Lecture 13 - Virtualization in the Field

How Linux, macOS and Windows build a hypervisor into the operating system they ship: KVM, Apple’s Hypervisor and Virtualization frameworks, and Hyper-V. Sources: Kivity et al., kvm: the Linux Virtual Machine Monitor (OLS 2007); Agache et al., Firecracker (NSDI 2020); Apple Developer and Microsoft Learn documentation.

Contents

01 From Xen to the host OS

02 Linux: KVM

03 Linux: virtio and pKVM

04 macOS: two frameworks

05 Windows: Hyper-V

06 Windows: virtualization-based security

07 Side by side

08 Takeaways

01

From Xen to the host OS

What changed after 2003

Lecture 11’s Xen rewrote the guest kernel because x86 couldn’t trap every sensitive instruction. Two hardware changes removed most of that need.

01

CPU virtualization

Intel VT-x (2005) and AMD-V (2006) add a guest mode below ring 0. Sensitive instructions exit to the hypervisor, so an unmodified kernel runs.

02

Nested paging

AMD NPT (2007) and Intel EPT (2008) walk the guest’s page tables, then a second table the hypervisor owns. Shadow page tables become optional.

03

I/O stays paravirtual

Emulating a real device costs an exit for every register access. Every system in this lecture still moves disk and network data through shared rings.

WHAT THIS MADE POSSIBLE

A hypervisor small enough to ship inside a general-purpose OS. Linux merged KVM in 2007, Windows Server 2008 shipped Hyper-V, and OS X 10.10 added Hypervisor.framework in 2014.

Three places to put the hypervisor

The systems in this lecture differ mainly in where the hypervisor runs and where devices are emulated.

SYSTEM	HYPERVISOR RUNS IN	HOST OS RUNS IN	DEVICES EMULATED BY
Xen (lecture 11)	its own small kernel, below every OS	Domain 0, a privileged guest	back-end drivers in Domain 0
Linux KVM	the host Linux kernel, as a module	the host itself; no extra layer	a user process such as QEMU
Windows Hyper-V	its own kernel, below Windows	the root partition, a privileged guest	drivers and a worker process in the root
macOS	the XNU kernel	the host itself; no extra layer	the app’s own process, or a system service

- Microsoft Learn, Hyper-V architecture; Apple Developer, Hypervisor; Linux Documentation/virt/kvm/api.rst.

One loop, three APIs

In all three operating systems a virtual CPU is a host thread that calls “run” and gets back an exit to handle.

```
/* Linux: KVM ioctl API */
for (;;) {
  ioctl(vcpu_fd, KVM_RUN, 0);
  switch (run->exit_reason) { /* ... */ }
}
/* macOS: Hypervisor.framework API */
for (;;) {
  hv_vcpu_run(vcpu);
  switch (exit->reason) { /* ... */ }
}
/* Windows: Windows Hypervisor Platform API */
for (;;) {
  WHvRunVirtualProcessor(part, 0, &ctx, sizeof ctx);
  switch (ctx.ExitReason) { /* ... */ }
}
```

WHERE THIS CODE COMES FROM

Written for this lecture, not copied from vendor code. Each loop calls a public, documented API; Apple’s and Microsoft’s hypervisors themselves are closed source.

ONE VM, ONE PROCESS

Guest RAM is ordinary memory of the VMM process, mapped into guest-physical space: `KVM_SET_USER_MEMORY_REGION`, `hv_vm_map`, `WHvMapGpaRange`.

ONE VCPU, ONE THREAD

The run call blocks while the guest executes. The guest runs only while its thread is inside that call.

EXITS ARE THE INTERFACE

Accesses the kernel doesn’t handle itself come back as an exit record: a port, an address, a size, the data.

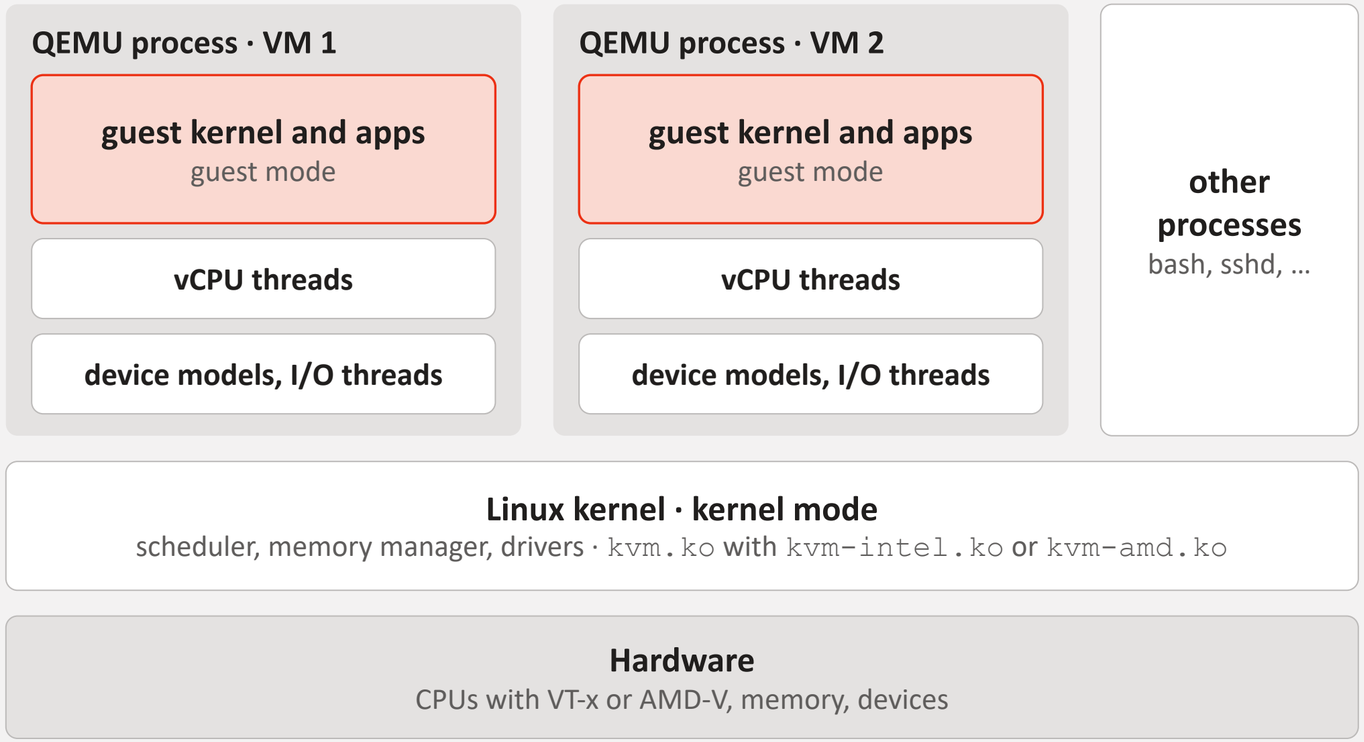
- APIs: Linux Documentation/virt/kvm/api.rst (KVM is open source); Apple Developer, Hypervisor (macOS SDK); Microsoft Learn, Windows Hypervisor Platform (WinHvPlatform.h, Windows SDK). Open code using all three: Microsoft’s OpenVMM, github.com/microsoft/openvmm.

02

Linux: KVM

KVM: the hypervisor is a kernel module

Kivity et al. (2007): add a guest mode to Linux and reuse everything else the kernel already does.



A THIRD MODE

Beside user mode and kernel mode, a thread can run in guest mode. `KVM_RUN` enters it; an exit leaves it.

REUSE, NOT REWRITE

Scheduling, NUMA placement, swapping, cgroups and page merging (KSM) apply to VMs because VMs are processes.

MERGED FAST

First posted in October 2006, KVM was in Linux 2.6.20, released in February 2007.

- A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, kvm: the Linux Virtual Machine Monitor, Ottawa Linux Symposium 2007.

The /dev/kvm API

```
int kvm = open("/dev/kvm", O_RDWR);
int vm  = ioctl(kvm, KVM_CREATE_VM, 0);

/* guest RAM: 1 MB of this process's memory */
void *ram = mmap(NULL, 1 << 20, PROT_READ | PROT_WRITE,
                 MAP_SHARED | MAP_ANONYMOUS, -1, 0);
struct kvm_userspace_memory_region slot = {
  .slot = 0, .guest_phys_addr = 0x1000,
  .memory_size = 1 << 20, .userspace_addr = (__u64)ram };
ioctl(vm, KVM_SET_USER_MEMORY_REGION, &slot);

int vcpu = ioctl(vm, KVM_CREATE_VCPU, 0);
int size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, 0);
struct kvm_run *run = mmap(NULL, size,
    PROT_READ | PROT_WRITE, MAP_SHARED, vcpu, 0);

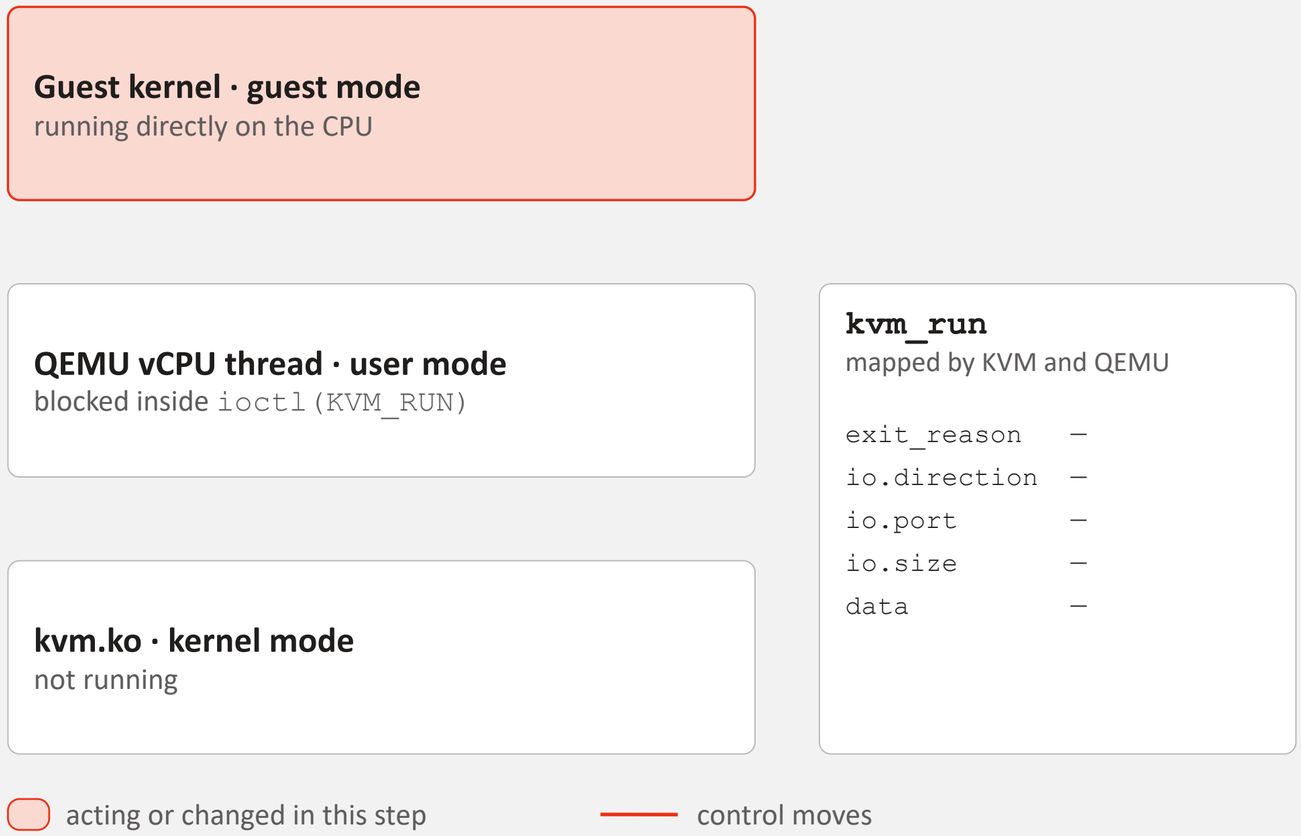
for (;;) {
  ioctl(vcpu, KVM_RUN, 0);
  if (run->exit_reason == KVM_EXIT_IO)  { /* emulate */ }
  if (run->exit_reason == KVM_EXIT_HLT) break;
}
```

A complete VM needs a handful of `ioctl` calls and one loop.

1. Each object is a file descriptor: the KVM system, the VM, each vCPU. Closing them tears the VM down.
2. Guest RAM is a range of the process's own memory, registered as a memory slot at a guest-physical address.
3. The `kvm_run` page is mapped by both KVM and the process, so exit details arrive without a copy.
4. The loop is the whole VMM core. Everything else is device emulation.

- Written for this lecture from the API in Linux Documentation/virt/kvm/api.rst; register setup and error handling omitted. LWN, "Using the KVM API" (2015), has a complete version.

One port write, step by step



STEP 1 OF 6

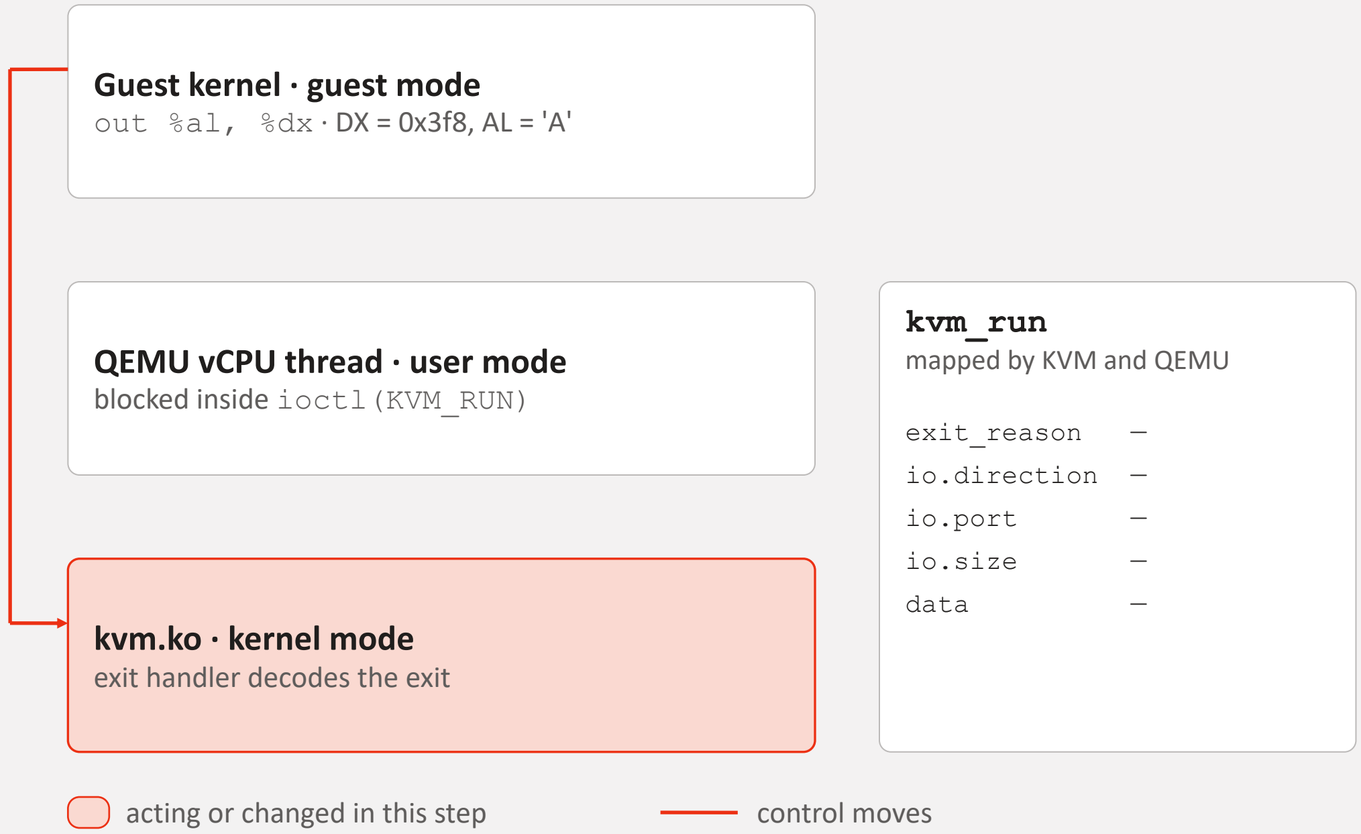
The guest is running

QEMU's vCPU thread called `ioctl(KVM_RUN)`. KVM loaded the guest's state and entered guest mode. The thread is now blocked in the kernel while the CPU runs guest code directly.

