VIRTIO DRIVER IMPLEMENTATION

2016-06-06

Virtio

Virtio is a standard for implementing device drivers on a virtual machine. Normally, a host would emulate a device like a rtl8139 network card, the VM would detect such a device and would load a driver for it. But this adds a lot of overhead because the host is emulating that device so it must translate the data it receives into a format that is understood by the real hardware. The more complicated the emulated device it, the more challenge it will be for the host to keep latency low.

Virtio solves this problem by letting the host expose a Virtio device. A virtio device is a fake device that will be used by the VM. Virtio devices are very simple to use compared to other real hardware devices. For example, a host may implement a Virtio network card. The VM would detect such a device and start using it as its network card. Of course, the end-user wouldn't really notice this. The simplicity of the device is seen by the device driver implementers.

So to use Virtio, the host must support it. Currenly, KVM does. Then the guest must install the appropriate device drivers. Virtio device drivers are included in the linux kernel already, so there is no need to download separate drivers. On windows, drivers must be downloaded separately.

Virtio can be seen as a two layer device architecture. The first layer is the communication layer between the host and the guest. This is how both exchange information to say "Here's a packet I want you to send on the real hardware" or "Here's a packet I just received from the real hardware". Note that the driver knows it is running in a virtual environment and can implement optimizations in that effect. But the rest of the OS, using the Virtio driver, doesn't know that. It only knows that it is using a network card with a driver like any other ones. Every Virtio device drivers communicate with the host using the same model. This means that the code for this layer can be shared between all Virtio drivers.

The second layer is the protocol used over the first layer. Every virtio device use a different protocol. For example, a virtio-net driver will speak differently than a virtio-block driver to the guest. But they would both convey the messages the same way to the host.

My OS

The reason I was interested in virtio was because my hobby operating system required some device drivers to work. I had already written an ATA driver and a netcard driver (rtl8139) but those are old devices and I wanted to learn something new anyway. By having implemented virtio drivers in my OS, I should, technically, be able to run my OS on any host that support virtio. I don't need to worry about developing several device drivers because different hosts support different hardware. If virtio because a widely accepted standard (maybe it is already), then my OS should be fine on all different hosts. Note that I will still need to implement several different drivers if I want to support my OS on real hardware. But running it in a VM for now is just fine.

My code

These are the drivers. Note that, without the full code of my OS, these drivers won't make much sense but I'm putting them here in case someone could use it as an example whenever trying to write such a driver.
virtnet.c
virtblock.c
virtio.c
virtio.h

Information

Implementing the virtio drivers was very simple. I was able to do so by using only two sources of information:

- The virtio spec
- And this introduction on osdev

Implementation

I'm going to describe the implementation using pseudo-code and will skip some of the basic details. Things like PCI enumeration will be left out since it is out of the scope of this document.

Device setup

Pci enumeration

The first thing to do is to discover the device on the PCI bus. You will be searching for a device with vendor ID 0x1AF4 with device ID between 0x1000 and 0x103F. The subsystemID will indicate which type of device it is. For example, susbsystem ID 1 is a netcard. So after finding device on the PCI bus, you will obtain the base IO address and IRQ number. You can then proceed to attaching your device driver interrupt handler to that IRQ and setup the device using the IObase address..

```
foreach pci_dev
   if pci_dev.vendor == 0x1AF4 && pci_dev.device >= 0x1000 && pci_dev.device <= 0x103F && pci_dev.subsystem == 1
   {
      return [pci_dev.iobase, pci_dev.irq];
   }
```

Init sequence

The device initialization is very well described in the spec so there is no need to go into much details here. Here is the sequence:

```
//Virtual I/O Device (VIRTIO) Version 1.0, Spec 4, section 3.1.1:  Device Initialization

// Tell the device that we have noticed it
OUTPORTB(VIRTIO_ACKNOWLEDGE,iobase+0x12);
// Tell the device that we will support it.
OUTPORTB(VIRTIO_ACKNOWLEDGE | VIRTIO_DRIVER,iobase+0x12);

// Get the features that this device supports. Different host may implement different features
// for each device. The list of device-specific features can be found in the spec
INPORTL(supportedFeatures,iobase+0x00);

// This is called the "negotiation". You will negotiate, with the device, what features you will support.
// You can disable features in the supportedFeatures bitfield. You would disable
// features that your driver doesn't implement. But you cannot enable more features
// than what is currently specified in the supportedFeatures.
negotiate(&supportedFeatures);
OUTPORTL(supportedFeatures,iobase+0x04);

// Tell the device that we are OK with those features
OUTPORTB(VIRTIO_ACKNOWLEDGE | VIRTIO_DRIVER | VIRTIO_FEATURES_OK,iobase+0x12);

// Initialize queues
init_queues();

c |= VIRTIO_DRIVER_OK;
OUTPORTB(VIRTIO_ACKNOWLEDGE | VIRTIO_DRIVER | VIRTIO_FEATURES_OK,iobase+0x12);
OUTPORTB(c,dev->iobase+0x12);
```

