Virtio Block Device Driver

Haiyang Han
Northwestern University
HaiyangHan2020@u.northwestern.edu

Zhebin Shen
Northwestern University
ZhebinShen2016@u.northwestern.edu

ABSTRACT
Virtio is a virtualization standard for network and disk device drivers where just the guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor. In this project, we set out to build a driver for a Virtio block device for the Nautilus AeroKernel. In the end our driver can achieve the functionality of writing to an attached block disk. Although functions such as reading from the block disk is still incomplete.

CODE REPOSITORY
https://github.com/haiyang1992/NautilusVirtio

1. INTRODUCTION
In a nutshell, Virtio is an abstraction layer of I/O devices in a paravirtualized hypervisor which means Virtio is a virtualization standard for network and disk device drivers where just the guest's device driver "knows" it is running in a virtual environment, and cooperates with the hypervisor.

There are 2 major virtualization paradigm nowadays. The full virtualization vs. paravirtualization. In full virtualization, the guest operating system runs on top of a hypervisor that sits on the bare metal. The guest is unaware that it is being virtualized and requires no changes to work in this configuration. Conversely, in paravirtualization, the guest operating system is not only aware that it is running on a hypervisor but includes code to make guest-to-hypervisor transitions more efficient.

In the full virtualization environment, the hypervisor must enumerate the hardware, which is emulating at the lowest level. This kind of emulation is clean and easy to use for upper layer operating system but is also an inefficient and highly complicated way; while under the paravirtualization scheme, the operating system is aware of the existence of the virtual device which requires modifications in operating system (the downside of this approach) but make the emulating of device more efficient.

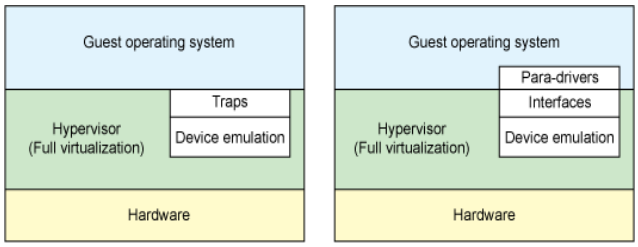

Figure 1. Device emulation in full virtualization and paravirtualization environments

The modifications of operating system to support paravirtualized device are where the Virtio comes in. There are some other Virtio alternatives in the planet, Xen provides paravirtualized device driver and VMware also provides what they called Guest tools in their suites. As illustrated in figure 1, a typical paravirtualized I/O device consists of 2 parts: 1) para-driver with in the guest OS; 2) a Virtio device interface and actual implementations on hypervisor which performs the efficient I/O operations in the real world. As for this project, only para-drivers are involved.

This project is aimed at building a simple block device driver via Virtio for the Nautilus, an extremely light weighted kernel framework for parallel runtime systems. Nautilus is basically a light weighted kernel optimized for parallel computing and with the support of Virtio it will achieve high IO performance (the purpose of this kernel) when running on a hypervisor.

2. BACKGROUND
2.1 Virtio
Virtio was first introduced by IBM system lab and be used to address a series of efficient, well-maintained Linux driver which can be adapted to several different hypervisors as well as guest operating systems. Virtio is built on the top of PCI interface which lead to lots operational overlap with PCI devices. The modification to an operating system to support paravirtualized device for performance reason is hard and there are lots of repeated but unavoidable work for different guest OS to support different VMM's paravirtualized device, while the Virtio provide both the guest OS (the driver) and the VMM (the virtual device) a standard way of interaction and made the modification of operating system to be minimal and be supported by various kinds of VMMs. In short Virtio created a general virtual I/O mechanism which is efficient, works on multiple hypervisor and platforms.

To reduce the duplication in the virtual device driver, good abstraction for each layer are important such that drivers can share code. The first abstraction here is Virtio configuration. Typically a Virtio device driver register themselves to handle a specific 32 device type the driver's' probe function is called when a suitable device is found and Virtio configuration data structure are passed in such that the driver can do the configuration jobs. The configuration operation consists of 4 ops, 1) reading and writing feature bits; 2) reading and writing the configuration space; 3) reading and writing the status; 4) reset the device. Another thing is configuration space which associate themselves with each device's own specific information. The information above is about the configuration APIs used in Virtio devices.

A more performance related abstraction layer is about transport abstraction. This part involved the actual I/O mechanism. The abstraction in this part is called the virtqueue and used for device to transport large trunk of data between device and OS. Each virtqueue occupies at least 2 continuous 4K pages and can be divided into three parts as below. The number for different device are different. For a block device only one virtqueue is used while a network device typically used two virtqueue, one for receiving data, another for sending data. A virtqueue can be viewed as a queue for guest to add buffer and host to consume buffer. Each buffer is a scatter-gather array of data which depends on which type of device are used.