STATE

CPU in guest mode · no exit yet

One port write, step by step



STEP 2 OF 6

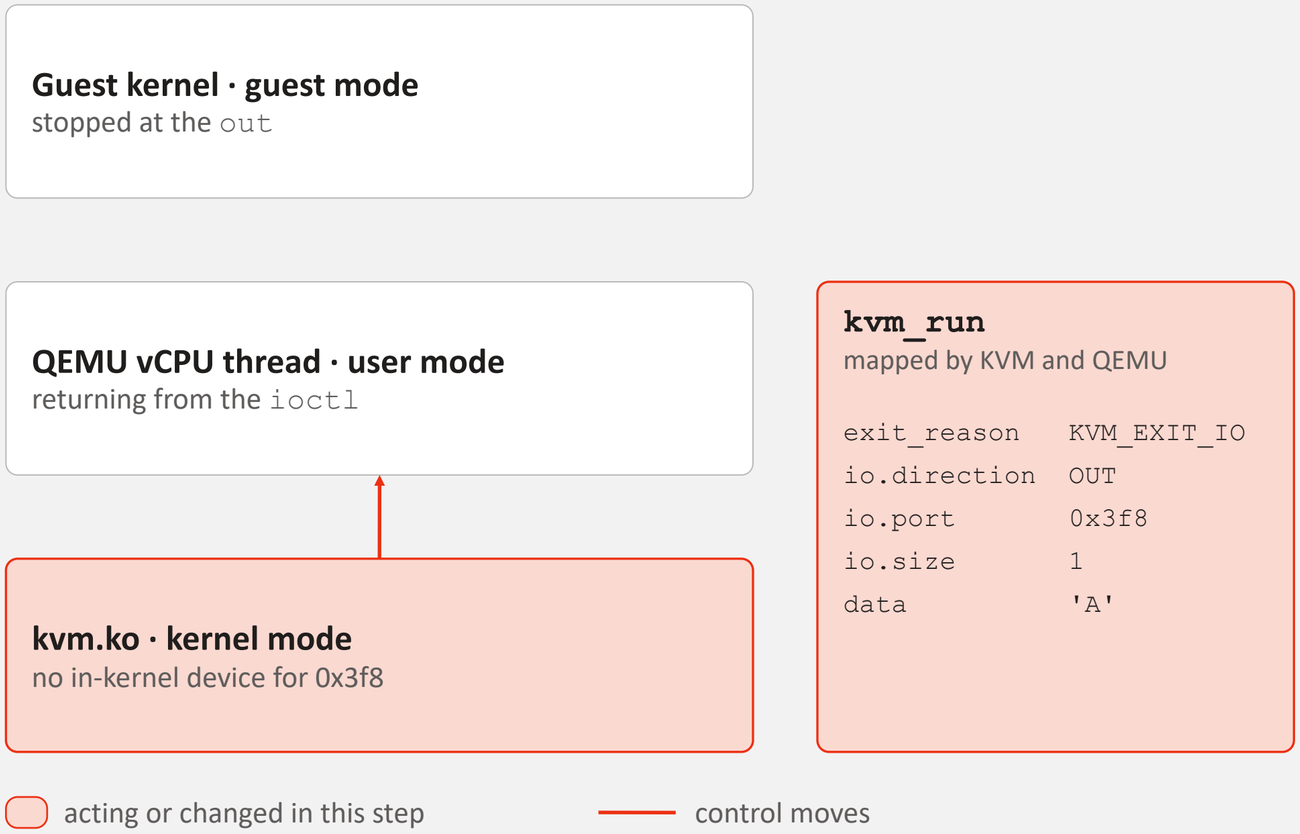
A port write exits

The guest's serial driver executes `out %al, %dx` with DX = 0x3f8. The VMCS says I/O to this port must exit, so the CPU saves the guest's state and jumps to KVM's exit handler.

STATE

VM exit · basic reason 30, I/O instruction

One port write, step by step



STEP 3 OF 6

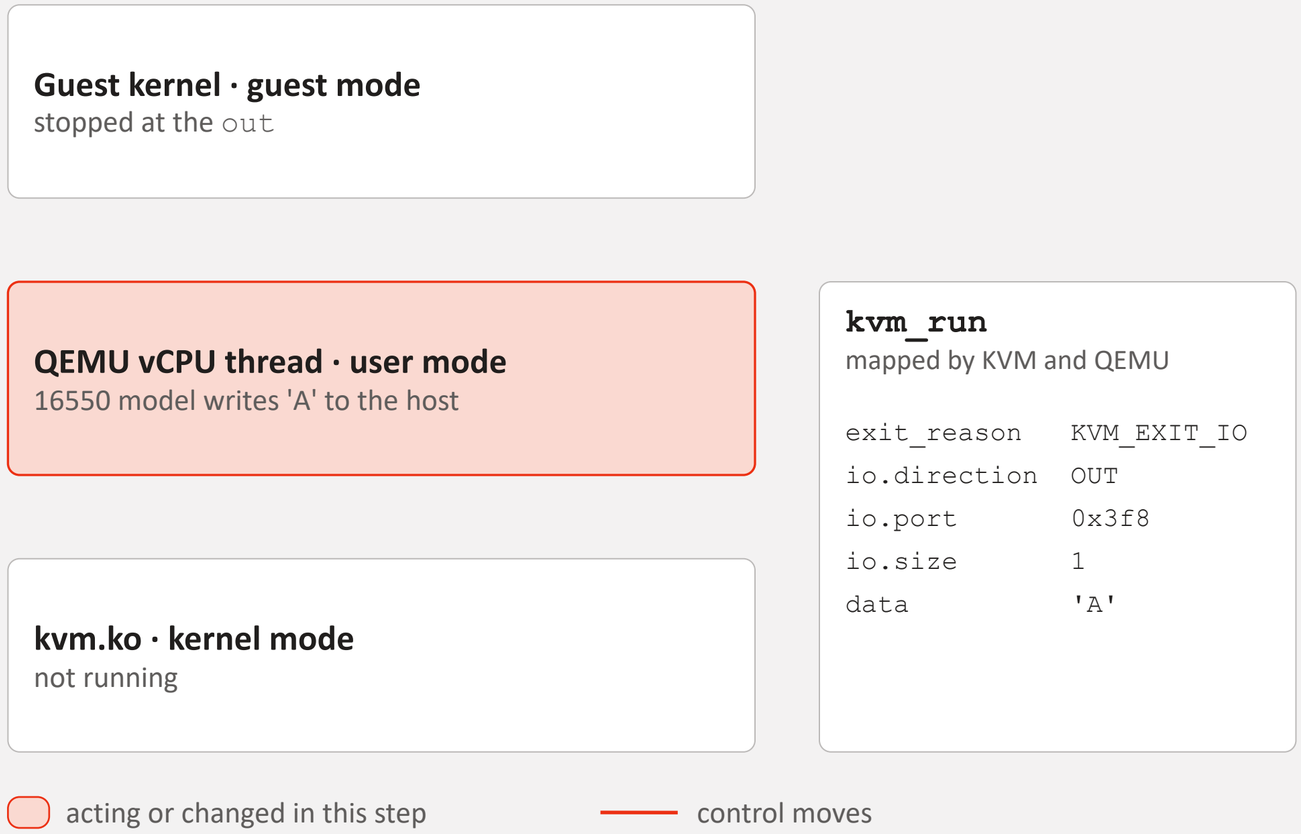
KVM hands the exit to QEMU

KVM checks its in-kernel devices and ioeventfds. None claims port 0x3f8, so it writes the details into `kvm_run` and returns from the `ioctl` with `KVM_EXIT_IO`.

STATE

ioctl returns · exit_reason = KVM_EXIT_IO

One port write, step by step



STEP 4 OF 6

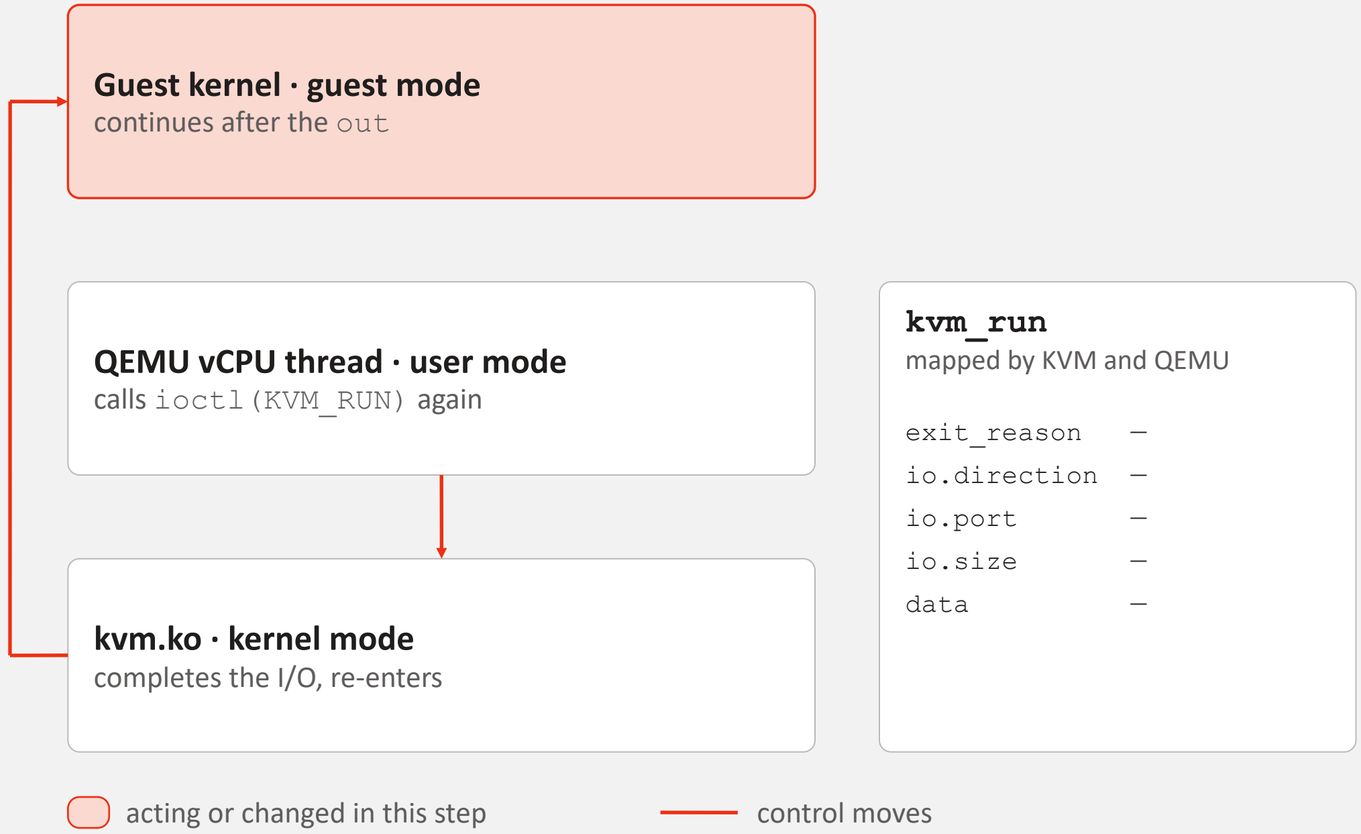
QEMU emulates the UART

The vCPU thread reads `exit_reason` and the byte from the same shared page, then calls its 16550 UART model. The model prints ‘A’ on the host terminal and updates the line status register.

STATE

device emulated in user space

One port write, step by step



STEP 5 OF 6

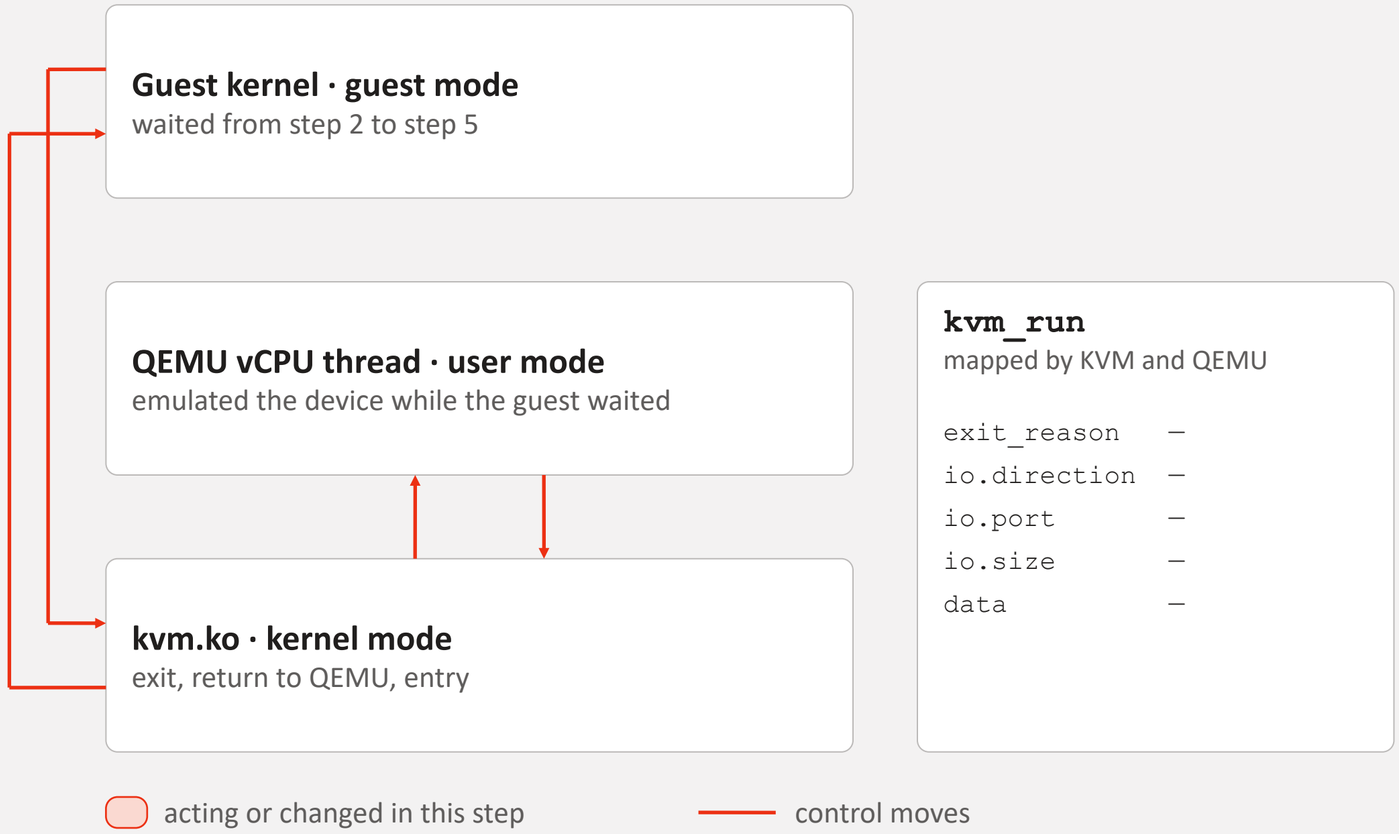
Back into the guest

QEMU loops and calls `ioctl(KVM_RUN)` again. KVM completes the instruction, moves the guest's instruction pointer past it, and resumes the guest with VMRESUME.

STATE

VM entry · guest resumes after the `out`

One port write, step by step



STEP 6 OF 6

What one byte cost

The guest ran no code from the exit in step 2 to the entry in step 5. Its thread went from guest mode into the kernel, out to QEMU in user space, and back. The guest waited while QEMU emulated the device. A disk or network card can’t pay that on every request; the next walkthrough shows how virtio avoids most of it.

STATE

1 exit · 2 kernel–user crossings · 1 entry

eventfd: a counter you can wait on

```
#include <sys/eventfd.h>
int efd = eventfd(0, 0);            // counter starts at 0

/* signaling side: any thread, process or the kernel */
uint64_t one = 1;
write(efd, &one, sizeof one);       // counter = 1, wakes waiters
write(efd, &one, sizeof one);       // counter = 2

/* waiting side */
struct pollfd p = { .fd = efd, .events = POLLIN };
poll(&p, 1, -1);                    // readable once counter > 0
uint64_t n;
read(efd, &n, sizeof n);            // n == 2, counter back to 0
```

A file descriptor that wraps one 64-bit counter in the kernel. KVM uses it as a doorbell.

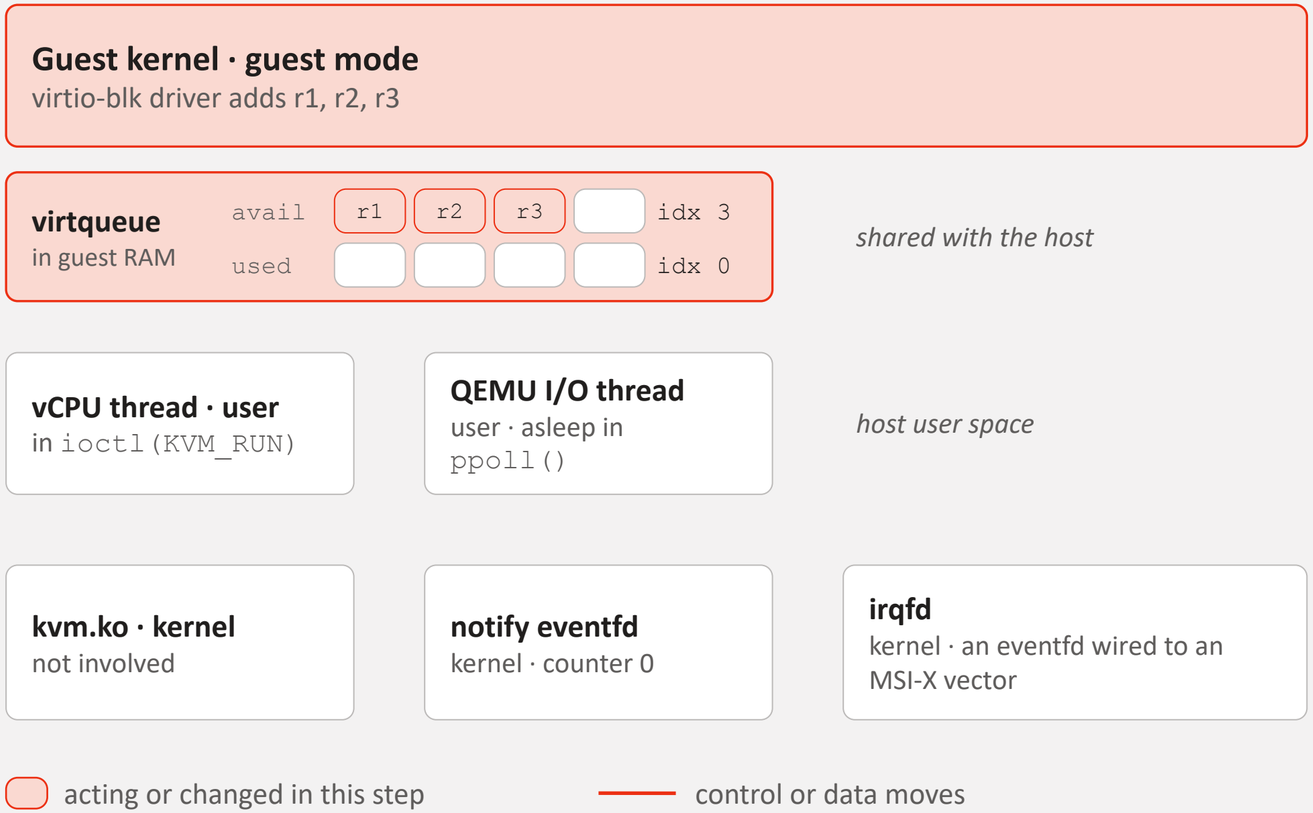
1. A write adds its 8-byte value to the counter and wakes any thread blocked in `read`, `poll` or `epoll`.
2. A read returns the whole count and resets it to 0, so several signals cost one wakeup.
3. Kernel code can signal or wait on the same counter, so it links kernel and user space.

