Chapter 6

Understanding Device Drivers

Device drivers are an important part of any operating system—without them, the kernel (and thus the applications) can't communicate with physical hardware attached to the system.

Most full virtualization solutions provide emulated forms of simple devices. The emulated devices are typically chosen to be common hardware, so it is likely that drivers exist already for any given guest. Examples of hardware emulated include simple IDE hard disks and NE2000 network interfaces. This is a reasonable solution in cases where the guest cannot be modified, and is used by Xen in HVM domains where unmodified guests are run.

Paravirtualized guests, however, need to be modified in order to run anyway. As such, the requirement for the virtual environment to use existing drivers disappears. Making guest kernel authors write a lot of code, however, would not be a very good design decision, and so Xen devices must be simple to implement. They should also be fast; if they are not, they have no advantage over emulated devices.

The Xen approach is to provide abstract devices that implement a high-level interface that corresponds to a particular device category. Rather than providing a SCSI device and an IDE device, Xen provides an abstract block device. This supports only two operations: read and write a block. This is implemented in a way that closely corresponds to the POSIX `readv` and `writev` calls, allowing operations to be grouped in a single request (which allows I/O reordering in the Domain 0 kernel or the controller to be used effectively). The network interface is slightly more complicated, but still relatively easy for a guest to implement.

6.1 The Split Driver Model

Supporting the range of hardware available for a commodity PC would be a daunting task for Xen. Fortunately, most of the required hardware is already supported by the guest in Domain 0. If Xen can reuse this support, it gets a large amount of hardware compatibility for free.

In addition, it is fairly common for an operating system to already provide some multiplexing services. The purpose of an operating system (as opposed to running applications directly on the hardware) is to provide an abstraction of the real hardware. One of the features of this abstraction in a modern OS is that applications are, in general, not aware of each other. Two applications can use the same physical disk, network interface, or sound device, without worrying about others. By piggy-backing on this capability, Xen can avoid writing a lot of new and untested code.

This multiplexing capability is quite important. Some devices on high-end systems, particularly mainframes, are virtualization-aware. They can be partitioned in the firmware, and each running operating system can interact with them directly. For consumer-grade hardware, however, this is not common. Most consumer-level devices assume a single user, and require the running operating system to perform any required multiplexing. In a virtualized environment, device access must be multiplexed before it is handed over to the operating system.

As discussed earlier, the hypervisor provides a simple mechanism for communicating between domains: shared memory. This is used by device drivers to establish a connection between the two components. The I/O ring mechanism, described later in this chapter, is typically used for this.

One important thing to note about Xen devices is that they are not really part of Xen. The hypervisor provides the mechanisms for device discovery and moving data between domains; the drives are split across a pair of guest domains. Typically, this pair is Domain 0 and another guest, although it is also possible to use a dedicated driver domain instead of Domain 0. The interface is specified by Xen; however, the actual implementation is left up to the domains.

Figure 6.1 shows the structure of a typical split device driver. The front and back ends are isolated from each other in separate domains, and communicate solely by mechanisms provided by Xen. The most common of these is the I/O ring, built on top of the shared memory mechanism provided by Xen.

Shared memory rings alone would require a lot of polling, which is not always particularly efficient, although it can be fast where there is pending data in a large percentage of the polled cases. This need is eliminated by the Xen event mechanism, which allows asynchronous notifications. This is used to tell the back end that a request is waiting to be processed, or to tell a front end that there is a response waiting. Handling and delivering events is discussed in the next chapter.

The final part of the jigsaw puzzle is the XenStore. This is a simple hierarchical

