Implementation of Xen PVHVM drivers in *Open*BSD

Mike Belopuhov
Esdenera Networks GmbH
mike@esdenera.com

Abstract

OpenBSD 5.9 will include a native implementation of Xen PVHVM drivers. It was written from scratch to facilitate simplicity and maintainability. One of major goals of this effort is to run OpenBSD images in the Amazon cloud.

1 Introduction

Xen virtual machine monitor provides two types of guest hosting depending on the underlying hardware: paravirtualized and hardware assisted virtualization mode when a CPU with virtualization extensions (AMD-V or Intel VT-x) is used.

At the same time guests running in the hardware assisted virtualization mode are not restricted access to the paravirtualized facilities via the hypercall interface normally used by the paravirtualized instances.

We will explore what facilities are provided and how an HVM guest can combine emulated PCI device tree and interfaces provided via paravirtualization.

2 Guest domain initialization

In order to gain access to the paravirtualized services the guest must take a few steps in order to identify the hypervisor and setup the hypercall interface.

In OpenBSD a *pvbus(4)* pseudo bus abstraction takes care of identifying the type of the hypervisor via a CPUID signature and probes for child devices using a standard *config(9)* framework for that.

Xen nexus device that performs domU setup is implemented as a *xen(4)* device driver that also acts as an attachment point for other virtual devices configured in the virtual machine settings.

3 The hypercall interface

In order to be able to perform different operations like configuring devices, virtual interrupts or simply fetching information from the dom0, Xen makes use of a hypercall interface which works similar to the *syscall(9)* interface that provides a *VMEXIT* event on the hypervisor side.

The guest allocates a page of memory within the kernel's code segment and communicates its location in the physical memory to the hypervisor via an MSR write that fills it with content. Upon inspection the content of the page contains *SGDT* instructions at offsets representing different hypercalls.

Since OpenBSD virtual memory subsystem doesn't implement a proper way to allocate memory pages that can be later called into for various reasons, a code segment of the kernel itself had to be extended by one page. And

while this is a rather straightforward modification, perhaps a randomized location would suit it better.

Via the established hypercall interface other parameters of the system can be learned, for example extended version, enabled virtual machine features, etc.

Unlike other implementations, OpenBSD uses a single hypercall function that is defined as a variable argument function and expands the parameter list in order to construct hypercall arguments.

4 Shared Information Page

One of several basic ways of communicating information to the hypervisor and back to the guest system is using shared memory pages. The Shared Information Page is a specialized page of memory that provides guests access to the bitmap of masked and pending event channel ports events as well as other information, such as RTC, TSC, and information about NMIs.

The guest system must allocate a page of memory that has both physical and virtual mappings, for example via *malloc(9)*, and communicate it's *frame number* (a number of page sized increments) to the hypervisor via a *memory operation* hypercall.

Shared Information Page also includes a running cycle counter and a wall clock so it should be possible in the future to turn this into a system timecounter. In fact the *PVCLOCK* interface used by Linux is implemented this way.

5 The interrupt subsystem

There are two ways for a Xen hypervisor to inject an interrupt request into the system: via an Interrupt Descriptor Table vector that has been allocated by the guest solely for these purposes and via a virtual PCI device, the *XenStore Platform Device*.

Once triggered a guest operating system must run an interrupt vector that traverses a pending event channel ports bitmap inside the Shared Information Page to establish which ports have triggered the event.

A *xen_intr_establish()* method is provided in order to setup a callback that will be executed by the interrupt vector when associated event port is pending in the event channel ports bitmap. In many cases the event port number is not known in advance and can be allocated by the aforementioned method itself. Likewise a *xen_intr_disestablish()* can be called to remove the binding.

During system startup and device driver initialization interrupts remain masked and are unmasked after the root filesystem is mounted. Device drivers are required to operate in the polling mode until interrupts are enabled.

After startup is finished, device drivers can mask and unmask their interrupt sources at will via calls to *xen_intr_mask()* and *xen_intr_unmask()*.

Unlike other implementations, we have included support for marking Xen upcall interrupts as pending to integrate interrupt processing better with the rest of the system, e.g. to ensure that interrupt handler is not reentrant.

