Implementation of Xen PVHVM drivers in OpenBSD

Mike Belopuhov
Esdenera Networks GmbH
mike@esdenera.com

Tokyo, March 12 2016
The goal

Produce a minimal well-written and well-understood code base to be able to run in Amazon EC2 and fix potential problems for our customers.
The challenge

Produce a minimal well-written and well-understood code base to be able to run in Amazon EC2 and fix potential problems for our customers.
Requirements

Need to be able to:

- boot
Requirements

Need to be able to:

- boot: already works!
Requirements

Need to be able to:

- boot: *already works!*
- mount root partition
Requirements

Need to be able to:

- boot: *already works!*
- mount root partition: *already works!*
Requirements

Need to be able to:

- boot: *already works!*
- mount root partition: *already works!*
- support SMP
Requirements

Need to be able to:

▶ boot: already works!
▶ mount root partition: already works!
▶ support SMP: didn’t work on amd64
Requirements

Need to be able to:

- boot: already works!
- mount root partition: already works!
- support SMP: fixed shortly
Requirements

Need to be able to:

▶ boot: *already works!*
▶ mount root partition: *already works!*
▶ support SMP: *fixed shortly*
▶ perform “cloud init”
Requirements

Need to be able to:

▶ boot: *already works!*

▶ mount root partition: *already works!*

▶ support SMP: *fixed shortly*

▶ perform “cloud init”: requires PV networking driver. *Snap!*
Requirements

Need to be able to:

- boot: already works!
- mount root partition: already works!
- support SMP: fixed shortly
- perform “cloud init”: requires PV networking driver
- login into the system via SSH...
Requirements

Need to be able to:

- boot: *already works!*
- mount root partition: *already works!*
- support SMP: *fixed shortly*
- perform “cloud init”: requires PV networking driver
- login into the system via SSH... Same thing.
Outlook on the FreeBSD implementation

- Huge in size
Outlook on the FreeBSD implementation

- Huge in size

  "du -csh" reports 1.5MB vs. 124KB in OpenBSD as of 5.9
  35 C files and 83 header files vs. 4 C files and 2 headers
Outlook on the FreeBSD implementation

- Huge in size
- Needlessly complex

Overblown XenStore API, interrupt handling, . . .

Guest initialization, while technically simple, makes you chase functions all over the place.
Outlook on the FreeBSD implementation

- Huge in size
- Needlessly complex
- Clash of coding practices
Outlook on the FreeBSD implementation

- Huge in size
- Needlessly complex
- Clash of coding practices

Lots of code has been taken verbatim from Linux (where license allows)
Outlook on the FreeBSD implementation

- Huge in size
- Needlessly complex
- Clash of coding practices
- Questionable abstractions
Outlook on the FreeBSD implementation

- Huge in size
- Needlessly complex
- Clash of coding practices
- Questionable abstractions

Code-generating macros, e.g. `DEFINE_RING_TYPES`. Macros to “facilitate” simple producer/consumer arithmetics, e.g. `RING_PUSH_REQUESTS_AND_CHECK_NOTIFY` and friends.

A whole bunch of things in the XenStore: `xs_directory` dealing with an array of strings, use of `sscanf` to parse single digit numbers, etc.
Porting plans...

...were scrapped in their infancy.
Single device driver model

In OpenBSD a pvbus(4) driver performs early hypervisor detection and can set up some parameters before attaching the guest nexus device:

xen0 at pvbus?

The xen(4) driver performs HVM guest initialization and serves as an attachment point for PVHVM device drivers, such as the Netfront, xnf(4):

xnf* at xen?
HVM guest initialization

- The hypercall interface
Instead of defining a macro for every type of a hypercall we use a single function with variable arguments:

```c
xen_hypercall(struct xen.softc *, int op, int argc, ...)
```

Xen provides an ABI for amd64, i386 and arm that we need to adhere to when preparing arguments for the hypercall.
The hypercall page