Table 1. Layout for a virtqueue

Descriptor Table	Available Ring (with padding)	Used Ring

A normal work flow for virtqueue to work is show as below: driver wants to send a buffer to the device, it will fit in a slot in Descriptor table (or chain several together) and write the descriptor index into the available ring. Then it will notify the device (kick); when the device finished a buffer, it writes the descriptor into the used ring and give host an interrupt. Here the word virtqueue is the abstract idea to explain how transport layer works while a virtio_vring is actually one of the implementation of the idea of virtqueue which would be described in detail later.

2.2 Nautilus

Nautilus is the first example of an AeroKernel that is available for public use and development. AeroKernels are extremely lightweight OS kernels that are intended to support the Hybrid Runtime (HRT) model, in which a parallel runtime enjoys full access to the entire machine. An AeroKernel like Nautilus provides a minimal set of functionality to the runtime. As indicated above, Nautilus now is really a tiny OS and the support for Virtio will definitely improve the I/O performance under VMMs which lead to overall performance increase.

3. OUR VIRTIO BLOCK DRIVER

3.1 Structures of files added

The general Virtio data structures, functions and some of their declarations are in virtio_pci.c and virtio_pci.h. The Virtio ring related structures are in virtio_ring.h. The Virtio block device specific structures and functions are in virtio_block.c and virtio_block.h.

3.2 Virtio ring

We defined the data structures for the virtq, descriptor table, used ring and available ring the same way as the appendix of the Virtio specification document.

3.3 Virtio PCI

We wrote some functions and changed a few functions provided in the patch.

- free_descriptor(): we made the function recursive so that it could free a descriptor chain, starting from the head node.
- virtio_enque_request(): this function is in charge of queueing one buffer into the virtqueue. We made sure that we only updated the available ring (with the index of the head node) and the available index once for each descriptor chain.
- virtio_dequeue_response(): within the Virtio framework, our thought is that the last_seen_used variable from the driver's perspective, the last element it saw in the used ring and the used index denotes the current last element in the ring. As a result, the driver should process the buffers until the last_seen_used catches up with the used index. We also deleted the callback function, because it was not needed. We could locate the address of the buffers by looking into the used ring, which gives us the buffer's location in the descriptor table.
- virtio_block_handler(): this function is the interrupt handler of the Virtio block device. Its main job is to read the ISR status register, which sets it to 0, and calls the blockrq_dequeue() function to start the dequeue process.
- virtio_block_init(): this function registers the virtio_block_handler with interrupts with the id 228, activates the device. We also put our test code in this function, creating a write and a read block request, queueing them up, and finally notifying the device by writing the queue number (0 for a block device) in the QUEUE_NOTIFY register.
- bringup_device(): We moved the code for writing the DEVICE_STATUS register and the GUEST_FEATURES register to this function, to make

sure that they are written before the virtqueue is initialized.

- process_descriptor(): this function is called by virtio_dequeue_response() and is in charge of determining the type of buffer in the ring to process, whether it be a read, write, or flush and calling the corresponding processing function. It also checks the status of used buffers to make sure the requests returned from the device were OK.
- read_request_process(): it recursively goes through a block read buffer and tries to output the data read from the disk.
- write_flush_request_process(): this function is used to process write and flush requests. In fact, this function doesn't actually do anything. But we put it there in case of future modifications.
- check_status(): this is called by process_descriptor() and tries to locate the last descriptor in a descriptor buffer chain, where the status field is modified by the device, and checks if the status of the request was OK.

3.4 Virtio block

We defined the data structure of a block request as the following:

```
struct virtio_block_request{
  /* read */
  #define VIRTIO_BLK_T_IN       0
  /* write */
  #define VIRTIO_BLK_T_OUT      1
  /* flush */
  #define VIRTIO_BLK_T_FLUSH    4
  uint32_t type;
  uint32_t priority;
  uint64_t sector;
  unsigned char data[512];

  /* success */
  #define VIRTIO_BLK_S_OK       0
  /* device or driver error */
  #define VIRTIO_BLK_S_IOERR    1
  /* request unsupported by device */
  #define VIRTIO_BLK_S_UNSUPP   2
  uint8_t status;
};
```

We also defined the block device interface and a block device but didn't have time to implement the details.

- blockrq_enqueue(): this function takes in an array of block requests and adds them, one by one into the virtqueue, giving each enqueue the necessary information so that the flags and next fields in the descriptor can be set according to the type of request as well as its relative position within a descriptor chain.
- blockrq_dequeue(): this is called by the interrupt handler, which calls the virtio_dequeue_response() for each ring the device has. In our case, ring 0.