5.1 Interrupts: the IDT method

When indicated by the virtual machine features a guest system may communicate an allocated Interrupt Descriptor Table vector to the hypervisor to deliver the interrupt directly into the system without the help of an emulated APIC.

To set up an IDT vector a system must establish a link between an IDT vector number in a range of 0-255 and a callback function via an IDT gate descriptor. OpenBSD groups IDT vector numbers according to which Interrupt Priority Level they represent. *IPL_NET* priority is used for Xen interrupt vector and therefore the first vector 0x70 in that group has been reserved for it.

Due to the fact that this interrupt vector is not established via a PIC-compatible interface low level interrupt stub functions that basically implement pending interrupt processing cannot be used for our interrupt vector. Instead a set of new functions akin to those used for the LAPIC timer is rolled to provide this functionality.

5.2 Interrupts: Platform Device

As an alternative to the IDT method, domU guest implementing PCI bus discovery can implement a driver for the XenSource Platform Device, 0x5853:0x0001. This device provides a level triggered interrupt wired to the emulated APIC and once it's set up, Xen can be made aware of it.

A driver *xspd(4)* has been implemented for this device that configures the Xen interrupt vector to call the Xen upcall when the IDT method is not available.

6 Grant tables

Grant tables represent a mechanism of passing references to pages of memory allocated by the guest across domains. In essence it's similar to the IOMMU mechanism where device visible addresses are translated into physical addresses but in this case device visible addresses are represented as indexes into the grant table and point to a *grant table entries*.

Grant table entries contain a frame number and access flags that are set up when one domain wants to provide access to its own memory to the other domain. Upon startup the hypervisor sets an upper limit on how many grant table frames can be used by the guest system.

OpenBSD implements a *bus_dma(9)* [1] abstraction on top of grant tables. It defines a new *bus_dma_tag* that contains methods that wrap underlying *_bus_dmamap_** functions in a way that the memory managed by this underlying methods gets accounted for by the grant tables as well.

This allows drivers for paravirtualized devices to take advantage of a standard approach to DMA memory management. The first step is to create a DMA map that records meta information about the mapping that will be performed later. It records number of segments, their sizes and a total size of the mapping. Due to the limitation of grant tables, only the page sized segments are currently supported. The wrapper of *_bus_dmamap_create()* allocates an additional array of entries that will be used to map physical addresses of map segments to grant table references. At the same time grant table entries for all map segments are reserved via *xen_grant_table_alloc()*. This array of entries is then set as a DMA map cookie. A wrapped *_bus_dmamap_destroy()* method can free those references and destroy the map.

Not all *bus_dma(9)* methods need to be wrapped. For instance memory allocation and KVA mapping functions *_bus_dmamem_alloc()* and *_bus_dmamem_map()* as well as their destructive counterparts don't need any special handling.

However in order to establish associations between physical addresses of DMA memory segments and grant table references wrapped versions of *_bus_dmamap_load()* family of functions is required. After calling the system *_bus_dmamap_load()* method a wrapper needs to go through all map segments represented by the *dm_segs* member of the *bus_dmamap_t* structure, associate them with grant table references and update entries in the grant table via *xen_grant_table_enter()*. Upon success the physical address of the segment *ds_addr* is replaced with the grant table reference. After this call, the driver can pass this reference to the other domain via a ring descriptor or a similar mechanism.

To remove the mapping a *_bus_dmamap_unload()* method wrapper calls the *xen_grant_table_remove()* and puts physical addresses back into the *ds_addr* before