The init_queues() function will discover all available queues for this device and initialize them. These queues are the core communication mechanism of virtio. This is what I was refering as the first layer. I will go in more details about queues a bit later. For now, to discover the queues, You just need to verify the queue size for each queue. If the size is not zero, then a queue exist. Queues are addressed with a 16bit number.

```
q_addr = 0
size = -1
while (size != 0)
{
   // Write the queue address that we want to access
   OUTPORTW(q_addr,iobase+0x08)
   // Now read the size. The size is not the byte size but rather the element count.
   INPORTW(size,iobase+0x12)

   if (size > 0) init_queue(q_addr, size)
   q_addr++
}
```

For each queue, you must prepare a rather large structure containing information about the queue and slots for buffers to send in the queue. The structure is created in memory (anywhere you want, as long as it sits on a 4k boundary) and the address will be given to the device driver. I find that the structure that is detailed in the spec is a bit confusing because the structure can't really be defined as a struct since it has many elements that must be dynamically allocated since their size depends on the queue size.

Field	Format	Size
Buffer Descriptors	u64 address; u32 length; u16 flags; u16 next;	queue_size
Available buffers header	u16 flags; u16 index;	1
Available buffers	u16 rings	queue_size
Padding to next page	byte	variable
Used buffers header	u16 flags; u16 index;	1
Used buffers	u32 index; u32 length;	queue_size

This is how I create the structure in memory

```
typedef struct
{
   u64 address;
   u32 length;
   u16 flags;
   u16 next;
} queue_buffer;

typedef struct
{
   u16 flags;
   u16 index;
   u16 rings[];
} virtio_available;

typedef struct
```


HOME AUTOMATION SYSTE
EZFLORA IRRIGATION SYST
SUMP PUMP MONITORING
BUILDING A HOSTED MAILSERVER SERVICE
I AM NOW HOSTING MY OW DNS AND MAIL SERVERS ON AMAZON EC2
DEPLOYING A LAYER3 SWIT ON MY NETWORK
ACD SERVER WITH RESIPROCATE
C++ JSON LIBRARY
IMPLEMENTING YOUR OWN MUTEX WITH CMPXCHG
WAKEUPCALL SERVER USIN RESIPROCATE
FFT ON AMD64
CLONING A HARD DRIVE
CONFIGURING AND USING KVM-QEMU
USING COUCHDB
INSTALLING COUCHDB ON SLACKWARE
NGW100 MY OS AND EDXS/L
NGW100 – MY OS
ASTERISK FILTER APPLICATION
CISCO ROUTER CONFIGURATION
AASTRA 411 XML APPLICATI
SPA941 PHONEBOOK
SPEEDTOUCH 780 DOCUMENTATION
AASTRA CONTACT LIST XMI APPLICATION
AVR32 OS FOR NGW100
ASTERISK SOUND INJECTIO APPLICATION
NGW100 – DIFFERENT PROBLEMS AND SOLUTIONS
AASTRA PRIME RATE XML APPLICATION
SPEEDTOUCH 780 CONFIGURATION
USING COUCHDB WITH PHP
AVR32 ASSEMBLY TIP
AP7000 AND NGW100 ARCHITECTURE
AASTRA WEATHER XML APPLICATION
NGW100 – GETTING STARTE
AASTRA ALI XML APPLICAT

```
{
    u32 index;
    u32 length;
} virtio_used_item;

typedef struct
{
    u16 flags;
    u16 index;
    virtio_used_item rings[];
} virtio_used;

typedef struct
{
    queue_buffer* buffers;
    virtio_available* available;
    virtio_used* used;
} virt_queue;

init_queue(index, queueSize)
    u32 sizeofBuffers = (sizeof(queue_buffer) * queueSize);
    u32 sizeofQueueAvailable = (2*sizeof(u16)) + (queueSize*sizeof(u16));
    u32 sizeofQueueUsed = (2*sizeof(u16))+(queueSize*sizeof(virtio_used_item));
    u32 queuePageCount = PAGE_COUNT(sizeofBuffers + sizeofQueueAvailable) + PAGE_COUNT(sizeofQueueUsed);
    char* buf = kernelAllocPages(queuePageCount);
    u32 bufPage = buf >> 12;

    vq->buffers = (u64)buf;
    vq->available = (virtio_available*)&buf[sizeofBuffers];
    vq->used = (virtio_used*)&buf[((sizeofBuffers + sizeofQueueAvailable+0xFFF)&~0xFFF)];
    vq->next_buffer = 0;

    // Tell the device what queue we are working on
    OUTPORTW(index,iobase+0x0E);

    // Now we have to tell the device what is the page number (of the physical address, not logical) of the structure
    // for that queue
    OUTPORTL(bufPage,iobase+0x08);

    vq->available->flags = 0;
```