TWO DOORBELLS IN KVM

- `KVM_IOEVENTFD`: KVM signals when the guest writes a given address; QEMU waits.
- `KVM_IRQFD`: QEMU signals; KVM injects an interrupt into the guest.

- Written for this lecture from eventfd(2), Linux man-pages; Linux Documentation/virt/kvm/api.rst (KVM_IOEVENTFD, KVM_IRQFD).

A virtio notification with ioeventfd



STEP 1 OF 5

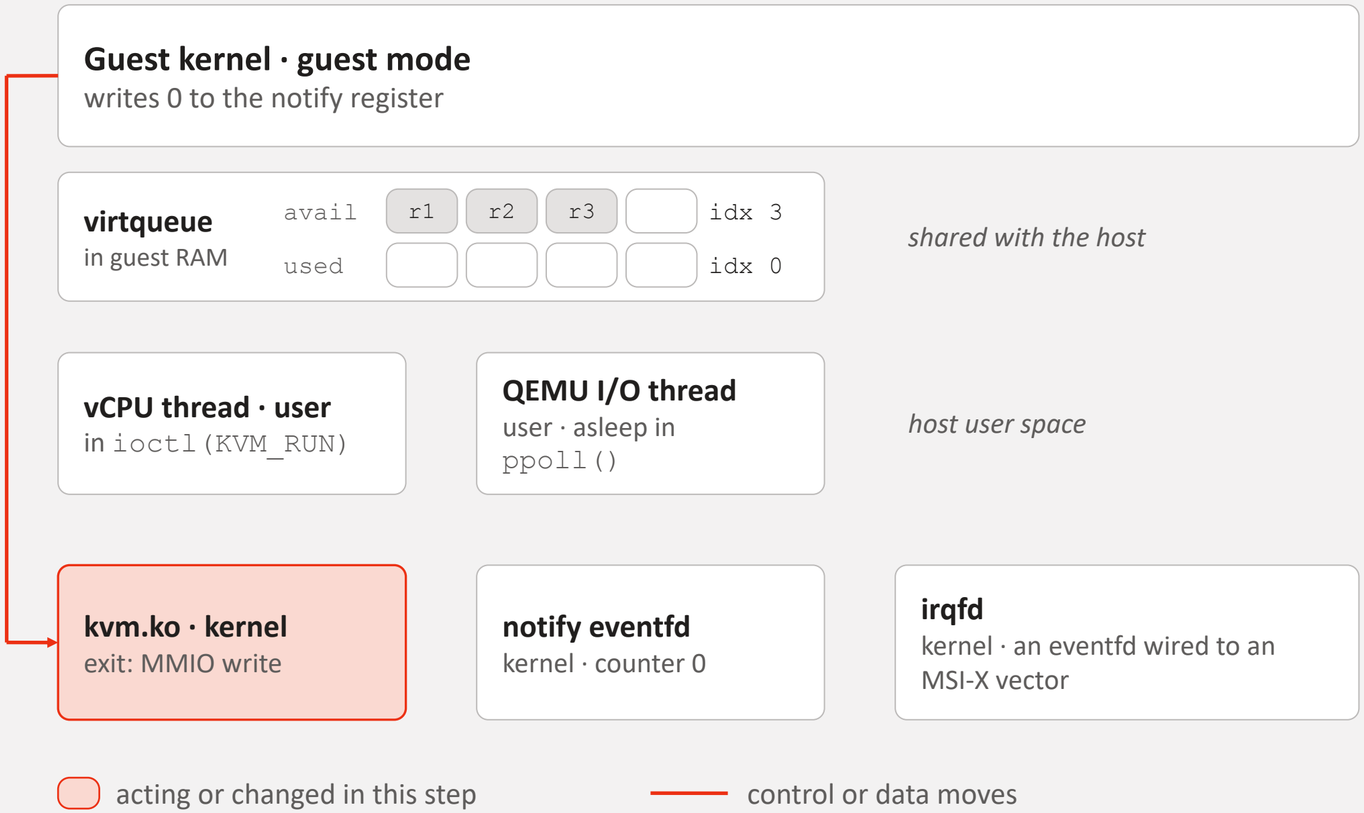
The guest queues three requests

The virtio-blk driver writes three request descriptors into the available ring and moves its index from 0 to 3. These are ordinary stores to guest RAM: nothing exits, and no host thread knows yet.

STATE

3 requests queued · 0 exits

A virtio notification with ioeventfd



STEP 2 OF 5

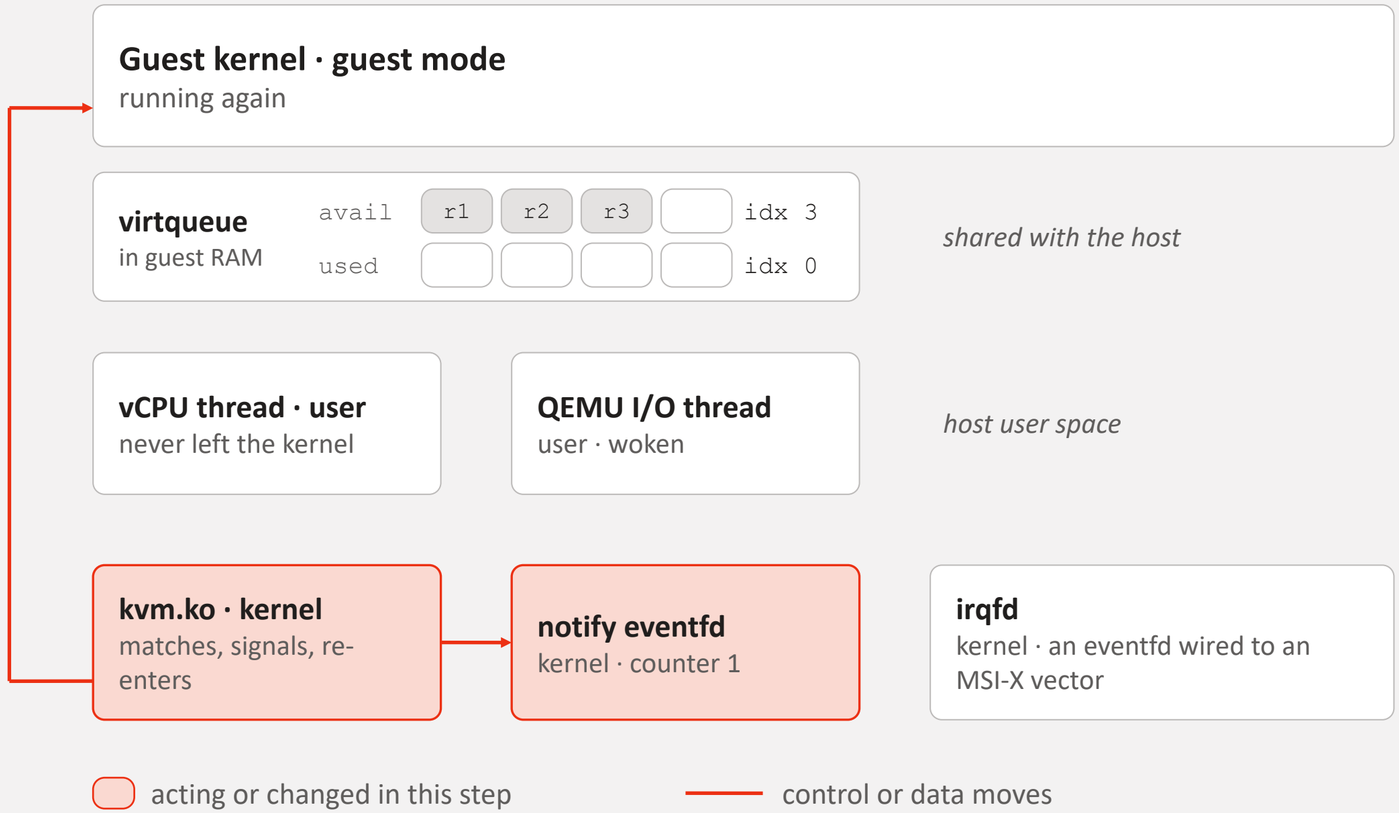
One notification for all three

The driver writes the queue number to the queue's notify register, a guest-physical MMIO address. Nothing is mapped there, so the write exits to KVM. One exit covers all three requests: this is where the guest batches.

STATE

1 VM exit for 3 requests

A virtio notification with ioeventfd



STEP 3 OF 5

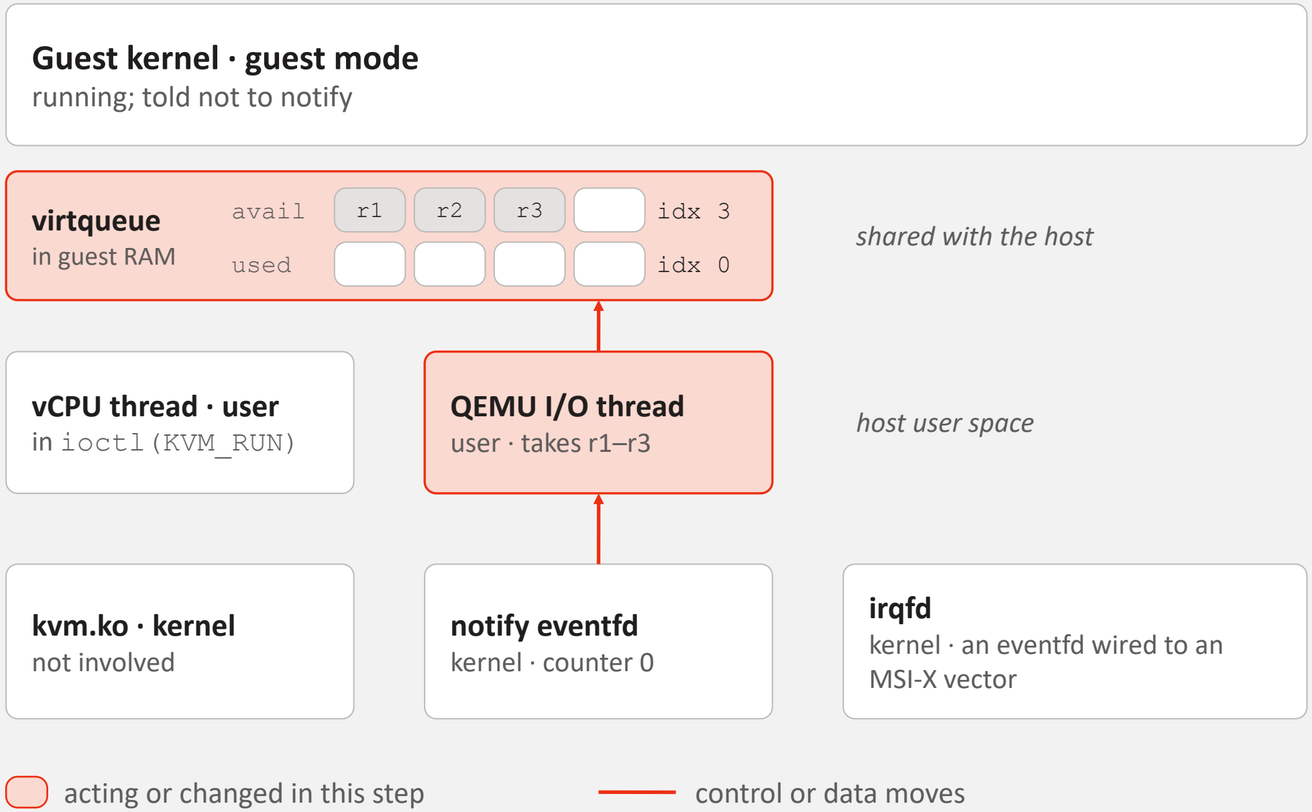
KVM signals an eventfd and resumes

At setup, QEMU registered the notify address with `KVM_IOEVENTFD`. KVM finds the address in that list, adds 1 to the eventfd's counter, which wakes the thread waiting on it, and re-enters the guest at once. This all runs in the kernel, on the vCPU thread, which never returns to QEMU.

STATE

kernel only · guest running again

A virtio notification with ioeventfd



STEP 4 OF 5

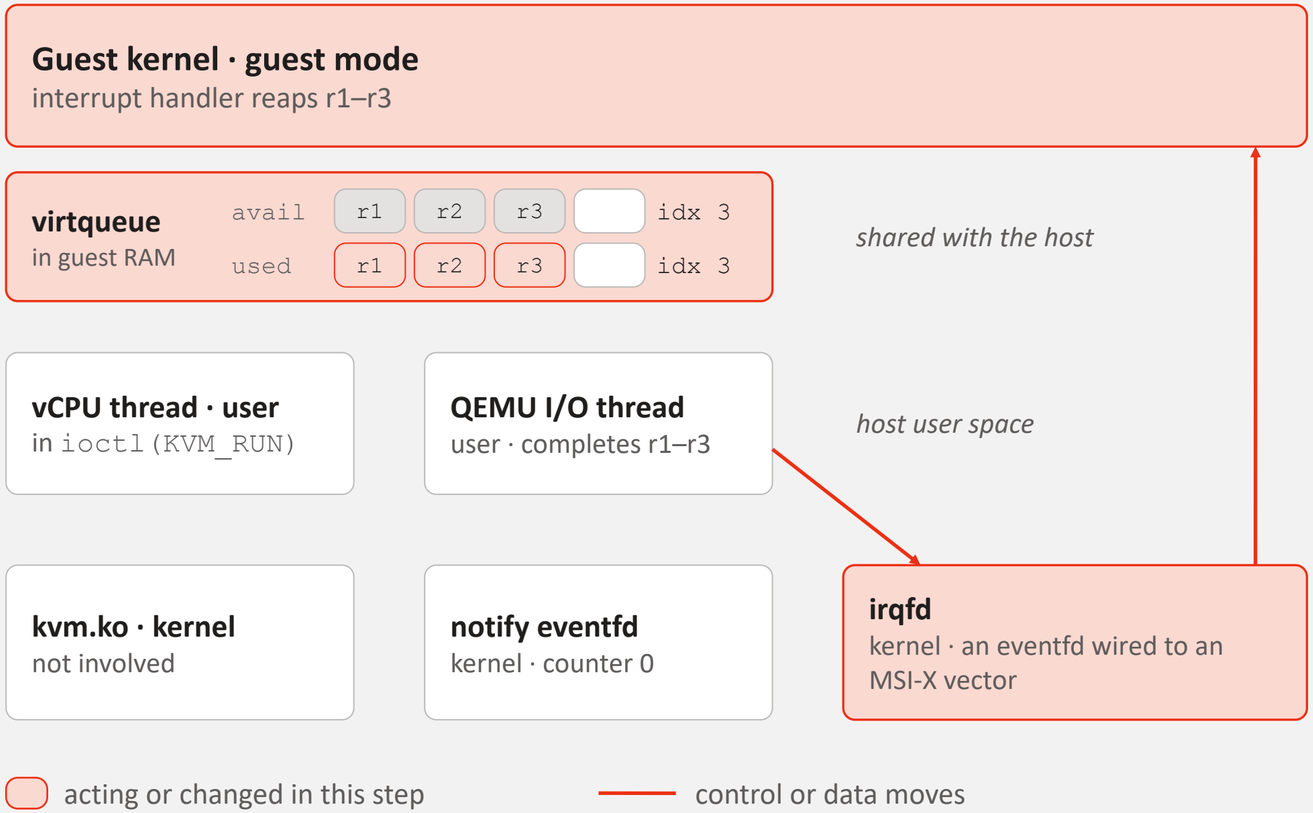
The I/O thread drains the ring

QEMU's I/O thread reads the eventfd, which returns the count and resets it to 0: several notifications cost one wakeup. It takes every request up to the available index, r1 to r3, and submits them to the host disk. While it works it sets a flag asking the guest not to notify, so new requests cause no exits.