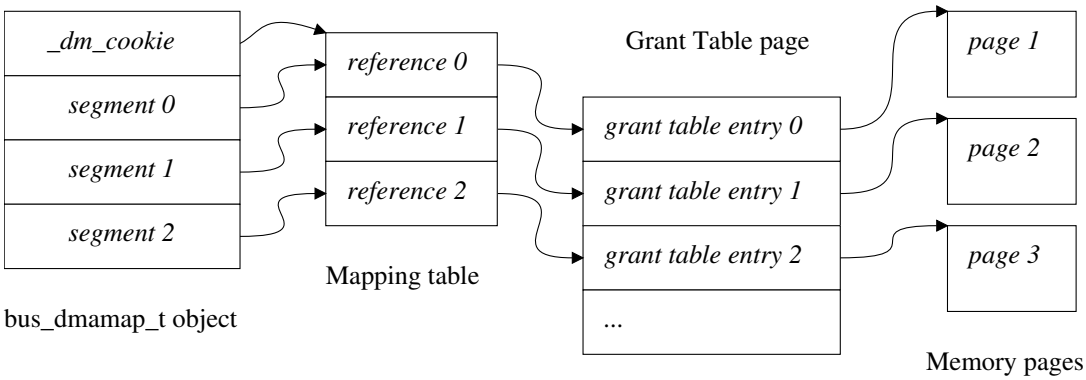

Figure 1: Interfacing Grant Tables with *bus_dma_load(9)*

calling to the system function.

7 XenStore

The XenStore is a hierarchical filesystem like property storage system that can be accessed via an interrupt driven producer/consumer ring interface in order to learn about configured virtual devices and their properties. This extends the hypercall interface with various information in the ASCII string format.

The XenStore ring page is usually allocated by the hypervisor and has to be mapped in by the guest into it's kernel virtual memory space via a call to the *pmap_kenter_pa()*. The physical address of the page is fetched with a hypercall. The event channel port for XenStore is also preallocated and can be learned via a hypercall as well.

The two most important hierarchies available throught the XenStore interface are "device/" and "backend/" that represent available virtual devices and their backend counterparts. Configuring these devices in large part requires setting properties within these hierarchies.

To issue a set of pre-defined commands a common interface akin to SCSI command submission was implemented. *xs_cmd()* takes an operation code, a node that the operation is performed upon as well as matrix of *iovec* structures. Depending on whether it's a *read* or a *write* operation the *iovec* matrix is ether allocated and returned by the XenStore driver or in case of a *write* operation by the callee. In case the matrix was allocated by the driver it needs to be disposed of by calling *xs_resfree()*.

A simpler property accessor interface *xs_getprop()* and *xs_setprop()* was introduced for when complexity of a generic *xs_cmd()* is not required.

Extensive string parsing in the kernel was avoided by providing a simple *iovec* based interface thus multiple independent strings are returned back to the caller as an array of vectors. Other parts of the infrastructure are required to perform data conversions and manipulations on their own which in most cases not strictly required as string comparisons cover most of the use cases.

8 Power management facilities

Xen provides a few power management capabilities to the controlling domain, namely it allows to signal halt, reboot and suspend events to the guest. In order to receive these events a guest is required to install a notification on the "control/shutdown" node via a XenStore *watch* operation.

Once installed, an asynchronous event mes-

sage will be delivered to the guest that will require it to read the "control/shutdown" property and act accordingly.

To execute an event callback in a safe environment it's scheduled via a *task_add(9)* to execute in the shared system task queue (*systq*) under the kernel lock. This allows us to perform graceful shutdown and reboot operations.

9 Virtual device attachment

In order for the guest to probe for virtual devices a XenStore directory listing operation must be issued for the "device/" node and then for every subdirectory of it another listing operation must be performed. This produces nodes like "device/vif/0" and "device/vif/1" that represent different virtual devices that have been configured for this virtual machine instance.

A standard procedure in this case is to utilize the BSD autoconfiguration framework [2] and specifically *config_found(9)* in order to attach devices configured in the kernel configuration file. For instance if "*xnf* at xen0*" configuration option is included in the kernel config file a specified match function for the *xnf(4)* device will be called and the driver can determine whether or not it should proceed with attachment. This decision is based on the provided *xen_attach_args* structure that records the node name and a few other bits of information to support device attachment procedure.

Default configuration of virtual networking interfaces employed by Xen allows for both paravirtualized and legacy PCI drivers to attach to the same network interface. It's done this way in order to simplify VM configuration and provide a simple fallback method. Thus it's imperative for the virtual network interface to claim ownership of the device and instruct the hypervisor to exclude the legacy PCI device from the PCI device tree. This is done via an I/O port operation once *xnf(4)* driver finishes the attachment process. This allows system operators to disable *xnf(4)* driver in the kernel via *config(8)* or User Kernel Config during boot and fall back to the legacy device driver without recompiling the kernel.