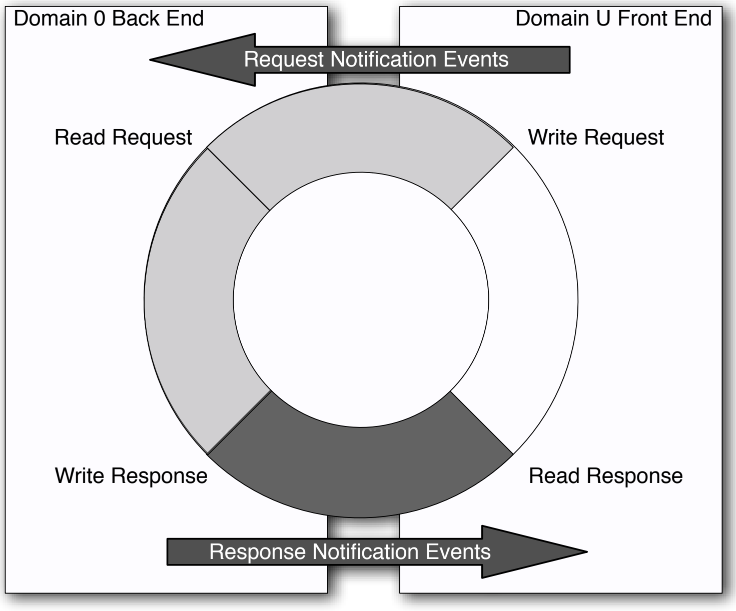

Figure 6.1: The composition of a split device driver

structure that is shared between domains. Unlike the grant tables, the interface is fairly high-level. One of the main uses for it is device discovery. In this rôle, it is analogous to the device tree provided by OpenFirmware, although it has additional uses. The guest in Domain 0 exports a tree containing the devices available to each unprivileged domain. This is used for the initial device discovery phase. The tree is traversed by the guest that wants to run front-end drivers, and any interesting devices are configured. The one exception to this is the console driver. It is anticipated that the console device is needed (or, at the very least, wanted) early on during the boot process, so it is advertised via the start info page.

The XenStore itself is implemented as a split device. The location of the page used to communicate is given as a machine frame number in the start info page. This is slightly different to other devices, in that the page is made available to the guest before the system starts, rather than being exported via the grant table mechanism and advertised in the XenStore.

6.2 Moving Drivers out of Domain 0

Xen provides a mechanism for delegating access to physical devices to domains other than Domain 0, known as *driver domains.* On platforms that don't contain an IOMMU or similar hardware that implements the protection elements of an IOMMU, it is not possible to do this securely.

The hypervisor can use the MMU to isolate the memory regions used by memory mapped I/O and grant pages within these regions to a driver domain. It can also prevent ring 1 processes from using I/O port instructions, and require them to be trapped and emulated by the hypervisor, adding significant overhead. It cannot, however, prevent the driver domain from issuing unsafe DMA requests without additional hardware support. This makes driver domains only semi-safe on most legacy hardware.

The use of driver domains provides two key advantages. The first is that it gives some extra isolation for components of the system. In the standard configuration, Domain 0 has two distinct responsibilities:

- Supporting hardware and running back-end devices
- Providing the administrative interface to Xen

One of the key features of Xen is that only a small amount of code runs in ring 0, which helps provide some security and stability. Code running in ring 1 in Domain 0, however, can still perform a lot of operations via the hypervisor that would usually be restricted to ring 0. By reducing the amount of code running in Domain 0, the security of a system is improved. Device drivers tend to make up the majority of the codebase of a modern operating system and, unfortunately, tend to be the most buggy parts of the kernel. This is understandable, because they are the least tested as not everyone is using the same drivers. Drivers also have to deal with (potentially undocumented) flaws in the physical hardware, as well as the wide range of potential interactions the device can have when functioning correctly. A device domain on a system with an IOMMU can only damage itself with erroneous DMA requests, because the IOMMU prevents accesses to other domains' address spaces.

The other advantage is that it allows support for more devices. If you choose to run Solaris in Domain 0, for example, you are not limited to the devices supported by Solaris. You could run Linux or NetBSD in a driver domain and gain access to other pieces of hardware. This could even be extended by hardware manufacturers to provide a "Xen native" device driver—a minimal kernel with the device driver embedded in it that could be run in a driver domain and reduce the need to provide OS-specific drivers.

6.3 Understanding Shared Memory Ring Buffers

The ring buffer is a fairly standard lockless data structure for producer-consumer communications. The variant used by Xen is somewhat unusual in that it uses free-running counters. A typical ring buffer has a producer and a consumer pointer. Each of these is tested by one and incremented by the other. When one pointer goes past the end of the buffer, it must be wrapped. This is relatively expensive, and because it involves programming for a number of corner cases, it is relatively easy to get wrong.

The Xen version avoids this by ensuring that the buffer size is a power of two. This means that the lowest n bits of the counter can be used to give an index within the buffer, and these bits can be obtained with a simple mask. Because the counters can run to more than the buffer size, you do not need to manually account for overflow; subtracting one from the other always gives you the amount of data in the buffer. The one special case comes from the fact that the counters themselves can overflow. Let's take a simple example—an 8-bit counter on a 32-element buffer—and see what happens when it overflows. The producer value is incremented to 258, which causes an overflow, wrapping the value around to two. The consumer value has not yet overflowed, so it is now bigger than the producer. What happens when we try a subtraction?