The communication layer

The way the driver talks to the device is by placing data in a queue and notifying the device that some data is ready. Data is stored in a dynamically allocated buffer. The buffer's physical address is then writen to the first free buffer descriptor in the queue. Buffers can be chained, but forget about that now (it will be usefull when you want to optimize). Then, you need to tell the device that a buffer was placed in the queue. This is done by writing the buffer index into the next free slot in the "available" array.

BTW: it's important to know that queue sizes will always be powers of 2. Making it easy to naturally wrap around, so you never need to take care of checking bounds.

```
    // Find next free buffer slot
    buf_index = 0;
    foreach desc in vq->buffers
        if desc.length == 0
            buf_index = index of this descriptor in the vq->buffers array
            break

    // Add it in the available ring
    u16 index = vq->available->index % vq->queue_size;
    vq->available->rings[index] = buffer_index;
    vq->available->index++;

    // Notify the device that there's been a change
    OUTPORTW(queue_index, dev->iobase+0x10);
```

Once the device has read your data, you should get an interrupt. You would then check the "used" ring and clear any used descriptors in vq->buffers that are referenced by the "used" ring (ie: set lenght back to 0)

To receive data, you would do it almost the same way. You would still place a buffer in the queue but you would set its lenght to the max size that you are expecting data (512bytes for a block device for example, or MTU for a net device). Then you would monitor the "used" ring to see when the buffer has been used by the device and filled up.

The transport interface

With this information, you should be able to write a generic virtio transport layer that provides 3 functions:

- init()
- send_buffer()
- receive_buffer()

The virtio-net implementation

MAC address

The MAC address can be found in the 6 bytes at iobase+0x14..0x19. You must access those bytes one by one.

To send a packet out, you need to create a buffer that contains a "net_header" and the payload. For simplicity, we'll assume that no buffer chaining is done. So sending a packet would be done like this:

```
typedef struct
{
    u8 flags;
    u8 gso_type;
    u16 header_length;
    u16 gso_size;
    u16 checksum_start;
    u16 checksum_offset;
} net_header;

send_packet(payload,size)
    char buffer[size+sizeof(net_header)];
    net_header* h = &buffer;
    h.flags = VIRTIO_NET_HDR_F_NEEDS_CSUM;
    h.gso_type = 0;
    h.checksum_start = 0;
    h.checksum_offset = size;
    memcpy(buffer[sizeof(net_header)],payload,size)
    virtio_send_buffer(buffer,size+sizeof(net_header));
```

To receive packets, just fill up the rx queue with empty buffers (with lenth=MTU) and set them all available. It's important to set them back available after you received data in it (ie: they've been added in the used ring) since you want to keep the queue full of ready buffers at all time.

I didn't talk about buffer chaining (it's very simple, and well described in the spec) but you should obviously use that. You could use one buffer for the header and another one for the data. You could use the address of the data buffer supplied by the calling function in the descriptor directly (as long as you convert it to physical address) instead of copying the entire frame. This allows you to implement a zero-copy mechanism.

The virt-block implementation

Block devices are similar to net device but they use one queue only and instead of a net_header, they use a block_header

```
typedef struct
{
    u32 type;
    u32 reserved;
    u64 sector;
} block_header;
```

To write, fill the header with type = 1, sector = sector number to write. Followed by the 512 bytes of data and send the buffer. To read, fill the header with type = 0, sector = sector number to read. Followed by a 512 bytes empty buffer. The device will fill the buffer and will put the buffer descriptor in the used ring.

I think you need to separate the header and the data buffer into 2 descriptors that are chained. That's the way I did it anyway, but I think I read that it won't work if you don't do that.

Conclusion

This was a very rough explanation of virtio but it should be enough to get you started and have something working. Once this is done, I suggest going through the specs again since it has a lot of information what will be needed for handling failure scenarios, optmization and multi-platform support. The driver I wrote works for my OS only and has only been tested with KVM. I am not doing any real feature negoiation nor am I handling any failure cases. Things could surely be optmized also since virtio allows very easy zero-copy buffer passing.