STATE

3 requests per wakeup · user space

A virtio notification with ioeventfd



STEP 5 OF 5

One interrupt for three completions

As the requests finish, the I/O thread writes them to the used ring and moves its index to 3. It then writes to a second eventfd, registered with `KVM_IRQFD`. KVM injects the MSI-X interrupt into the guest from the kernel, and the guest's handler collects all three completions.

STATE

1 interrupt for 3 completions

ioeventfd: where each step runs, what gets batched

Every notification still exits. ioeventfd removes the return to QEMU's vCPU thread, and lets the guest run while the device works.

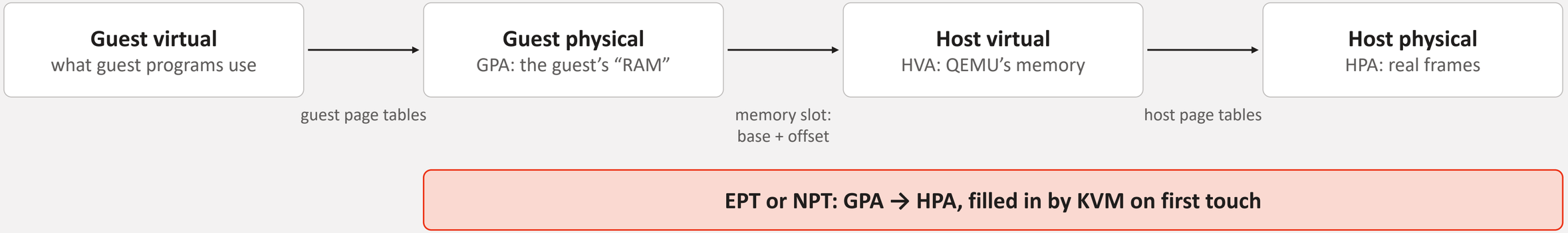
STEP	RUNS ON	MODE	WHAT GETS BATCHED
Queue requests	the guest's vCPU	guest mode	many requests go into the ring before one notification
Notify	the guest's vCPU	guest mode, then exit	one exit per batch; none while the device asks for no notifications
Signal, resume	vCPU thread, in `kvm.ko`	host kernel	notifications before the I/O thread runs add up in one counter
Process the ring	QEMU I/O thread	host user space	every available request in one pass
Complete	I/O thread, then `kvm.ko`	user, then kernel	one interrupt for many used entries

WITH VHOST

vhost-net for networks, or vhost-scsi for disks, moves the "process the ring" row into a kernel worker thread. Then no step of the data path runs in user space.

Guest memory: three address spaces

KVM maps guest-physical addresses through its memory slots to host-virtual ones, then lets Linux's own page tables do the rest.



Built lazily

The EPT starts empty. A first access exits to KVM, which turns the GPA into an HVA through the memory slot, asks Linux for the frame behind that HVA, and installs GPA → frame.

Linux still owns the memory

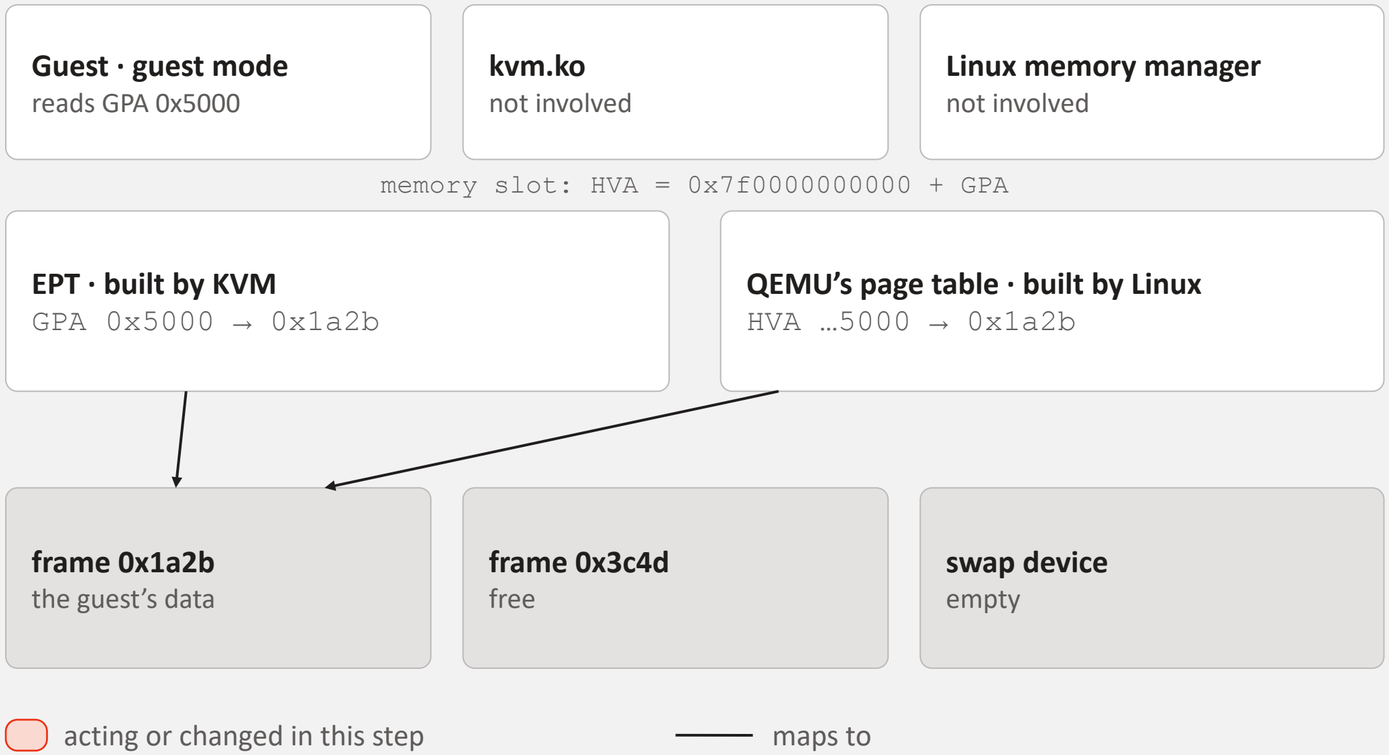
Guest RAM is anonymous process memory, so Linux can swap it, back it with huge pages, or merge identical pages across VMs with KSM.

Linux must tell KVM first

The EPT entry is a copy of Linux's translation. Before Linux swaps or moves the page, an MMU notifier makes KVM delete that copy. Next: one page, step by step.

- Linux Documentation/virt/kvm (api.rst: KVM_SET_USER_MEMORY_REGION); Kivity et al., OLS 2007; Linux include/linux/mmu_notifier.h.

Swapping out one guest page, step by step



STEP 1 OF 6

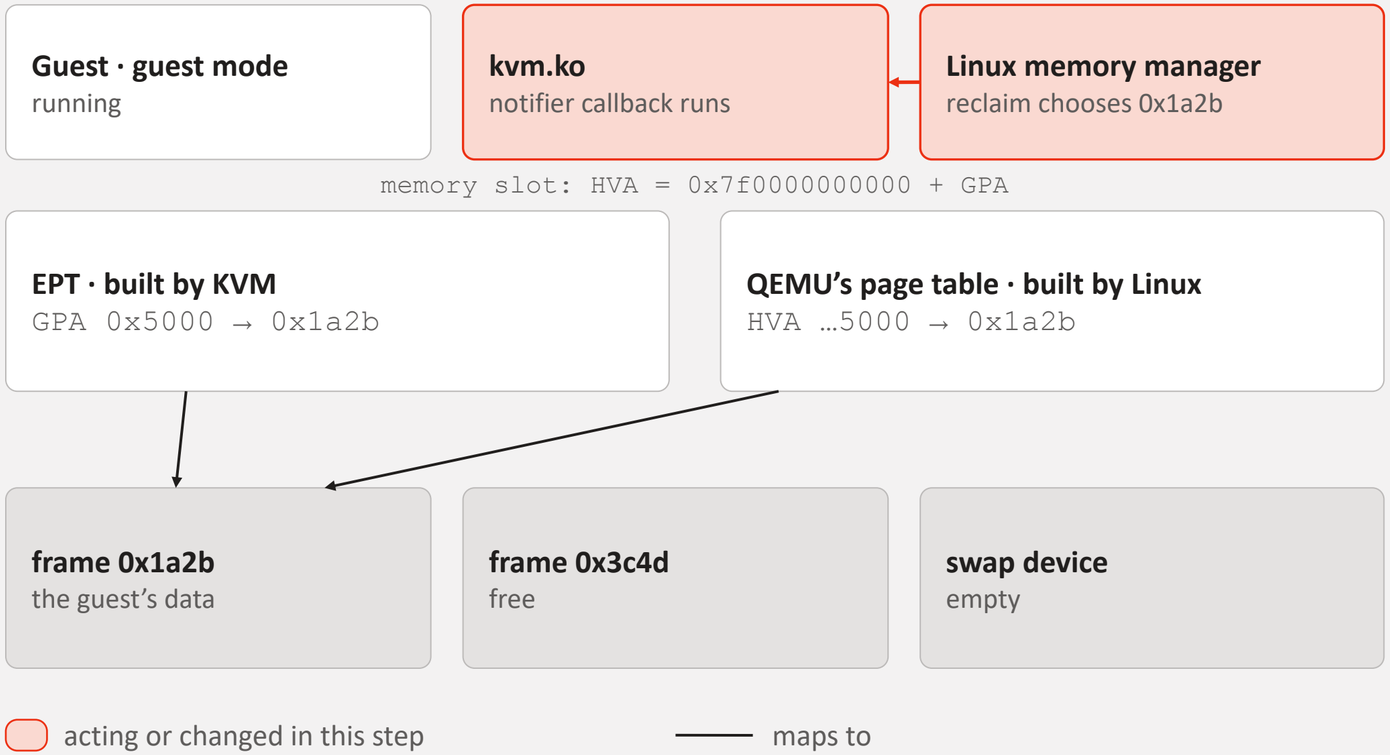
Two tables point at one frame

The guest reads GPA 0x5000 and the MMU follows the EPT to frame 0x1a2b. KVM built that entry on the first touch: the memory slot turned the GPA into QEMU's HVA 0x7f0000005000, and Linux's page table for QEMU gave the frame.

STATE

EPT and Linux's page table agree

Swapping out one guest page, step by step



STEP 2 OF 6

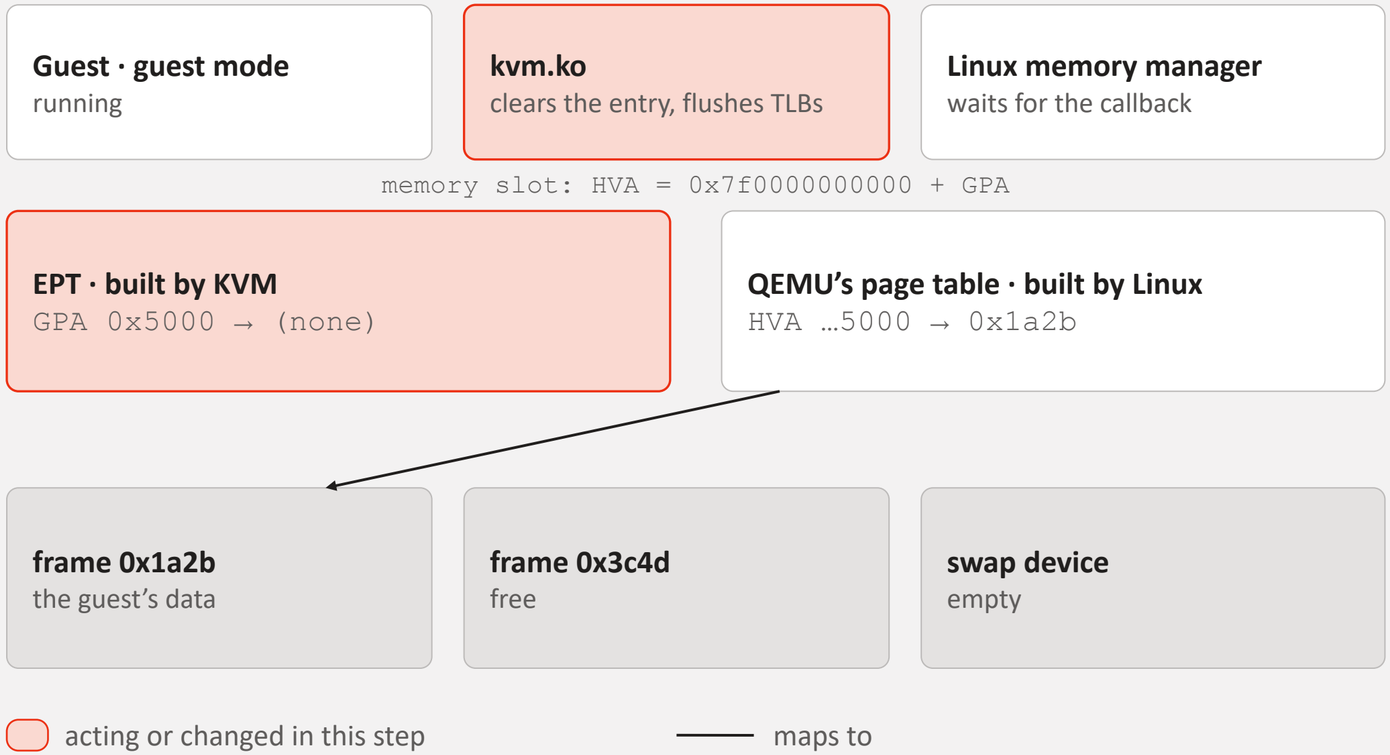
Linux picks this frame to reclaim

Memory is short. To Linux, frame 0x1a2b is an ordinary anonymous page of the QEMU process, so it may be swapped out. Before Linux changes that page's mapping, it calls the MMU notifiers registered on QEMU's address space: `invalidate_range_start` for this HVA.

STATE

notifier call: invalidate_range_start(HVA …5000)

Swapping out one guest page, step by step



STEP 3 OF 6

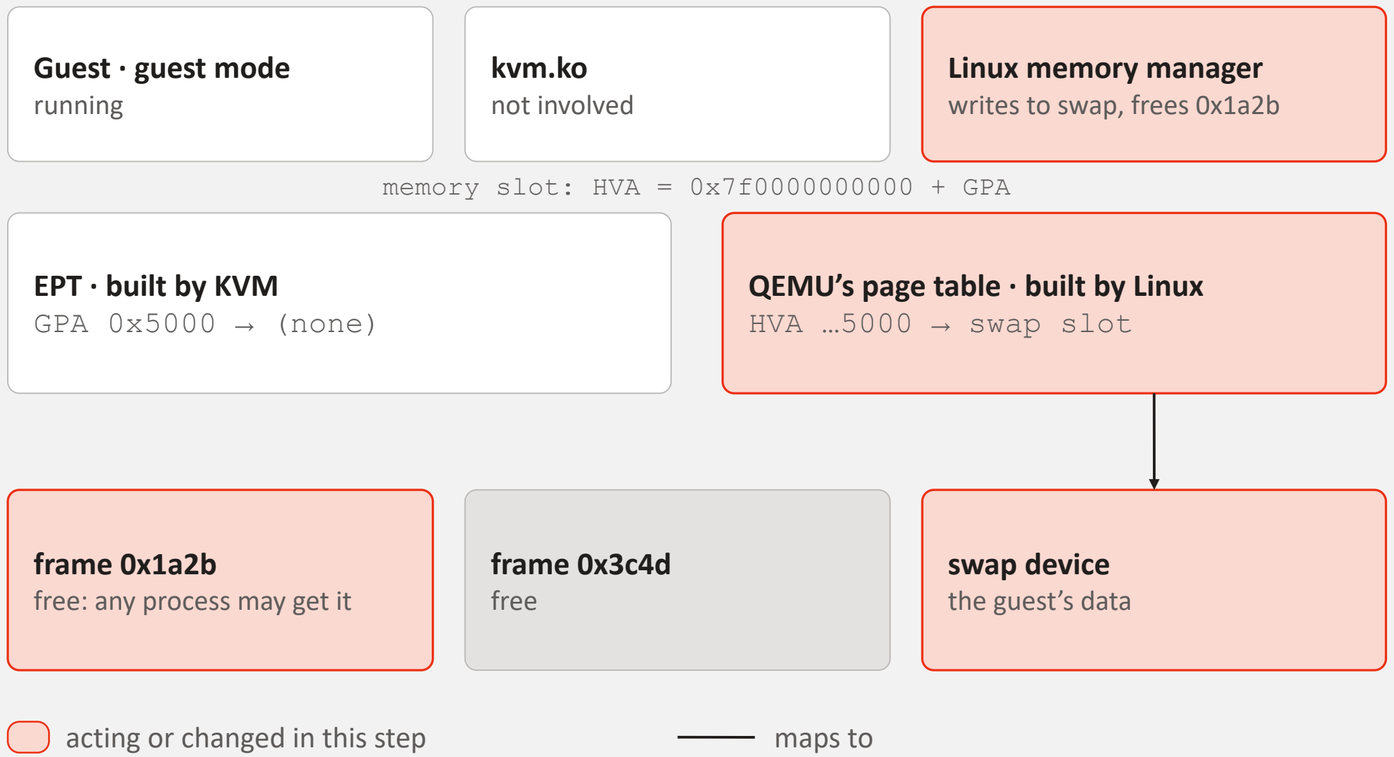
KVM deletes its copy first

KVM's callback turns the HVA back into GPA 0x5000 through the memory slot, clears that EPT entry, and flushes the guest TLB on every vCPU of the VM. Only then does the callback return. No path from the guest reaches frame 0x1a2b any more.