	0	0	0	0	0	0	1	0	(Producer value of 2)
−	1	1	1	1	1	1	0	0	(Consumer value of 252)
1	0	0	0	0	0	1	1	0	(Subtraction gives −250)
	0	0	0	0	0	1	1	0	(Truncation gives 6)

The leading 1 in the subtraction result comes from the use of 1's complement arithmetic. If the result of a calculation is negative, the leading bit will be 1, and the remaining bits will be the inverse of their positive values. This representation is used in all modern CPUs, because it simplifies a number of cases (adding signed values can be implemented using the same logic, irrespective of their signs). One nice side effect of this is that subtraction still works if one of the values has overflowed. Note that the leading bit is truncated, because it doesn't fit in the 8-bit value. Most CPUs will store this in a condition register somewhere; however, it can simply be discarded for our purposes.

One thing to note is that this value will be wrong if the producer value overflows twice before the consumer value overflows. Fortunately, this can never happen, because that would require the buffer to be larger than the counter, meaning that no mask exists that would convert the counter value to a buffer index.

The other interesting thing about the Xen ring mechanism is that each ring contains two kinds of data, a request and a response, updated by the two halves of the driver. In principle, this could cause some problems. If the responses are

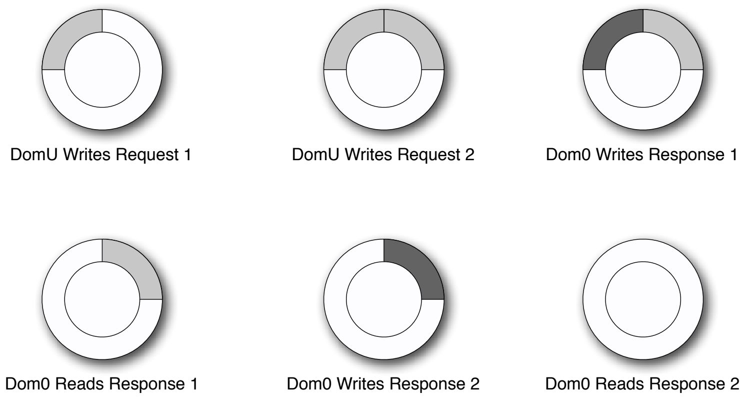

Figure 6.2: A sequence of actions on a ring buffer

larger than the requests, the response pointer catches up with the request pointer, preventing the back end from writing any more responses.

Xen solves this by only permitting responses to be written in a way that overwrites requests. Figure 6.2 shows a typical sequence of actions in an I/O ring. In this sequence, the front half of the driver issues two requests, and the back half fills them in.

The first two operations in this sequence show the front half of the driver writing a pair of requests into the ring. After writing each request, it increments a counter indicating the used part of the ring. The back half then writes a response over the first request. After doing this, it increments the response counter, indicating that the response has been filled in. The front half can then read this response and increment a counter indicating where the back end is. This is then repeated for the second response.

There are three counters of relevance here. The first indicates the start of the requests. This is only ever incremented by the front end, and never decremented. If the back end reads this value, it can guarantee that the value will never be lower than this value.[1] It can then use any space from the back of the request segment to the front in order to store responses.

After storing a response, the back end increments a counter indicating that the response has been stored. The front end can read this counter, and know that anything between it and the back of the tail counter contains responses. It can then read them, and update the tail counter to any value that doesn't cause it to

[1]The only requirement for this to work is that memory writes are atomic. This is true on most modern CPUs. Note that a memory barrier is not required; the old value for the counter is always safe to use, because it is monotonic.

pass the response counter. The tail counter is then used again by the front end, to ensure that it does not overwrite existing requests with new ones.

The ring can only run out of space for responses if there are no outstanding requests, but if this is the case, it will not need to generate any. If it runs out of space for requests, this implies one of two things. Either the ring is full of requests, in which case the front end should back off for a bit and allow the back end to process them, or it contains some responses, in which case, it should consider processing some of them before adding more requests.

6.3.1 Examining the Xen Implementation

Most of the definitions relating to Xen's I/O rings can be found in `xen/interface/public/io/ring.h`. This file contains a number of macros for creation, initialization, and use of I/O rings. This generalized interface is not used by all drivers. Some use a simpler mechanism where there are two unidirectional rings. Others use more complex interfaces. The macros described here are used by both the network and block interfaces, and are suited to any interface that has a one-to-one mapping between requests and responses. This is obviously not the case for the console, where reading from the keyboard and writing to the screen are unrelated operations.

