Virtualization under *BSD
The case of Xen

Jean-Yves Migeon – jym@NetBSD.org

EuroBSDcon 2011

18 août 2012

Table of Contents I

1. So Xen. . . What is that thing ?
   - A brand ?
   - Why did it become so central ?
   - So Xen. . .
2. Xen, the hypervisor
   - Overview
3. The privileged domain : dom0
   - whoami : dom0
   - Xen tools
   - XenStore
4. Diving into Xen's world
   - The smallest system : Xen and dom0
   - Virtual memory
   - Virtual memory layout : 32 bits
   - Virtual memory layout : 64 bits
   - Impacts of para-virtualization

Table of Contents II

5 Virtualizing devices
- Split device drivers
- Grant table(s)
- I/O rings
- Event channels/ports

6 domUs : the unprivileged brethren
- What are they ?
- The big picture !

7 Virtualization and ahead
- From PV to hardware
- Hardware Assisted Virtualization
- Device virtualization : IOMMU and SR-IOV
- High availability : live migration, Remus

8 Conclusion

9 Backup
- Xen's history

Table of Contents III

- Turning an OS into a dom0

So Xen. . . A brand ?

An innocent question, really. . .

XenDesktop

CITRIX

XenApp

Xen TM

Why did it become so central ?

x86 was not really virtualization friendly for kernels :

- certain sensitive instructions (SIDT) could not be trapped
- complex model : 4 privilege levels (rings), memory model uses segments & pages, real mode. . .

Solutions up to then :

emulation	very slow and error-prone (QEMU)
binary rewriting	complex to get right (VMWare)

Xen chose a different path : para-virtualization . Guest knows that it runs in a pseudo-virtualized x86 environment. Sensitive instructions are replaced with hypervisor calls.

Open Source communities rapidly took interest in it, which brought lots of momentum to the OS virtualization movement.

So Xen. . .

Let's look at what the majority sees through *Xen* nowadays :

- first and foremost, an **hypervisor** , designed (a bit) like a microkernel :
  - minimalist in nature
  - drives : CPUs, (virtual) memory, and everything that is security critical and requires high privileges
- a privileged domain, known as **dom0** :
  - implements most of the drivers (for hardware support)
  - focal point for hypervisor's management
- tools and services used to manage Xen's system : start/stop VMs, expose system information, control ressources. . .
  - through `xend` , `xm(1)` , XenStore . . .

Xen, the hypervisor

Minimalist in nature : drives hardware (like a pilot), delegates navigation to dom0.

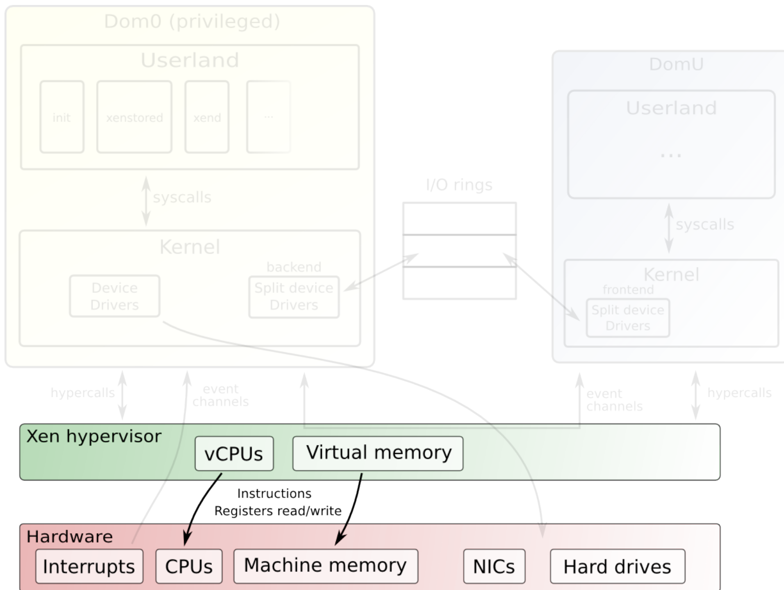
Some people classify Xen as a "Type 1" VMM, meaning that it runs directly on metal .

It does not give the steering wheel away, so it has to offer abstractions somehow :

Syscalls	Hypercalls
Signals/Interrupts	Events/Ports
`shm*`, `mmap(2)`	update_va_mapping()
POSIX `mqueue(3)`	I/O rings
ACPI tables, `sysctl(7)`	XenStore, Xenbus

Looks rather familiar, heh ☺

Piling up...



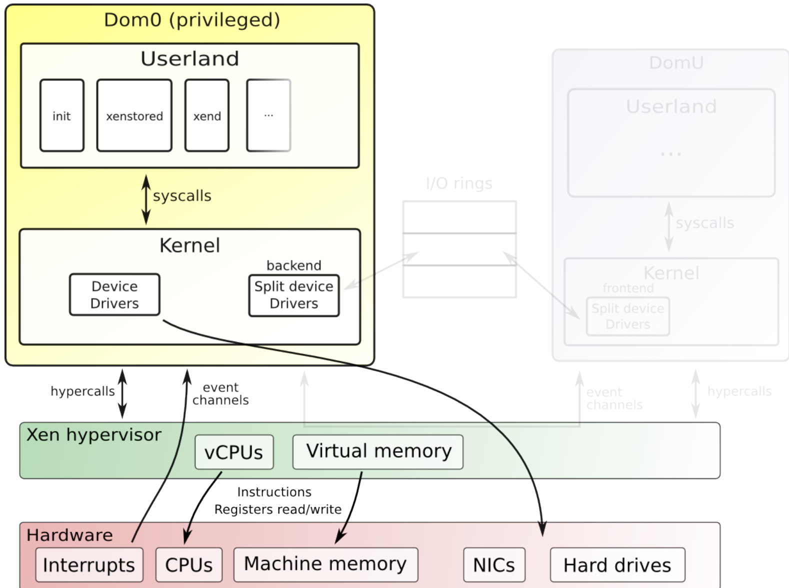
The privileged domain : dom0

Xen is minimalistic : almost no drivers. It needs a co-pilot that tells him what to do to arrive at destination : the **dom0** .

- the first general purpose OS to boot just after hypervisor
- para-virtualized – Xen does not hide from him (yet), dom0 has no direct access to hardware
- makes hypercalls to Xen to manage system :
  - HDs, NICs, USB controllers,...
  - administrativia : `xm/xl(1)`
- should be fairly small and reliable, as it virtualizes hardware through driver backends (99,9% of the time)

Although it doesn't drive, its almost as critical as Xen for everyone's safety.

Piling up...



Xen tools

Lastly, we need tools and services to manage the whole virtualization environnement.

They evolved over time, especially in :

functionality: ballooning, CPU pinning, scheduling,...

low level APIs: tools from one revision are not compatible with an hypervisor of another rev (no backwards compat ☹)

Main control command :

Xen 3.$*$ series : xm(1), written in Python.

Xen 4.$*$ series : xl(1), in C, lighter than `xm` (deprecated).

Used for pretty much everything :

- start/pause/stop/suspend/migrate VMs
- get information from hypervisor (dmesg , info)
- control/monitor ressources (vcpu-$*$, mem-$*$, sched-$*$, block/network activity with xm top)

Xen tools

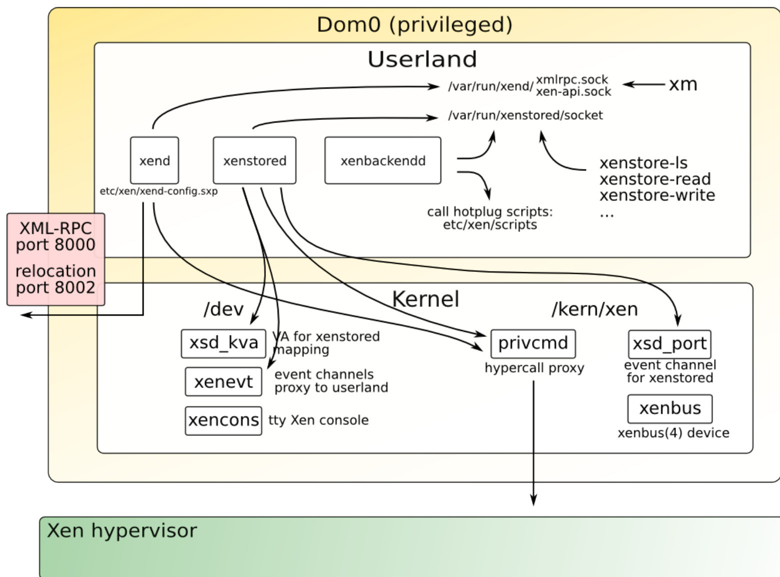
There are more things happening in the background :

- **xend** daemon that handles commands submitted via xm/xl
- **xenstored** XenStore facility ; centralizes data about VMs and virtual drivers. Used by domains to publish information about them.
- **xenbackendd** backend manager daemon ; handles events that concerns backend drivers.

They used to implement their own mechanisms on-top of low level libraries (libxc, libxenguest,. . .).

Recent versions of Xen (4 and up) provide `libxenlight` as a solid foundation for the whole toolstack. Written in C, aims at being lightweight.

Xen : system's overview



XenStore

Central storage, accessible to running guests (dom0 & domUs) and tools. It is a very simple tree that stores key ⇒ value pairs .

Accessible via :

- `xenstore-ls`, `xenstore-read`,. . . from userland
- `xenbus(4)`, a communication channel between domains' kernels

XenStore, via `xenbus(4)`, allows a domain to query (and publish) information about it .

Main use of this feature is for configuring split device drivers : domains can register "watches" that will be triggered when something happens in XenStore (like a device's state change).

XenStore

```
# xenstore-ls /local/domain/35
...
device = ""
 vbd = ""
  768 = ""
   state = "4"
   backend = "/local/domain/0/backend/vbd/35/768"
   ring-ref = "511"
   event-channel = "5"
 vif = ""
  0 = ""
   mac = "00:16:3e:00:00:32"
   vifname = "xennet0"
   tx-ring-ref = "510"
   rx-ring-ref = "509"
   ...
memory = ""
 target = "65536"
```

Diving into Xen's world

Xen + dom0 is the smallest possible system you can encounter. Xen is nothing more than a hardware abstraction layer here.

As said, it hides the mess of x86 behind it; which means that you have to port your OS on Xen before it can acts as a dom0 .

Xen has ≃ 40 hypercalls [1]. They cover low level x86 operations and replace privileged ones : MMU PD/PT updates, interrupt/channel setup, VCPU control, domains management.

As dom0 cannot have access to hardware anymore, Xen provides structures to pass down information during boot : `xen_start_info`, `shared_info_t`.

Moving a kernel out of its "reserved" ring 0 has consequences.

1. documented in Xen headers, sys/arch/xen/include/xen3-public/xen.h

Start info, shared info structures

when domain starts...
address in %esi:

```
struct start_info {
    ...
    /* MACHINE address of shared info struct. */
    unsigned long shared_info;
    ...
    /* XenStore shared page and event channel */
    /* MACHINE page number of shared page */
    xen_pfn_t store_mfn;
    /* Event channel for XenStore communication. */
    uint32_t store_evtchn;
    ...
    int8_t cmd_line[MAX_GUEST_CMDLINE];
};
```

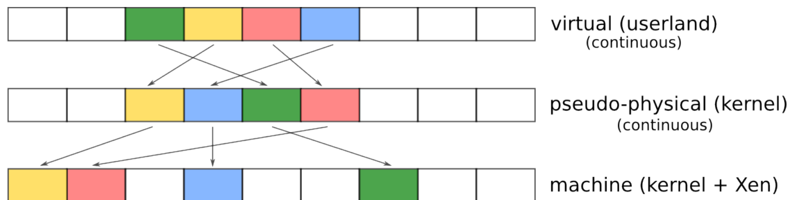
```
struct shared_info {
    /* stores VCPUs information */
    struct vcpu_info vcpu_info[MAX_VIRT_CPUS];
    /* masks for pending event channels interrupts */
    unsigned long evtchn_pending[sizeof(unsigned long) * 8];
    unsigned long evtchn_mask[sizeof(unsigned long) * 8];

    /* time-tracking */
    uint32_t wc_version;
    uint32_t wc_sec;
    uint32_t wc_nsec;

    /* MD stuff: PFN => MFN lists, maximum address, ... */
    struct arch_shared_info arch;
}
```

```
struct vcpu_info {
    /* per-VCPU event channel interrupts */
    uint8_t evtchn_upcall_pending;
    uint8_t evtchn_upcall_mask;
    ...
    /* MD specific info, like %cr2 for page faults */
    struct arch_vcpu_info arch;
    ...
};
```

Virtual memory

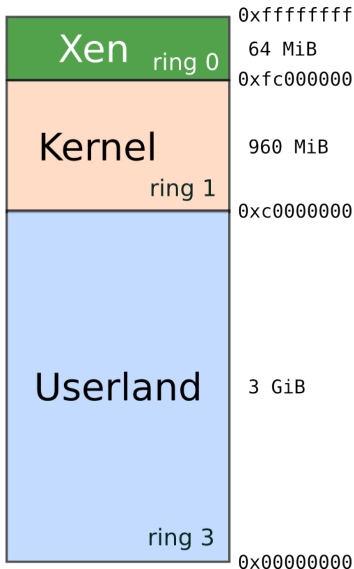
As Xen virtualizes memory between different OS, it adds a third level of indirection in the typical VM model :

- **virtual** addresses the majority knows and uses daily
- **machine** real addresses, managed by kernel via the MMU and. . .
- **pseudo-physical** implementation choice made by Xen [2] : only low level parts have to know about virtualization and MFNs, rest uses GPFNs.



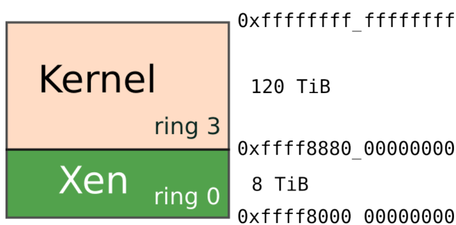
2. to ease portability

Virtual memory layout : 32 bits

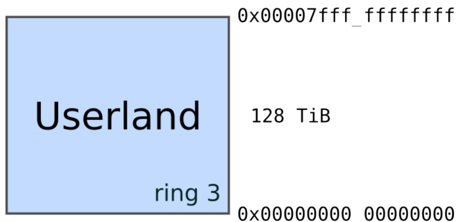


- 4 level rings with 32 bits
  - Xen runs in ring 0
  - steals VM in the upper part
    - 64 MiB (without PAE)
    - 128 MiB (with PAE)
  - pushes kernel to ring 1
- TLB flushes heavy with x86
  - avoid full context switch
  - use segmentation to protect hypervisor from kernel

Virtual memory layout : 64 bits



non-canonical addresses



- 2 level rings with 64 bits
  - Xen runs in ring 0
  - pushes kernel to ring 3
  - steals VM in the low portion of upper part
- TLB flushes heavy with x86
  - cannot use segmentation to protect hypervisor here
  - kernel and userland address spaces are not mapped together
  - userland gets mapped in kernel address space when needed

Impacts of para-virtualization

Luckily, the majority of a Unix kernel does not require supervisor mode to run. Affected parts are mainly MD components :

- virtual memory handling , which is the most significant issue
  - Xen helps here with assistance mechanisms via `vm_assist()` hypercall : shadow page tables, writable page tables
- initialization : OS boots in protected mode
  - real mode is not managed, neither is V86.
- kernel runs in an unprivileged mode
  - all privileged instructions have to be converted to their hypercall equivalent
  - kernel is not all-powerful anymore, so it cannot perform operations that were allowed before : PD/PT overwrites, segment games,...

Virtual memory is the difficult part : it is performance critical, and hypercalls are not "free" (context switch).

Split device drivers

The Xen virtual drivers are "split" in two parts :

backend handles multiplexing for the real device

frontend generic virtual driver, used by domUs

Type	Backend	Frontend
Block	xbdback(4)	xbd(4)
Network	xvif(4)	xennet(4)
PCI	pciback(4)	xpci(4)

Except for network and block devices, each Xen split driver type implements its own communication model.

All split drivers rely on these to work :

- grant tables mechanism
- I/O rings
- event channels (also called "ports")

Grant table(s)

Grant tables are the main facility used to establish memory mappings between domains, and build IPCs; most notably, I/O rings .

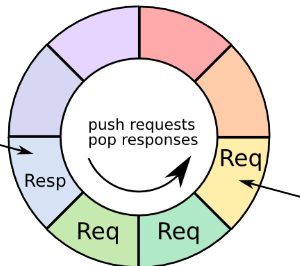
Grant tables are setup through the `grant_table_op` hypercall. When a page gets used as a grant table, it contains entries like :

```
struct grant_entry {
    /* GTF_xxx: various type and flag information. */
    uint16_t flags;
    domid_t  domid; /* domain being granted privileges */
    uint32_t frame; /* MFN (real address >> PAGE_SHIFT) */
};
```

Once setup, a grant reference is returned. This ref is used by domains to establish the mapping later, and is generally stored in XenStore.

I/O rings

```
struct blkif_response {
  /* copied from request */
  uint64_t id;
  /* copied from request */
  uint8_t  operation;
  /* RESP status */
  int16_t  status;
};
```




```
struct blkif_request {
  ...
  uint8_t  nr_segments; /* number of segments  */
  uint64_t id;          /* private guest value */
  ...
  struct blkif_req_seg seg[SEGMENTS_PER_REQUEST];
};
```

```
struct blkif_request_segment {
  grant_ref_t gref; /* grant ref buffer frame  */
  uint8_t     first_sect, last_sect;
};
```

Event channels/ports

Event channels (aka. ports) are like virtual interrupts . They are managed by a domain through the `event_channel_op()` hypercall. They can be categorized into three types :

- **VIRQ** interrupts associated with a virtual device, like a timer (`VIRQ_TIMER`).
- **PIRQ** physical interrupts, bound to a real device. This is mainly used by Xen and dom0 to virtualize real IRQs.
- **interdomain** used to establish a virtual interrupt line between two domains. Used in conjunction with I/O rings to mimic real device functionality.

For interdomain communications, the event identifier is typically stored in XenStore, and fetched by split device drivers to establish the channel.

domUs : what are they ?

The main interest of all this : running unprivileged guests, aka. domUs .

Typically not allowed to access hardware , even via hypercalls.

It uses virtual devices for communication, via frontend drivers :

- XenStore, so it can query for virtual device configuration
- virtual console, `xencons`
- network device, `xennet(4)`
- block device, `xbd(4)`

These are very simple, generic devices : the main reason why it is easier to port and run an OS as domU rather than a dom0.

What are they

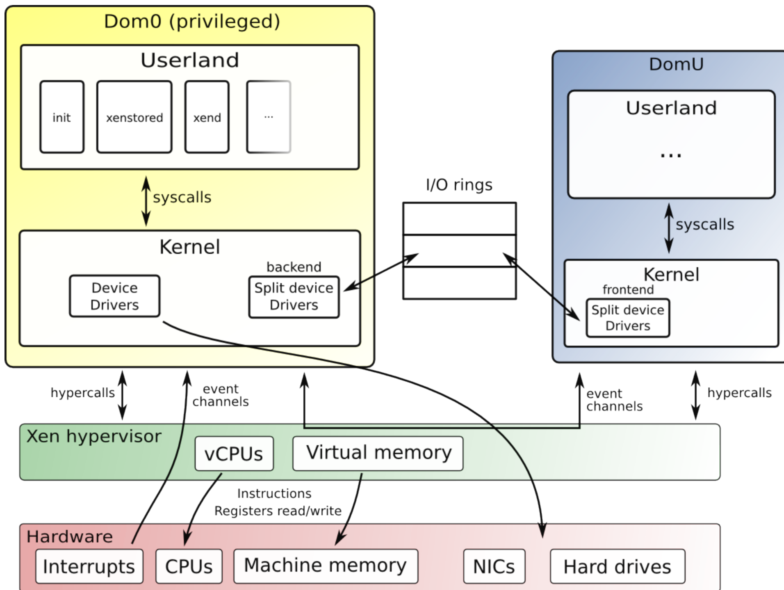
As virtualization capacities of x86 evolve, the frontier between dom0 and domU becomes blurry . A domU can :

- run real devices , via PCI passthrough frontend (`xpci(4)`)
  - with a compromise on security, especially without IOMMU...
- host virtual driver backends , when you want to move things out of dom0 [3]

domUs are stripped down versions of a dom0 ; even more so now that hardware has improved virtualization support, where an unmodified OS can run as a domU (with degraded I/O performance).

3. see Qubes' architecture for Storage Domain : http ://qubes-os.org/files/doc/arch-spec-0.3.pdf

The big picture !



From PV to hardware

In the early days of Xen, virtualization was handled at software level :

- patching source to replace instructions with hypercalls
- add abstractions (pseudo-physical addresses) for virtual memory management
- writing down virtual drivers to act as real devices
- rely on a general purpose OS to perform multiplexing :
  - bridging, routing for network
  - using generic frameworks like `disk(9)`
  - PCI ring to proxy PCI commands towards dom0

These can be replaced with hardware alternatives, except for the code that has to configure them (Xen, and ≃dom0).

Hardware Assisted Virtualization, HVM

Emerged in 2006, rapidly supported by Xen (3.0) ; popularized the technology : you could run Windows in a VM at ≃native speed (except I/Os).

HVM avoids modifying the domU OS ⇒ no requirement for PV :

- domU do not bother about virtualization : hypervisor does.
- hmm, remember the `vmxassist` issues when booting FreeBSD, because Intel HVM cannot handle real mode ?

Para-virtualization	Intel (VT-x)	AMD (AMD-V)
PV	cpuid : VMX flag	cpuid : SVM flag
Hypervisor calls	native x86 instructions	
Pseudo-physical addr.	EPT [4]	RVI/NPT [5]
PV frontends	emulated devices, via QEMU-dm	

4. Extended Page Tables
5. Rapid Virtualization Indexing/Nested Page Tables

Device virtualization : IOMMU and SR-IOV

Device virtualization encompasses different topics :

- **IOMMU** for I/O device ⇒ machine address translations
- **SR-IOV** for sharing a device between different VMs

IOMMU is mainly handled by Xen [6] : it configures the unit so that a device cannot access memory not belonging to the domU.

Single Root I/O virtualization allows any PCIe device to announce physical and virtual functions. SR-IOV can be viewed as a device virtualization solution, just like dom0 does with real hardware and virtual drivers , except that it is implemented in hardware.

Both are transparent to domU guests. SR-IOV requires support in dom0 (not supported with NetBSD). I/O virtualization is still an evolving technology, at least under x86.

---

6. by passing "iommu=1" to Xen during boot

High availability : live migration, Remus

Migration is the act of moving a domU from host A to host B . It is "live" when done at runtime with minimal downtime (≃200ms).

Xen controls VM usage, so it knows what/when a guest modifies its memory. Useful for migration : can propagate changes on-the-fly while keeping the domU running.

Atop of that we can implement simple active-passive high availability : have a VM act as the active part, and let changes propagate to a passive one.

When a (hardware) failure occurs, zap. This is what Remus does, broadly (Xen 4 and up). The challenge being that guest should remain unaffected by the fault.

HVM is a bit more tricky to suspend though : more states have to be preserved by Xen, like CPUs and emulated devices.

Challenges

Xen :

- strong competition with KVM
  - performance is often pushed forward
    - not much about security (critical for an hypervisor)
  - toolstack makes the difference (especially for cloud builders)
    - KVM lacks maturity here, compared to Xen (EC2, Rackspace)
- not as user-friendly as a Virtualbox or KVM/QEMU

*BSD :

- difficult to get people on board
  - especially to interact with Xen community
  - although it has lots of interesting subjects : kernel, operating system, userland, toolstack, GUIs. . .
- stack continues to grow : when does it stop ?
  - HVM ⇒ Xen ⇒ kernel ⇒ jails ⇒ virtual machines/emulators
  - less obvious for NetBSD : missed jails, so pushed Xen really hard
- all efforts concentrated on Linux
  - though running the same thing everywhere is kind of boring ☹

Questions

Questions ?

Xen... a tale ?

And to make things easier : its history for 3.x was... a bumpy ride :

2003-2005	1.0, 2.0	First stable releases : PV [7] only, Linux and NetBSD. block/device virtual drivers.
2005	3.0	HVM [8] guests, 64-bits, SMP, new concepts : XenStore and xenbus(4) **Buggy PAE. No support in upstream Linux**
2007	3.1	Lots of bug fixes, 32 bits PAE domains with 64-bits Xen XenSource acquired by Citrix.

...

7. Para-Virtualization
8. Hardware Virtual Machine, e.g. Hardware virtualization

Xen... a tale ?

2008	3.2	ACPI, PCI pass-through & IOMMUs, bug fixes **Xen Interface changes, incompatibilities between Linux distros**
	3.3	larger VT-x/VT-d support, power management. **Dumped !PAE support. XenSource's Linux stucked at 2.6.18**
2009	3.4	Focusing on RAS [9], power management. FreeBSD-8 runs as PV domU.
2010	4.0	Upstream support for Linux. Dumped `xm/xend` (Python) for a lighter altnerative : xenlight (`xl`).
2011	4.1	CPU pools, super pages, memory inspection API (≃VMsafe), `xl` refinements.

9. Reliability, Availability, Serviceability

Turning an OS into a dom0

Coincidently, making a dom0 is not really difficult when you already have a PV domU. The biggest differences being :

- dom0 handles real devices, so interfaces have to be adapted :
  - BIOS and ACPI mapping, for periph. enumeration.
  - bus_dma(9), bus_space(9), and IRQs (through event channels).
- dom0 runs backend drivers, notably block (`xbdback(4)`) and network (`xvif(4)`) devices.
- userland mechanisms expected by Xen tools : /kern/xen (or /dev/xen) entries.

Portability helps a lot here : the ugliness does not spread to MI parts, maintenance burden is acceptable. It took years for Linux to have upstream support for dom0, NetBSD got Xen 3 dom0 with -4 with a port that remained stable.