STATE

EPT entry cleared · guest TLBs flushed

Swapping out one guest page, step by step



STEP 4 OF 6

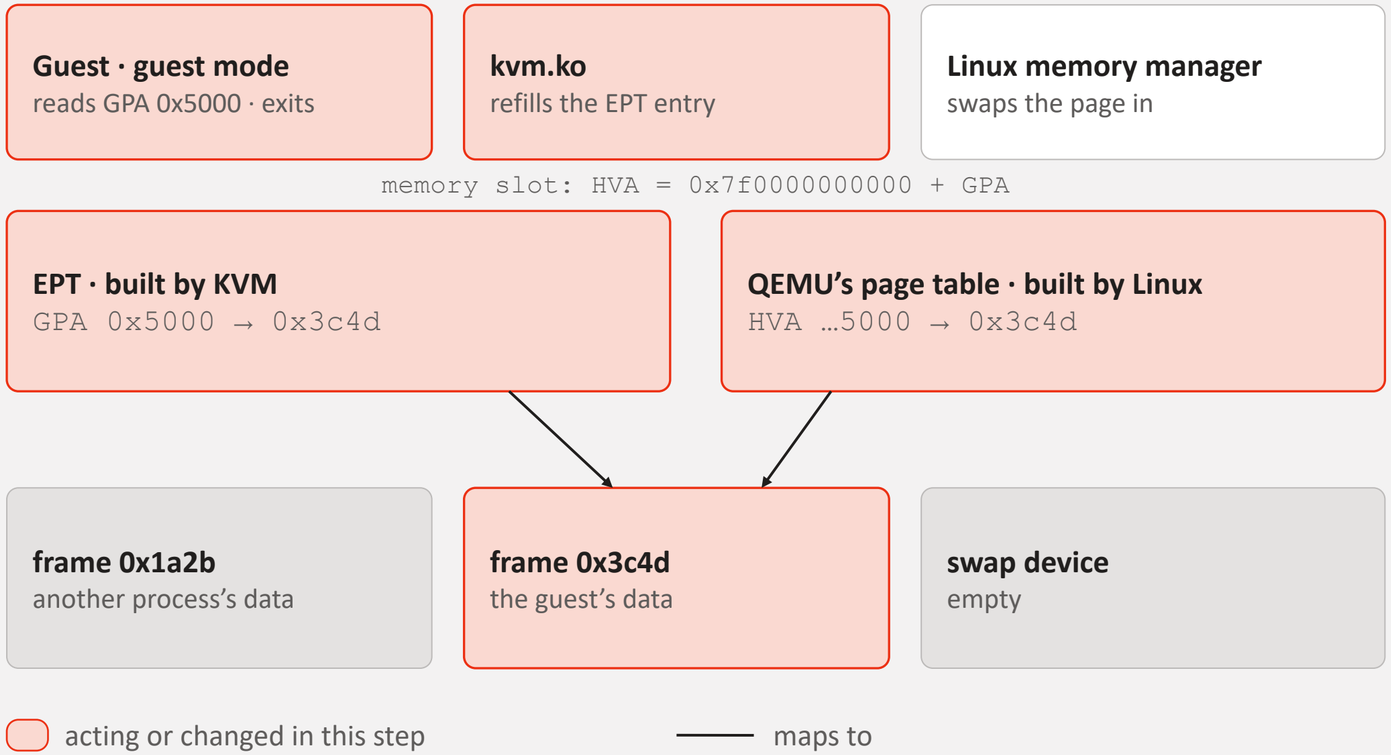
Linux swaps the page out

Linux clears its own page-table entry, writes the page to the swap device and frees frame 0x1a2b. The page allocator may now give that frame to any process. Linux then calls `invalidate_range_end`.

STATE

frame 0x1a2b free · data on swap

Swapping out one guest page, step by step



STEP 5 OF 6

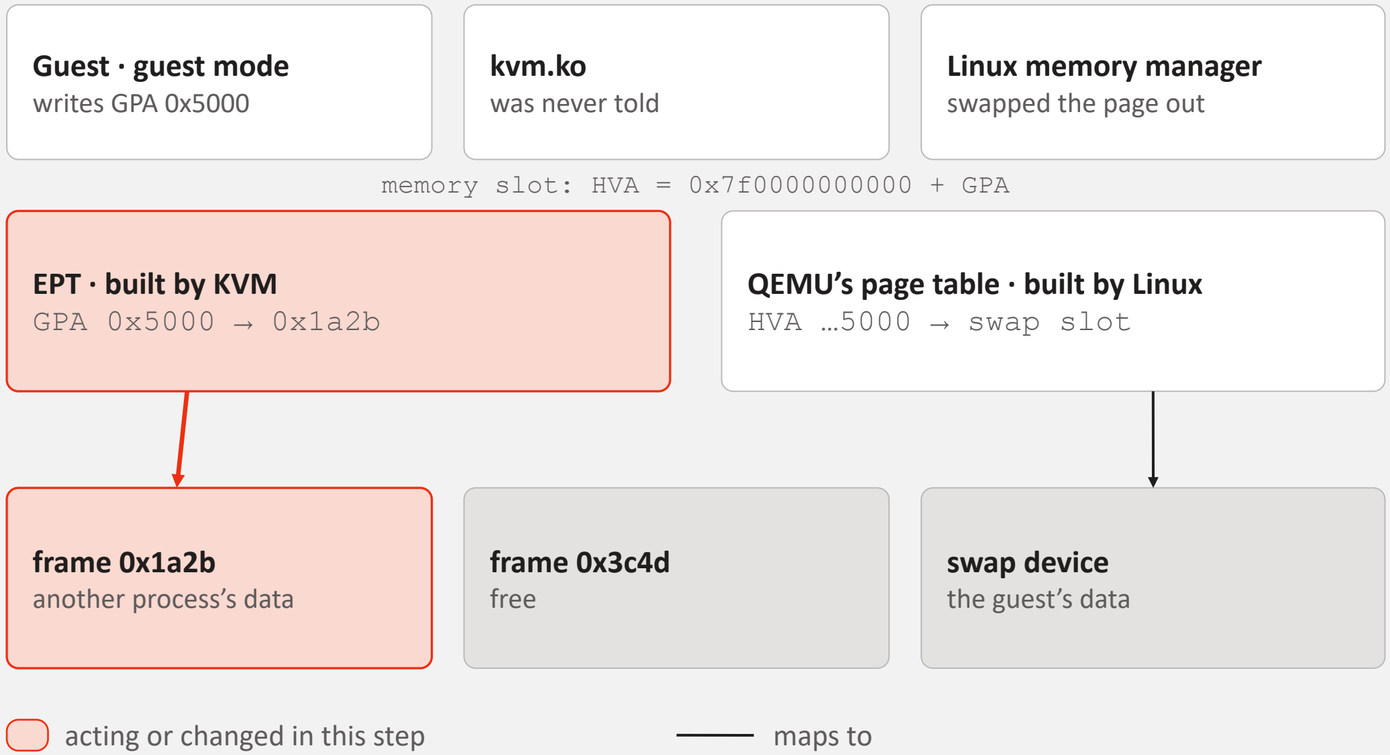
The guest touches the page again

The guest reads GPA 0x5000. The EPT entry is empty, so the CPU exits with an EPT violation. KVM finds the HVA, and Linux's ordinary page-fault path reads the page back from swap into a new frame, 0x3c4d. KVM installs GPA 0x5000 → 0x3c4d and resumes the guest, which never noticed.

STATE

both tables agree again, on frame 0x3c4d

Swapping out one guest page, step by step



STEP 6 OF 6

Why Linux must notify KVM

Skip step 3 and the EPT still says GPA 0x5000 → 0x1a2b after step 4. The guest goes on reading and writing frame 0x1a2b, which now belongs to another process. Before MMU notifiers, KVM avoided this by pinning guest memory, so Linux could not swap, migrate or merge it.

STATE

stale EPT entry: the guest writes another process's memory

What KVM keeps in the kernel

Whatever sits on the path of a frequent exit moves into the kernel. Everything else stays in the VMM process.

COMPONENT	WHERE IT RUNS	WHY THERE
VM entry and exit, vCPU state	`kvm.ko`	every exit starts here
EPT or NPT management	`kvm.ko`	the first touch of each guest page faults here
Local APIC, I/O APIC, PIT	`kvm.ko` (optional)	guests program timers and send interrupts constantly
ioeventfd, irqfd	`kvm.ko`	notify a device or inject an interrupt without user space
virtio-net data path (vhost-net)	host kernel thread	packets move between virtqueue and tap device in the kernel
Device models, firmware, migration	QEMU or another VMM	large, and rarely on the fast path

- Linux Documentation/virt/kvm/api.rst (KVM_CREATE_IRQCHIP, KVM_IOEVENTFD, KVM_IRQFD); drivers/vhost.

03

Linux: virtio and pKVM

virtio: lecture 11’s I/O ring, standardized

Rusty Russell’s virtio (2008) gave Linux one paravirtual device interface for every hypervisor. Its pieces map onto Xen’s.

XEN (LECTURE 11)	VIRTIO	WHAT CHANGED
shared I/O ring page	virtqueue: descriptor table, available ring, used ring	descriptors chain buffers; the rings carry only indices
grant reference	guest-physical address in a descriptor	the host can map all guest RAM, so no grants
event channel to the back end	a write to the queue’s notify register	KVM turns it into an eventfd signal
event channel to the guest	an MSI-X interrupt, injected through irqfd	a standard PCI interrupt
XenStore	PCI configuration space, or virtio-mmio registers	discovered like real hardware
`req_event`, `rsp_event`	`used_event`, `avail_event`	same idea: notify only when asked

- R. Russell, virtio: towards a de-facto standard for virtual I/O devices, ACM SIGOPS OSR 42(5), 2008; OASIS VIRTIO specification.

Three back ends for one virtio-net queue

The guest driver is the same in all three. Only the code that consumes the virtqueue on the host changes.

01

QEMU in user space

The notify write exits; an ioeventfd wakes a QEMU I/O thread, which reads the virtqueue and writes the packet to a tap device. Simple, but every packet crosses into user space.

02

vhost-net

A host kernel thread owns the virtqueue and moves packets between it and the tap device. QEMU only sets it up and handles control messages.

03

vhost-user

A separate user process, such as a DPDK virtual switch, maps guest memory and polls the virtqueue. No kernel code sits on the data path.

WHY THE SPLIT PAYS OFF

Operators move the data path between QEMU, the kernel and a switch process without touching any guest, because the ring is the interface.

- Linux drivers/vhost/net.c; QEMU docs/interop/vhost-user.rst.

Why Google built pKVM

In 2020, Android devices already had hypervisors, but none that Android could use, update or trust.

01

Hypervisors were fragmented

SoC vendors and device makers shipped their own EL2 code as firmware. Deacon called it the wild west: highly privileged, hard to update, and running no VMs for Android to use.

02

TrustZone code was over-privileged

DRM and other third-party binaries run in the secure world, which can read all of memory. Android, the main OS, ran in the least privileged part of the system.

03

Android needed a different threat model

A guest's data must stay private even if the host kernel is compromised. Ordinary KVM with VHE puts the host kernel in charge of all guest memory.

04

The parts were already there

Every arm64 Android device has virtualization hardware and two-stage MMUs. KVM has run on arm64 since Linux 3.11 (2013), and GKI ships one kernel to every device.

- W. Deacon, Virtualisation for the Masses: Exposing KVM on Android, KVM Forum 2020 (slides); J. Edge, KVM for Android, LWN, November 11, 2020.

What runs in Android’s VMs

AOSP lists three uses. Two need a protected VM; the Linux terminal deliberately runs in an ordinary one.

01

Isolated compilation

After a Mainline update, a protected VM compiles boot-classpath and system-server code before the reboot. It signs the output only if every step succeeded, so Android can make it fail but not change it.

02

On-device content safety

Play Protect live threat detection runs its classifier in a protected VM. The server encrypts its protections to a key only that VM holds, even on rooted devices.

03

Linux terminal

The Terminal app runs a Debian-based image in a VM on select devices. This VM is not protected: it holds nothing Android must not see.

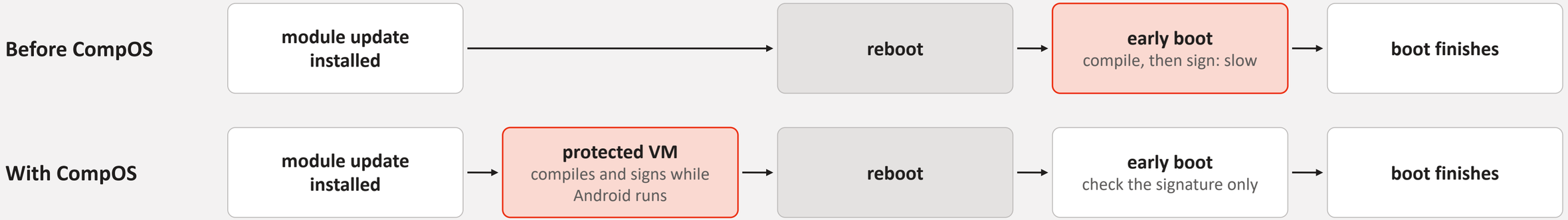
DEVICE MAKERS TOO

OPPO runs an on-device risk-control service for apps in an AVF virtual machine.

- AOSP, AVF use cases (source.android.com/docs/core/virtualization/usecases).

Isolated compilation: why compile on the phone at all?

Android ships compiled code in its system image, but a module update can make that code stale. The question is who may rebuild it.



Why compile on the phone

Since Android 12, ART is a Mainline module updated apart from the OS. Its compiled code for boot-classpath and system-server JARs must match the exact JARs installed, so an update can force a rebuild.

Why the output is sensitive

Zygote and system_server load this code, so tampering would survive reboots with system privileges. odsign signs it with a key usable only in early boot, when only trusted code runs.

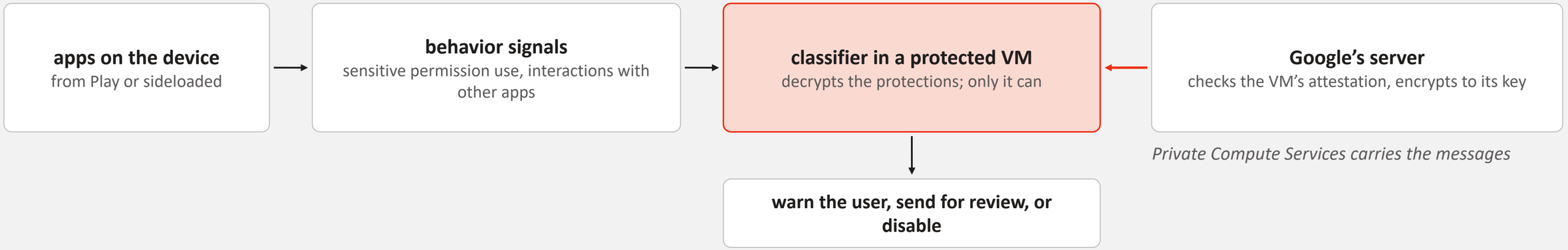
Why a protected VM

Compiling before the reboot saves boot time, but by then Android may be compromised. The VM signs only a clean compile, so Android can make it fail, not change it.

- AOSP: On-device signing architecture (source.android.com/docs/security/features/verifiedboot); AVF use cases, “Isolated compilation”.

Play Protect’s live threat detection

Two protections in opposite directions: the user’s data stays on the phone, and Google’s detection logic stays hidden from the phone.



What Play Protect is

Google’s malware protection, part of Google Play services. Google says it scans 200 billion apps a day and protects more than 3 billion users.

What live threat detection adds

Announced in May 2024: on-device AI watches how apps behave, to catch apps that hide their behavior or stay dormant. Pixel 6 and later got it first.

Why a protected VM

Malware authors want to read the model to evade it, and a rooted phone would let them. In a protected VM, AOSP says, it resists reverse engineering even on rooted devices.

- Google Security Blog, I/O 2024: What’s new in Android security (May 2024); AOSP, AVF use cases, “Content Safety On-Device”.

Linux terminal: why a VM, and why not a protected one

The Terminal app uses AVF to run Debian, but in an ordinary VM: here the VM protects Android, not the other way round.

Why a VM at all

- An Android app is sandboxed: its own user ID, SELinux and seccomp limits, no root.
- Android’s C library is Bionic; Debian’s packages expect glibc and a normal Linux user space.
- Running a distribution by translating system calls, as proot under Termux does, is slow and incomplete.
- A VM gives a real Linux kernel, root inside the guest, and apt, with Android untouched.

Why not a protected VM

- It holds nothing that must be hidden from Android.
- Protected VMs pay for bounce buffers, memory that can’t be swapped, and a short device list.
- An ordinary VM can get network access through Android’s tethering, and forward ports to Android.

ISOLATION RUNS ONE WAY

Whatever runs as root in Debian stays inside the VM and can’t reach Android’s apps or data; Android can still read the VM, which is fine for a scratch space.

- AOSP, AVF use cases, “Linux development environment”; our reasoning for the sandbox and cost points.

What each use case protects, and from whom

Each needs a result or a secret that Android itself can’t tamper with, while Android keeps running everything else.

USE CASE	PROTECTS	FROM WHOM	WITHOUT A PROTECTED VM
Isolated compilation	the integrity of compiled boot code	a compromised Android, kernel included	compile during early boot, before untrusted code runs, which slowed the first boot after each update
On-device content safety	the detection model and its key	the device’s owner, even with root	the model sits where root can read and change it
Third-party code now in TrustZone, such as DRM	Android and other apps	that code itself	it runs in the secure world, which can read all of memory
Linux terminal	nothing secret	no one	an ordinary VM is enough; it is not protected

THE COMMON THREAD

For one job, Android is treated as untrusted. A process can’t give that, and neither can ordinary KVM, whose host reads all guest memory.