Statically allocated in the kernel code segment:

```assembly
.text
.align NBPG
.globl _C_LABEL(xen_hypercall_page)
_C_LABEL(xen_hypercall_page):
.skip 0x1000, 0x90
```
(gdb) disassemble xen_hypercall_page
<xen_hypercall_page+0>:    mov $0x0,%eax
<xen_hypercall_page+5>:    sgdt
<xen_hypercall_page+6>:    add %eax,%ecx
<xen_hypercall_page+8>:    retq
<xen_hypercall_page+9>:    int3
...
<xen_hypercall_page+32>:    mov $0x1,%eax
<xen_hypercall_page+37>:    sgdt
<xen_hypercall_page+38>:    add %eax,%ecx
<xen_hypercall_page+40>:    retq
<xen_hypercall_page+41>:    int3
...
HVM guest initialization

- The hypercall interface
- The shared info page
HVM guest initialization

- The hypercall interface
- The shared info page
- Interrupt subsystem
Interrupts

- Allocate an IDT slot

  Pre-defined value of 0x70 (start of an IPL_NET section) is used at the moment.
Interrupts

- Allocate an IDT slot
- Prepare interrupt, resume and recurse vectors

Xen upcall interrupt is executing with an IPL_NET priority.
Xintr_xen_upcall is hooked to the IDT gate.
Xrecurse_xen_upcall and Xresume_xen_upcall are hooked to the interrupt source structure to handle *pending* Xen interrupts.
Interrupts

- Allocate an IDT slot
- Prepare interrupt, resume and recurse vectors
- Communicate the slot number with the hypervisor

A XenSource Platform PCI Device driver, xspd(4), serves as a backup option for delivering Xen upcall interrupts if setting up an IDT callback vector fails.
Interrupts

- Allocate an IDT slot
- Prepare interrupt, resume and recurse vectors
- Communicate the slot number with the hypervisor
- Implement API to (dis-)establish device interrupt handlers and mask/unmask associated event ports.

```c
int xen_intr-establish(evtchn_port_t, xen_intr_handle_t *, void (*handler)(void *), void *arg, char *name);
int xen_intr-disestablish(xen_intr_handle_t);
void xen_intr-mask(xen_intr_handle_t);
int xen_intr-unmask(xen_intr_handle_t);
```
Interrupts

- Allocate an IDT slot
- Prepare interrupt, resume and recurse vectors
- Communicate the slot number with the hypervisor
- Implement API to (dis-)establish device interrupt handlers and mask/unmask associated event ports.
- Implement events fan out

```c
Xintr_xen_upcall(xen_intr()):
    while(pending_events?)
        xi = xen_lookup_intsrc(event_bitmask)
        xi->xi_handler(xi->xi_arg)
```
Almost there: XenStore

- Shared ring with a producer/consumer interface
Almost there: XenStore

- Shared ring with a producer/consumer interface
- Driven by interrupts
Almost there: XenStore

- Shared ring with a producer/consumer interface
- Driven by interrupts
- Exchanges ASCII NUL-terminated strings
Almost there: XenStore

- Shared ring with a producer/consumer interface
- Driven by interrupts
- Exchanges ASCII NUL-terminated strings
- Exposes a hierarchical filesystem-like structure
Almost there: XenStore

- Shared ring with a producer/consumer interface
- Driven by interrupts
- Exchanges ASCII NUL-terminated strings
- Exposes a hierarchical filesystem-like structure

```
device/
device/vif
device/vif/0
device/vif/0/mac = '06:b1:98:b1:2c:6b'
device/vif/0/backend = '/local/domain/0/backend/vif/569/0'
```
Almost there: XenStore

References to other parts of the tree, for example, the backend
/local/domain/0/backend/vif/569/0:

- domain
- script
- mac
- type
- feature-rx-copy
- handle
- state
- online
- feature-sg
- feature-rx-copy
- uuid
- frontend
- frontend-id
- feature-gso-tcpv4
- feature-rx-flip
- hotplug-status
Almost there: Device discovery and attachment

- **device/**
  - **device/vif/**
    - **device/vif/0** (Netfront 0)
      - **xnf0**
    - **device/vif/1** (Netfront 1)
      - **xnf1**
  - **device/vbd/**
    - **device/vbd/768** (Diskfront)
      - **xdf0**
Enter Netfront

...or not!
Enter Netfront

Grant Tables are required to implement receive and transmit rings.
What’s in a ring?

Producer

Consumer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5
What’s in a ring?

Producer

Consumer

Buffer 1

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5
What’s in a ring?

