Article

Virtio: An I/O virtualization framework for Linux

Paravirtualized I/O with KVM and lguest

Save Like

Site feedback

By M. Jones

Published 28 January 2010

In a nutshell, `virtio` is an abstraction layer over devices in a paravirtualized hypervisor. `virtio` was developed by Rusty Russell in support of his own virtualization solution called `lguest`. This article begins with an introduction to paravirtualization and emulated devices, and then explores the details of `virtio`. The focus is on the `virtio` framework from the 2.6.30 kernel release.

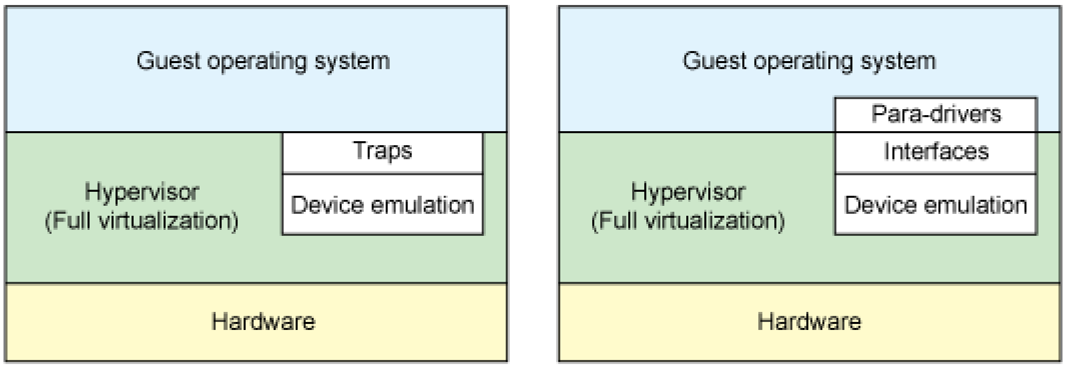
Linux is the hypervisor playground. Linux offers a variety of hypervisor solutions with different attributes and advantages. Examples include the Kernel-based Virtual Machine (KVM), `lguest`, and User-mode Linux. Having these different hypervisor solutions on Linux can tax the operating system based on their independent needs. One of the taxes is virtualization of devices. Rather than have a variety of device emulation mechanisms (for network, block, and other drivers), `virtio` provides a common front end for these device emulations to standardize the interface and increase the reuse of code across the platforms.

Full virtualization vs. paravirtualization

Let's start with a quick discussion of two distinct types of virtualization schemes: full virtualization and 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 (see Figure 1).

In the full virtualization scheme, the hypervisor must emulate device hardware, which is emulating at the lowest level of the conversation (for example, to a network driver). Although the emulation is clean at this abstraction, it's also the most inefficient and highly complicated. In the paravirtualization scheme, the guest and the hypervisor can work cooperatively to make this emulation efficient. The downside to the paravirtualization approach is that the operating system is aware that it's being virtualized and requires modifications to work.

Figure 1. Device emulation in full virtualization and paravirtualization environments



Hardware continues to change with virtualization. New processors incorporate advanced instructions to make guest operating systems and hypervisor transitions more efficient. And hardware continues to change for input/output (I/O) virtualization, as well (see resources on the right to learn about Peripheral Controller Interconnect [PCI] passthrough and single- and multi-root I/O virtualization).

But in traditional full virtualization environments, the hypervisor must trap these requests, and then emulate the behaviors of real hardware. Although doing so provides the greatest flexibility (namely, running an unmodified operating system), it does introduce inefficiency (see the left side of Figure 1). The right side of Figure 1 shows the paravirtualization case. Here, the guest operating system is aware that it's running on a hypervisor and includes drivers that act as the front end. The hypervisor implements the back-end drivers for the particular device emulation. These front-end and back-end drivers are where `virtio` comes in, providing a standardized interface for the development of emulated device access to propagate code reuse and increase efficiency.