- AOSP, AVF use cases; W. Deacon, KVM Forum 2020. Last column: AOSP for isolated compilation, our reasoning for the other rows.

pKVM in brief: what it is, and what it isn’t

01

Built from Linux, shipped in the same image

The EL2 code lives in `arch/arm64/kvm/hyp/nvhe` and is linked into the kernel image in its own ELF sections, with every symbol renamed `__kvm_nvhe_*` so it can’t call Linux by accident.

02

Not Linux running at EL2

It has its own address space, page tables and page allocator. It can’t block or schedule, and it shares only `memcpy`-style helpers with Linux.

03

Not all of KVM either

The `/dev/kvm` ioctls, memory slots, vCPU threads and interrupt emulation stay in the host kernel at EL1, untrusted. EL2 does the world switch, stage-2 tables, page ownership and checks on firmware calls.

04

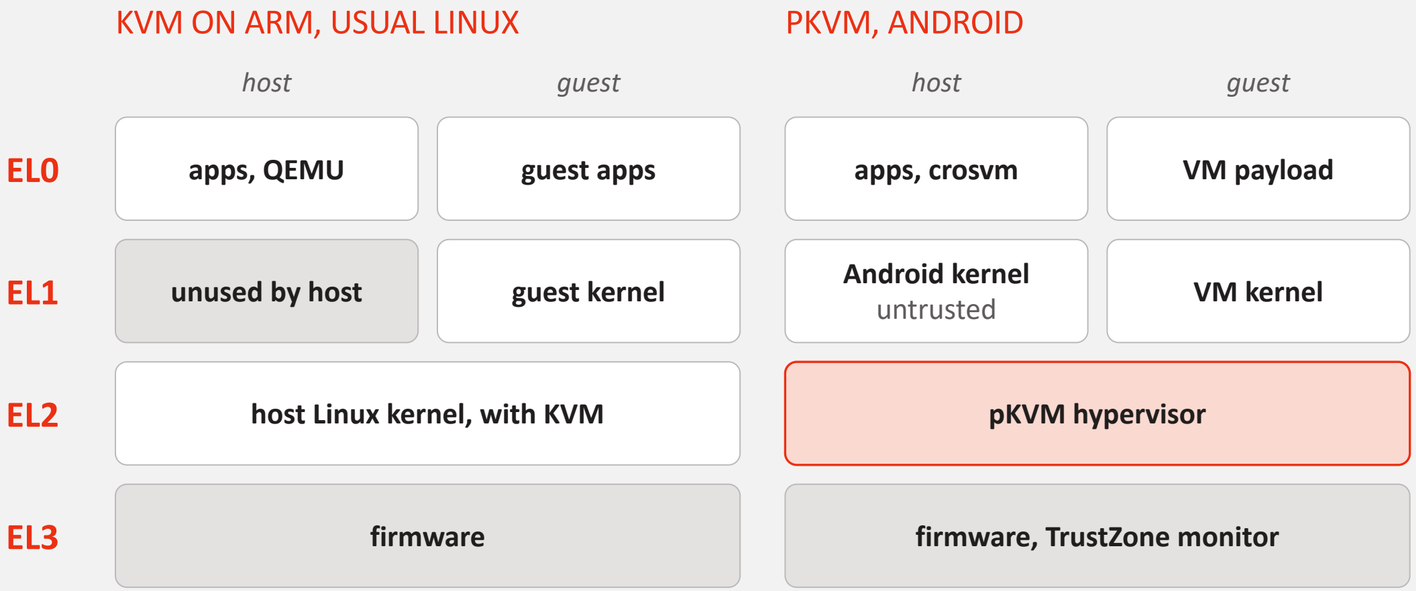
Small, with a narrow door

About 20,000 lines at EL2, against about 63,000 for KVM’s host side, reached through a fixed table of under 60 host hypercalls.

- Mainline Linux 7.4-rc3 (commit 4405c18, November 18, 2026), arch/arm64/kvm/hyp/nvhe/Makefile and line counts; LWN, KVM for Android (2020).

pKVM: where each piece runs

Android’s protected KVM keeps KVM’s code, but moves the host kernel out of EL2, so the host can no longer read protected guests.



With VHE (Armv8.1 and later) the whole host kernel runs at EL2.

Only the small hypervisor part of KVM keeps EL2.

ONE IMAGE, TWO PROGRAMS

The EL2 hypervisor is built from Linux’s source tree into the same kernel image, but it runs as its own small program, not as Linux.

THE HOST STILL DOES MOST WORK

Scheduling, devices, virtio and interrupt emulation stay in the host kernel at EL1, now as untrusted code.

COMPARE XEN AND HYPER-V

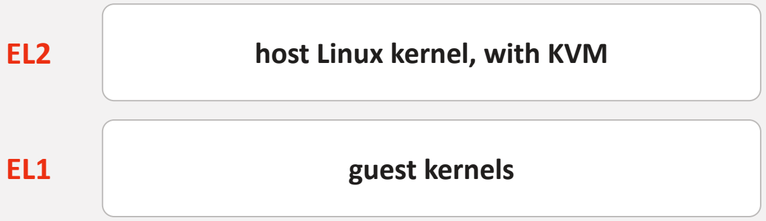
There, Domain 0 or the root partition can map any guest’s memory. With pKVM, the host OS can’t.

- Android Open Source Project, AVF architecture (source.android.com/docs/core/virtualization/architecture), “Background” and “Hypervisor”.

Three ways to put a hypervisor under Android

Deacon’s 2020 talk weighed three layouts. Only one keeps the host kernel out of the trusted code without moving Android into a VM.

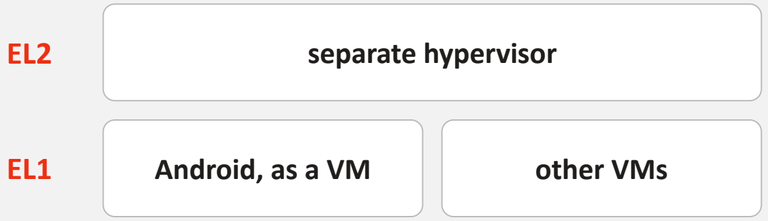
KVM WITH VHE



Fast: the host kernel is already at EL2, so no world switch is needed to reach it. But the whole host kernel is trusted and can read every guest.

Rejected: breaks the threat model

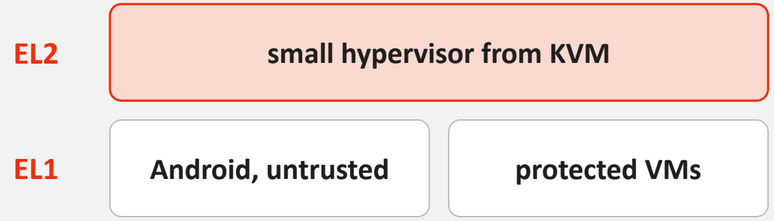
ANDROID AS A GUEST



The Xen and Hyper-V layout. It removes the host from the trusted code, but Android’s interrupts and devices pass through the hypervisor. Deacon cited interrupt latency and IOMMUs not ready for device pass-through.

Not chosen in 2020

PKVM: SPLIT KVM



Android keeps its devices, drivers and scheduler. Only the EL2 code is trusted, and it is small enough that the team planned to apply formal verification.

Chosen

- J. Edge, KVM for Android, LWN, November 11, 2020, reporting W. Deacon’s KVM Forum 2020 talk.

What runs at EL2, and what doesn’t

The EL2 code comes from the same source tree and the same kernel image as Linux, but it runs as a separate small program.

One kernel image, one build

Linux kernel · runs at EL1, untrusted
all of Linux, plus KVM’s host side: `/dev/kvm`, memory slots, vCPU threads, interrupt emulation
KVM host side: about 63,000 lines (`arch/arm64/kvm`, `virt/kvm`)

a fixed table of host hypercalls: under 60 entries

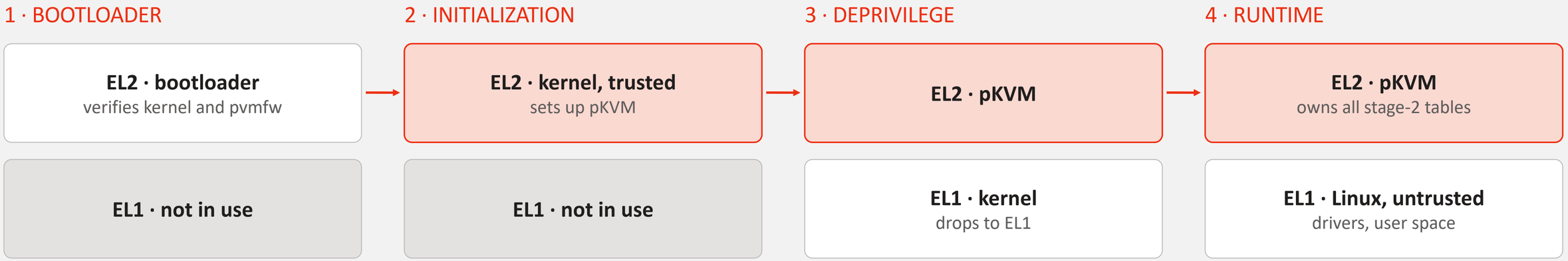
EL2 hypervisor object · runs at EL2
own ELF sections; every symbol renamed `__kvm_nvhe_*`; own page tables and page allocator
no scheduler, not preemptible; shares only `memcpy`-style helpers with Linux
about 20,000 lines (`arch/arm64/kvm/hyp`)

EL2 FILE	ITS JOB
`switch.c`	world switch between host and guests
`mem_protect.c`	stage-2 tables and page ownership
`pkvm.c`	protected VM and vCPU state
`hyp-main.c`	the host’s hypercall table
`ffa.c`, `psci-relay.c`	checks the host’s calls to EL3 firmware

- Counted in mainline Linux 7.4-rc3 (commit 4405c18, November 18, 2026): .c, .S and .h lines including comments. arch/arm64/kvm/hyp/nvhe/Makefile; LWN, KVM for Android (2020).

Boot: the kernel gives up EL2

Trust is highest at power-on. pKVM is set up before any untrusted code runs, and then the kernel steps down.



The bootloader verifies the kernel image, loads and verifies pvmfw, and derives pvmfw's secrets from the device's root of trust.

The bootloader enters the kernel at EL2. Trusted kernel code sets up pKVM and its modules. Nothing untrusted has run yet.

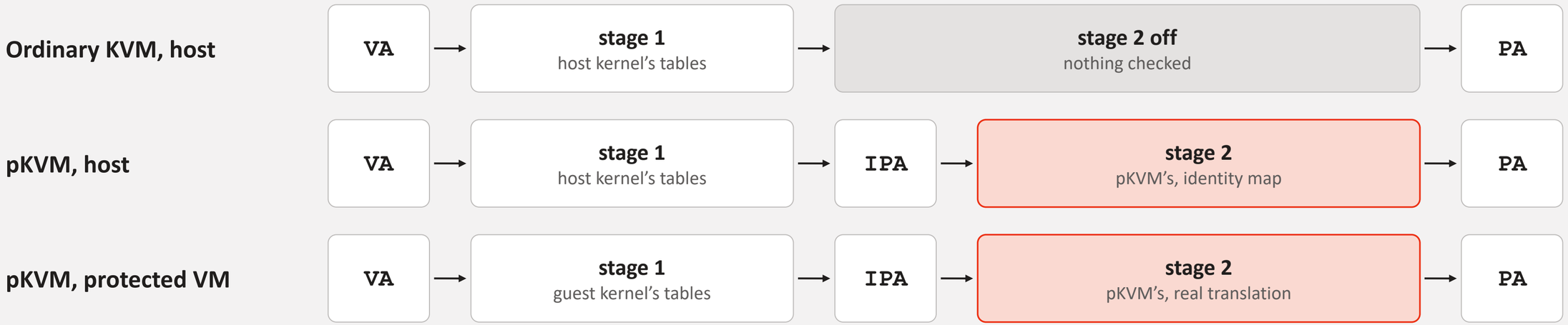
The kernel sees it is running at EL2 and drops itself to EL1. Only pKVM stays at EL2.

Linux boots as usual. pKVM now treats it as untrusted, and keeps pvmfw's memory hidden from it until shutdown.

- AOSP, AVF architecture: "Boot procedure" and "pVM firmware: Device boot".

Stage-2 translation for the host too

Ordinary KVM turns stage 2 off while the host runs. pKVM leaves it on, so every host access is checked against page ownership.

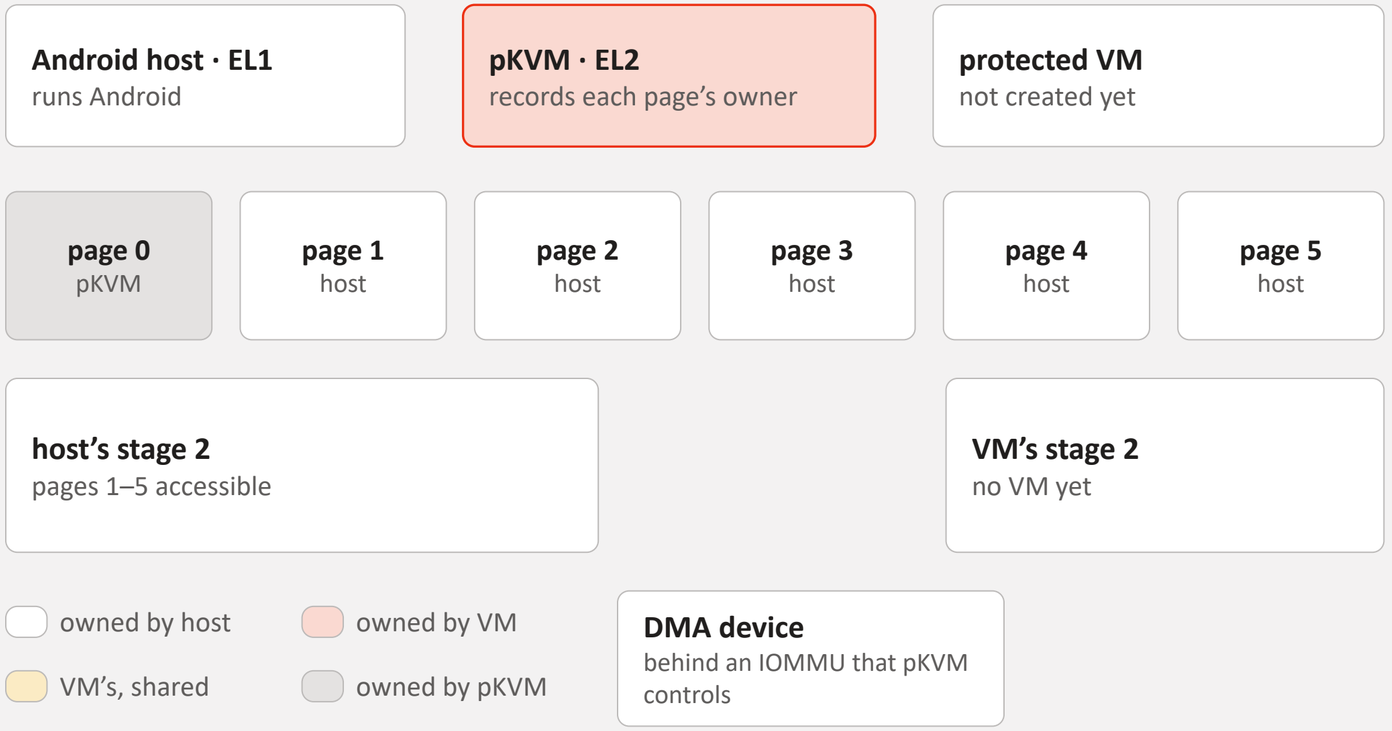


WHY AN IDENTITY MAP FOR THE HOST

For the host, IPA equals PA, so the table can be indexed by physical address. pKVM records each page's owner and sharing state in spare, software-defined bits of these entries, and large block mappings keep TLB pressure low.

- AOSP, AVF architecture: "CPU memory access protection" and "Memory ownership".