The first macro is used for creating ring structures, in the following way:

```
DEFINE_RING_TYPES(name, request_t, response_t);
```

This creates the structures for a ring where requests were specified by a `request_t` and responses by a `response_t`. This defines three structures representing the ring—one is the shared data structure, which goes in a shared memory page. The other two contain private variables, one for the front end and one for the back. These used by the other macros. Also defined is a union of the request and response types. This is used to divide the rings into segments that can store either a request or a response.

Before a ring can be used, it must be initialized. This sets the producer and consumer indexes to their correct initial values. The macro for initializing the shared ring depends on the existence of `memset`, so ensure that you have a working implementation of this in your kernel, even if it's not properly optimized.

To give some idea of how these are used, we will look briefly at how the virtual block device uses them. The interface to this device is defined in the `io/blkif.h` public header file.

The block interface defines the `blkif_request` and `blkif_response` structures for requests and responses, respectively. It then defines the shared memory ring structures in the following way:

```
DEFINE_RING_TYPES(blkif, struct blkif_request, struct
    blkif_response);
```

Grant Table Use

To use the ring mechanism, both sides of the driver must be able to access it. This means that the underlying memory must be shared, and the standard mechanism for doing this is via the grant table.

The convention for the Xen split driver model is that the front-end driver should offer the grant reference, while the back end maps it. This means that the front end does not need to issue any hypercalls to set up the driver rings. Aside from consistency, this has the advantage that HVM guests can implement paravirtualized device drivers without the need to use any grant table hypercalls. It also aids migration. The shared rings belong to the front end, and are shared with the back end, so when the front end is moved to a different machine they stay in the same place (in the pseudo-physical address space) and can be remapped easily.

This convention applies to all Xen drivers that use the split model, not only those that use the generic ring macros. This means that the only hypercalls that a front-end driver needs to use are the ones that relate to the event channel.

This defines the blkif_sring_t type, representing the shared ring and two other structures representing the private variables used by the front and back halves of the ring. The front end must allocate the space for the shared structure, and then initialize the ring. The Linux implementation of this driver then initializes the ring as follows:

```
SHARED_RING_INIT(sring);
FRONT_RING_INIT(&info->ring, sring, PAGE_SIZE);
```

Here, the sring variable is a pointer to the shared ring structure. The info variable is a pointer to a structure that contains some kernel-specific bookkeeping information about the virtual block device. The ring field is a blkif_front_ring_t, which is a structure defined by the ring macros to store the private variables relating to the front half of the ring. These are required due to the way in which data is put on the ring.

The first step is to write the request to the ring. Then, the (shared) ring's request producer count is incremented. For efficiency, it is fairly common to write multiple requests into the ring, and then perform the update. For this reason, it is necessary to keep a private copy of the request producer counter, which is incremented to let the front end know where it should put new requests. This is then pushed to the shared ring later. The next available request in the ring is requested by the front end like this:

```
ring_req = RING_GET_REQUEST(&info->ring, info->ring.
    req_prod_pvt);
```

The request is then filled in with the correct information, and is ready to be pushed to the front end. This is done using the following macro:

```
RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&info->ring, notify);
```

Here, notify is a return parameter, which is used to indicate whether the front end needs to deliver an event to the back end. This macro sets the shared ring's request producer counter to be equal to the private copy. After doing this, it tests the request consumer counter. If this is still lower than the request producer counter value from before the update, it means that the back end is still processing requests that were enqueued before this update was pushed. If this is the case, there is no need to send an event to the back end, and so notify is set to false.

After a request has been sent, the back end processes it and inserts a response into the ring. The next response from the ring is fetched like this:

```
blkif_response_t *bret = RING_GET_RESPONSE(&info->ring, i);
```

This sets bret to the response at index i in the ring. The value of i needs to be somewhere between the shared ring's rsp_prod and the front half's rsp_cons values. These two fields represent the response producer and consumer indexes. After processing response i, the front half's rsp_cons value should be updated to reflect this. This value is used when inserting new requests into the ring, because it indicates where the back of the used segment of the ring is. No new requests may be inserted past this value. After fetching all waiting requests, the block driver performs the following check:

```
RING_FINAL_CHECK_FOR_RESPONSES(&info->ring, more_to_do);
```

The more_to_do value is set to true if the ring still has some requests waiting to process. After checking this, the event channel should be unmasked, the ring checked one final time, and then control returned.