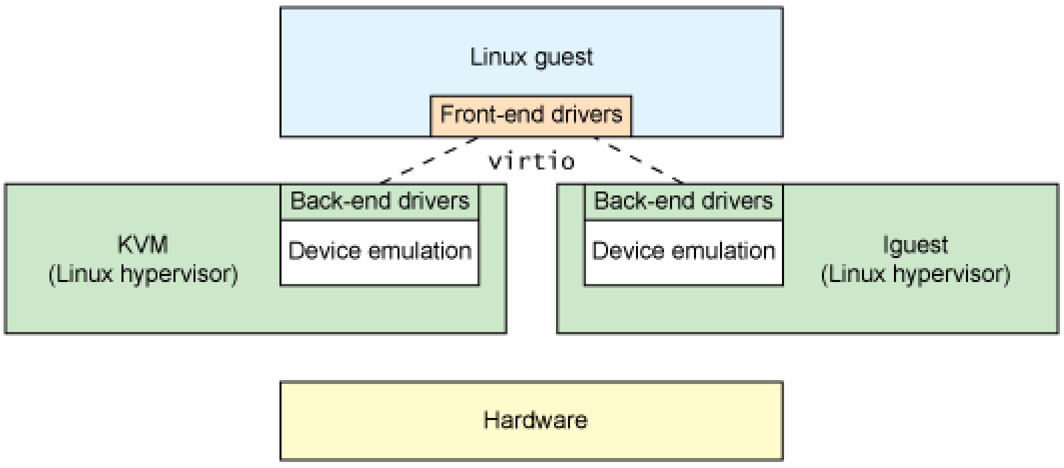
Virtio alternatives

`virtio` is not entirely alone in this space. Xen provides paravirtualized device drivers, and VMware provides what are called *Guest Tools.*

An abstraction for Linux guests

From the previous section, you can see that `virtio` is an abstraction for a set of common emulated devices in a paravirtualized hypervisor. This design allows the hypervisor to export a common set of emulated devices and make them available through a common application programming interface (API). Figure 2 illustrates why this is important. With paravirtualized hypervisors, the guests implement a common set of interfaces, with the particular device emulation behind a set of back-end drivers. The back-end drivers need not be common as long as they implement the required behaviors of the front end.

Figure 2. Driver abstractions with virtio



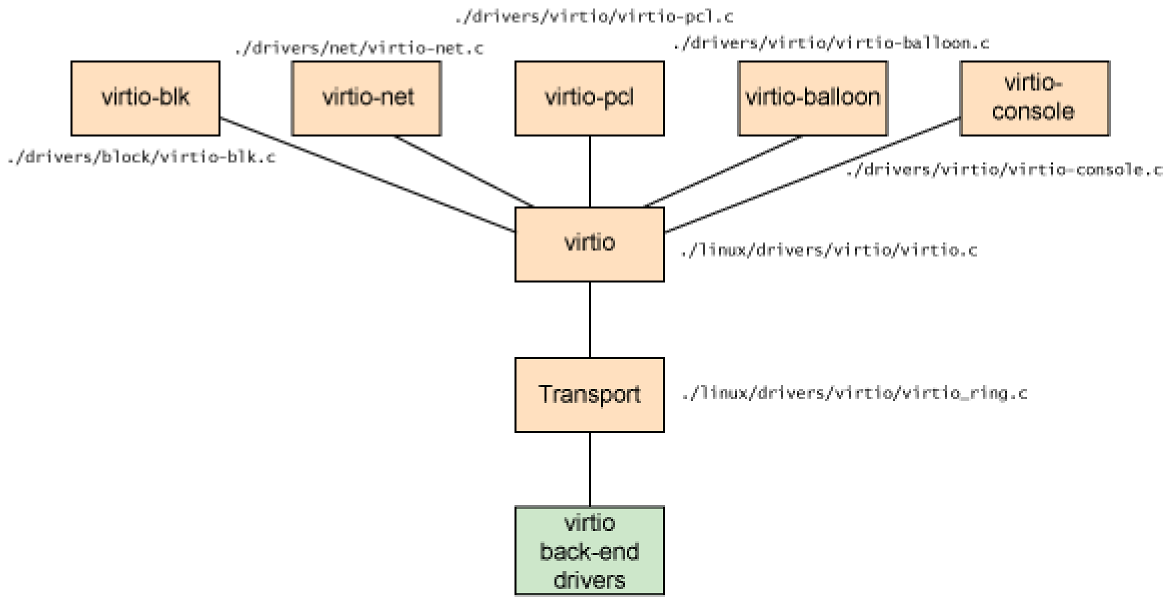
Note that in reality (though not required), the device emulation occurs in user space using QEMU, so the back-end drivers communicate into the user space of the hypervisor to facilitate I/O through QEMU. QEMU is a system emulator that, in additio guest operating system virtualization platform, provides emulation of an entire system (PCI host controller, disk, network, video hardware, USB controller, and other hardware elements).