10 Virtual network interface

A driver for the virtual network interface is based around the idea that receive and transmit ring descriptors take grant table references to the networking stack buffers instead of physical addresses that regular hardware counterparts do.

Both receive and transmit rings are allocated as contiguous chunks of memory so that they can be associated with a single grant table reference. These grant table references are passed to the hypervisor as device properties "rx-ring-ref" and "tx-ring-ref" accordingly.

Each receive and transmit descriptor has its own grant table reference pointing to a single buffer not exceeding a page in size. In order to support fragmented chains in the transmission code path we need to ensure that the chain is not longer than 18 fragments (when scatter-gather operation is supported by the backend) and then extract each individual memory buffer in order to load it into the transmit descriptor map and associate with a transmit descriptor. This effectively transposes a tree like structure of a ring of mbuf chains into a flat ring of buffers that Netfront implements.

In order to tell the hypervisor that the device is ready to receive network traffic a *state* property needs to be set to the *connected* state value.

Driver supports receive and transmit TCP/UDP checksum offloading for both IPv4 and IPv6 packets. One of the quirks when dealing with checksums turns out to be almost exact emulation of Linux checksum offloading by Xen Netfront that seems to conflict with how OpenBSD IPv4 checksum offloading is supposed to work in conjunction with protocol checksum offloading.

11 Operation in the Amazon EC2

Antoine Jacoutot and Reyk Flöter did a great job with providing OpenBSD development images in the Amazon cloud. In order to create and upload an OpenBSD image ready to be deployed as an AMI a script [1] by Antoine can be used. It depends on on the "sysutils/ec2-api-tools" package and automates creation of a freshly installed OpenBSD system with an additional *rc* script that fetches SSH key and configures the primary networking interface.

12 Future Work

One of the issues exposed by the Netfront is lack of hardware interrupt moderation for the receive ring and therefore it's becomes somewhat easy to mount DoS attacks against it. Since OpenBSD does not have an infrastructure to polling mode of operation for networking interfaces we need to implement additional countermeasures.

To ensure compatibility with existing migration techniques we need to implement full suspend/resume functionality. It's controlled by the same "control/shutdown" node. In order to suspend the system a few steps must be taken such as disabling interrupts and after that a hypercall is issued that upon return signals either resume or a cancelled suspend.

A *diskfront* driver can be implemented to support paravirtualized storage devices. Possible benefits include speed improvements and potentially support for hot-pluggable storage devices such as Amazon S3 volumes.

Support for a native timecounter should be fairly easy to implement given that the Shared Information Page provides most of the data already.

[1] `https://github.com/ajacoutot/aws-openbsd`

13 Conclusion

This effort has proven that all of the Xen PVHVM infrastructure can be implemented in about 2500 lines of code with comments (excluding device drivers) in just a few C files (*/sys/dev/pv/xen.c* and */sys/dev/pv/xenstore.c*) and a few header files (*/sys/dev/pv/xenvar.c* and */sys/dev/pv/xenreg.h*). Existing and well-known system abstractions such as *bus_dma(9)* can be used to keep device driver implementation simple and comprehensible.

Acknowledgments

The author would like to thank Esdenera Networks GmbH for funding this work, OpenBSD developers, especially Reyk Flöter, Mark Kettenis, Martin Pieuchot, Antoine Jacoutot, Mike Larkin and Theo de Raadt for productive discussions and code reviews. Huge thanks to all our users who took their time to test, report bugs, submit patches and encourage development.

Availability

Scheduled to become available in OpenBSD 5.9. The whole stack will be enabled in the default "GENERIC" kernel as well as install ramdisk kernel "RAMDISK_CD", not requiring users to build their own kernels or install media.

References

[1] Jason Thorpe, A Machine-Independent DMA Framework for NetBSD, Usenix 1998 Annual technical conference.

[2] Chris Torek, Device Configuration in 4.4BSD, Lawrence Berkeley Laboratory, 1992.