The back end has a similar interface, but uses the other private structure, and uses the queues the other way around.

6.3.2 Ordering Operations with Memory Barriers

Some operations on rings require *memory barriers* on some systems. On a purely in-order architecture, memory operations are guaranteed to be completed in the order in which they are issued. This means that, as long as the data is written into the ring before the counters are incremented, things will work correctly.

x86 began life as an in-order architecture. As it has grown, it has had to maintain compatibility with legacy code. For this reason, modern x86 CPUs are still *strongly ordered*. Memory operations on a strongly ordered CPU are guaranteed to complete in the order in which they are issued. On x86, this is only true for stores; loads may be reordered.

This is not true of all architectures supported by Xen. IA64, for example, is a *weakly ordered* architecture, and so no such guarantees exist. This is somewhat complicated by the fact that early versions of the Itanium were strongly ordered. This means that code written and tested on an early Itanium might suddenly fail on a newer version.

To prevent re-ordering, most architectures provide memory barrier instructions. These force all loads, stores, or loads and stores to be completed before continuing. The Xen code uses macros to implement these, and expands them based on a per-processor definition. When developing on x86, it is tempting to leave some of these out, because the write barrier expands to a no-op on x86. If you do this, be aware that your code will break on other platforms.

The Xen ring macros include the correct barriers, and work on all supported architectures. For other devices that don't use these macros, it is important to remember to add the barriers yourself. The most commonly used barrier macros are wmb() and mb(), which provide a write memory barrier and a full (read and write) barrier, respectively. Here is an example of a barrier being used in one of the standard ring macros:

```
#define RING_PUSH_REQUESTS(_r) do {
                                                            \
    wmb(); /* back sees requests /before/ updated producer
        index */        \
    (_r)->sring->req_prod = (_r)->req_prod_pvt;
                                                    \
} while (0)
```

The wmb() macro is used here to ensure that the requests that have been written to the ring are actually in memory before the private copy of the request producer counter is copied into the shared ring. A full memory barrier is used on the macros that check whether they need to notify the other end of new data, to ensure that any reads of the data in the ring by the remote end have completed before checking the consumer counter.

Note that x86 has no explicit barrier instructions. Write barriers are not needed, but read barriers are on some occasions. Instructions with the **LOCK** prefix are implicit read barriers, and so an atomic add instruction is used in place of an explicit barrier. This means that barriers can be omitted on x86 when they immediately follow an atomic operation.

6.4 Connecting Devices with XenBus

The XenBus, in the context of device drivers, is an informal protocol built on top of the XenStore, which provides a way of enumerating the (virtual) devices available to a given domain, and connecting to them. Implementing the XenBus interface is not required when porting a kernel to Xen. It is predominantly used in Linux to isolate the Xen-specific code behind a relatively abstract interface.

The XenBus interface is intended to roughly mirror that of a device bus such as PCI. It is defined in `linux-2.6-xen-sparse/include/xen/xenbus.h`.[2] Each virtual device has three major components:

- A shared memory page containing the ring buffers
- An event channel signaling activity in the ring
- A XenStore entry containing configuration information

These components are tied together in the bus interface by the structure shown in Listing 6.1. This device structure is passed as an argument to a number of other functions, which (between them) implement the driver for the device.

Listing 6.1: The structure defining a XenBus device [from: linux-2.6-xen-sparse/include/xen/xenbus.h]

```
struct xenbus_device {
    const char *devicetype;
    const char *nodename;
    const char *otherend;
    int otherend_id;
    struct xenbus_watch otherend_watch;
    struct device dev;
    enum xenbus_state state;
    struct completion down;
};
```

The exact definition of this structure is tied quite closely to Linux; the device struct, for example, represents a Linux device. It can, however, be used as a good starting point for building a similar abstraction layer for other systems.

The core component of the XenBus interface, indeed the only part that needs to be implemented by all systems wanting to use the paravirtualized devices available to Xen guests, is the xenbus_state enumerated type. Each device has such a type associated with it.

[2]Although the header is part of the sparse Linux tree, it is available under a more permissive license when not distributed as part of Linux.