A page’s life under pKVM, step by step



STEP 1 OF 6

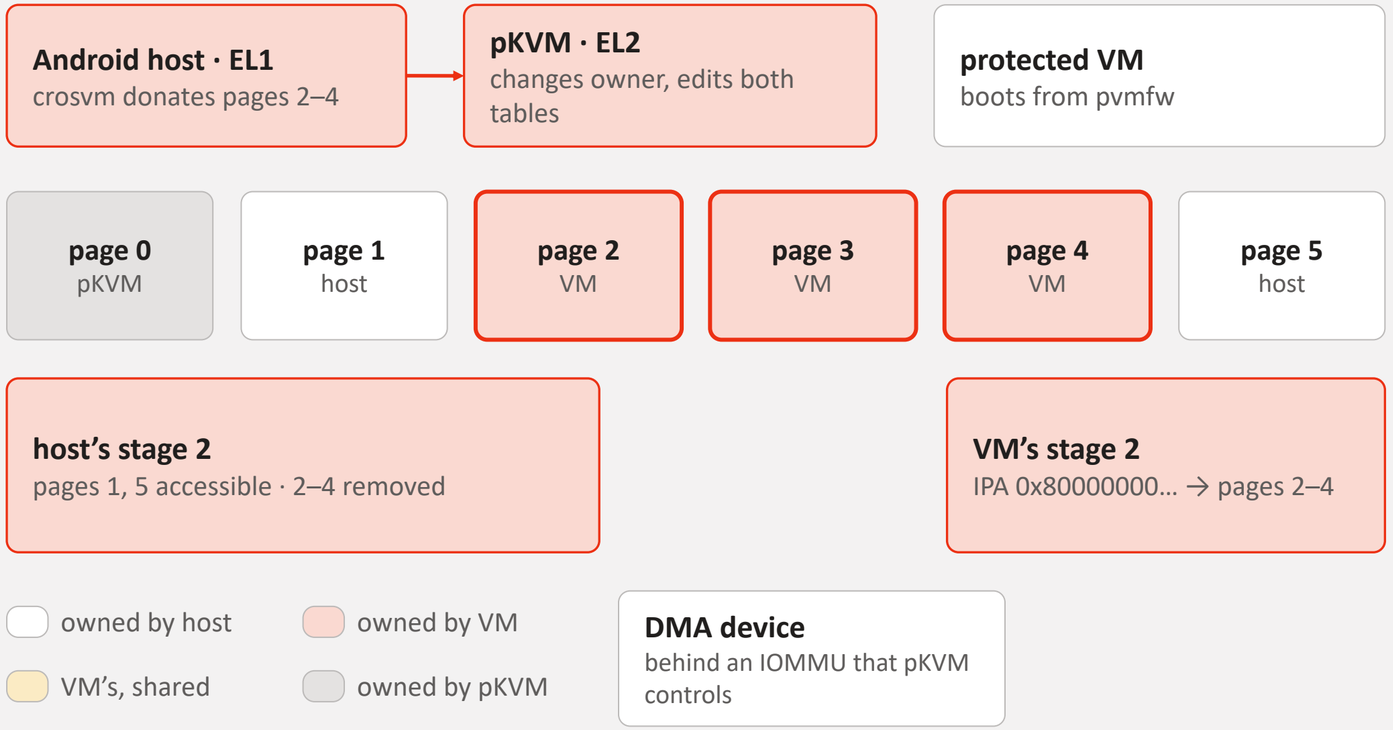
At boot, the host owns every page

pKVM assumes every page outside its own memory belongs to the host. The host’s stage-2 table is an identity map, so Linux can still touch pages 1 to 5, but pKVM now checks every access.

STATE

one owner recorded for every page

A page’s life under pKVM, step by step



STEP 2 OF 6

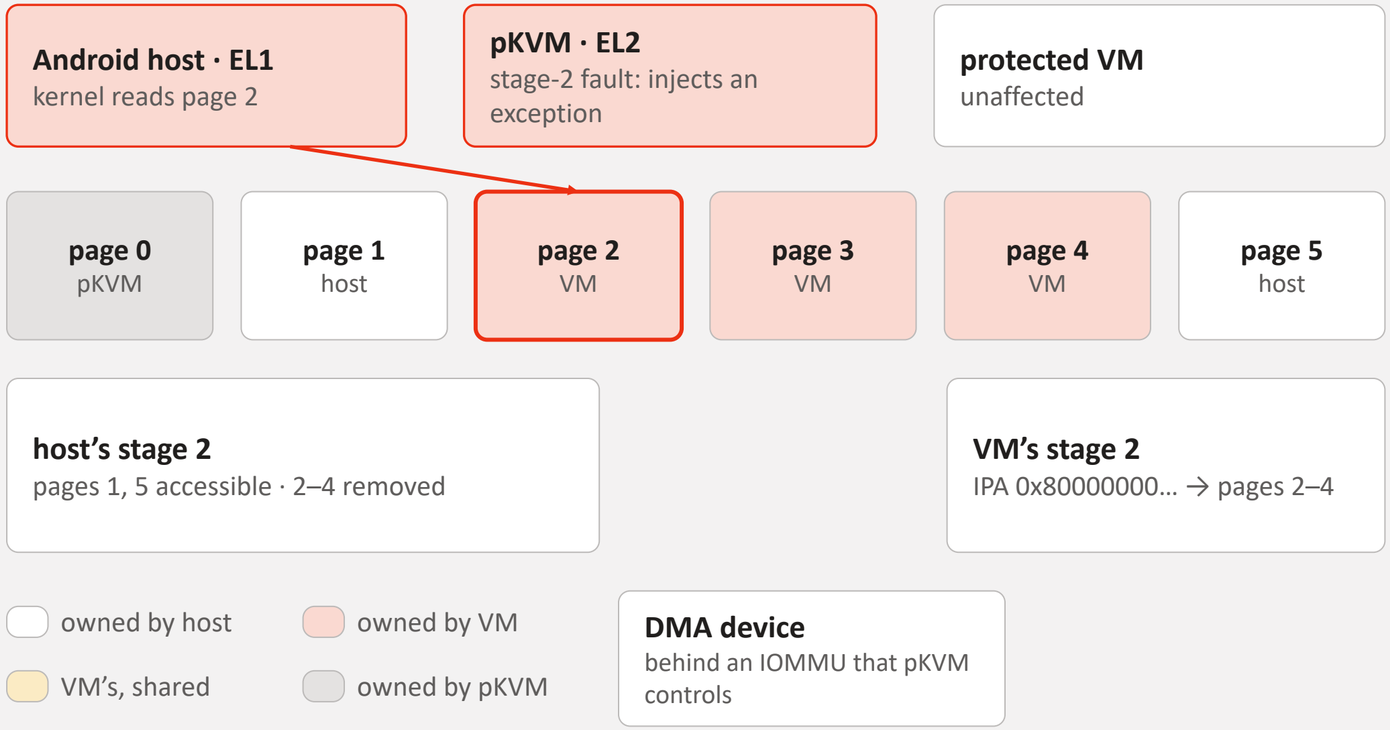
The host donates pages to a new VM

crosvm allocates guest RAM with mmap and registers it as a memory slot. When the protected VM is created, the host donates pages 2 to 4. pKVM records the VM as their owner, removes them from the host’s stage 2, and maps them into the VM’s. Linux stops swapping or merging them.

STATE

pages 2–4 owned by the VM

A page’s life under pKVM, step by step



STEP 3 OF 6

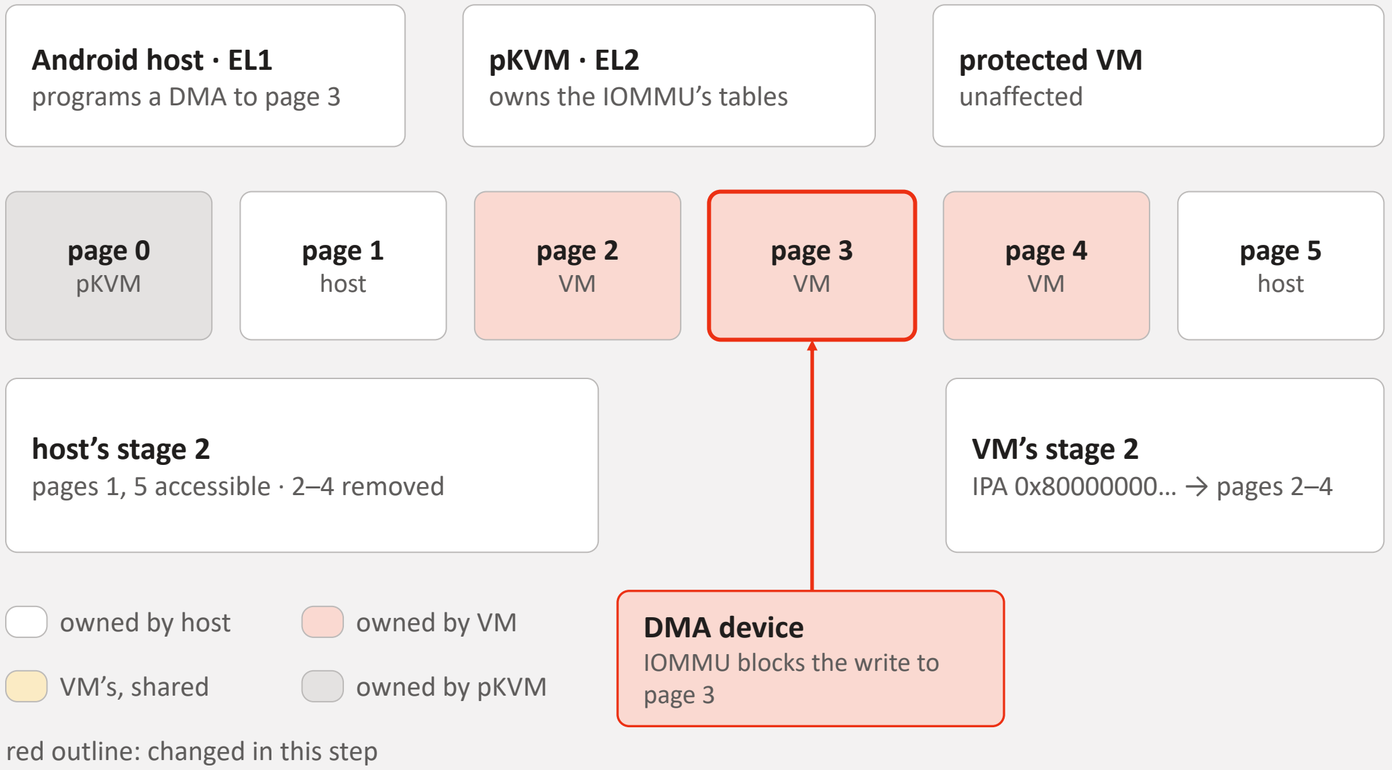
The host reads a donated page

Linux, even in kernel mode, reads page 2. Its own stage-1 tables allow it, but its stage-2 table doesn’t, so the access faults to EL2. pKVM injects an exception back into the host: the task gets SIGSEGV, or the kernel crashes.

STATE

denied by the host’s stage 2

A page’s life under pKVM, step by step



STEP 4 OF 6

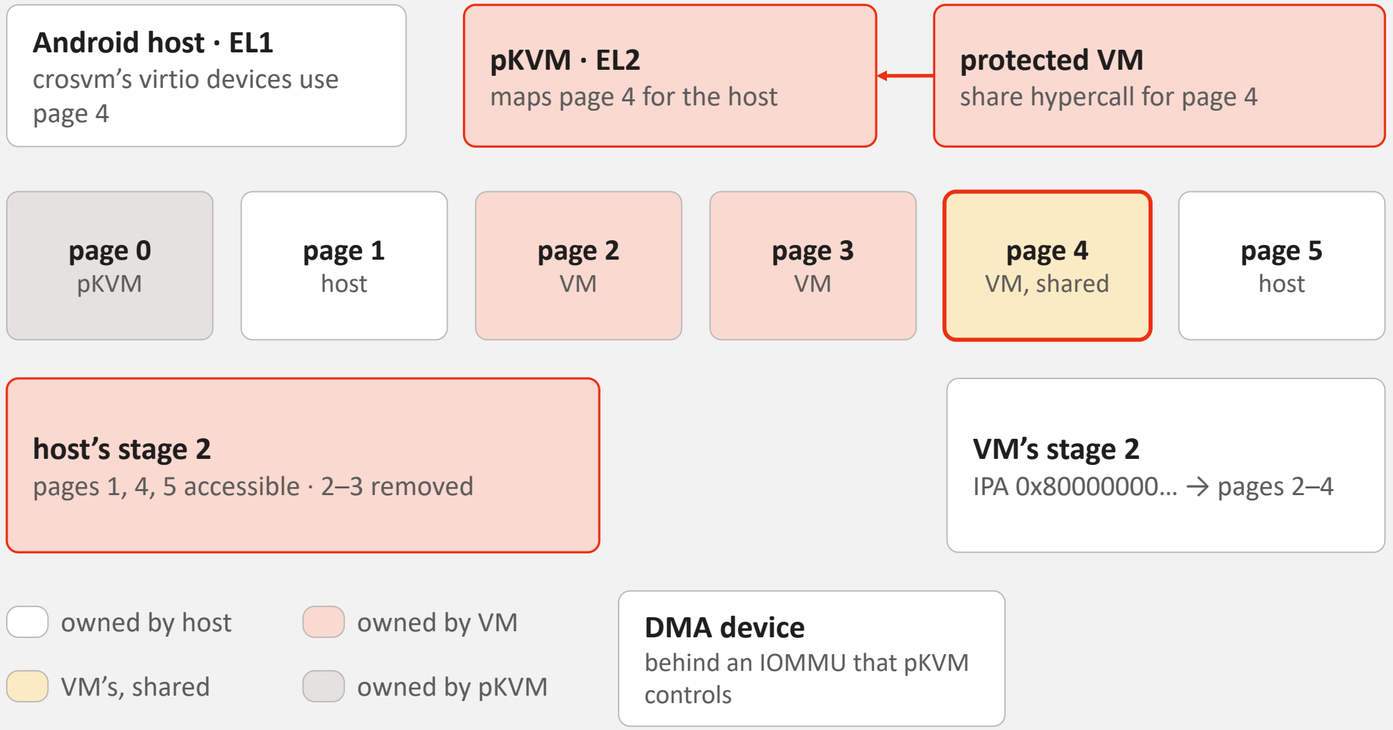
A device tries DMA into the VM

The host programs a DMA-capable device to write page 3. CPU page tables don’t apply to a device, so pKVM requires an IOMMU in front of every such device and controls its tables. The write is blocked.

STATE

denied by the IOMMU

A page’s life under pKVM, step by step



STEP 5 OF 6

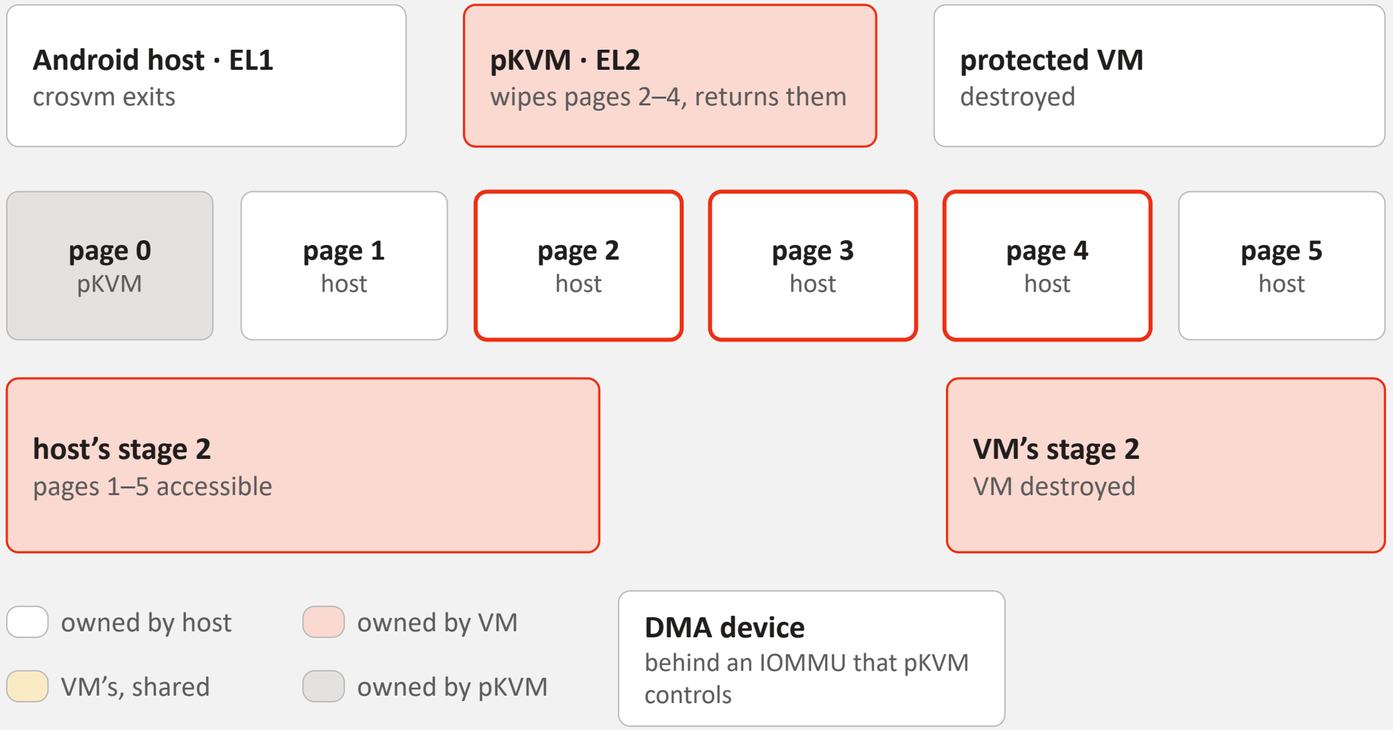
The guest shares a window back

virtio needs memory both sides can touch. The guest shares page 4 with a hypercall, and pKVM maps it into the host’s stage 2 again. The guest keeps its virtqueues and buffers in that window and copies data through it, so nothing else is exposed.

STATE

page 4 owned by the VM, shared with the host

A page’s life under pKVM, step by step



STEP 6 OF 6

The VM stops

When the VM is destroyed, pKVM wipes pages 2 to 4 and returns them to the host, which can map them again. A long-lived VM can also hand back single pages earlier with a relinquish hypercall, driven by the virtio balloon.