3.5 Flow of supplying a buffer to the device

To supply a request to the Virtio block device, the flow of operations is as follows:

1) Place the requests into the free entries in the descriptor table, for each free descriptor d: set the d.addr to the either the address of the header, data, or status fields; set d.len to the size of the data we are pointing to; set d.flags to VIRTQ_DESC_F_WRITE if the descriptor needs to be written by the device, otherwise we set it to VIRTQ_DESC_F_NEXT to show that the descriptors chain via the next field; finally d.next is set to the index of the next descriptor in the chain.
2) Update the available ring: we write the index of the head descriptor into the next free entry in the available ring and increment the available index by 1 for each chain enqueued.
3) A memory barrier is executed to ensure the device sees the updated descriptor table and available ring before the next step.
4) We kick the device by writing the ring index (in the Virtio block device case, 0) into the QUEUE_NOTIFY register.

3.6 Flow of receiving a buffer from the device

To receive a request from the Virtio block device, the flow of operations is as follows:

1) The dequeue process begins when we receive an interrupt from the device via the interrupt handler.
2) Disable interrupt on the ring by writing VIRTQ_AVAIL_F_NO_INTERRUPT into the flags field of the available ring.
3) Check the last_seen_used variable and the index of the used ring to determine if there are buffers in the used ring waiting to be processed by the handler.
4) Execute a memory barrier and check again to handle the case where, after the last check and before enabling interrupts, an interrupt has been suppressed by the device.
5) From the elements in the used ring, we can get the index of the head of the descriptor chain of the used buffer.
6) For each descriptor chain, we first check the status in the tail descriptor to make sure the request was completed by the device. Then, for a read request, we try to output the data field; and for write and flush requests, we don't do anything. Finally we free the descriptors by recursively setting their lengths to 0.
7) We increment the last_seen_used by 1 for each chain we processed.

4 EFFORTS, DIFFICULTIES, AND ACCOMPLISHMENTS

4.1 Getting started

- Understanding concepts

We looked through a couple of example Virtio driver source codes but they provided little help for us due to their complexity and high level of abstraction. We also found the Virtio documents difficult to read due to the large number of concepts presented to us in the early stages of the project. As time went on and we got familiar with the concepts, we were able to focus on coding and debugging the driver itself.

- Keep in mind that writing a driver

In the early stages of the project, we didn't quite grasp what it meant to write a driver and were not clear about exactly what we were to do. Later on we realized that following the established standards in the specifications is important in writing a driver because the device can only behave as intended if the driver follows the same protocols.

4.2 Virtio_pci_vring and virtq

We were confused about the structures virtio_pci_vring and virtq and how to manipulate them. It seemed to me that there were two ways to access the virtqueue. The virtq makes descriptor table, available ring and used ring easier to access from a driver's perspective. Whereas the virtio_pci_vring captures additional information needed for the driver code, but not used (or visible to) the device. Its address is stored in the virtio_pci_dev structure so we can allocate and deallocate memory space at initialization and de-initialization.

4.3 Writing to the disk

There were several challenges while we tried to write to a disk image file.

- qemu reports missing block header error

Previously each of our write requests only took up one entry in the descriptor table, and we would get a runtime error from qemu, "missing block header". This was solved by providing at least three descriptors in a chain to the device, the header, data and status. If the device sees fewer than 3 descriptors in a descriptor chain, it reports the above error.

- Idx of the used ring does not increase, no interrupt

qemu does not report errors now, but we cannot see the used ring index increment, which is what we should observe after notifying the device of the write request. We solved this by looking through the document again. Our mistake was having the available ring store every entry in the descriptor table. By having the available ring store the heads of the descriptor chains only, we were able to see the used index increment accordingly and get a response from the device as well.

- Write not getting through, status field doesn't change

At this stage, we could observe that the device has processed the requests. However when looking at the disk image files, we were not able to see any actual write taking place. It turned out that the address field in our descriptors should not point to the start of block requests, but point to the fields inside. For example, if we have a blockrq[3] array, previously we would have desc[i]->blockrq[i]. The correct way is to have desc[0]->blockrq[0].type, desc[1]->blockrq[1].data, and desc[2]->blockrq[2].status. Otherwise the device would not be able to process the requests correctly. Finally we were able to see the changes being made in a disk image after a write request.

4.4 Processing used ring

We spent a significant amount of time figuring out relationship between used.idx and last_seen_used. The Virtio document specified that we check if used.idx != last_seen_used in the interrupt handler. However when we implemented this we were never able to reach the code to process the request because the two values would be different to begin with and it would just skip the code for processing used requests and updating the last_seen_used variable. So we changed the != to >=, with the idea that we would process the waiting used buffers until what we last checked is equal to the current used index. However the used ring index might wrap around after a large of requests taking place and our current code does not address that issue.