The XenBus state, unlike the rest of the XenBus interface, is defined by Xen, in the `io/xenbus.h` public header. This is used while negotiating a connection between the two halves of the device driver. There are seven states defined. In normal operation, the state should be gradually incremented as the device is initialized, connected, and then disconnected. The possible states are:

- XenbusStateUnknown represents the initial state of the device on the bus, before either end has been connected.

- XenbusStateInitialising is the state while the back end is in process of initializing itself.

- XenbusStateInitWait should be entered by the back end while it is waiting for information before completing initialization. The source of the information can be hot-plug notifications within the Domain 0 kernel, or further information from the connecting guest. The meaning of this state is that the driver itself is initialized, but needs more information before it can be connected to.

- XenbusStateInitialised should be set to indicate that the back end is now ready for connection. After the bus is in this state, the front end may proceed to connect.

- XenbusStateConnected is the normal state of the bus. For most of the duration of a guest's run, the bus will be in this state indicating that the front and back ends are communicating normally.

- XenbusStateClosing is set to indicate that the device has become unavailable. The front and back halves of the driver are still connected at this point, but the back end is no longer doing anything sensible with the commands received from the front. When this state is entered, the front end should begin a graceful shutdown.

- XenbusStateClosed is the final state, once the two halves of the driver have disconnected from each other.

Not all drivers make use of the XenBus mechanism. The two notable exceptions are the console and XenStore. Both of these are mapped directly from the start info page, and have no information in the XenStore. In the case of the console, this is so that a guest kernel can start outputting debugging information to the console as soon as possible. In the case of the XenStore, it is obvious that the device can't use XenBus, because XenBus is built on top of the XenStore, and the XenStore cannot be used to get information required to map itself.

6.5 Handling Notifications from Events

Real hardware uses interrupt channels to notify the CPU of asynchronous events. These then cause the CPU to enter privileged mode and jump to a handler that has been configured for the event.

Xen provides an analog of this in the form of event channels. These are delivered asynchronously to guests and, unlike interrupts, are enqueued when the guest is not running. Interrupts, conventionally, are not aware of the existence of virtual machines and so are delivered immediately.[3]

When a driver needs to notify a device of some waiting data, it typically writes to a control register. Within Xen, the event mechanism also replaces this for notifying the back end of a split device driver, because both directions are an example of interdomain communication.

Events have a couple of major differences from interrupts. They are bidirectional and connection-oriented. An IRQ is delivered to a specified handler, but the concept of a connection does not arise. What can raise an interrupt is defined at the hardware level; the interrupt descriptor table indicates how privileged a program needs to be to trigger a given interrupt in software, and the hardware defines which devices can trigger them externally.

An event needs much better access control. A malicious guest could cause some serious problems by triggering large numbers of spurious events. For this reason, events may only be delivered on a given channel by one of the two domains to which it is connected. When an event channel is allocated by one domain, it explicitly states the number of the domain that is allowed to bind to the other end. The other domain must then explicitly request binding to that event channel. It is not until this point that either end may trigger the event delivery.

The event channel number for the device must be passed to the front end somehow, before it can connect.

In addition to providing asynchronous notifications sent from one domain to another, events can be used by a guest to receive real IRQs. These cannot be delivered using the normal mechanism because, as mentioned earlier, they might well end up being delivered to the wrong VM. The hypervisor must catch interrupts raised by devices and enqueue them so that the domain hosting the back end of the driver can receive them irrespective of which domain was running when the interrupt was generated.

Event channels are also used to deliver notifications from a (small) number of devices that run inside the hypervisor. The most common of these is the domain virtual time clock device. This is typically used for scheduling; it provides a notification when a certain amount of virtual time has elapsed, that is, when the domain has received $n$ms of CPU time.

[3]Modern, virtualization-aware hardware provides a mechanism for enqueuing interrupts.

6.6 Configuring via the XenStore

Among other things, the XenStore is the Xen equivalent of an OpenFirmware device tree, or the results of querying an external bus. It provides a central location for retrieving information about devices that are available to the domain. The XenStore is, itself, a device, and so must be bootstrapped using information from the start info page. Chapter 8 will discuss this process in more detail, and give a more comprehensive overview of the XenStore.

Each virtual machine has an entry in the XenStore in which all information about that VM is stored. This is true for both the front and back ends of the device. As mentioned earlier, the back ends are not always in Domain 0, so the front end needs to be able to know three things when connecting to a typical Xen device:

- The domain hosting the back end
- The grant reference of the shared memory page
- The event channel used for notifications

It may also need to know some other device-specific information. This could be passed in the shared memory page, but passing it in the store allows it to be inspected before the device is initialized, and allows extra information to be added without breaking the device's ABI. It also allows tools to access the information about the device, for various reasons.