The `virtio` API relies on a simple buffer abstraction to encapsulate the command and data needs of the guest. Let's look at the internals of the `virtio` API and its components.

Virtio architecture

In addition to the front-end drivers (implemented in the guest operating system) and the back-end drivers (implemented in the hypervisor), `virtio` defines two layers to support guest-to-hypervisor communication. At the top level (called *virtio*) is the virtual queue interface that conceptually attaches front-end drivers to back-end drivers. Drivers can use zero or more queues, depending on their need. For example, the `virtio` network driver uses two virtual queues (one for receive and one for transmit), where the `virtio` block driver uses only one. Virtual queues, being virtual, are actually implemented as rings to traverse the guest-to-hypervisor transition. But this could be implemented any way, as long as both the guest and hypervisor implement it in the same way.

Figure 3. High-level architecture of the virtio framework

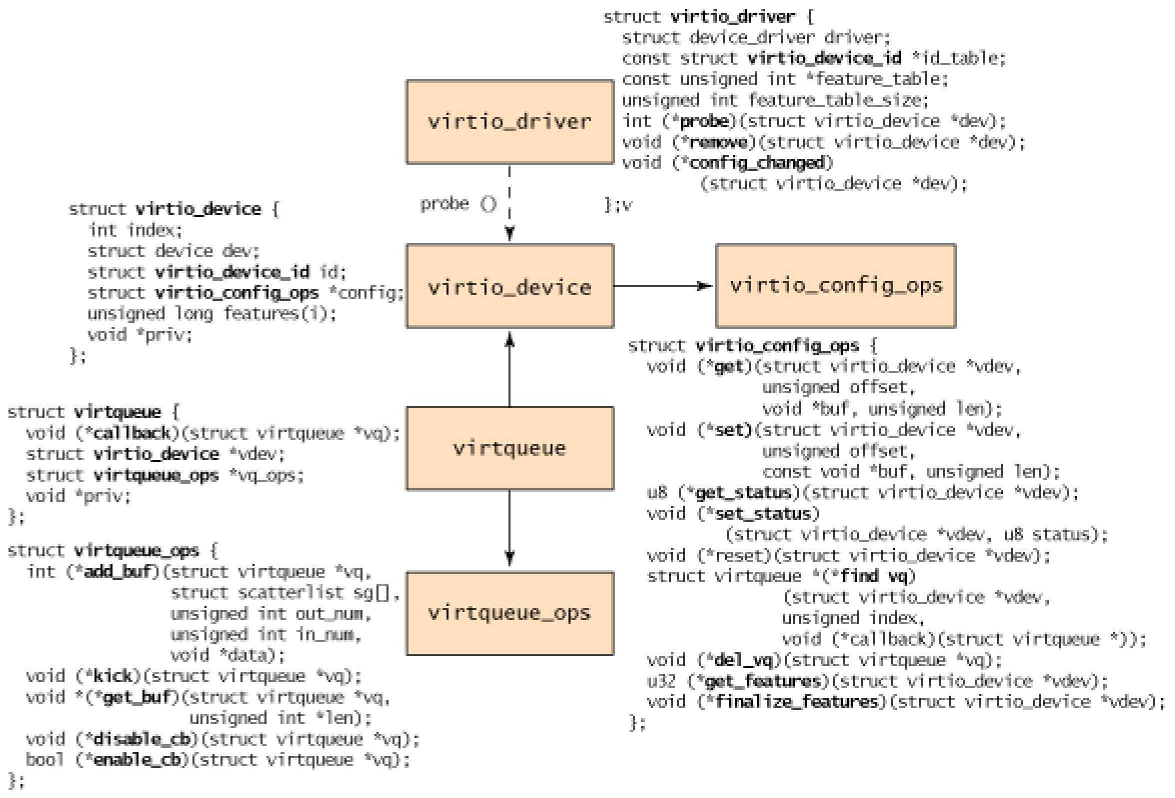

./drivers/virtio/virtio-pcl.c
./drivers/net/virtio-net.c
./drivers/virtio/virtio-balloon.c
virtio-blk
virtio-net
virtio-pcl
virtio-balloon
virtio-console
./drivers/block/virtio-blk.c
./drivers/virtio/virtio-console.c
virtio
./linux/drivers/virtio/virtio.c
Transport
./linux/drivers/virtio/virtio_ring.c
virtio back-end drivers