Producer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5

Consumer

Buffer 1
Buffer 2
What’s in a ring?

Consumer

Producer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5

Buffer 1
Buffer 2
Buffer 3
What’s in a ring?
What’s in a ring?

Producer
Consumer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5

Buffer 3
Buffer 4
Buffer 5
What’s in a ring?

Consumer

Producer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5
Consumer

Buffer 3
Buffer 4
Buffer 5
What’s in a ring?

Producer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5

Consumer
What’s in a ring?

Buffer

Descriptor 1
Descriptor 2
Descriptor 3
Descriptor 4
Descriptor 5

Physical memory

0x4000

0x5000

Virtual memory

0xffff4000

0xffff3000

memcpy
bus_dma(9)

Since its inception, bus_dma(9) interface has unified different approaches to DMA memory management across different architectures.
bus_dma(9): Preparing a transfer

- bus_dmamap_create to specify DMA memory layout

```c
struct bus_dmamap {
    ...
    void **_dm_cookie;
    bus_size_t dm_mapsize;
    int dm_nsegs;
    bus_dmamap_segment_t *dm_segs[1];
};
typedef struct bus_dmamap_segment {
    bus_addr_t ds_addr;
    bus_size_t ds_len;
    ...
} bus_dmamap_segment_t;
```
bus_dma(9): Preparing a transfer

- `bus_dmamap_create` to specify DMA memory layout
- `bus_dmamem Alloc` to allocate physical memory
bus_dma(9): Preparing a transfer

- `bus_dmamap_create` to specify DMA memory layout
- `bus_dmamem_alloc` to allocate physical memory
- `bus_dmamem_map` to map it into the KVA
An example of buffer spanning multiple pages
bus_dma(9): Preparing a transfer

- `bus_dmamap_create` to specify DMA memory layout
- `bus_dmamem_alloc` to allocate physical memory
- `bus_dmamem_map` to map it into the KVA
- `bus_dmamap_load` to connect allocated memory to the layout
Buffer loaded into the segment map
bus_dma(9): Preparing a transfer

- `bus_dmamap_create` to specify DMA memory layout
- `bus_dma_mem_alloc` to allocate physical memory
- `bus_dma_mem_map` to map it into the KVA
- `bus_dmamap_load` to connect allocated memory to the layout
- Signal *the other side* to start the DMA transfer
bus_dma(9): Transfer completion

- bus_dmamap_unload to disconnect the memory
bus_dma(9): Transfer completion

- bus_dmamap_unload to disconnect the memory
- bus_dmamem_unmap to unmap the memory from the KVA
bus_dma(9): Transfer completion

- `bus_dmamap_unload` to disconnect the memory
- `bus_dmamem_unmap` to unmap the memory from the KVA
- `bus_dmamem_free` to give the memory back to the system
bus_dma(9): Transfer completion

- bus_dmamap_unload to disconnect the memory
- bus_dmamem_unmap to unmap the memory from the KVA
- bus_dmamem_free to give the memory back to the system
- bus_dmamap_destroy to destroy the DMA layout
Netfront RX ring

Consists of a 64 byte header and a power-of-2 number of 8-byte descriptors that fit in one page of memory.

```c
#define XNF_RX_DESC 256
struct xnf_rx_ring {
    uint32_t rxr_prod;
    uint32_t rxr_prod_event;
    uint32_t rxr_cons;
    uint32_t rxr_cons_event;
    uint32_t rxr_reserved[12];
    union xnf_rx_desc rxr_desc[XNF_RX_DESC];
} __packed;
```
Netfront RX ring

Each descriptor can be a “request” (when announced to the backend) or a “response” (when receive is completed):

```c
union xnf_rx_desc {
    struct xnf_rx_req rxd_req;
    struct xnf_rx_rsp rxd_rsp;
} __packed;
```
Netfront RX ring

Descriptor contains a reference (rxq_ref) of a page sized memory buffer:

```c
struct xnf_rx_req {
    uint16_t rxq_id;
    uint16_t rxq_pad;
    uint32_t rxq_ref;
} __packed;
```
bus_dma(9) usage for the Netfront RX ring

Create a shared page of memory for the ring data:

- `bus_dmamap_create` a single entry segment map
bus_dma(9) usage for the Netfront RX ring

Create a shared page of memory for the ring data:

- **bus_dmamap_create** a single entry segment map
- **bus_dmamem_alloc** a single page of memory for descriptors
bus_dma(9) usage for the Netfront RX ring

Create a shared page of memory for the ring data:
- bus_dmamap_create a single entry segment map
- bus_dmamem_alloc a single page of memory for descriptors
- bus_dmamem_map the page and obtain a VA
bus_dma(9) usage for the Netfront RX ring

Create a shared page of memory for the ring data:

- `bus_dmamap_create` a single entry segment map
- `bus_dmamem_alloc` a single page of memory for descriptors
- `bus_dmamem_map` the page and obtain a VA
- `bus_dmamap_load` the page into the segment map
bus_dma(9) usage for the Netfront RX ring

Now we can communicate the location of this page with a backend, but first we need to create packet maps for each descriptor (256 in total) so that we can connect memory buffers (mbuf clusters) with references in the descriptor.

We don’t need to allocate memory for buffers since they’re coming from the mbuf cluster pool.
bus_dma(9) usage for the Netfront RX ring

Whenever we need to put the cluster on the ring we just need to perform a `bus_dma_map_load` operation on an associated DMA map and then set the descriptor reference to the value stored in the DMA map segment...

*Right?*
bus_dma(9) usage for the Netfront RX ring

Whenever we need to put the cluster on the ring we just need to perform a `bus_dma_map_load` operation on an associated DMA map and then set the descriptor reference to the value stored in the DMA map segment...

*Right? Wrong!*

RX and TX descriptors use `references`, not physical addresses!
Grant Table reference

entry 0
entry 1
entry 2
entry 3
entry 4
...
entry 511

Grant Table
page 0 (0-511)

Physical memory

Buffer

Virtual memory

0xffff5000
0xffff4000
0xffff3000

0x5000
0x4000
0x3000
Grant Table entry

Grant Table entry version 1 contains a frame number, flags (including permissions) and a domain number to which the access to the frame is provided.
Grant Table entry

Grant Table entry *version 1* contains a *frame number*, flags (including permissions) and a domain number to which the access to the *frame* is provided.

If only we could add a translation layer to the bus_dma(9) interface to convert between physical address and a frame number.
bus_dma(9) and Grant Tables

Luckily bus_dma(9) interface allows us to use custom methods:

```c
struct bus_dmamap_tag xen_bus_dmamap_tag = {
    NULL,     // <-- another cookie
    xen_bus_dmamap_create, xen_bus_dmamap_destroy,
    xen_bus_dmamap_load, xen_bus_dmamap_load_mbuf,
    NULL, NULL, xen_bus_dmamap_unload,
    xen_bus_dmamap_sync, _bus_dmamem_alloc,
    NULL, _bus_dmamem_free,
    _bus_dmamem_map, _bus_dmamem_unmap,
};
```
Xen bus_dma(9) interface

After creation of the DMA segment map structure via `bus_dmamap_create`, we can create an additional array for the purpose of mapping Grant Table references to physical addresses of memory segments loaded via `bus_dmamap_load` and set it to be a DMA map cookie!
Xen bus_dma(9) interface

After creation of the DMA segment map structure via bus_dmamap_create, we can create an additional array for the purpose of mapping Grant Table references to physical addresses of memory segments loaded via bus_dmamap_load and set it to be a DMA map cookie!

We have to preallocate Grant Table references at this point so that we can perform bus_dmamap_load and bus_dmamap_unload sequences fast. Since we create DMA maps in advance, xen_grant_table_alloc can take time to increase the number of Grant Table pages if we’re running low on available references.
Xen `bus_dma(9)` interface

When we’re ready to put the buffer on the ring we call `bus_dmamap_load` that populates the DMA map segment array with physical addresses of buffer segments.
Xen bus_dma(9) interface

When we’re ready to put the buffer on the ring we call bus_dmamap_load that populates the DMA map segment array with physical addresses of buffer segments.

Once it’s done we can punch those addresses into Grant Table entries that we have preallocated and set appropriate permission flags via xen_grant_table_enter.
Xen bus_dma(9) interface

When we’re ready to put the buffer on the ring we call bus_dmamap_load that populates the DMA map segment array with physical addresses of buffer segments.

Once it’s done we can punch those addresses into Grant Table entries that we have preallocated and set appropriate permission flags via xen_grant_table_enter.

We record physical addresses in our reference mapping array and swap values in the DMA map segment array to Grant Table references. This allows the Netfront driver to simply use these values when setting up ring descriptors.
Xen bus_dma(9) interface

During bus_dmamap_unload we perform the same operations backwards: xen_grant_table_remove clears the Grant Table entry, we swap physical addresses back and call into the system to finish unloading the map.

If we wish to destroy the map, bus_dmamap_destroy will deallocate Grant Table entries via xen_grant_table_free and then destroy the map itself.
Announcing Netfront rings

In order to announce locations of RX and TX rings, Netfront driver needs to set a few properties in its “device” subtree via XenStore API.
Announcing Netfront rings

In order to announce locations of RX and TX rings, Netfront driver needs to set a few properties in its “device” subtree via XenStore API.

A Grant Table reference for the RX ring data needs to be converted to an ASCII string and set as a value for the “rx-ring-ref” property.
Announcing Netfront rings

In order to announce locations of RX and TX rings, Netfront driver needs to set a few properties in its “device” subtree via XenStore API.

A Grant Table reference for the RX ring data needs to be converted to an ASCII string and set as a value for the “rx-ring-ref” property.

TX ring location is identified by the backend with the “tx-ring-ref” property.
Operation in the Amazon EC2

Amazon Machine Image (AMI) is required to contain some knowledge of the EC2 cloud to be able to obtain an SSH key during the instance creation.
Operation in the Amazon EC2

Amazon Machine Image (AMI) is required to contain some knowledge of the EC2 cloud to be able to obtain an SSH key during the instance creation.

Since the information is provided by the EC2 via an internal HTTP server, it’s required that the first networking interface comes up on startup with a DHCP configuration and fetches the SSH key.
Operation in the Amazon EC2

Amazon Machine Image (AMI) is required to contain some knowledge of the EC2 cloud to be able to obtain an SSH key during the instance creation.

Since the information is provided by the EC2 via an internal HTTP server, it’s required that the first networking interface comes up on startup with a DHCP configuration and fetches the SSH key.

This procedure is called “cloud-init” and obviously requires some additions and adjustments to the OpenBSD boot procedure.
Operation in the Amazon EC2

- Public images of 5.8-current snapshots were provided regularly by Reyk Flöter (reyk@) and Antoine Jacoutot (ajacoutot@) in several “availability zones”.

Operation in the Amazon EC2

- Public images of 5.8-current snapshots were provided regularly by Reyk Flöter (reyk@) and Antoine Jacoutot (ajacoutot@) in several “availability zones”.

- Antoine has created a few scripts to automate creation and upload of OpenBSD images to the EC2 using ec2-api-tools as well as perform minimal “cloud-init” on the VM itself.
Operation in the Amazon EC2

- Public images of 5.8-current snapshots were provided regularly by Reyk Flöter (reyk@) and Antoine Jacoutot (ajaxoutot@) in several “availability zones”.

- Antoine has created a few scripts to automate creation and upload of OpenBSD images to the EC2 using ec2-api-tools as well as perform minimal “cloud-init” on the VM itself.

- We would like to provide an OpenBSD 5.9-release image in the Amazon Marketplace.
Future work

- Support for the PVCLOCK timecounter
Future work

- Support for the PVCLOCK timecounter
- Support for suspend and resume
Future work

- Support for the PVCLOCK timecounter
- Support for suspend and resume
- Driver for the Diskfront interface
Future work

- Support for the PVCLOCK timecounter
- Support for suspend and resume
- Driver for the Diskfront interface
- Support for the PCI pass-through
I’d like to thank Reyk Flöter and Esdenera Networks GmbH for coming up with this amazing project, support and letting me have a freedom in technical decisions.

I’d also like to thank 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.

Special thanks to Wei Liu and Roger Pau Monné from Citrix for being open to questions and providing valuable feedback as well as other present and past contributors to the FreeBSD port. Without it, this work might not have been possible.
Questions?
Thank you for attending the talk!

ありがとうございました！