS(klass);

    // can add additional class props as well.
    // props must be part of your device struct.
    device_class_set_props(dc, virtio_xblk_properties);

    set_bit(DEVICE_CATEGORY_MISC, dc->categories);

    vdc->realize = virtio_xblk_device_realize; // needed, else QEMU assert fail
    vdc->unrealize = virtio_xblk_device_unrealize;
    vdc->get_features = virtio_xblk_device_get_features; // needed, else QEMU assert fail (virtio-bus.c)
}
```

```
// qemu/hw/virtio/virtio-xblk.c
static Property virtio_xblk_properties[] = {
    DEFINE_PROP_BIT64("in-order-buf", VirtIOXBlk, host_features,
                      VIRTIO_F_IN_ORDER, true),
    DEFINE_PROP_BIT64("test-feature-0", VirtIOXBlk, host_features,
                      VIRTIO_F_XBLK_TEST_0, true),
    DEFINE_PROP_BIT64("test-feature-1", VirtIOXBlk, host_features,
                     VIRTIO_F_XBLK_TEST_1, false),
    DEFINE_PROP_END_OF_LIST(),
};
```

```
static uint64_t virtio_xblk_device_get_features(VirtIODevice *vdev, uint64_t features,
                                                Error **errp)
{
    printf("requesting features\n");
    VirtIOXBlk *dev = VIRTIO_XBLK(vdev);
    features |= dev->host_features;
    // virtio_add_feature(&features, VIRTIO_F_IN_ORDER);
    return features;
}
```

Step 3 of enabling device attach: Ensure the #.class_init of the device backend defines #.realize() and #.get_features() methods. This ensures that `qdev_realize()` from the pci bindings file succeeds.

On successful attach, the new virtio-pci device is listed as a child of the virtio-pci bus.

- `(qemu) info qtree` can be used to inspect buses and realized devices.

Writing a VirtIO Backend for QEMU

Reading and writing to VirtQueues

The `VirtqueueElement` is the representation of a message in the virtqueue. It contains references to the actual sg_in and sg_out, i.e. the driver-initialized sg buffers meant to be respectively written to and read from by the device.

a. **To retrieve a request from a virtqueue, use `virtqueue_pop()`**