As shown in Figure 3, five front-end drivers are listed for block devices (such as disks), network devices, PCI emulation, a balloon driver (for dynamically managing guest memory usage), and a console driver. Each front-end driver has a corresponding back-end driver in the hypervisor.

Concept hierarchy

From the perspective of the guest, an *object hierarchy* is defined as shown in Figure 4. At the top is the `virtio_driver`, which represents the front-end driver in the guest. Devices that match this driver are encapsulated by the `virtio_device` (a representation of the device in the guest). This refers to the `virtio_config_ops` structure (which defines the operations for configuring the `virtio` device). The `virtio_device` is referred to by the `virtqueue` (which includes a reference to the `virtio_device` to which it serves). Finally, each `virtqueue` object references the `virtqueue_ops` object, which defines the underlying queue operations for dealing with the hypervisor driver. Although the queue operations are the core of the `virtio` API, I provide a brief discussion of discovery, and then explore the `virtqueue_ops` operations in more detail.

Figure 4. Object hierarchy of the virtio front end



```
struct virtio_driver [
  struct device_driver driver;
  const struct virtio_device_id *id_table;
  const unsigned int *feature_table;
  unsigned int feature_table_size;
  int (*probe)(struct virtio_device *dev);
  void (*remove)(struct virtio_device *dev);
  void (*config_changed)
        (struct virtio_device *dev);
};v
```

```
struct virtio_device {
  int index;
  struct device dev;
  struct virtio_device_id id;
  struct virtio_config_ops *config;
  unsigned long features(i);
  void *priv;
};
```

```
struct virtio_config_ops {
  void (*get)(struct virtio_device *vdev,
              unsigned offset,
              void *buf, unsigned len);
  void (*set)(struct virtio_device *vdev,
              unsigned offset,
              const void *buf, unsigned len);
  u8 (*get_status)(struct virtio_device *vdev);
  void (*set_status)
        (struct virtio_device *vdev, u8 status);
  void (*reset)(struct virtio_device *vdev);
  struct virtqueue *(*find vq)
              (struct virtio_device *vdev,
              unsigned index,
              void (*callback)(struct virtqueue *));
  void (*del_vq)(struct virtqueue *vq);
  u32 (*get_features)(struct virtio_device *vdev);
  void (*finalize_features)(struct virtio_device *vdev);
};
```

```
struct virtqueue {
  void (*callback)(struct virtqueue *vq);
  struct virtio_device *vdev;
  struct virtqueue_ops *vq_ops;
  void *priv;
};
```

```
struct virtqueue_ops {
  int (*add_buf)(struct virtqueue *vq,
                struct scatterlist sg[],
                unsigned int out_num,
                unsigned int in_num,
                void *data);
  void (*kick)(struct virtqueue *vq);
  void *(*get_buf)(struct virtqueue *vq,
                unsigned int *len);
  void (*disable_cb)(struct virtqueue *vq);
  bool (*enable_cb)(struct virtqueue *vq);
};
```

The process begins with the creation of a `virtio_driver` and subsequent registration via `register_virtio_driver`. The `virtio_driver` structure defines the upper-level device driver, list of device IDs that the driver supports, a features table (dependent upon the device type), and a list of callback functions. When the hypervisor identifies the presence of a new device that matches a device ID in the device list, the `probe` function is called (provided in the `virtio_driver` object) to pass up the `virtio_device` object. This object is cached with the management data for the device (in a driv

Depending on the driver type, the `virtio_config_ops` functions may be invoked to get or set options specific to the device (for example, getting the Read/Write status of the disk for a `virtio_blk` device or setting the block size of the block device).