4.5 Reading from the disk

- Status wrong(second buffer gives a status of OK)

We tried to enqueue a descriptor chain containing three descriptors, with the data part set to NULL. When we process the returned buffers, the status field in the last descriptor would have a value other than 0, 1, 2, which is not part of the specification. However the status bit in the second buffer does give a value of OK.

- Used element->len is 1

For each used element in the used ring, we can read the len value and get the number of bytes the device has written. For a write, that value is what we expected (length of data + status). But for read, we only get 1, which probably means that only the status field (which is exactly 1 byte in size) is modified by the device, and the data field is left untouched. This is confusing. Maybe Virtio block read requests are designed to include only two buffers? But where has the read data gone to? That is something left to find out in the future.

- Reading data field gives blank characters

Naturally, when we try to output the data field in the finished request buffers, we can only print blank characters, indicating that although the read request itself might be successful. We cannot really read data from the Virtio block device yet.

5. LIMITS AND FUTURE WORK

5.1 Getting read to work

Figures 2 shows the status of the used ring when we receive an interrupt from the device after a write and a read request. The number of bytes written by the device in the read is only 1. Figure 3 shows the data inside a read request after it has been processed by the device. Notice the data does not contain any value.

```
DEBUG: used[0] = 0 with length 201
DEBUG: used[1] = 3 with length 1
DEBUG: used[2] = 0 with length 0
DEBUG: used[3] = 0 with length 0
DEBUG: used[4] = 0 with length 0
```

Figure 2. Debug information of used ring

```
DEBUG: VIRTIO_PCI: DEBUG: Processing read request
VIRTIO_PCI: Data is at: 302e20
VIRTIO_PCI: Data is:
VIRTIO_PCI: Data is at: 303248
VIRTIO_PCI: Data is:
```

Figure 3. Data of returned read request.

Our project has been relatively unsuccessful with reading from the disk image. Getting the read function to work properly would be a future priority.

5.2 Block interface

We currently do not have a functional interface for reading and writing disks for the operating system. We also did not finish on implementing a general block device structure.

5.3 Advanced features

Some of the advanced features in Virtio block devices, such as SCSI commands are not supported.

5.4 Descriptor chaining

In our current implementation, a descriptor i can only chain to descriptor i+1. This poses a problem when we want to wrap around the descriptor table, having the last entry point to the first entry. There is also the problem of fragmentation when the total number of free descriptors is sufficient for us to enqueue a long descriptor chain but there is no contiguous space in the descriptor table to fit it. In this case we would need a descriptor be able to point to any other descriptor in the table, instead of only to the next entry in the table.

5.5 Testing performance

The goal of Virtio is to increase the performance of guest operating systems running on VMMs. We have yet to test the performance of writing to block devices to see if Virtio really provides faster operations compared to conventional block device drivers, where the OS is unaware that it is running on a virtual platform.

6. Conclusion

In this project we built a partially functional Virtio block device driver for the Nautilus kernel. It is able to write to a virtual block device but unable to perform a complete read operation. This was our first time working with device drivers, and we gained quite a bit of experience during the process, reading specification documents, experiencing with low-level code, and debugging under a kernel environment.

7. Reference

[1] Jones, M. (2010, January 29). Virtio: An I/O virtualization framework for Linux. Retrieved from http://www.ibm.com/developerworks/library/l-virtio/

[2] Rusty Russell. 2008. Virtio: towards a de-facto standard for virtual I/O devices. SIGOPS Oper. Syst. Rev. 42, 5 (July 2008), 95-103. DOI=http://dx.doi.org/10.1145/1400097.1400108

[3] Virtio. (n.d.). In *OSDev.org*. Retrieved from http://wiki.osdev.org/Virtio

[4] **[VIRTIO-v1.0]** *Virtual I/O Device (VIRTIO) Version 1.0*. Edited by Rusty Russell, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. 29 October 2015. OASIS Committee Specification Draft 05 / Public Review Draft 05. http://docs.oasis-open.org/virtio/virtio/v1.0/csprd05/virtio-v1.0-csprd05.html. Latest version: http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html.

[5] FreeBSD kernel VIRTIO device code Documentation. Retrieved from http://www.leidinger.net/FreeBSD/dox/dev_virtio/html/index.html

[6] Yang, J. (2014, October 27). virtio guest side implementation: PCI, virtio device, virtio net and virtqueue. Retrieved from https://jipanyang.wordpress.com/2014/10/27/virtio-guest-side-implementation-pci-virtio-device-virtio-net-and-virtqueue/

[7] Russel, R. (2012, May 7). Virtio PCI Card Specification v0.9.5 DRAFT. Retrieved from http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf