Lecture 11 - Xen and the Art of Virtualization

Barham, Dragovic, Fraser, Hand, Harris, Ho, Neugebauer, Pratt and Warfield (Cambridge) · SOSP 2003. A hypervisor that runs commodity operating systems at near-native speed by changing the guest kernel instead of hiding x86's flaws. Implementation details from David Chisnall, The Definitive Guide to the Xen Hypervisor (2007).

Contents

01 The x86 problem

02 Xen’s design

03 CPU: rings and hypercalls

04 Memory

05 Events, time and scheduling

06 I/O: split drivers and rings

07 Evaluation

08 After the paper

09 Takeaways

01

The x86 problem

Why x86 resisted virtualization

Popek and Goldberg (1974): a machine is virtualizable if every sensitive instruction is privileged, so it traps. Robin and Irvine (2000) found 17 x86 instructions that break this rule.

01

Silent failures

Some privileged operations just do nothing outside ring 0 instead of trapping, so the hypervisor never gets a chance to emulate them.

02

Leaky state

LAR, LSL and SIDT read segment and descriptor state without trapping, so a guest can see what the hypervisor rearranged.

03

Hardware page tables

The MMU walks page tables itself and the TLB is untagged, so every address-space switch flushes it.

WHY BOTHER WITH X86

Alpha would have been far easier to virtualize, but its installed base was tiny. x86 ran the world’s servers, so it was the architecture worth solving.

Robin and Irvine: where x86 breaks the rule

Their 2000 analysis of the Pentium found 17 instructions that are sensitive but don’t trap. Grouped by what goes wrong in ring 1:

INSTRUCTIONS	WHAT GOES WRONG OUTSIDE RING 0	HOW XEN COPES
`SGDT SIDT SLDT SMSW STR`	they read where the system tables live and other control state, without trapping	harmless once the guest knows it’s virtualized; ports don’t rely on them
`PUSHF POPF`	**POPF silently ignores changes to the interrupt flag, so a kernel’s “restore interrupts” does nothing**	**interrupts are disabled with the event mask in the shared info page (section 05)**
`LAR LSL VERR VERW, PUSH CS, MOV from segments`	they report the real privilege level and descriptors, so ring 1 shows through	acceptable: a PV kernel knows it runs in ring 1
`far CALL JMP INT RET`	privilege checks use the real ring and the real descriptor tables	Xen validates every descriptor a guest installs (section 03)

- J. S. Robin and C. E. Irvine, USENIX Security 2000: the paper’s reference [36]. Grouping follows their two categories, sensitive registers and protection-system references.

Four ways around it

Bochs interprets everything; VMware rewrites binaries at run time; Xen rewrites the guest at design time; Intel and AMD later fixed the hardware.

APPROACH	HOW SENSITIVE OPERATIONS ARE HANDLED	GUEST KERNEL	COST
Full emulation (Bochs)	interpret every instruction in software; nothing runs directly	unmodified, even for a different host ISA	tens of times slower than native, even well tuned (lecture 10)
Binary rewriting (VMware)	scan code as it runs and patch unsafe instructions to call an emulator	unmodified	80–97% of native in Chisnall's estimate, worse on kernel-heavy code
Paravirtualization (Xen)	port the kernel to an idealized x86 that calls the hypervisor directly	modified; applications are not	porting effort, and it needs the kernel's source
Hardware assist (VT-x, AMD-V)	a new CPU mode below ring 0 traps the sensitive operations	unmodified	arrived in 2005–06, two years after this paper

- Chisnall: paravirtualization is binary rewriting moved to compile time. The guest is fixed once, in its source.

What a paravirtualized guest must change

Everywhere the kernel assumes it owns the machine. This is the map for sections 03 to 06.

AREA	WHY IT BREAKS UNDER A HYPERVISOR	XEN'S REPLACEMENT	SECTION
Privilege level	the kernel can't keep ring 0 once Xen owns it	ring 1, or ring 3 with two page tables on x86-64	03
Privileged instructions	cli, hlt, lidt and CR3 writes fault or fail silently	event mask, sched_op, trap table, hypercalls	03, 05
Exceptions	handlers need CR2, readable only in ring 0	Xen takes the fault, saves CR2, bounces it	03
Page tables	the guest could map any machine frame	read directly; every write validated by Xen	04
Interrupts	device IRQs belong to whichever guest runs	event channels, queued per domain	05
Time	a guest gets only part of the CPU	real, virtual and wall-clock time from Xen	05
Devices	real hardware can't be shared directly	split front-end and back-end drivers	06
Boot	real mode and the BIOS belong to Xen	start in protected mode with a start info page	08

02

Xen’s design

Four design principles from the paper

01

Keep application binaries unmodified

Users won’t switch otherwise, so Xen virtualizes every architectural feature the standard ABIs rely on.

02

Run full operating systems

A guest is a real multi-application OS, so a complex server setup fits inside one virtual machine.

03

Paravirtualize x86

On an uncooperative architecture, it’s the only route to both high performance and strong resource isolation.

04

Don’t hide everything

Even on friendly hardware, hiding real time and real addresses hurts guests’ correctness and performance.

Mechanism in Xen, policy in the guests

Chisnall’s summary of the philosophy behind the design, and how it shaped later releases.

01

Separate the two

Xen provides grant tables, events and the XenStore. How devices are shared is left to the guests that use them.

02

Less is more

A bug in Xen compromises every VM, so each release tries to do less. Network multiplexing left Xen after 1.0.

03

Reuse the guest OS

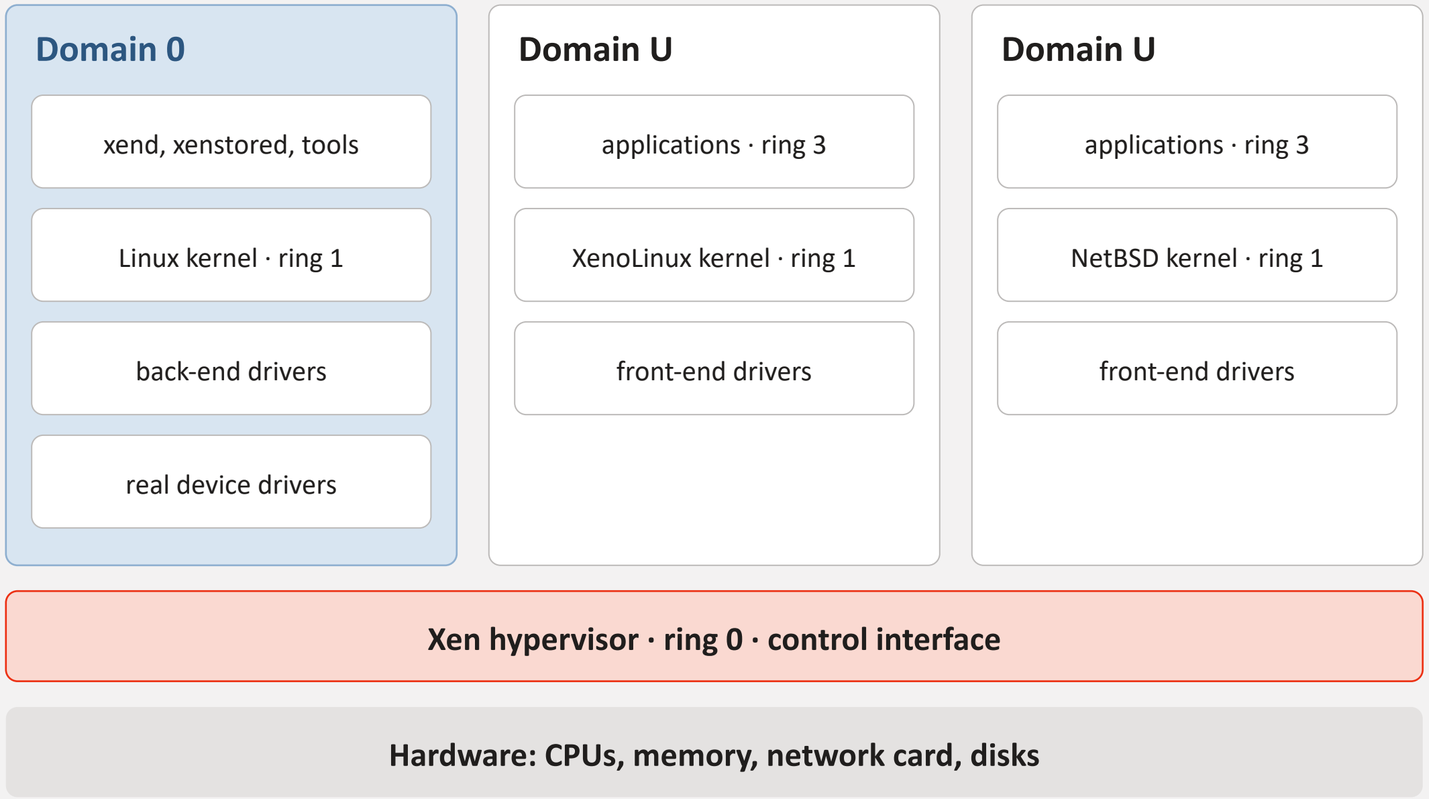
Xen has no device drivers. Domain 0’s drivers, bridging and firewall tools do that work.

A SMALL INTERFACE

Chisnall counts 37 hypercalls in Xen 3, against 64 system calls in Version 7 UNIX and about 450 in FreeBSD.

Who runs where

The hypervisor owns the CPU, memory and interrupts. Everything else, including the drivers, lives in a guest.



XEN

Schedules CPUs, partitions memory, validates page tables and routes interrupts as events. Nothing more.

DOMAIN 0

Booted with Xen. Runs the management tools, the XenStore daemon and the real drivers, and creates other domains.

DOMAIN U

Unprivileged. Reaches devices only through front-end drivers that talk to back ends in Domain 0.

Xen's UNIX vocabulary

Chisnall's analogy: Xen is to a guest kernel what a UNIX kernel is to a process.

XEN	UNIX COUNTERPART	WHAT IT DOES	SECTION
Hypercalls	system calls	the guest kernel asks Xen for a privileged operation	03
Events	signals	a one-bit asynchronous notification, delivered by upcall	05
Grant tables	POSIX shared memory	one domain lets another map, or receive, specific pages	06
XenStore	filesystem	a small hierarchical store for discovery and configuration	06
Domains	processes	an address space plus one or more virtual CPUs	02

- The analogy has limits: events only arrive on channels a guest has set up, so one guest can't send another the equivalent of SIGSEGV.

The paravirtualized x86 interface

The paper's Table 1: what a guest kernel sees instead of the bare machine.

AREA	WHAT THE GUEST SEES	SECTION
Segmentation	no fully privileged descriptors; nothing may overlap Xen's region	04 Memory
Paging	reads its page tables directly; Xen batches and validates updates	04 Memory
Protection	the kernel runs below Xen's privilege level, in ring 1	03 CPU
Exceptions	handlers registered with Xen; only the page fault handler changes	03 CPU
System calls	a "fast" handler lets applications enter the kernel without Xen	03 CPU
Interrupts	replaced by a lightweight event mechanism	05 Events
Time	a timer interface that reports both real and virtual time	05 Events
Network, disk	simple virtual devices over asynchronous I/O rings	06 I/O

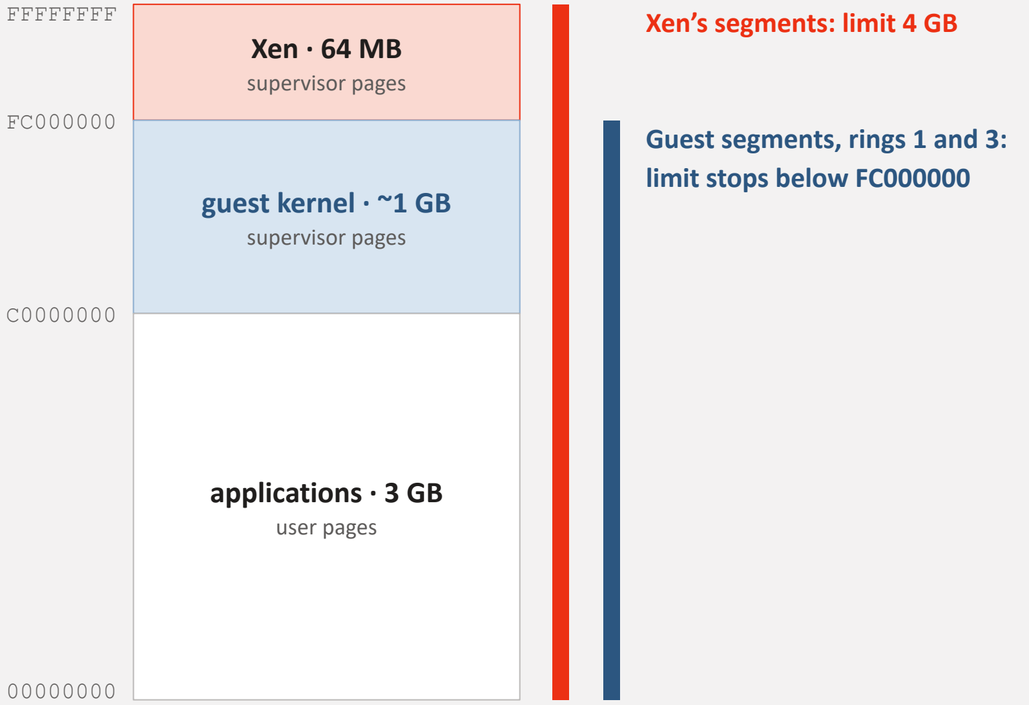
03

CPU: rings and hypercalls

x86-32: segments guard Xen, pages guard the kernel

Page protection has only two levels, user and supervisor. Segment limits add the third boundary that Xen needs.

ONE ADDRESS SPACE, NOT TO SCALE



CODE RUNNING IN	USER PAGES	KERNEL PAGES	XEN'S 64 MB
Ring 3 · apps	yes	no: U/S bit	no: U/S bit
Ring 1 · kernel	yes	yes	**no: segment limit**
Ring 0 · Xen	yes	yes	yes

WHY THIS IS CHEAP

Xen and the kernel are mapped in every address space, so a hypercall or a system call changes rings but never page tables, and the TLB survives.

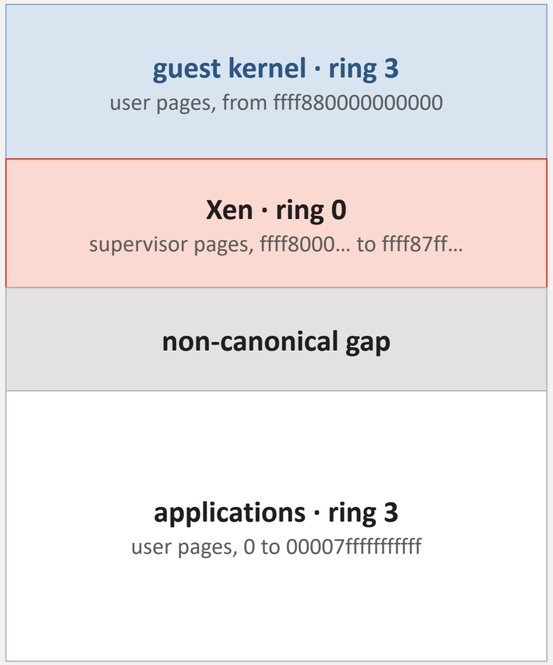
GUESTS CAN'T WIDEN THEIR SEGMENTS

Xen validates every descriptor a guest puts in its GDT or LDT: none may reach into the top 64 MB.

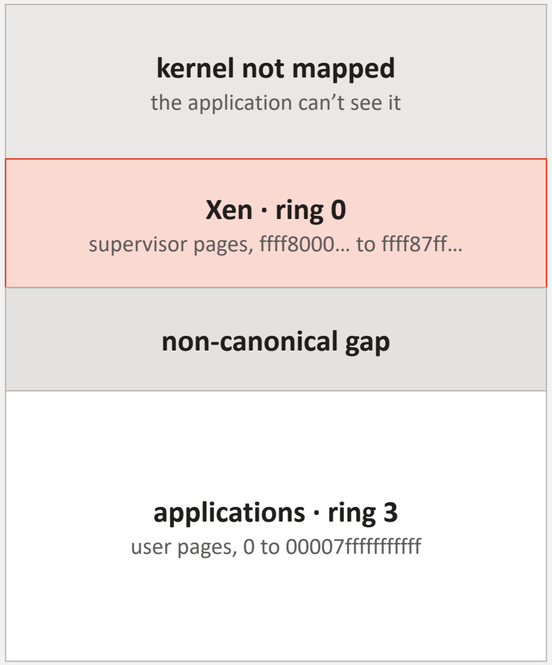
x86-64: no segment limits, so two page tables per process

64-bit mode ignores segment limits. The only boundary left is user versus supervisor pages, and Xen needs it for itself.

KERNEL-MODE PAGE TABLE



USER-MODE PAGE TABLE



ONE SYSTEM CALL

1. The application executes syscall. The CPU enters Xen in ring 0, not the guest kernel.
2. Xen switches CR3 to the process’s kernel-mode table and jumps to the guest’s handler, still in ring 3.
3. The kernel runs. Xen’s supervisor pages stay out of its reach; the kernel’s pages stay out of the app’s.
4. To return, the kernel makes an iret hypercall, and Xen switches back to the user-mode table.

THE COST

Two trips through Xen and two page-table switches per system call. Without tagged TLBs, each switch can flush the TLB.

Hypercalls: from int 0x82 to the hypercall page

A hypercall is a system call one level down. Xen 3 replaced the trap with a call into a page that Xen fills in.

```
; Xen 1.x and 2.x: a software interrupt
mov  eax, __HYPERVISOR_sched_op
mov  ebx, SCHEDOP_yield
int  0x82          ; system calls use 0x80

; Xen 3: call a 32-byte stub in a page
; that Xen maps into the guest at boot
call hypercall_page + __HYPERVISOR_sched_op*32
```

Simplified from Chisnall, Listing 2.1.

REGISTERS, NOT THE STACK

Arguments start in EBX, the result comes back in EAX, as in Linux system calls.

32 BYTES PER STUB

A 4 KB page holds 128 stubs. Xen 3 defines 37 hypercalls.

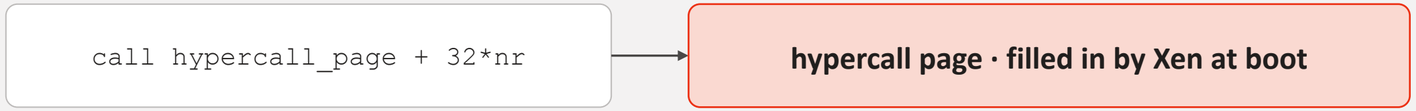
STILL COMPATIBLE

int 0x82 still works in Xen 3, but is deprecated in favour of the page.

- Why Xen, rather than the guest, writes the instructions in those stubs: next slide.

Why Xen writes the hypercall page itself

The best way to enter Xen depends on the CPU and on the kind of guest, and a compiled guest kernel can’t know which it will run on.



GUEST RUNNING AS	WHAT XEN WRITES IN EACH STUB (SIMPLIFIED)
32-bit PV	`mov $nr,%eax; int $0x82; ret`
64-bit PV	`mov $nr,%eax; syscall; ret`
HVM, Intel VT-x	`mov $nr,%eax; vmcall; ret`
HVM, AMD SVM	`mov $nr,%eax; vmmcall; ret`

- The 64-bit stub also saves the registers that syscall overwrites.

ONE BINARY, ANY CPU

VMCALL and VMMCALL are different opcodes. A kernel that hard-coded one would fault on the other vendor’s processors.

ROOM TO CHANGE

Xen can switch to a faster entry path later without recompiling or re-releasing any guest kernel.

A FAMILIAR TRICK

Linux’s vsyscall page did the same for system calls, choosing int 0x80 or SYSENTER at boot.

Why not let the guest load the real IDT?

A native kernel loads its own IDT. Xen keeps the real one for itself and gives guests a trap table instead.

CONCERN	IF A GUEST LOADED THE REAL IDT	WITH XEN'S TRAP TABLE
Privilege	every gate names a code segment; one pointing at ring 0 would hand the guest the machine	**Xen checks each entry once, when it is registered: no handler may run in ring 0**
Device interrupts	the disk or timer interrupt would go to whichever guest happens to be running	**Xen fields every hardware interrupt and queues an event for the domain it belongs to**
Faults Xen must see	page faults would skip Xen, so no one could rescue CR2 for a ring-1 handler	**Xen's handler runs first, saves what the guest needs, then bounces the fault**
Switching guests	Xen would load and recheck a raw descriptor table on every domain switch	**one real IDT per CPU; only the fast int 0x80 gate changes per guest**

- LIDT is privileged, so a ring-1 kernel can't load the table anyway. The trap table is the paravirtual replacement for that one instruction.
- Chisnall: the trap table is a simple, generic structure on purpose, not a copy of the hardware format, so it's easy to validate and to port.

The trap table: a virtual interrupt descriptor table

The guest registers its handlers with one hypercall, and Xen delivers exceptions to them.

```
struct trap_info {
  uint8_t  vector;    /* exception number      */
  uint8_t  flags;     /* rings allowed to INT  */
  uint16_t cs;        /* handler's segment     */
  unsigned long address;    /* handler         */
};
{ 14,   0, KERNEL_CS, page_fault  },
{ 0x80, 3, KERNEL_CS, system_call },
HYPERVISOR_set_trap_table(trap_table);
```

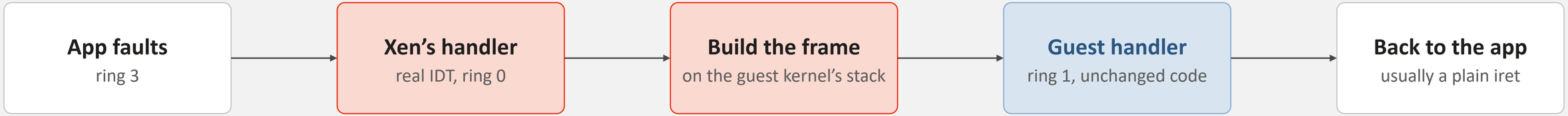
WHAT XEN CHECKS

Only that each handler’s code segment is not ring 0. The rest is checked lazily, if a fault ever happens.

THE FLAGS FIELD

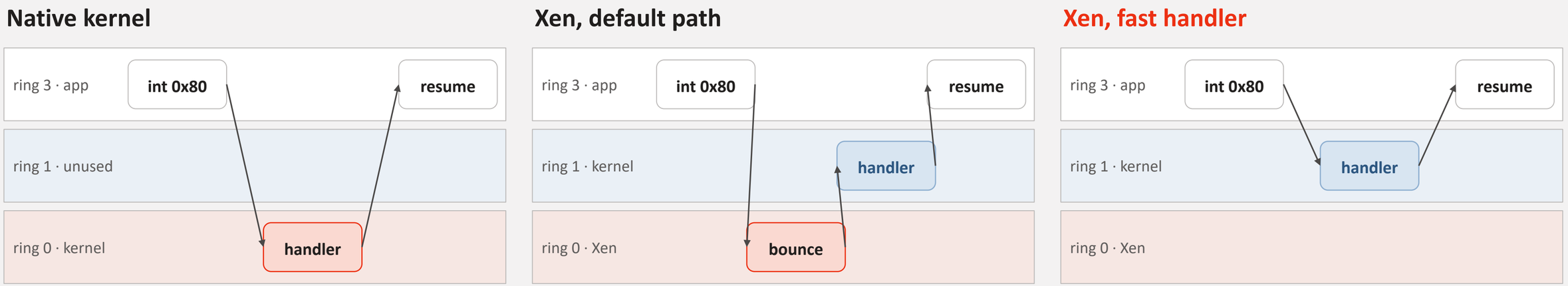
Which rings may raise the vector with INT: 3 lets applications use int 0x80; 0 means only the CPU can.

DELIVERING A FAULT FROM AN APPLICATION



Three paths for a system call

SYSENTER and SYSCALL always enter ring 0, which belongs to Xen, so 32-bit PV guests keep using int 0x80.



HOW THE FAST PATH STAYS SAFE

Xen copies the guest's int 0x80 gate into the real IDT. The gate leads to a ring-1 segment Xen has checked, so it can never reach ring 0.

WHAT IT SAVES

Two crossings of Xen per system call. lmbench's null system call: 0.46 µs on Xen against 0.45 µs native.

When Xen has to step in

The cases a PV kernel can’t handle alone, and how Xen avoids checking handlers in advance.

1	**Page faults** only ring 0 can read CR2, so Xen copies the faulting address for the guest, then bounces	**Always via Xen**
2	**Privileged instructions** cli, hlt or a CR3 write fault in ring 1; ports use the event mask and hypercalls instead	**Replaced by the port**
3	**Going idle** SCHEDOP_block sleeps the virtual CPU until an event, such as a timer set with set_timer_op	**Sleep until an event**
4	**A broken handler** a handler that isn’t mapped faults inside Xen’s bounce code; Xen recognizes its own address	**Guest is killed**

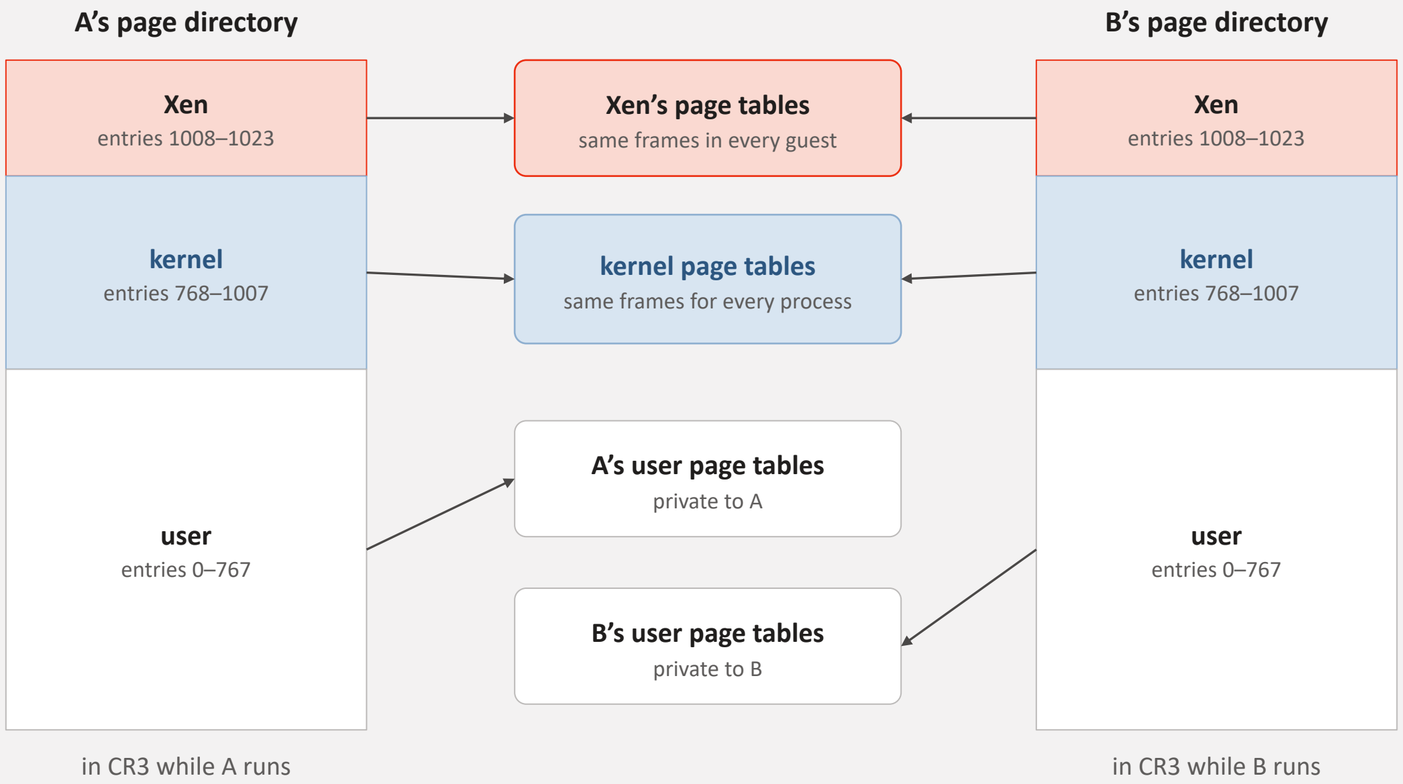
- Lazy checking is safe and cheap: a bad handler can only hurt its own domain, and Xen never validates guest memory up front.

04

Memory

What each page directory maps (x86-32)

Every process’s page directory points at the same kernel and Xen page tables. Only the user entries differ.



A SWITCH ONLY CHANGES THE USER PART

Going from A to B loads a new CR3, but Xen’s and the kernel’s entries are identical, and their page tables are shared frames.

WHAT XEN’S 64 MB HOLDS

Xen’s code and data, per-domain mappings, and a read-only machine-to-physical table every guest may read.

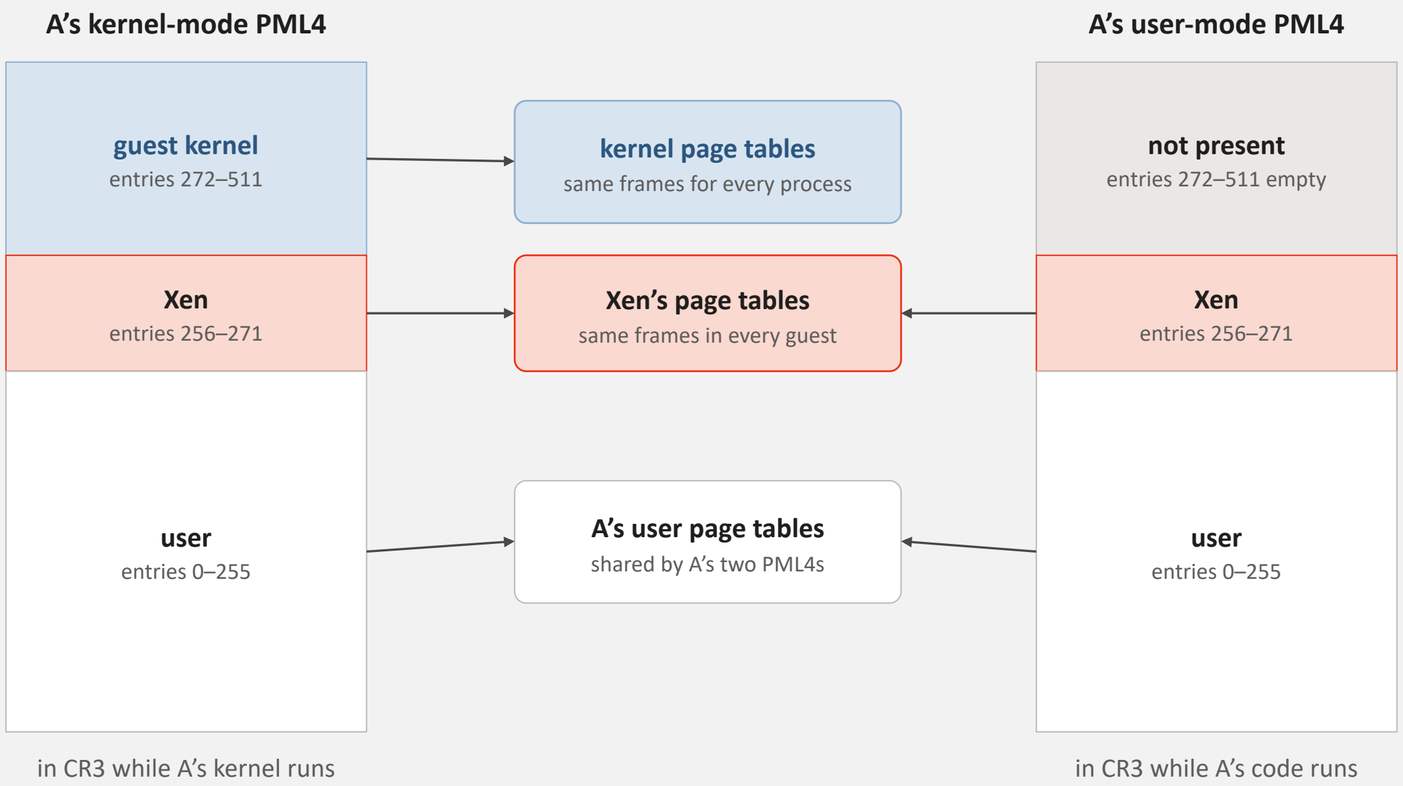
WHO WRITES THE XEN ENTRIES

Xen copies them in itself when it validates a new page directory. Guests can never change them.

- Simplified: non-PAE x86-32 with a 1 GB kernel. Each directory entry maps 4 MB, so Xen’s 64 MB is 16 entries.

What each top-level table maps (x86-64)

Each process has two top-level tables. Both carry Xen and the user page tables; only the kernel-mode one maps the kernel.



TWO TOP-LEVEL TABLES PER PROCESS

The kernel-mode PML4 maps everything; the user-mode one leaves the kernel out. Xen switches CR3 between them on every system call and return.

SHARED BELOW THE TOP

Both point at the same user page tables, and every process’s kernel-mode PML4 points at the same kernel tables.

XEN’S 16 ENTRIES

Copied into every PML4 Xen validates, and mapped supervisor-only, so the ring-3 kernel can’t touch them.

- Each PML4 entry maps 512 GB, so Xen’s 16 entries reserve 8 TB at ffff800000000000. That’s why 64-bit Linux’s direct map starts at ffff880000000000.

Three kinds of address

A PV guest sees a tidy pseudo-physical range for most of the kernel, and real machine frames where performance needs them.

ADDRESS	WHAT IT NUMBERS	WHO MAINTAINS THE MAPPING
Virtual	addresses used by processes and the kernel	guest page tables, validated by Xen
Pseudo-physical (PFN)	a contiguous 0 to N range the kernel treats as its RAM	the guest: an array from PFN to machine frame
Machine (MFN)	real host frames, possibly scattered across memory	Xen: a machine-to-physical table every guest can read

WHY NOT USE MACHINE FRAMES ONLY

Kernels assume contiguous RAM, and a guest resumed or migrated elsewhere gets different frames.

WHAT PAGE TABLES HOLD

Machine frames: x86 walks them in hardware, so the guest translates before every update.

Naming the frames: MFN, PFN and GFN

A GFN is whatever frame number the guest writes into its page tables. What that means depends on the kind of guest.

NAME	WHAT IT NUMBERS	IN A PV GUEST	IN AN HVM GUEST
MFN · machine frame number	a real frame of host memory	what its page tables hold	never seen by the guest
PFN · pseudo-physical frame number	the guest's contiguous 0 to N view of its own RAM	used by most of the kernel; the guest maps PFN to MFN itself	what the guest believes is physical memory
GFN · guest frame number	**the frame number the guest puts in its page tables**	**the same as the MFN**	**the same as the PFN; Xen's p2m maps it to an MFN**

WHO TRANSLATES TO MACHINE FRAMES

PV: the guest, before every page-table write, so the MMU can walk its tables directly. HVM: Xen, through its p2m table, via shadow or nested page tables.

IN THE SLIDES THAT FOLLOW

Shadow paging shows an HVM guest, so its page tables hold GFNs. Direct paging and fork() show a PV guest, so they hold MFNs.

Who builds the page tables the MMU walks?

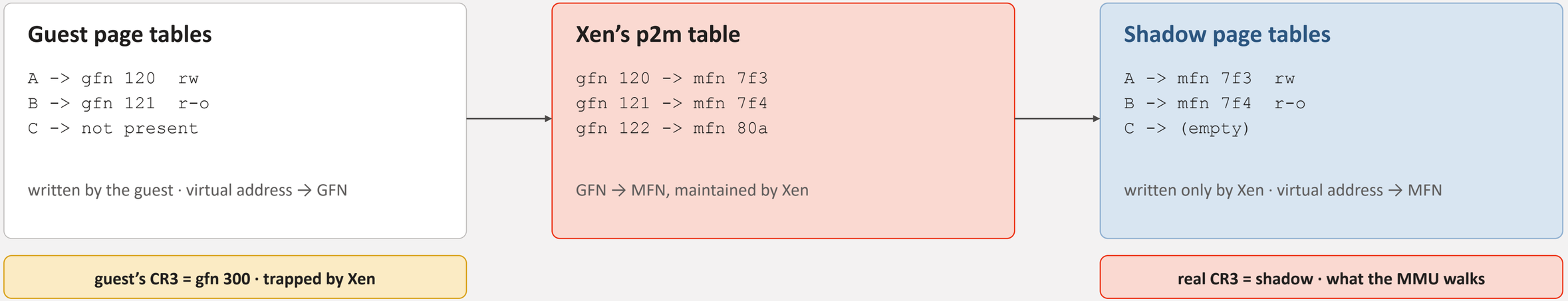
Xen 3 has four answers. First shadow paging, the approach the paper avoided; then the paper's direct paging, ending with fork() step by step.

TECHNIQUE	HOW IT WORKS	USED FOR	COST
Shadow paging	guest tables hold GFNs and are write-protected; Xen builds the real tables, with MFNs, from them	HVM guests; any guest during live migration; VMware in 2003	a trap on every page-table write
Direct paging	guest tables hold MFNs and the MMU walks them; every write is a validated hypercall	PV guests: the paper's design	porting effort
Writable page tables	a direct-paging variant: Xen unhooks a page, lets the guest write it, revalidates on next use	PV ports that reuse existing MMU code	a fault and a revalidation
Nested paging	hardware walks the guest's tables, then Xen's p2m	HVM guests on newer CPUs	longer TLB-miss walks

- Slides on shadow paging and on direct paging are marked as such in the red heading at the top. AMD's NPT shipped first; Intel's EPT followed in 2008.

Shadow paging: two tables, only one loaded

For HVM guests. The guest’s page tables hold guest frame numbers (GFNs), so the MMU can’t use them: Xen builds a shadow copy with machine frames (MFNs) and loads that.



WHEN XEN USES THEM

HVM guests on CPUs without nested paging, and any guest during live migration.

WHAT KEEPS THEM IN SYNC

The guest’s page-table frames are write-protected, so every write to them traps to Xen.

TWO KINDS OF FAULT

Hidden faults fill the shadow and the guest never knows. Real faults are passed on to the guest.

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	gfn 120	rw	A0	D0
B	gfn 121	r-o	A0	D0
C	not present			

XEN'S P2M

120->7f3 121->7f4 122->80a

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

mov to CR3 → exit to Xen

A	(empty)
B	(empty)
C	(empty)

STEP 1 OF 6

Guest loads CR3

The guest switches to a process. Its CR3 write exits to Xen, which finds or builds a shadow for guest table 300, write-protects that table, and loads the shadow into the real CR3.

STATE

MMU walks an empty shadow

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	**gfn 120**	**rw**	**A1 D0**	
B	gfn 121	r-o	A0 D0	
C	not present			

XEN'S P2M

120->7f3 121->7f4 122->80a

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

read A → shadow empty → fault to Xen

A	**mfn 7f3**	**r-o**	
B	(empty)		
C	(empty)		

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

STEP 2 OF 6

Hidden fault: fill on a read

The guest reads A. The shadow entry is empty, so the CPU faults into Xen. Xen walks the guest's table, translates gfn 120 to machine frame 7f3, sets the guest's accessed bit and fills the shadow. The guest never sees a fault.

STATE

shadow A = 7f3, read-only

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	**gfn 120**	**rw**	**A1**	**D1**
B	gfn 121	r-o	A0	D0
C	not present			

XEN'S P2M

120->7f3 121->7f4 122->80a

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

write A → shadow read-only → fault to Xen

A	**mfn 7f3**	**rw**
B	(empty)	
C	(empty)	

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

STEP 3 OF 6

Hidden fault: the first write

Xen left A read-only because the guest's dirty bit was clear. The first write faults, Xen sets the dirty bit in the guest's entry, and makes the shadow writable. Later writes run at full speed.

STATE

guest A dirty · shadow A rw

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	gfn 120	rw	A1	D1
B	gfn 121	r-o	A0	D0
C	**not present**			

XEN'S P2M

120->7f3 121->7f4 122->80a

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

touch C → guest table says not present

A	mfn 7f3	rw
B	(empty)	
C	(empty)	

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

STEP 4 OF 6

Real fault: pass it on

The guest touches C. This time Xen finds no mapping in the guest's own table either, so it injects a page fault into the guest, exactly as the hardware would have.

STATE

guest's fault handler runs

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	gfn 120	rw	A1	D1
B	gfn 121	r-o	A0	D0
C	**gfn 122**	**rw**	**A0**	**D0**

XEN'S P2M

120->7f3 121->7f4 122->80a

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

store to table 300 → write-protected → fault

A	mfn 7f3	rw	
B	(empty)		
C	**mfn 80a**	**r-o**	

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

STEP 5 OF 6

Guest writes its page table

The guest's handler maps C by writing its page table. That table is write-protected, so the store faults. Xen emulates the store, then updates the matching shadow entry so the two agree.

STATE

write emulated · shadow C synced

Shadow paging, step by step

GUEST'S PAGE TABLE · GFN 300

guest CR3 = gfn 300

write-protected in the shadow

A	gfn 120	rw	A1	D1
B	gfn 121	r-o	A0	D0
C	gfn 122	rw	A0	D0

XEN'S P2M

120->7f3 121->7f4 122->80a

changed in this step

valid shadow entry

Xen traps writes here

A0/A1, D0/D1: the guest's accessed and dirty bits

SHADOW · WHAT THE MMU WALKS

real CR3 = shadow 9e1

write A → read-only in shadow → Xen logs it

A	**mfn 7f3**	**r-o, now rw (logged)**	
B	**mfn 7f4**	**r-o**	
C	**mfn 80a**	**r-o**	

dirty bitmap: 120=1 121=0 122=0

STEP 6 OF 6

Log-dirty mode for migration

During live migration Xen makes every shadow entry read-only. A page's first write faults, sets its bit in the dirty bitmap and restores write access. Each copying round reads and clears the bitmap.

STATE

dirty: gfn 120

What shadow paging costs

Shadows let unmodified guests run, but they put Xen back in the path of every page-table change.

01

Page-table writes trap

Each guest PTE write costs a fault and an emulated instruction, where native costs one store.

02

Address-space switches

Every CR3 write exits to Xen. Xen caches recent shadows so switching back doesn’t rebuild them.

03

Memory and guessing

Shadows take memory, and Xen must guess when a page stops being a page table, or keep trapping its writes.

WHY NESTED PAGING HELPED

With AMD NPT or Intel EPT, the hardware walks the guest’s tables and then Xen’s p2m: no shadows and no write traps, but a TLB miss can take up to 24 memory references.

Direct paging: validating every update

The paper’s approach for PV guests: the MMU walks the guest’s own tables, but every write to them is a request Xen checks.

```
struct mmu_update {
    uint64_t ptr;   /* machine address of PTE */
    uint64_t val;   /* new contents of PTE    */
};

/* queue many writes, then one hypercall */
mmu_update_t req[2048];
HYPERVISOR_mmu_update(req, n, &done);
```

From xen/include/public/xen.h, via Chisnall Listing 5.6.

FRAME TYPES

Every frame is a page directory, page table, LDT, GDT or writable, with a reference count. Never two at once.

PINNING

A new page table is validated once and pinned, so switching to it costs no revalidation.

BATCHING

One hypercall can carry 2,048 updates: 8 MB of new address space.

- No guest can map a frame it doesn’t own, or map its own page tables writable. Sharing goes through grant tables.

The type system, rule by rule

Xen enforces one invariant: a frame is either a page table or writable by the guest, never both. Each update is checked against a few rules.

WHEN THE GUEST…	XEN REQUIRES	AND THEN
maps a frame writable	the guest owns the frame, and its type is writable or not yet set	writable count +1; the frame can’t become a page table until it’s 0
maps a frame read-only	the guest owns the frame, or holds a grant for it	any type is fine, so guests can read their own page tables
points a directory entry at a frame	the frame validates as a page table: every entry obeys these rules	page-table count +1; already-validated frames aren’t rechecked
pins a frame	it validates as the requested type	the type is held even when nothing references it
unpins and drops a page table	its page-table count reaches 0	it’s ordinary memory again, free to map writable

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  rw
stack -> 1c3  rw
```

CHILD

not allocated yet

not allocated yet

MMU UPDATES IN THIS STEP

nothing to send

TOTALS SO FAR

hypercalls	0
entries checked	0

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 1 OF 9

fork() enters the kernel

The parent calls fork(). Its int 0x80 reaches the guest kernel in ring 1 through the fast handler, so Xen isn't involved yet.

STATE

0 hypercalls

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  rw
stack -> 1c3  rw
```

CHILD

Child directory · 3c0 — writable

(zeroed page)

Child table · 3c1 — writable

(zeroed page)

MMU UPDATES IN THIS STEP

nothing to send

TOTALS SO FAR

hypercalls	0
entries checked	0

changed in this step

page-table type: read-only to guest

read-only, not yet a page table

ordinary writable frame

STEP 2 OF 9

Allocate the child’s tables

The kernel takes two free frames from its own reservation. To Xen they are ordinary memory, mapped writable in the kernel’s address space.

STATE

2 frames · type: writable

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1   user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  rw
stack -> 1c3  rw
```

CHILD

Child directory · 3c0 — writable

```
  0 -> PT 3c1   user
768 -> kernel PTs
1008 -> (empty)
```

Child table · 3c1 — writable

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

nothing to send

TOTALS SO FAR

hypercalls	0
entries checked	0

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 3 OF 9

Fill them with plain stores

The kernel copies its own entries into the new directory and the parent's entries into the new table, making writable pages read-only for copy-on-write. These frames aren't page tables yet, so every store is a normal write.

STATE

still 0 hypercalls

- Why this is safe: nothing points at 3c0 or 3c1 yet, so the MMU can't use them. Xen checks every entry before they can become page tables (step 7).

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  rw
stack -> 1c3  rw
```

CHILD

Child directory · 3c0 — writable

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> (empty)
```

Child table · 3c1 — writable

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
2a1[data]  = 1b7 r-o
2a1[stack] = 1c3 r-o
```

queued, not sent

TOTALS SO FAR

hypercalls	0
entries checked	0

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 4 OF 9

Queue the parent’s changes

Copy-on-write needs the parent’s writable pages made read-only too. The parent’s table is live and read-only to the guest, so these changes go into the batch.

STATE

2 updates queued

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

CHILD

Child directory · 3c0 — writable

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> (empty)
```

Child table · 3c1 — writable

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
2a1[data]  = 1b7 r-o
2a1[stack] = 1c3 r-o
```

sent · 1 hypercall

TOTALS SO FAR

hypercalls	**1**
entries checked	**2**

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 5 OF 9

Flush the batch

One mmu_update hands Xen the whole batch. Xen checks each entry: the frame belongs to this domain, and a read-only mapping is allowed for any type. Then the parent's stale TLB entries are flushed.

STATE

1 hypercall · 2 entries checked

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

CHILD

Child directory · 3c0 — read-only

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> (empty)
```

Child table · 3c1 — read-only

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
kernel map of 3c0 = r-o
kernel map of 3c1 = r-o
```

sent · 1 hypercall

TOTALS SO FAR

hypercalls	**2**
entries checked	**4**

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 6 OF 9

Give up write access

Xen won’t accept a frame as a page table while the guest can still write it. The kernel makes its own mappings of 3c0 and 3c1 read-only, in one batched call.

STATE

writable count of 3c0, 3c1 → 0

- Xen refuses to give a frame a page-table type while any writable mapping of it exists, so this must happen before the pin.

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

CHILD

Child directory · 3c0 — PD · pinned

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> Xen PTs
```

Child table · 3c1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
pin 3c0 as a directory
```

sent · 1 hypercall

TOTALS SO FAR

hypercalls	**3**
entries checked	**4 + child tree**

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 7 OF 9

Pin: Xen validates the tree

The kernel asks Xen to pin 3c0. Xen checks every entry, types 3c1 as a page table and 3c0 as a directory, and writes its own 16 entries at the top. None of this is rechecked until the child exits.

STATE

child's tree validated once

- All or nothing: one bad entry, such as a frame the child doesn't own or a page table mapped writable, and the pin fails.

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

CHILD

Child directory · 3c0 — PD · pinned

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Child table · 3c1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
new base pointer = 3c0
```

sent · 1 hypercall

TOTALS SO FAR

hypercalls	**4**
entries checked	**4 + child tree**

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 8 OF 9

Run the child

When the scheduler picks the child, the kernel asks Xen to load 3c0 into CR3. The directory is already pinned, so Xen checks nothing.

STATE

CR3 = 3c0

fork() in XenoLinux

PAGE-TABLE FRAMES

PARENT

Parent directory · 2a0 — PD · pinned

```
  0 -> PT 2a1    user
768 -> kernel PTs
1008 -> Xen PTs
```

Parent table · 2a1 — PT

```
code  -> 1a2  r-x
data  -> 1b7  r-o
stack -> 1c3  r-o
```

CHILD

Child directory · 3c0 — PD · pinned

```
  0 -> PT 3c1    user
768 -> kernel PTs
1008 -> Xen PTs
```

loaded in CR3

Child table · 3c1 — PT

```
code  -> 1a2  r-x
data  -> 4d2  rw
stack -> 1c3  r-o
```

MMU UPDATES IN THIS STEP

```
3c1[data] = 4d2 rw
```

sent · 1 hypercall

TOTALS SO FAR

hypercalls	**5**
entries checked	**5 + child tree**

- changed in this step
- page-table type: read-only to guest
- read-only, not yet a page table
- ordinary writable frame

STEP 9 OF 9

Copy-on-write fault

The child writes its data. The entry is read-only, so the CPU faults into Xen, which saves CR2 and bounces the fault. The kernel copies the page to 4d2 and maps it writable with update_va_mapping.

STATE

2 trips into Xen per fault

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

xm mem-set → XenStore memory/target → watch fires in the guest

XENSTORE memory/target

512 MB

GUEST'S SWAP DEVICE

0 MB used

THIS GUEST'S MEMORY HELD BY XEN

0 MB

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 1 OF 7

Domain 0 sets a target

The administrator asks for 512 MB. Domain 0 writes the new target into the guest's XenStore directory, and the balloon driver's watch on that key fires.

STATE

guest holds 1,024 MB · target 512 MB

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

alloc_page() succeeds from the free list

XENSTORE memory/target	GUEST'S SWAP DEVICE	THIS GUEST'S MEMORY HELD BY XEN
512 MB	**0 MB used**	**192 MB**

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 2 OF 7

Inflate: free memory first

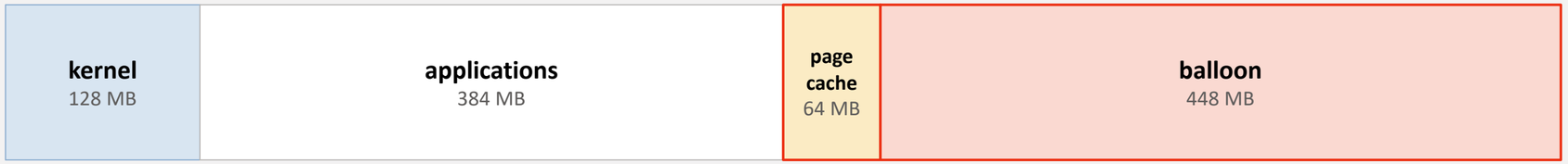
The driver allocates pages from the guest kernel like any other kernel code. Free pages go first. It unmaps each one, marks its p2m entry invalid, and returns the frame to Xen with decrease_reservation.

STATE

192 MB handed back to Xen

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

free list empty → the allocation enters the kernel's reclaim

XENSTORE memory/target

512 MB

GUEST'S SWAP DEVICE

0 MB used

THIS GUEST'S MEMORY HELD BY XEN

448 MB

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 3 OF 7

Reclaim: drop page cache

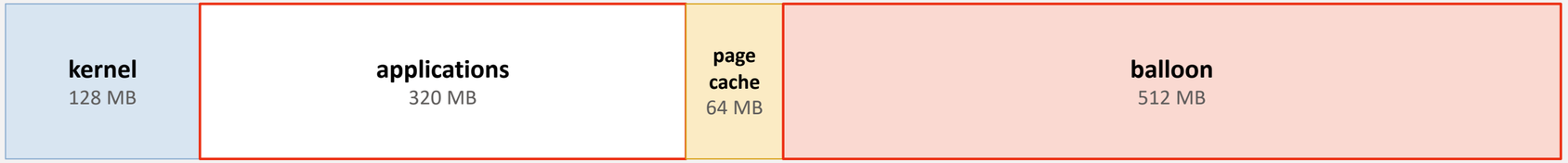
With no free pages left, each allocation makes the kernel reclaim memory, exactly as if an application had asked. Clean page cache is cheapest, so it goes first, and the freed pages land in the balloon.

STATE

256 MB of cache dropped

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

reclaim writes the least recently used anonymous pages to the guest's swap device

XENSTORE memory/target	GUEST'S SWAP DEVICE	THIS GUEST'S MEMORY HELD BY XEN
512 MB	**64 MB used**	**512 MB**

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 4 OF 7

Swap: cold pages go to disk

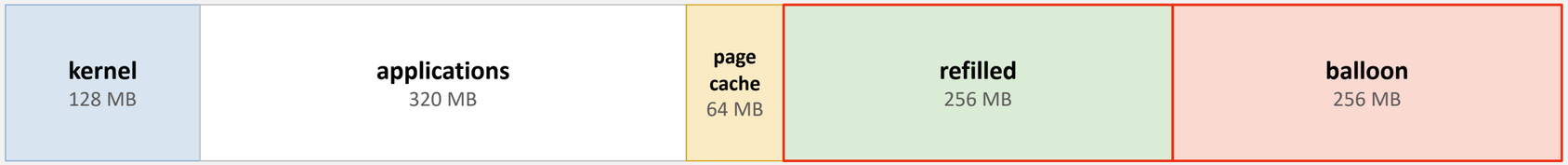
Still 64 MB short. The kernel's reclaim now picks the least recently used application pages and writes them to its own swap device. The balloon reaches the target.

STATE

target reached · 64 MB swapped by the guest

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

target 768 MB → increase_reservation → driver maps the new frames

XENSTORE memory/target

768 MB

GUEST'S SWAP DEVICE

64 MB used

THIS GUEST'S MEMORY HELD BY XEN

256 MB

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 5 OF 7

Deflate: frames come back

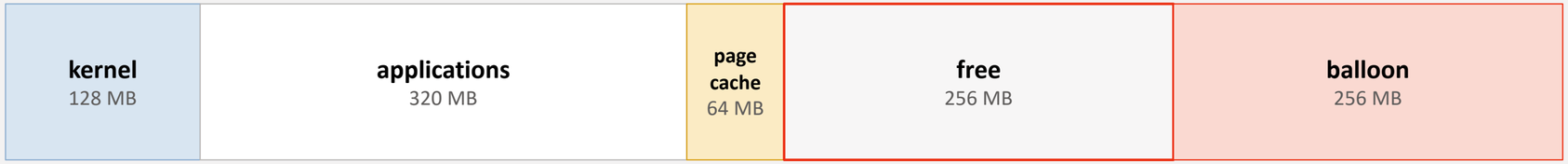
The target rises to 768 MB. The driver gets 256 MB of machine frames from Xen and puts them behind 256 MB of its empty PFNs. The driver still owns these pages.

STATE

256 MB of PFNs backed again

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

free_page() on each refilled page → the kernel's free list

XENSTORE memory/target

768 MB

GUEST'S SWAP DEVICE

64 MB used

THIS GUEST'S MEMORY HELD BY XEN

256 MB

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 6 OF 7

The driver frees them

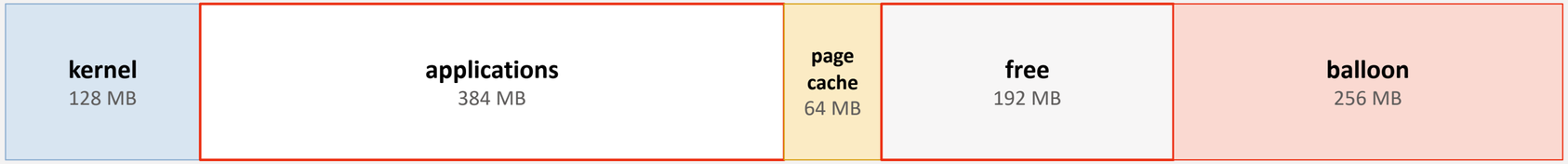
The driver releases the pages to the guest's page allocator. To the kernel they are simply 256 MB of new free memory.

STATE

kernel sees 256 MB free

Ballooning, step by step

GUEST'S PSEUDO-PHYSICAL MEMORY · 1,024 MB



Balloon: guest PFNs that currently have no machine frame behind them. Drawn as one block; really they are scattered.

an application touches a swapped page → page fault → read back from swap

XENSTORE memory/target	GUEST'S SWAP DEVICE	THIS GUEST'S MEMORY HELD BY XEN
768 MB	**0 MB used**	**256 MB**

kernel · applications · page cache · refilled · free · balloon

Red outline: changed in this step.

STEP 7 OF 7

Swapped pages return on use

Nothing brings the swapped-out pages back eagerly. When an application touches one, an ordinary page fault reads it from the swap device into a free page.

STATE

64 MB back in RAM, on demand

Why a balloon instead of swapping in Xen

Each guest grows or shrinks its own reservation, and decides for itself what to give up.

6.2 MB

smallest XenoLinux domain

RH 7.2 daemons, sshd and Apache, no swap

4.2 MB

with a swap device

pages handed back to Xen until needed

20 kB

Xen’s own state per domain

no shadow page tables to keep

XEN CAN’T CHOOSE WELL

To Xen every frame looks alike. It can’t tell free pages from page cache from hot data, so it would evict the wrong ones.

DOUBLE PAGING

If Xen had swapped a page out and the guest then chose to swap it too, Xen would have to read it back just so the guest could write it out.

ISOLATION

Paging in the hypervisor lets one guest’s thrashing slow every other guest. The paper rejects it for this reason.

05

Events, time and scheduling

Event channels replace interrupts

01

Physical IRQs

Real device interrupts, bound to channels in Domain 0 or a driver domain. Queued while that domain isn’t running.

02

Virtual IRQs

Events raised by Xen itself, such as each domain’s timer, which runs in that domain’s virtual time.

03

Interdomain

One domain opens an unbound port and advertises it; the other binds to it. Every split driver uses one.

04

Intradomain

Both ends in the same domain: the equivalent of an interprocessor interrupt between virtual CPUs.

Event state lives in the shared info page

One page per domain, readable and writable by both the guest and Xen. Checking for events is an ordinary memory read.

shared info page · one per domain, mapped by the guest

evtchn_pending

0	0	1	0	0	0	0	1	0	0	0	0	0	0

one bit per event channel: port 0, 1, 2, … (8 words in all)

evtchn_mask

0	0	0	0	0	0	0	1	0	0	0	0	0	0

a set bit means: don’t interrupt me for this port

vcpu_info[0]

```
upcall_pending   1
upcall_mask      0
pending_sel      0001
```

vcpu_info[1]

```
upcall_pending   0
upcall_mask      1
pending_sel      0000
```

Also in the page: time, the wall clock, and the guest’s p2m frame list (sections 04 and 05).

FIELD	XEN	GUEST
pending bits	sets 0 to 1	clears after handling
mask bits	never touches	sets and clears
upcall_mask	reads only	set = cli, clear = sti
upcall_pending, selector	sets	clears while scanning

NO HYPERCALL TO LOOK

Because the bits are plain memory in its address space, a guest can check for events whenever it likes.

Delivering an event

Xen can stop short of an upcall at three points. Each stop leaves a bit set that the guest can find later.



COALESCED, NOT COUNTED

Ten events before the guest looks are one set bit. The driver checks its ring to see how much work there is.

STOPPED IS NOT LOST

Every stop leaves a bit set in the shared page. Unlike a UNIX signal, nothing raised while masked disappears.

- Xen’s side of delivery, after Chisnall’s Figure 7.1. Amber boxes are decisions; red boxes are writes Xen makes to the shared page.

Disabling interrupts is just a store

A ring-1 kernel can’t execute cli or sti. It writes its VCPU’s upcall mask instead, and enters Xen only if an event is waiting.

```
/* "cli": a store, no hypercall */
vcpu->evtchn_upcall_mask = 1;

/* ... critical section ... */

/* "sti": also a store ... */
vcpu->evtchn_upcall_mask = 0;
barrier();
/* ... unless something arrived meanwhile */
if (vcpu->evtchn_upcall_pending)
    force_evtchn_callback();  /* any hypercall */
```

Simplified from the Linux PV port.

THE COMMON CASE IS FREE

If nothing arrived during the critical section, enabling costs one store and one load.

WHY IT MATTERS

Kernels disable and enable interrupts constantly. A trap into Xen each time would dominate.

FORCING DELIVERY

Any cheap hypercall will do: Xen delivers pending upcalls on its way back to the guest.

- The barrier matters: the check must come after the unmask, or an event arriving in between would wait for the next one.

Polling the pending bits

01

Mask a busy channel and poll it

Upcalls are costly to enter and leave. Under heavy load a driver masks its channel and checks the bit when convenient.

02

Recheck before returning

Clear the bit first, then handle the work, then look again: anything raised in the meantime has set it again.

03

Wait with SCHEDOP_poll

Spinning on a bit wastes a CPU another guest could use. Xen sleeps the VCPU until one of the listed ports is pending.

04

Clear bits atomically

Xen may set bits from another CPU at the same moment, so the guest clears them with atomic test-and-clear.

Three clocks

A guest gets only part of the CPU, so it must keep elapsed time and its own running time apart.

```
struct vcpu_time_info *t = &shared->vcpu_info[0].time;

uint64_t d   = rdtsc() - t->tsc_timestamp;
uint64_t sys = t->system_time
             + ((d << t->tsc_shift)
                * t->tsc_to_system_mul);
uint64_t now = shared->wc_sec * 1000000000ULL
             + shared->wc_nsec + sys;
```

Simplified from Chisnall 3.3; the real code uses a fixed-point multiply.

REAL (SYSTEM) TIME

Nanoseconds since boot, refreshed by Xen and extended between refreshes with the TSC.

VIRTUAL TIME

Advances only while the domain runs. The guest scheduler uses it to share its slices fairly.

WALL-CLOCK TIME

An offset added to real time, so the clock can be set without disturbing real time.

- Domain 0 runs NTP to correct TSC drift; other guests inherit the corrected time from Xen.

From BVT to the Credit scheduler

The paper used Borrowed Virtual Time. By Xen 3, Credit was the default.

SCHEDULER	IDEA	GOOD AT	WEAK AT
BVT (the paper)	weighted virtual time; a woken domain may “warp” back to run sooner	low-latency wake-up, such as TCP acks	no cap; I/O-heavy domains needed tuning
Atropos	each domain runs n ms in every m ms, soft real time	latency guarantees	no overcommit; bursts
SEDF	earliest deadline first over (slice, period) pairs	periodic workloads	being phased out
Credit (Xen 3)	a weight sets each share and a cap sets a limit; 30 ms slices	fair sharing across CPUs	soft real time

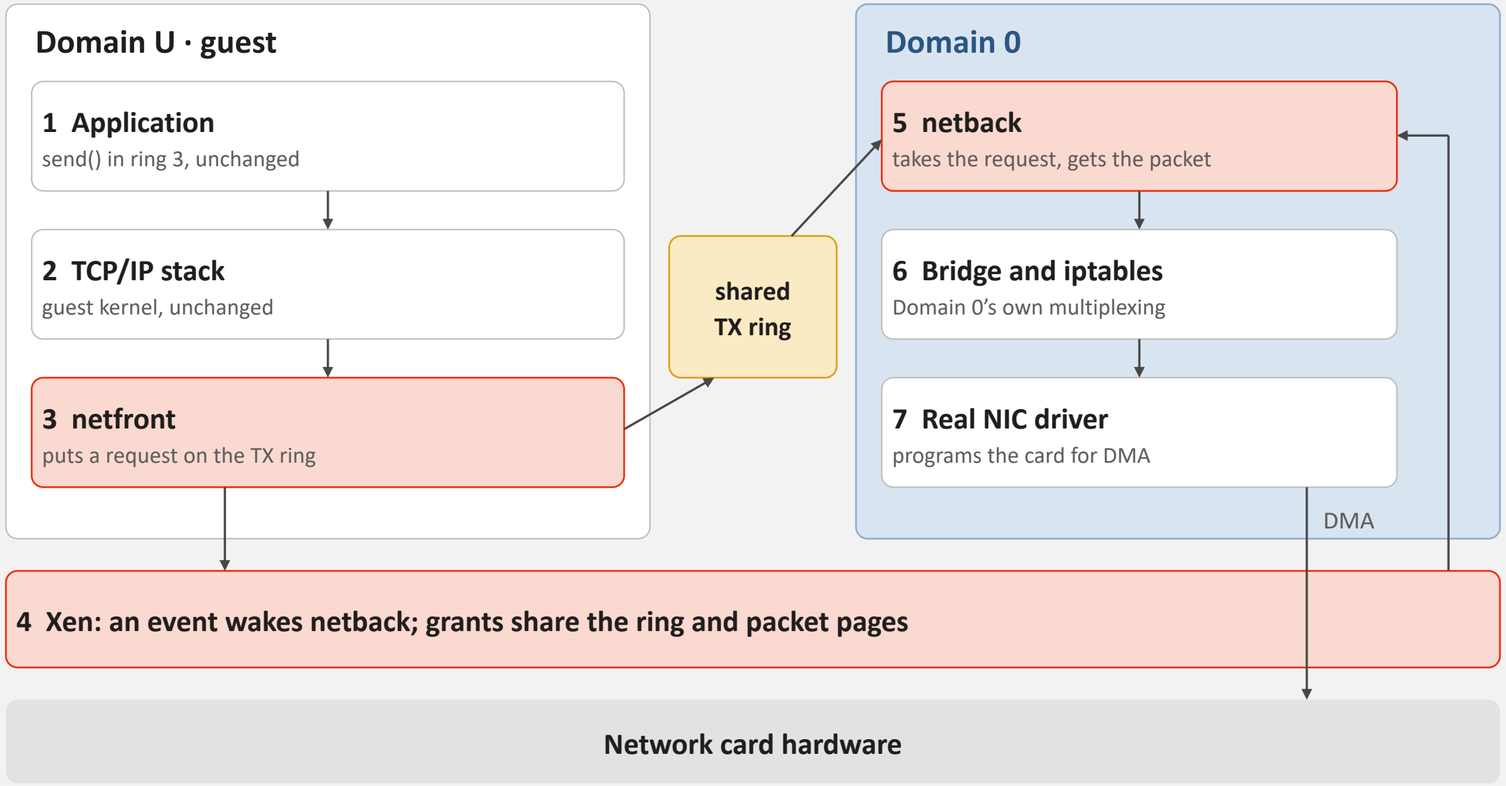
- In the paper’s test, 128 domains lost 7.5% of throughput with 5 ms slices. 50 ms slices, ESX’s default, closed the gap but hurt latency.

06

I/O: split drivers and rings

A packet from a guest to the wire

Each step runs in a guest. Xen, underneath, only provides the shared pages and the notification.



IN DOMAIN U · STEPS 1–3

Applications and TCP/IP are unchanged. Only the bottom driver knows it's running on Xen.

IN XEN · STEP 4

One event and a grant mapping. Xen never parses or routes the packet.

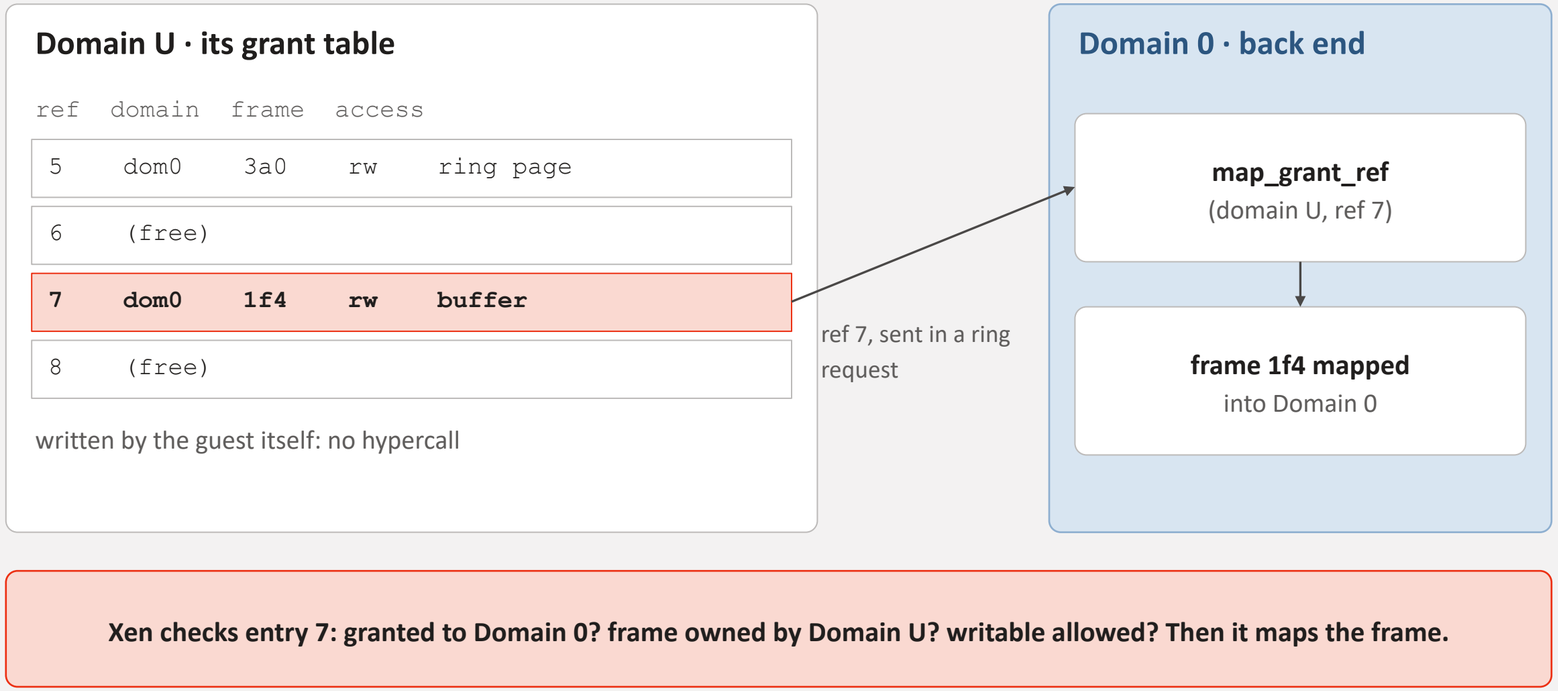
IN DOMAIN 0 · STEPS 5–7

Linux's own bridging, firewall and NIC driver do the multiplexing and the hardware work.

- In the 2003 paper, Xen itself filtered and routed packets in a "virtual firewall-router". Xen 2.0 moved that into Domain 0.

Grant tables: how a guest shares a page

A back end has to touch guest pages. The guest decides which pages, and for whom, in its grant table.



Xen checks entry 7: granted to Domain 0? frame owned by Domain U? writable allowed? Then it maps the frame.

A REFERENCE IS JUST AN INDEX

A small integer into the granter’s table: cheap to put in a ring request or in the XenStore.

THE GUEST CONSENTS

Only pages the guest granted, to the domain it named, can be mapped, and it can revoke them.

ANY DOMAIN CAN BE A BACK END

Grants don’t need Domain 0’s privilege, so an unprivileged driver domain can serve them too.

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	**dom0**	**1f4**	**rw**	
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · private to Domain U, empty

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

(no request yet)

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

ring <- domain U, ref 5

XEN

not involved in this step

STEP 1 OF 6

Grant the buffer page

The block front end wants Domain 0 to read disk data into its page 1f4. It fills grant entry 7: Domain 0 may map frame 1f4, writable.

changed in this step

a plain write into Domain U's own grant table · no hypercall

STATE

ref 7 granted, not in use

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	dom0	1f4	rw	
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · private to Domain U, empty

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

```
req id=1  read sector 2048  gref 7
```

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

```
ring      <- domain U, ref 5
```

XEN

not involved in this step

STEP 2 OF 6

Send the reference

The request on the ring names the buffer by grant reference, not by address. The front end advances req_prod and notifies Domain 0.

changed in this step

ref 7 travels inside the ring request; an event wakes Domain 0

STATE

Domain 0 knows: domain U, ref 7

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	**dom0**	**1f4**	**rw**	**in use**
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · also mapped in Domain 0

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

```
req id=1  read sector 2048  gref 7
```

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

```
ring      <- domain U, ref 5
buffer    <- domain U, ref 7
```

XEN

map_grant_ref(domain U, 7): granted to Domain 0? frame owned by U? write allowed? Yes: map it, mark the entry in use.

changed in this step

one hypercall: GNTTABOP_map_grant_ref

STEP 3 OF 6

Domain 0 maps it

Domain 0 asks Xen to map grant 7 of domain U. Xen checks the entry, maps frame 1f4 into Domain 0, and marks the entry as being written.

STATE

1 hypercall · entry marked in use

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	dom0	1f4	rw	in use
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · holds sector 2048

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

```
req id=1   read sector 2048   gref 7
```

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

```
ring      <- domain U, ref 5
buffer    <- domain U, ref 7
```

XEN

not involved in this step

changed in this step

Domain 0's real disk driver DMAs straight into frame 1f4

STEP 4 OF 6

The disk fills the page

The back end issues the read to Domain 0's disk driver with frame 1f4 as the target. The data lands directly in the guest's page: no copy.

STATE

data in U's page, zero copies

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	**dom0**	**1f4**	**rw**	
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · holds sector 2048

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

```
rsp id=1   OK
```

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

```
ring      <- domain U, ref 5
```

XEN

unmap_grant_ref: remove the mapping from Domain 0, clear the in-use flag in entry 7.

STEP 5 OF 6

Unmap and respond

When the read completes, Domain 0 unmaps the page; Xen clears the in-use flag. The back end puts the response on the ring and notifies the guest.

changed in this step

one hypercall: GNTTABOP_unmap_grant_ref; then a response on the ring

STATE

Domain 0 no longer maps 1f4

One grant, step by step

DOMAIN U'S GRANT TABLE

ref	domain	frame	access	
5	dom0	3a0	rw	in use
6	(free)			
7	**(free)**			
8	(free)			

ref 5 is the ring page, mapped by Domain 0 since the device connected

DOMAIN U'S BUFFER

frame 1f4 · private again, with the data

RING AND DOMAIN 0

RING REQUEST OR RESPONSE

```
rsp id=1   OK
```

DOMAIN 0'S MAPPINGS OF DOMAIN U'S PAGES

```
ring      <- domain U, ref 5
```

XEN

not involved in this step

STEP 6 OF 6

End the grant

The front end ends access by clearing entry 7, atomically and only if it isn't in use. Frame 1f4 is private again and ref 7 can be reused.

changed in this step

the guest clears entry 7, but only if it isn't in use

STATE

page private · ref 7 free

Three grant operations: map, transfer, copy

Mapping is what we just walked through. Xen offers two other ways to move data between domains.

OPERATION	WHAT HAPPENS	COST	USED FOR
Map	the granter names a frame and a domain in its grant table; that domain maps the page by reference	one mapping, then plain memory access	rings, block buffers
Transfer	ownership of a frame moves to the other domain, which must have offered to accept it	high but constant: page tables and TLBs change	the paper’s network receive; ballooning
Copy	Xen copies bytes between granted pages, since it already maps all of memory	grows with size, small fixed cost	network receive in Xen 3

- Flipping pages turned out to cost more in page-table work than it saved in copying, so Xen moved to copies (Chisnall, 2.5).

The I/O ring

One shared page holds a circular array of descriptors, plus the counters that say who has produced what.

```
struct sring {
  RING_IDX req_prod, req_event;  /* front end */
  RING_IDX rsp_prod, rsp_event;  /* back end  */
  union { request_t req;
          response_t rsp; } ring[8];
};
/* private: req_cons (back), rsp_cons (front) */
slot = counter & (8 - 1);
```

Simplified from xen/include/public/io/ring.h.

DESCRIPTORS, NOT DATA

Entries point to buffers granted separately, so bulk data never passes through the ring.

OUT OF ORDER

Each request carries an id that its response echoes, so the back end may reorder work.

NOTIFY WHEN ASKED

*_event tells the producer when the other side wants an event, so most writes send none.

- Requests and responses share one array: a response may only overwrite a request the back end has already taken.

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 1	req_cons 0
rsp_prod 0	rsp_cons 0

written this step
request waiting
taken by the back end
response waiting

SHARED RING PAGE

`req_prod & 7 = 0` → **slot #0**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	req id=1 read sector 2048
#1	free
#2	free
#3	free
#4	free
#5	free
#6	free
#7	free

STEP 1 OF 6

Write request 1

The guest's block front end fills slot 0 with a request: an id, a sector, and grant references for the pages to read into. Then it advances req_prod.

STATE

req_prod 0 → 1 · no event yet

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 2	req_cons 0
rsp_prod 0	rsp_cons 0

written this step
request waiting
taken by the back end
response waiting

SHARED RING PAGE

`req_prod & 7 = 1` → **slot #1**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	req id=1 read sector 2048
#1	req id=2 read sector 9000
#2	free
#3	free
#4	free
#5	free
#6	free
#7	free

STEP 2 OF 6

Write request 2, then notify

A second request goes in slot 1. Only now does the front end send one event on its channel: two requests, one notification.

STATE

req_prod 2 · one event to Dom0

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 2	**req_cons 2**
rsp_prod 0	rsp_cons 0

- written this step
- request waiting
- taken by the back end
- response waiting

SHARED RING PAGE

`req_prod - req_cons = 2` → **2 to take**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	req id=1 taken by back end
#1	req id=2 taken by back end
#2	free
#3	free
#4	free
#5	free
#6	free
#7	free

STEP 3 OF 6

Back end takes both

The event wakes blkback in Domain 0. It copies out every request between req_cons and req_prod and passes both to the real disk driver.

STATE

req_cons 0 → 2 · two reads in flight

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 2	req_cons 2
rsp_prod 1	rsp_cons 0

written this step
request waiting
taken by the back end
response waiting

SHARED RING PAGE

`rsp_prod & 7 = 0` → **slot #0**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	rsp id=2 OK
#1	req id=2 taken by back end
#2	free
#3	free
#4	free
#5	free
#6	free
#7	free

STEP 4 OF 6

Request 2 finishes first

The disk completes request 2 first. Responses fill slots in order, but each carries its request's id, so the front end can still match them.

STATE

rsp_prod 0 → 1 · out of order

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 2	req_cons 2
rsp_prod 2	rsp_cons 0

written this step
request waiting
taken by the back end
response waiting

SHARED RING PAGE

rsp_prod & 7 = 1 → **slot #1**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	rsp id=2 OK
#1	rsp id=1 OK
#2	free
#3	free
#4	free
#5	free
#6	free
#7	free

STEP 5 OF 6

Respond to request 1, notify

The second response lands in slot 1. The back end checks rsp_event, sees the front end wants waking, and sends one event for both.

STATE

rsp_prod 2 · one event to DomU

One block read through the ring

WHO ACTS

DomU · block front end
makes requests, consumes responses

Dom0 · block back end
owns the disk driver

COUNTERS

req_prod 2	req_cons 2
rsp_prod 2	**rsp_cons 2**

written this step
request waiting
taken by the back end
response waiting

SHARED RING PAGE

rsp_prod - rsp_cons = 2 → **2 to read**

granted by DomU, mapped by Dom0 · 8 slots

RING SLOTS · INDEX = COUNTER & 7

#0	free	
#1	free	
#2	free	(next request: 2 & 7)
#3	free	
#4	free	
#5	free	
#6	free	
#7	free	

STEP 6 OF 6

Front end collects responses

The front end reads both responses, completes each I/O by id, and advances rsp_cons. The counters keep counting; the next request uses slot 2.

STATE

rsp_cons 0 → 2 · ring empty

Why free-running counters work

Counters only ever increase. Masking gives the slot; subtracting gives how many entries are in use.

INDEXING A RING OF 8

```
slot = counter & (8 - 1)
used = req_prod - rsp_cons
full = (used == 8)
```

No wrap-around cases to get wrong: the size is a power of two, so a mask replaces the modulo.

WHEN AN 8-BIT COUNTER WRAPS

```
producer 258 -> 0000 0010
consumer 252    1111 1100
2 - 252 = -250  keep 8 bits:
                0000 0110 = 6
```

Subtraction modulo 256 still gives the right count, as long as the ring is smaller than the counter’s range.

- Chisnall’s example uses an 8-bit counter and a 32-slot ring. Real rings use 32-bit counters, so wrapping is rare but still safe.

The XenStore: discovery and configuration

A small hierarchical store, kept by a daemon in Domain 0, that guests use to find each other’s rings and ports.

```
# xenstore-ls /local/domain/1
console
  ring-ref = "5400"
  port = "2"
name = "Mini-OS"
domid = "1"
memory
  target = "32768"
store
  ring-ref = "5401"
  port = "1"
```

A Mini-OS domain, from Chisnall 8.2.

SMALL VALUES ONLY

Grant references, event ports, device state and targets. Bulk data goes through rings.

TRANSACTIONS AND WATCHES

Updates can be grouped atomically, and a guest can watch a key, as the balloon driver watches memory/target.

A SPLIT DEVICE ITSELF

A guest finds its XenStore ring in the start info page, then finds everything else in the store.

Connecting a front end to a back end

Device setup uses all three mechanisms: grant tables, event channels and the XenStore.

1 **Front end allocates a ring**
an empty shared ring page inside the guest — DomU

2 **Grant it and advertise it**
grant the page to Domain 0; write its ring-ref and an event port to the XenStore — XenStore write

3 **Back end connects**
Domain 0’s watch fires; it maps the grant and binds the event channel — Map and bind

4 **Traffic flows**
requests and responses on the ring, events only when asked for — No XenStore

- After migration the same handshake runs against the new host’s back end: rings are reconnected, not copied (Chisnall, 5.7.3).

Network: near-native at full-size packets

The paper’s ttcp bandwidth in Mb/s over Gigabit Ethernet (higher is better).

SYSTEM	SEND, MTU 1500	RECEIVE, MTU 1500	SEND, MTU 500	RECEIVE, MTU 500
Native Linux	897	897	602	544
Xen	897	897	516 (−14%)	467 (−14%)
VMware Workstation	291	615	101	137
User-mode Linux	165	203	61.1	91.4

PAGE FLIPPING (2003)

On receive, Xen swaps the page holding the packet for a free page from the guest: no copy, so the cost per byte is tiny.

THE SMALL-PACKET COST

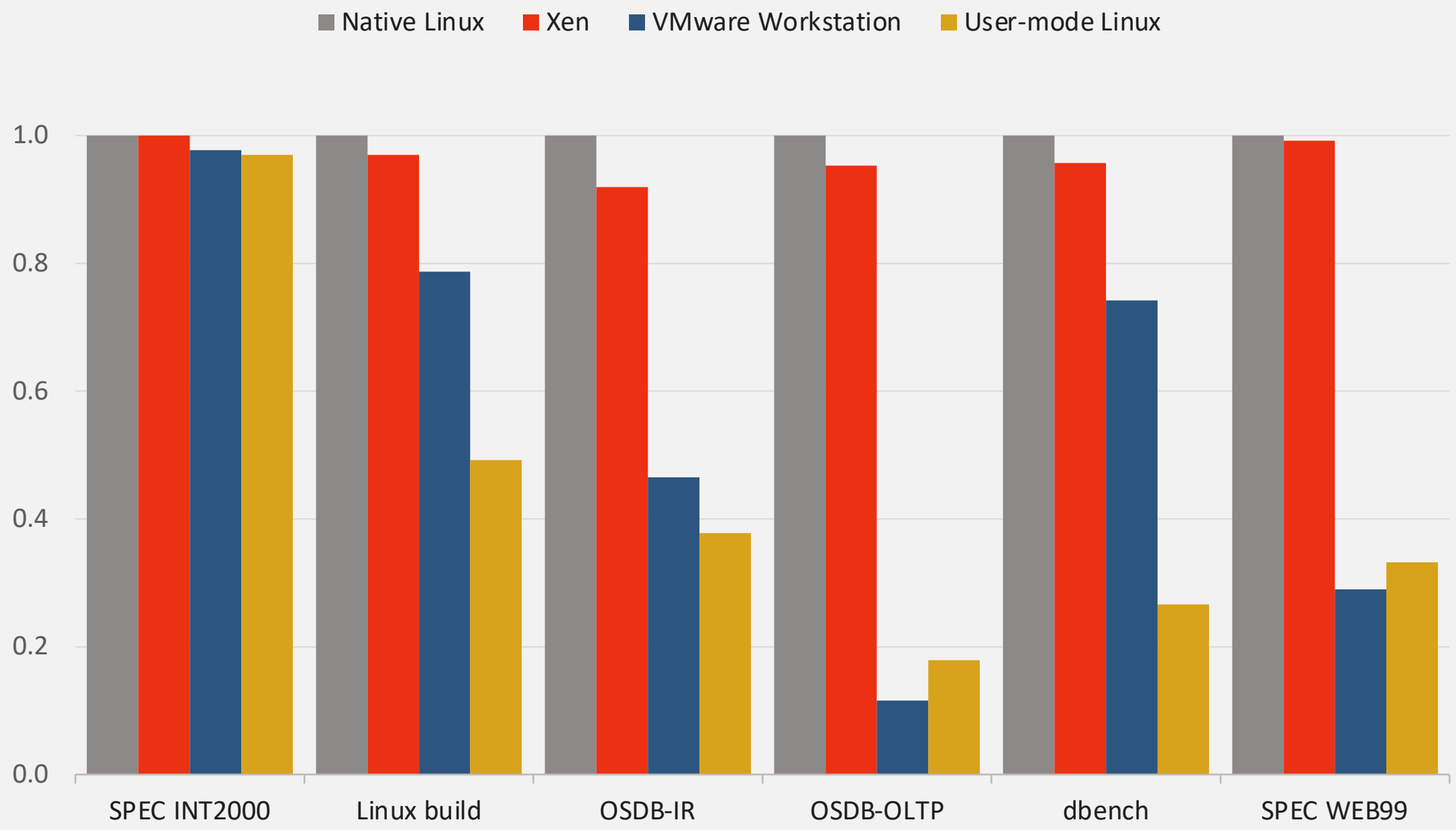
At MTU 500, per-packet costs dominate: filtering, demultiplexing and page swaps cost Xen 14%.

07

Evaluation

Close to native on whole applications

Score relative to native Linux (higher is better). The paper’s Figure 3, median of seven runs.



SETUP

Dual 2.4 GHz Xeon with one CPU enabled, 2 GB RAM, Linux 2.4.21 in every configuration.

XEN

At least 92% of native on every benchmark, and within 1% on SPEC WEB99.

THE OS-HEAVY CASES

OSDB-OLTP: Xen reaches 95% of native, VMware Workstation 12%.

Where the overhead lives: lmbench

Microseconds per operation (lower is better). Shaded: Xen's largest penalties, all from page-table work.

OPERATION	LINUX	XEN	VMWARE WS	UML	WHY XEN PAYS
Null system call	0.45	0.46	0.73	24.7	fast handler skips Xen
fork	110	**198**	874	21,000	every page-table write validated
exec	530	**768**	2,300	33,000	same, for a new image
Context switch, 2 procs	0.77	**1.97**	18.1	15.5	a hypercall to switch page tables
Page fault	1.42	**2.73**	12.4	26.3	two trips into Xen per fault
mmap latency	68.0	**139**	620	1,400	faults plus page-table updates

- Native column is the uniprocessor Linux kernel. Xen tracks it closely on 24 of the 37 lmbench tests.
- XenoLinux was 0.75 µs faster than native at signal handling, which the authors blame on lucky cache alignment.

Isolation and scale

Four domains with equal shares, two of them running deliberately antisocial workloads; then up to 128 domains.

2–4%

slowdown beside hostile domains

SPEC WEB99 and OSDB-IR next to a fork bomb, disk and memory hogs

7.5%

throughput lost at 128 domains

CPU-bound SPEC CINT2000 subset, 5 ms slices

5.4 ms

response time of an idle domain

UDP round trip with 128 busy domains running

NATIVE LINUX, SAME TEST

The disruptive processes made the machine unusable for the two benchmarks: almost all CPU time went to the OS.

THE AUTHORS' CAVEAT

They call the isolation result somewhat fortuitous, given Xen's simple disk scheduler at the time.

Reading the evaluation critically

The headline claim, that Xen beats the competition, rests on a narrower comparison than it first appears.

01

No ESX numbers

ESX’s licence barred publishing benchmarks, so the comparison is VMware Workstation, a hosted product.

02

One guest OS

Only XenoLinux was measured. The Windows XP and NetBSD ports were incomplete.

03

One CPU

Neither Xen nor VMware ran SMP guests, so most tests used a single processor.

INDEPENDENT REPLICATION

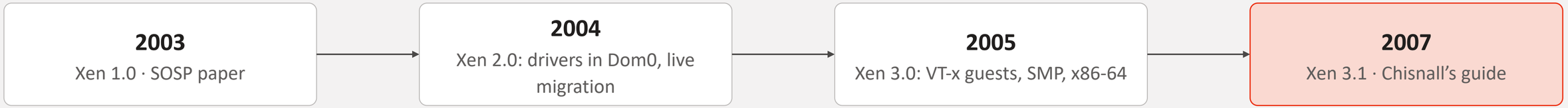
A team at Clarkson University reran the experiments on different hardware a year later: “Xen and the Art of Repeated Research” (USENIX 2004).

08

After the paper

Four years, three major releases

The mechanisms stayed; what changed was where work happens and which guests can run.



WHAT STAYED

Hypercalls, event channels, page-table validation and I/O rings. Later features are built from the 2003 mechanisms.

WHAT MOVED

Device drivers and network multiplexing left the hypervisor for Domain 0, and then began moving out of Domain 0.

- AMD’s SVM support followed in Xen 3.0.2. Chisnall’s book was first printed in November 2007.

The paper versus Xen 3

Side by side: the SOSP prototype and the system Chisnall documents.

AREA	SOSP 2003	XEN 3 (CHISNALL, 2007)
Hypercalls	int 0x82	a call into the hypercall page
Scheduler	BVT with 5 ms slices	Credit: weights, caps, 30 ms slices
Network	page flipping; filtering inside Xen	grant copies; bridging in Domain 0
Discovery	control interface used by Domain 0	the XenStore, with watches
Unmodified guests	Windows XP port in progress	HVM guests on VT-x and AMD-V
64-bit	x86-32 only	x86-64, with the kernel in ring 3

HVM: running unmodified guests

With VT-x and AMD-V, Xen can run guests that don’t know they are virtualized. Almost everything at the boundary changes.

ASPECT	PARAVIRTUALIZED GUEST	HVM GUEST
Boot	protected mode, with a start info page from the builder	real mode, with an emulated BIOS
Kernel privilege	ring 1 on x86-32, ring 3 on x86-64	ring 0, in a guest mode below Xen
Devices	split front-end drivers	emulated devices from QEMU
Page tables	guest-maintained, Xen-validated	shadow or nested page tables
System calls	through Xen, unless the fast handler is set	the native SYSENTER fast path
Finding Xen	the ELF header names the hypercall page	CPUID leaf 0x40000000: “XenVMMXenVMM”

Paravirtualization didn’t go away

Hardware support made unmodified guests possible. Chisnall argues the best results come from combining the two.

01

PV drivers in HVM

An unmodified guest can load front-end drivers. Linux uses the same driver source for PV and HVM guests.

02

HVM features in PV

Fast system calls and hardware nested paging help a paravirtualized kernel too.

03

Port in stages

Start unmodified, then paravirtualize drivers, then timekeeping, then memory, in that order of payoff.

WHY EMULATED I/O HURTS

On full-virtualization systems an I/O-heavy guest can keep the host busy over 90% of the time just emulating devices (Chisnall, 13.7).

Shrinking Domain 0

Domain 0 kept Xen small, but it became the largest body of trusted code on the machine.

01

Driver domains

Run a device’s driver in its own guest. Without an IOMMU, rogue DMA makes this only semi-safe.

02

Stub domains

Move each HVM guest’s device emulator out of Domain 0, so its CPU time is charged to that guest.

03

Finer privilege

Replace “is the caller Domain 0?” checks with a security policy for each hypercall.

THE DOMAIN 0 QUESTION

Domain 0 can map any guest’s memory, so a root exploit there owns the machine. Split up, Xen starts to look like a multiserver microkernel.

09

Takeaways

Four things to remember

01

Change the machine, not the apps

About 3,000 changed lines in Linux bought near-native speed on hardware that couldn’t be virtualized cleanly.

02

Mechanism in Xen, policy outside

Hypercalls, events, grant tables and the XenStore are the whole toolkit. Guests build drivers from them.

03

Batch and share, don’t trap

Batched page-table updates, I/O rings and the shared info page keep crossings into Xen rare.

04

The interface outlived its hardware

HVM, x86-64 and PV drivers in HVM guests all reuse the 2003 mechanisms.

Check your understanding (1 of 2)

01 Why did Xen choose paravirtualization over full virtualization on x86, and what did it give up?

---

02 How does Xen use the x86 rings to protect itself, and why does it reserve the top 64 MB of every address space?

---

03 Follow-up: with only two usable privilege levels, as on x86-64, where must the guest kernel run, how does it stay safe from its applications, and what does that cost?

Check your understanding (2 of 2)

04 How does Xen handle page-table updates differently from shadow page tables, and why are fork and exec still slower than native?

---

05 Describe the I/O ring. Why are request ids and decoupled notifications useful?

---

06 What limits how far you can trust the paper’s claim that Xen outperforms its competitors?

For discussion

01 Each Xen release tried to do less, pushing work into Domain 0. Did that shrink the trusted code, or just move it?

---

02 Hardware assist arrived two years after this paper. Why did paravirtual drivers stay valuable even for unmodified guests?

---

03 The authors couldn’t publish ESX numbers. How should a field treat comparisons it isn’t allowed to see?

---

04 Split into driver and stub domains, Xen looks like a microkernel. Is a hypervisor just a microkernel with a better story?