This function returns a `VirtqueueElement.` Subsequent access to the inbufs and outbufs occur through the `in_sg` and `out_sg` attributes.

b. **To write a response to a virtqueue, use `virtqueue_push()`**
For a device writing a response to an inbuf, the popped virtqueue element #.in_sg can be filled using `iov_from_buf()`, and subsequently the element can be pushed back the virtqueue.

So Far:

- Enabled users to `realize` (attach) a custom Virtio device to the virtio-pci bus of a compatible QEMU machine: (default is pc-i440fx-9.2)

Pause

Next Steps:

- Understand `VirtIO` device interactions from the guest's perspective.

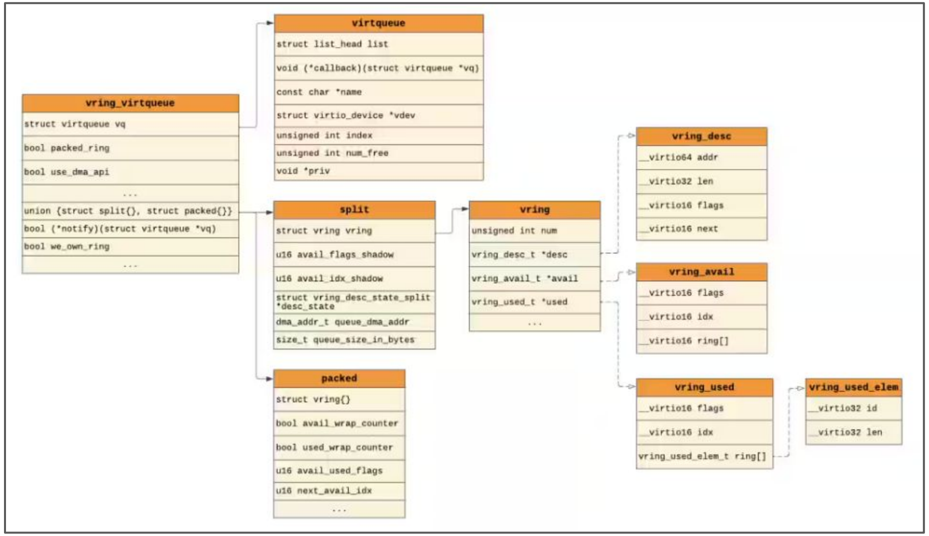
Appendix

The VirtQueue

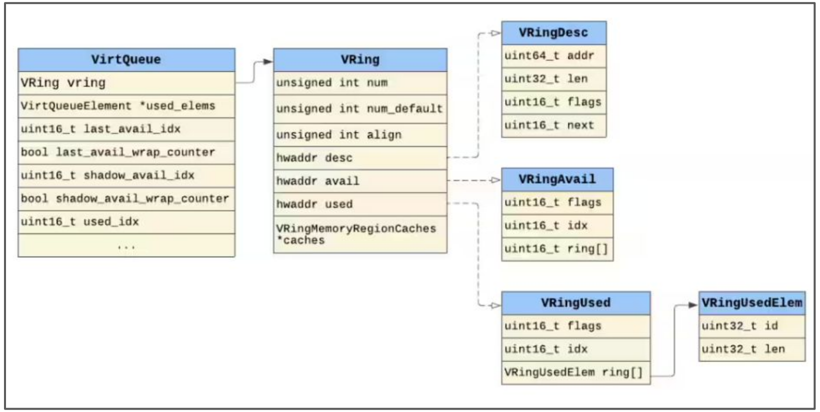
a. A nested data structure that implements a `VRing`, located in a shared page in guest physical memory.

b. Three `VRings` per `VirtQueue`: Descriptor Ring, Available Ring and Used Ring.

c. Descriptor Ring entries point to data buffers, Available and Used Ring entries manage VirtQueue metadata.



VirtQueue implementation: Linux Kernel



VirtQueue implementation: QEMU

The Linux Kernel and QEMU both implement APIs for developing VirtIO drivers and devices.

Appendix

The VirtQueue API

Linux Kernel
Full API reference in `$kernel/virtio.h`

a. **`sg_init_one()`**
Given a kernel-allocated physically contiguous region (storing custom struct, array of ints etc.), *create a scatter-gather list containing 1 scatter-gather (sg) entry.*

- Each sg entry is now linked to a fixed memory region in kernel space.

b. **`virtqueue_add_sgs()`**
Given a list of sg entries for outgoing data (outbufs), empty buffers to store any device-written data (inbufs), and an associated `token`, this function:

- Create descriptor chains, stored in the descriptor ring of the specified virtqueue and
- Update the available ring according to virtio spec.

A single sg entry would create a single descriptor

c. **`virtqueue_get_buf()`**