The XenStore provides an abstract way of discovering information about devices, and about other aspects of the system. It is one of the first devices that must be supported, because it allows the information required for other devices to be found.

The XenStore is a simple hierarchical namespace containing strings. This eliminates a number of potential problems. When running x86 and x86-64 guests on the same machine, word size becomes an issue for binary interfaces. An even worse situation can occur on some architectures, such as PowerPC, ARM, or SPARC, which are bi-endian. Big-endian and little-endian guests could be running on the same system; passing anything other than a string of characters (bytes) would require careful use of things like the `htonl` macros to ensure consistent storage. Another advantage of the text-based nature of the XenStore is that it makes it much easier for tools written in scripting languages to parse the data.

6.7 Exercise: The Console Device

The console device is simpler than many of the others. Instead of using the generic ring mechanism, where a single ring contains requests and responses, it provides two rings containing input and output characters, respectively. This is because console interaction on the dumb terminal model is intrinsically composed of two unidirectional systems. The keyboard writes data into the system in response to the user, whereas the screen displays text without providing any information.

The console interface itself is very simple. Listing 6.2 describes the structure found on the shared memory page used by the console. The machine address of this page is passed to the guest in the start info structure.

Listing 6.2: Xen Console interface structure [from: xen/include/public/io/console.h]

```
struct xencons_interface {
    char in[1024];
    char out[2048];
    XENCONS_RING_IDX in_cons, in_prod;
    XENCONS_RING_IDX out_cons, out_prod;
};
```

Before the console can be used, the guest needs to map it into its address space. This is done as per the example in Chapter 5. After this has been accomplished, the event channel for console events must be bound. This will be covered in more detail in the next chapter; for now, we will treat the console as a write-only device, and try to display a boot message.

Listing 6.3 shows how to map the console. We will leave some space here for setting up the event handler, and come back to that in the next chapter, after detailed discussion of the events system. We will keep a record of the event channel that is being used for the console in the console_evtchn variable.

Listing 6.3: Mapping the console [from: examples/chapter6/console.c]

```
/* Initialise the console */
int console_init(start_info_t * start)
{
    console = (struct xencons_interface*)
        ((machine_to_phys_mapping[start->console.domU.mfn] <<
            12)
         +
        ((unsigned long)&_text));
    console_evt = start->console.domU.evtchn;
    /* TODO: Set up the event channel */
    return 0;
}
```

When we want to write something to the screen, we have to copy it into the buffer, assuming that there is space. If there is insufficient space, we have to wait until there is. Typically, copying a load of data would be done using memcpy. Because we are not linking against the C standard library, however, this is not an option for us. Instead, we have to implement the copy operation ourselves. The version shown here is not at all optimized; it is possible to make it significantly faster. This is not particularly important for a console driver, because the console is typically a fairly low data rate device; however, for other drivers, it is probably worth copying a well-optimized memcpy implementation, assuming you don't already have one in your kernel.

Listing 6.4 contains a simple function for writing a string to the console. The function loops for each character on an input string until it encounters a NULL, copying the character from the input string to the buffer.

Listing 6.4: Writing data to the console [from: examples/chapter6/console.c]

```
/* Write a NULL-terminated string */
int console_write(char * message)
{
    struct evtchn_send event;
    event.port = console_evt;
    int length = 0;
    while(*message != '\0')
    {
        /* Wait for the back end to clear enough space in the
            buffer */
        XENCONS_RING_IDX data;
        do
        {
            data = console->out_prod - console->out_cons;
            HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
            mb();
        } while (data >= sizeof(console->out));
        /* Copy the byte */
        int ring_index = MASK_XENCONS_IDX(console->out_prod,
            console->out);
        console->out[ring_index] = *message;
        /* Ensure that the data really is in the ring before
            continuing */
        wmb();
        /* Increment input and output pointers */
        console->out_prod++;
        length++;
        message++;
    }
    HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
    return length;
}
```

Note the use of the MASK_XENCNOS_IDX macro. This is used because the console rings, like other I/O rings in Xen, use free-running counters. The least-significant n bits of these are used to indicate the position within the ring. The advantage of this approach is that there is no need to test whether the counters have overflowed the buffer. In addition, comparisons of the form $producer - consumer$ will always give the correct result, as long as the size of the maximum value of the index variable is greater than twice the size of the buffer, and the *producer* counter doesn't overflow twice before the *consumer* counter overflows once.