Note that the `virtio_device` includes no reference to the `virtqueue` (but the `virtqueue` does reference the `virtio_device`). To identify the `virtqueues` that associate with this `virtio_device`, you use the `virtio_config_ops` object with the `find_vq` function. This object returns the virtual queues associated with this `virtio_device` instance. The `find_vq` function also permits the specification of a callback function for the `virtqueue` (see the `virtqueue` structure in Figure 4), which is used to notify the guest of response buffers from the hypervisor.

The `virtqueue` is a simple structure that identifies an optional callback function (which is called when the hypervisor consumes the buffers), a reference to the `virtio_device`, a reference to the `virtqueue` operations, and a special `priv` reference that refers to the underlying implementation to use. Although the `callback` is optional, it's possible to enable or disable callbacks dynamically.

But the core of this hierarchy is the `virtqueue_ops`, which defines how commands and data are moved between the guest and the hypervisor. Let's first explore the object that is added or removed from the `virtqueue`.

Virtio buffers

Guest (front-end) drivers communicate with hypervisor (back-end) drivers through buffers. For an I/O, the guest provides one or more buffers representing the request. For example, you could provide three buffers, with the first representing a Read request and the subsequent two buffers representing the response data. Internally, this configuration is represented as a scatter-gather list (with each entry in the list representing an address and a length).

Core API

Linking the guest driver and hypervisor driver occurs through the `virtio_device` and most commonly through `virtqueues`. The `virtqueue` supports its own API consisting of five functions. You use the first function, `add_buf`, to provide a request

This request is in the form of the scatter-gather list discussed previously. To `add_buf`, the guest provides the `virtqueue` to which the request is to be enqueued, the scatter-gather list (an array of addresses and lengths), the number of buffers that serve as out entries (destined for the underlying hypervisor), and the number of in entries (for which the hypervisor will store data and return to the guest). When a request has been made to the hypervisor through `add_buf`, the guest can notify the hypervisor of the new request using the `kick` function. For best performance, the guest should load as many buffers as possible onto the `virtqueue` before notifying through `kick`.

Responses from the hypervisor occur through the `get_buf` function. The guest can poll simply by calling this function or wait for notification through the provided `virtqueue callback` function. When the guest learns that buffers are available, the call to `get_buf` returns the completed buffers.

The final two functions in the `virtqueue` API are `enable_cb` and `disable_cb`. You can use these functions to enable and disable the callback process (via the `callback` function initialized in the `virtqueue` through the `find_vq` function). Note that the callback function and the hypervisor are in separate address spaces, so the call occurs through an indirect hypervisor call (such as `kvm_hypercall`).

The format, order, and contents of the buffers are meaningful only to the front-end and back-end drivers. The internal transport (rings in the current implementation) move only buffers and have no knowledge of their internal representation.

Example virtio drivers

You can find the source to the various front-end drivers within the ./drivers subdirectory of the Linux kernel. The `virtio` network driver can be found in ./drivers/net/virtio_net.c, and the `virtio` block driver can be found in ./drivers/block/virtio_blk.c. The subdirectory ./drivers/virtio provides the implementation of the `virtio` interfaces (`virtio` device, driver, `virtqueue`, and ring). `virtio` has also been used in High-Performance Computing (HPC) research to develop inter-virtual machine (VM) communications through shared memory passing. Specifically, this was implemented through a virtualized PCI interface using the `virtio` PCI driver. You can learn more about this work in the resources section on the right.

You can exercise this paravirtualization infrastructure today in the Linux
need is a kernel to act as the hypervisor, a guest kernel, and QEMU for d

You can use either KVM (a module that exists in the host kernel) or with Rusty Russell's `lguest` (a modified Linux guest kernel). Both of these virtualization solutions support `virtio` (along with QEMU for system emulation and `libvirt` for virtualization management).

The result of Rusty's work is a simpler code base for paravirtualized drivers and faster emulation of virtual devices. But even more important, `virtio` has been found to provide better performance (2-3 times for network I/O) than current commercial solutions. This performance boost comes at a cost, but it's well worth it if Linux is your hypervisor and guest.

Going further

Although you may never develop front-end or back-end drivers for `virtio`, it implements an interesting architecture and is worth understanding in more detail. `virtio` opens up new opportunities for efficiency in paravirtualized I/O environments while building from previous work in Xen. Linux continues to prove itself as a production hypervisor and a research platform for new virtualization technologies. `virtio` is yet another example of the strengths and openness of Linux as a hypervisor.