Look up the used ring, update metadata. Finally, return a pointer to the updated driver `token` that was registered via `virtqueue_add_sgs`().

d. **`virtqueue_kick()`**
Given a virtqueue in guest kernel, send a notification to the corresponding virtqueue in the backend.

Appendix

The VirtQueue API

QEMU
Full API reference in `qemu/include/hw/virtio.h`

a. `iov_from_buf()`
Adds a device buffer to the specified sg-entry. The device backend will have written directly to the location in shared memory of the guest's response inbuf through the DMA mapping created by the virtio subsystem.

b. `iov_to_buf()`
Copy data in a VirtQueueElement to a device buffer.

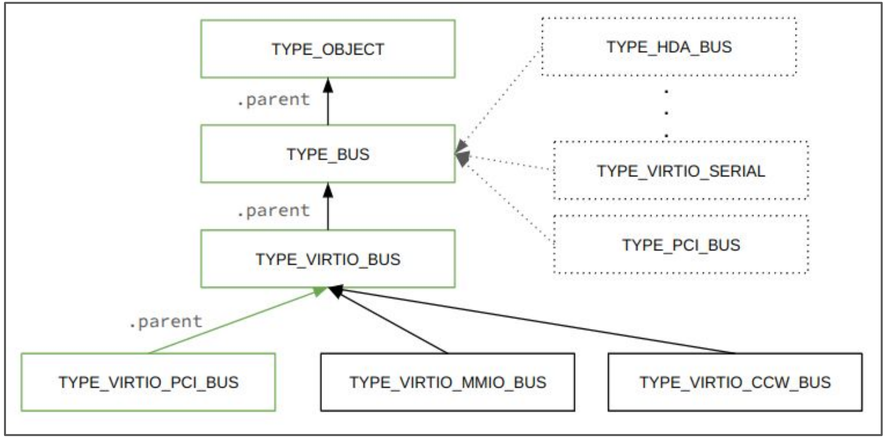
c. `virtqueue_pop()`
Retrieve the next VirtQueueElement in the virtqueue.

d. `virtqueue_push()`
Place a VirtQueueElement onto the virtqueue

e. `virtio_notify()`
Send an interrupt to the virtqueue mapped to the current.

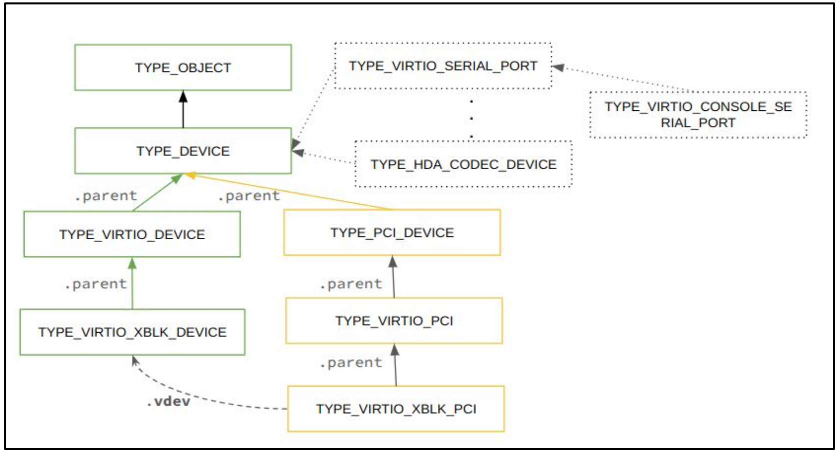
Appendix

QOM Representation of Virtio Device



QEMU's `TypeInfo` hierarchy for Buses. The Bus hierarchy for VirtIO devices over PCI is highlighted.

- All VirtIO devices are attached to the VIRTIO_PCI_BUS.
- *The BUS enables device<->machine communication.*



QEMU's `TypeInfo` hierarchy of Device backends and PCI transport bindings.

- The PCI bindings .c file is used to define an association between the TYPE_VIRTIO_XBLK_PCI and the TYPE_VIRTIO_XBLK_DEVICE classes.
- *The PCI bindings enable device<->guest communication.*

The class hierarchies are recursively built during qemu init by the #.class_init() method of each `type_init()`ed Type.

Hands-on VirtIO

B. virtio-xblk-pci

Apply the `virtio-xblk-helper-patch.patch` to your QEMU directory. This will generate a few empty source files corresponding to a new **virtio-xblk-pci** device.

a. Implement `virtio-xblk-pci.c` and `virtio-xblk.c` and successfully attach the virtio-xblk-pci device to your VM. (hint: use the virtio-demo patch to discover additional necessary changes)

b. (extra) Write a Frontend Driver for the `virtio-xblk-pci` device.

Thank You