This means that we only need to test for one condition before adding something to a ring—that $producer - consumer < ring\ size$. Because $producer - consumer$ always gives you the amount of data in the ring, this does not allow you to proceed if the ring is full. In a full implementation of the console, we would wait until an event is received before proceeding here.

After putting data in the buffer, we need to signal the back end to remove it and display it. This is done via the event channel mechanism. Events will be discussed in detail in the next chapter, including how to handle incoming ones. For now, we will just signal the event channel, and look at what is actually happening in the next chapter.

Signaling the event channel is fairly simple, and can be thought of in the same way as sending a UNIX signal. The console_evtchn variable holds the number of the event channel being used for the console. This is similar to the signal number in UNIX, but is decided at runtime, rather than compile time. Note that sending an event after each character has been placed into the ring is highly inefficient. It is more efficient to only send an event at the end of sending, or when the buffer is full. This is left as an exercise for the reader.

To issue the signal, we use the HYPERVISOR_event_channel_op hypercall. The command we give to this tells it to send the event, and the control structure takes a single argument, indicating the event channel to be signaled.

We will add one more function for our simple half-console driver. This flushs the output buffer by blocking until the buffer is empty (that is, the consumer counter has caught up with the producer in the output ring). Because the function is just spinning while waiting for the back end to catch up, we issue a hypercall to notify the hypervisor that it should schedule other virtual machines while we are waiting. The memory barrier here is almost certainly not needed, because a hypercall (and the resulting ring transition) is a memory barrier, but is left in for clarity. Listing 6.5 shows the flush function.

Now that we have something that is hopefully a working implementation of a write-only console driver (we are not reading text input by the user yet), we should try using it. Let's create a console.h file that contains prototypes to our two functions, and then try calling them. Listing 6.6 gives the body of a kernel that writes "Hello world" to the console. "Hello world" is the obligatory first

Listing 6.5: Flushing the console output buffer [from: examples/chapter6/console.c]

```
/* Block while data is in the out buffer */
void console_flush(void)
{
    /* While there is data in the out channel */
    while(console->out_cons < console->out_prod)
    {
        /* Let other processes run */
        HYPERVISOR_sched_op(SCHEDOP_yield, 0);
        mb();
    }
}
```

output message from any system, but it's a bit boring. Let's print the Xen magic string, containing the running Xen version, as well.

Note that we call the console_flush() function before exiting. This is because the console ring buffer ceases to exist when the domain is destroyed, and if we are not very lucky, this will happen before the back end has read the contents of the buffer.

All that remains now is to add `console.o` to the `Makefile`, build, and test our kernel. When we launched our last simple kernel, we used `xm create`. This creates the new domain in the background. When we start this one, we want to see the output from the console. We do this by adding the `-c` flag, which tells `xm` to automatically attach to the console:

```
# xm create -c domain_config
Using config file "./domain_config".
Started domain Simplest_Kernel
Hello world!
Xen magic string: xen-3.0-x86_32p
#
```

The new domain is then destroyed, because we told the kernel to exit after writing the message. If this happens, everything is working as expected. We can now use the console for output during the rest of our boot procedure. When we have mapped the event channel, we can use it for input as well.

Currently, the console output is a little bit raw. It would be nice to add something like the C standard printf() function. This is left as an exercise to the reader. A good approach is to take a look at the vfprintf() function in an existing C library (the OpenBSD implementation is a good bet here) and replace the function or macro used for actually outputting characters with a call to our console_write() function.

Listing 6.6: The body of the "hello world" kernel [from: examples/chapter6/kernel.c]

```c
/* Main kernel entry point, called by trampoline */
void start_kernel(start_info_t * start_info)
{
    /* Map the shared info page */
    HYPERVISOR_update_va_mapping((unsigned long) &shared_info,
            __pte(start_info->shared_info | 7),
            UVMF_INVLPG);
    /* Set the pointer used in the bootstrap for reenabling
     * event delivery after an upcall */
    HYPERVISOR_shared_info = &shared_info;
    /* Set up and unmask events */
    init_events();
    /* Initialise the console */
    console_init(start_info);
    /* Write a message to check that it worked */
    console_write("Hello world!\r\n");
    /* Loop, handling events */
    while(1)
    {
        HYPERVISOR_sched_op(SCHEDOP_block,0);
    }
}
```