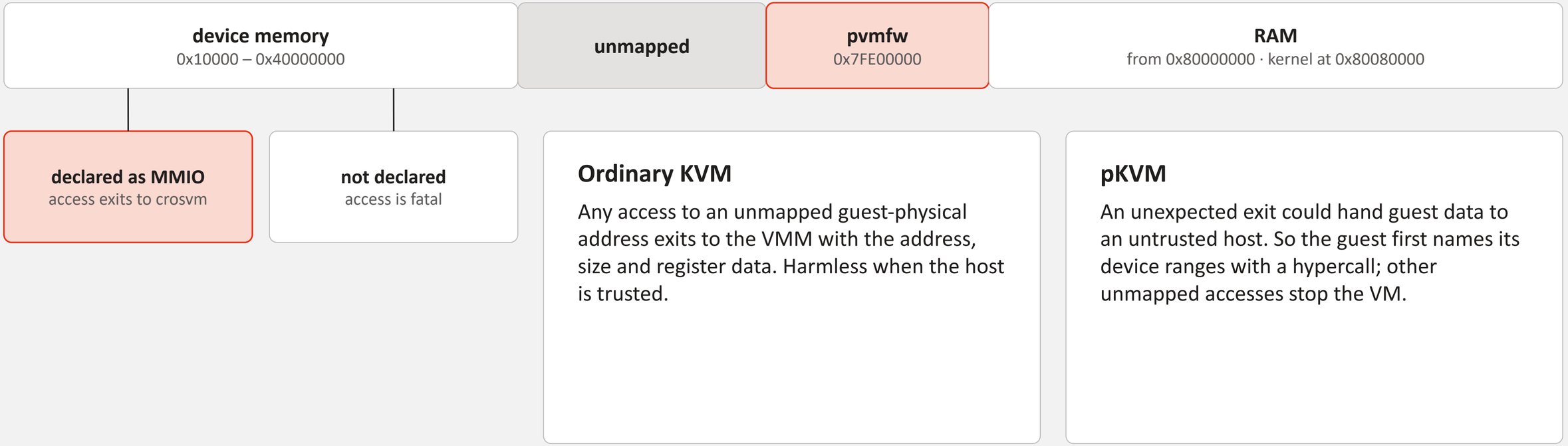
STATE

pages wiped, owned by the host again

MMIO guard: exits that could leak

An exit tells the host the address, the register and maybe the data. pKVM allows that only for ranges the guest declared as devices.

THE PROTECTED VM'S GUEST-PHYSICAL MEMORY, AS CROSVM LAYS IT OUT (NOT TO SCALE)



Ordinary KVM

Any access to an unmapped guest-physical address exits to the VMM with the address, size and register data. Harmless when the host is trusted.

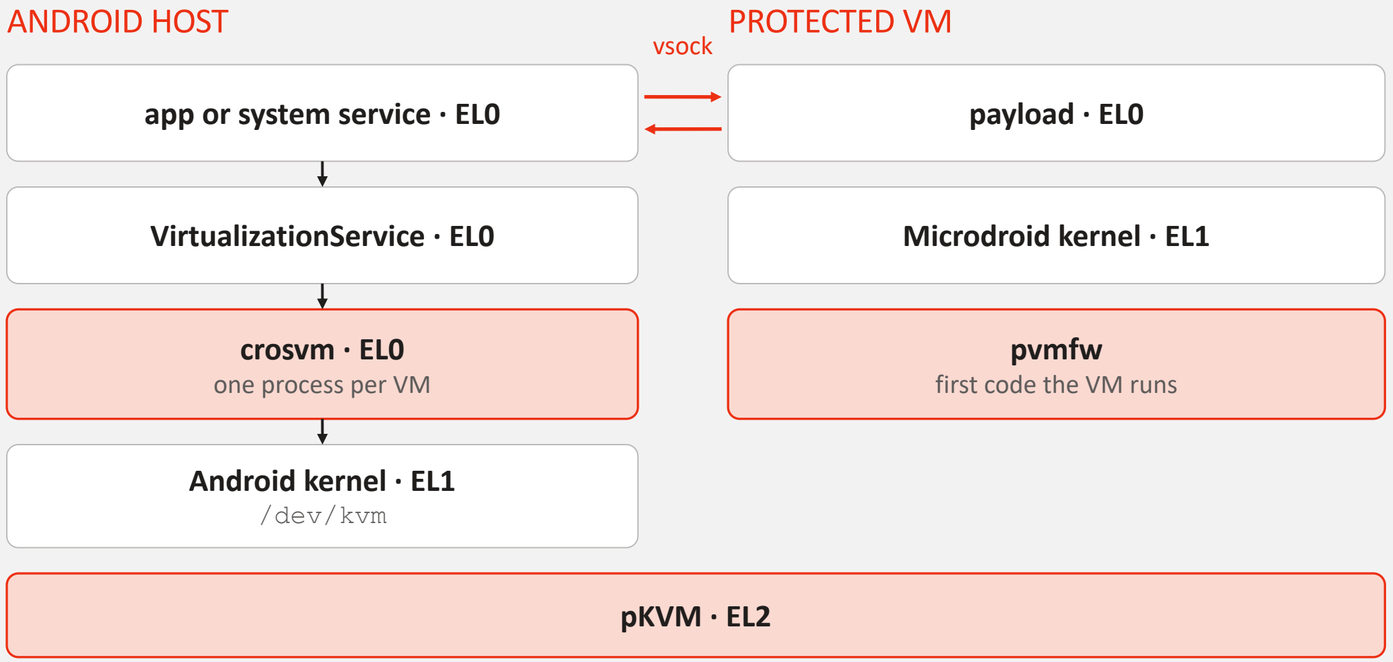
pKVM

An unexpected exit could hand guest data to an untrusted host. So the guest first names its device ranges with a hypercall; other unmapped accesses stop the VM.

- AOSP, AVF architecture: "pKVM hypercalls" (MMIO guard) and "Memory allocation" (crosvm's memory layout).

Android Virtualization Framework: who runs what

Android drives pKVM with the same pieces as any KVM host: a VMM process, /dev/kvm and a run loop.



CROSVM

A Rust VMM from ChromeOS. Each crosvm process runs one VM, and each vCPU is a POSIX thread calling `KVM_RUN`: the loop from section 01.

ITS DEVICES

virtio-blk, vhost-vsock, virtio-pci, a pl030 clock and a 16550a UART, inside a sandbox that protects the host kernel.

PVMFW

Checks the payload’s signature, derives the VM’s secret, wipes its own copy and jumps to the payload. Any failure resets the VM.

- AOSP, AVF architecture: “Virtual machine monitor” and “pVM firmware”; AOSP, Microdroid.

Every channel between a protected VM and Android

vsock is not the only channel. Each channel crosses an untrusted host, so each has its own reason to be safe.

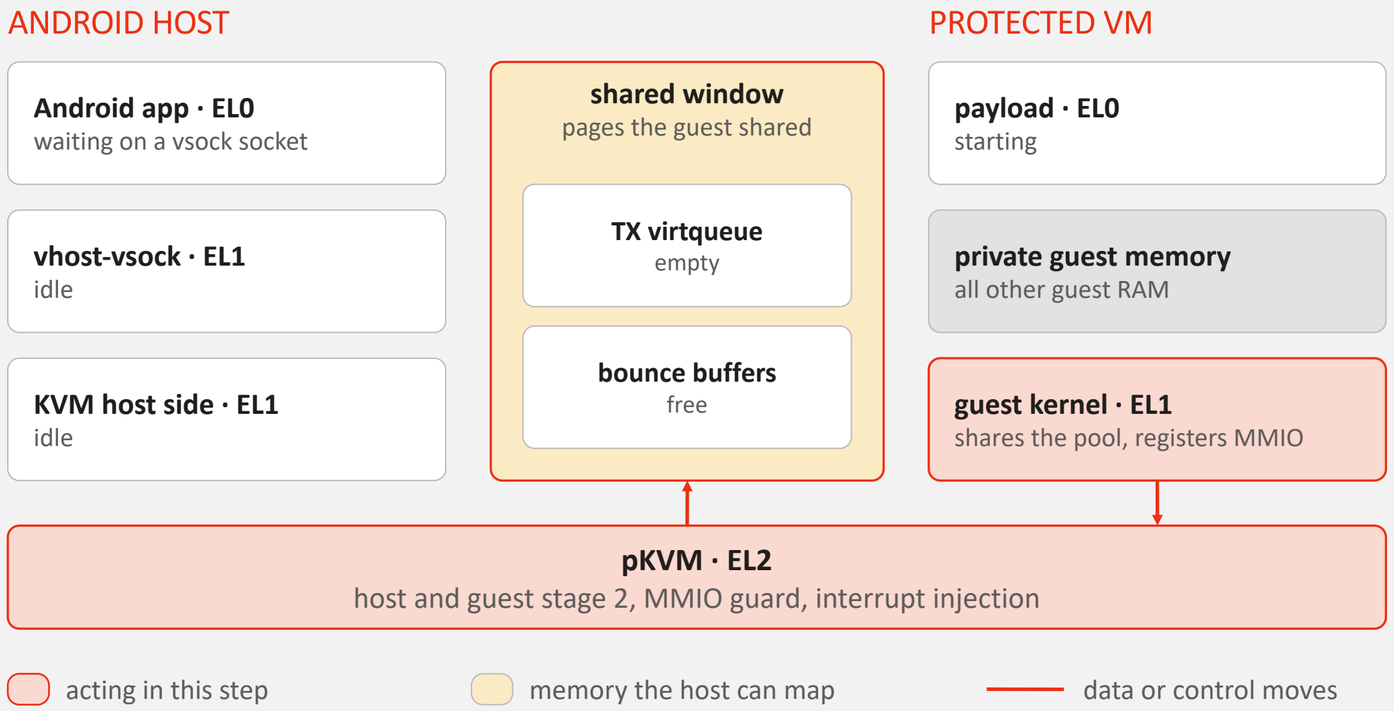
CHANNEL	CARRIES	WHY AN UNTRUSTED HOST CAN'T ABUSE IT
virtio-blk: OS and payload disks	Microdroid's partitions, the app's APK, APEXes	read-only, and checked by dm-verity against hashes from the signed boot chain
virtio-blk: instance disk	per-VM boot identity, keys, rollback counters	encrypted with a secret only the VM has; Android just stores it
vhost-vsock	Binder RPC to Android apps; file exchange (AuthFS)	the VM treats every message as untrusted input; AuthFS checks file contents against hashes
16550a UART	console and log output	output only; the VM chooses what to print
hypercalls to pKVM	page sharing, MMIO guard, random numbers	they go to pKVM at EL2, never to Android

ALL VIRTIO CHANNELS SHARE ONE RULE

Virtqueues and buffers live in a window of pages the guest shared on purpose; the next six slides follow one message through it.

- AOSP: AVF architecture ("Virtual machine monitor", "Virtual I/O device", "Guest interfaces"); Microdroid ("Boot sequence", "Binder RPC", "File exchange").

One vsock message from a protected VM, step by step



STEP 1 OF 6

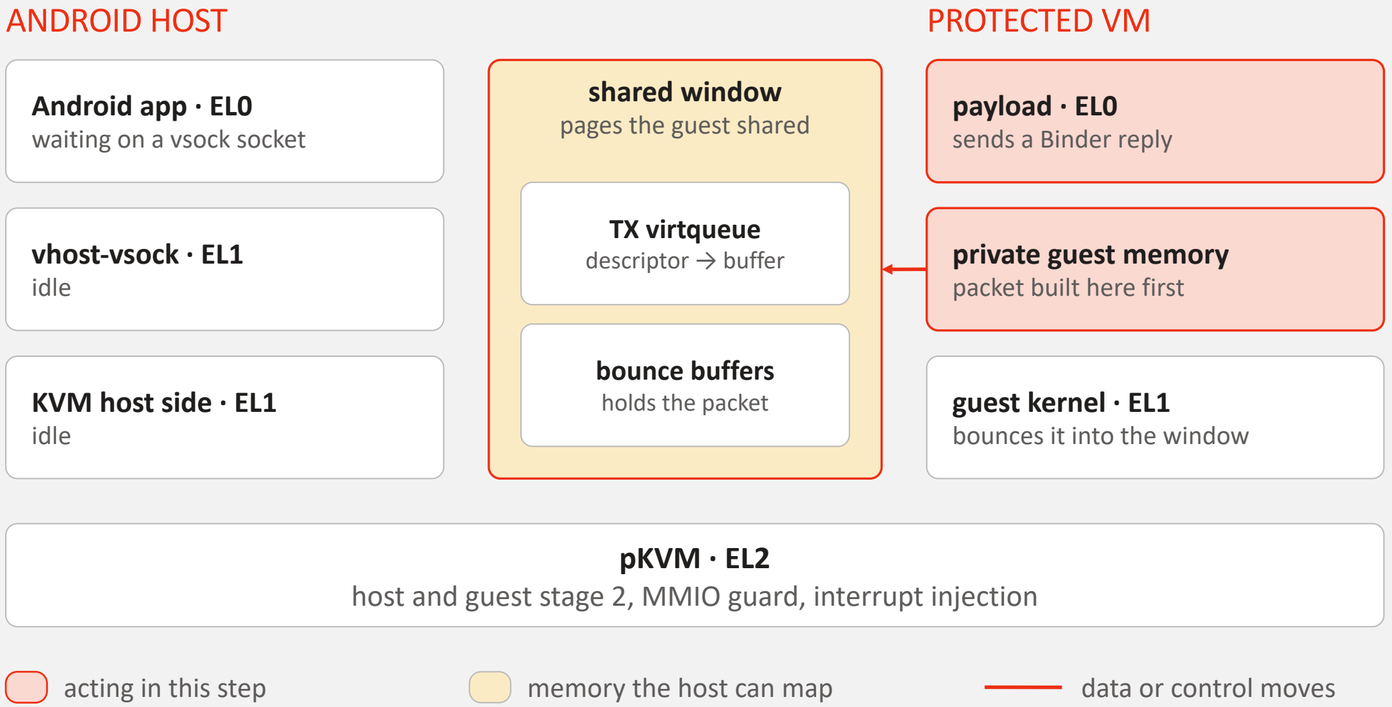
At boot: share a window

The guest kernel reserves a small pool of its memory for all virtio traffic and shares those pages with a hypercall; pKVM maps them into the host's stage 2. It also registers the virtio-pci register ranges with the MMIO guard.

STATE

host can see the window and nothing else

One vsock message from a protected VM, step by step



STEP 2 OF 6

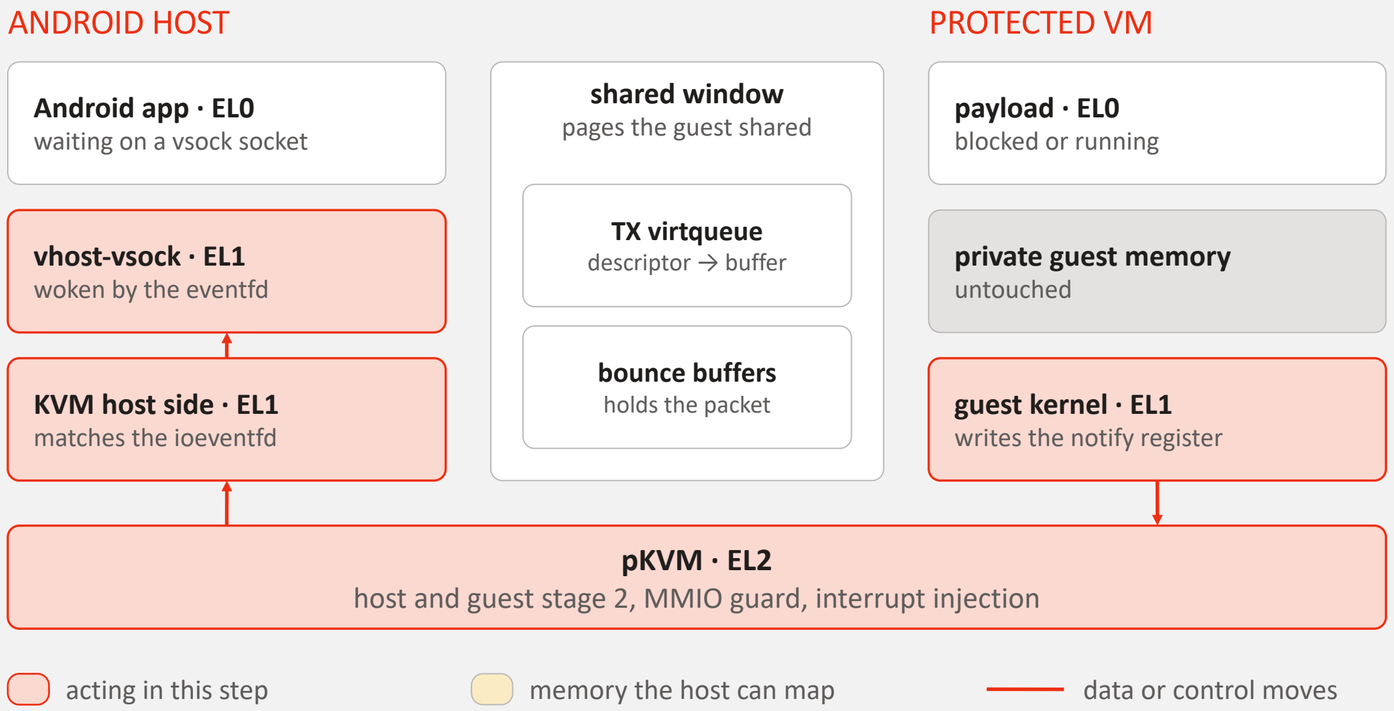
The payload sends a reply

The payload writes a Binder RPC reply to its vsock socket. The guest’s vsock driver builds the packet in private memory, then copies it into a bounce buffer in the shared window and puts a descriptor for it on the transmit virtqueue, also in the window.

STATE

one copy: private memory → shared window

One vsock message from a protected VM, step by step



STEP 3 OF 6

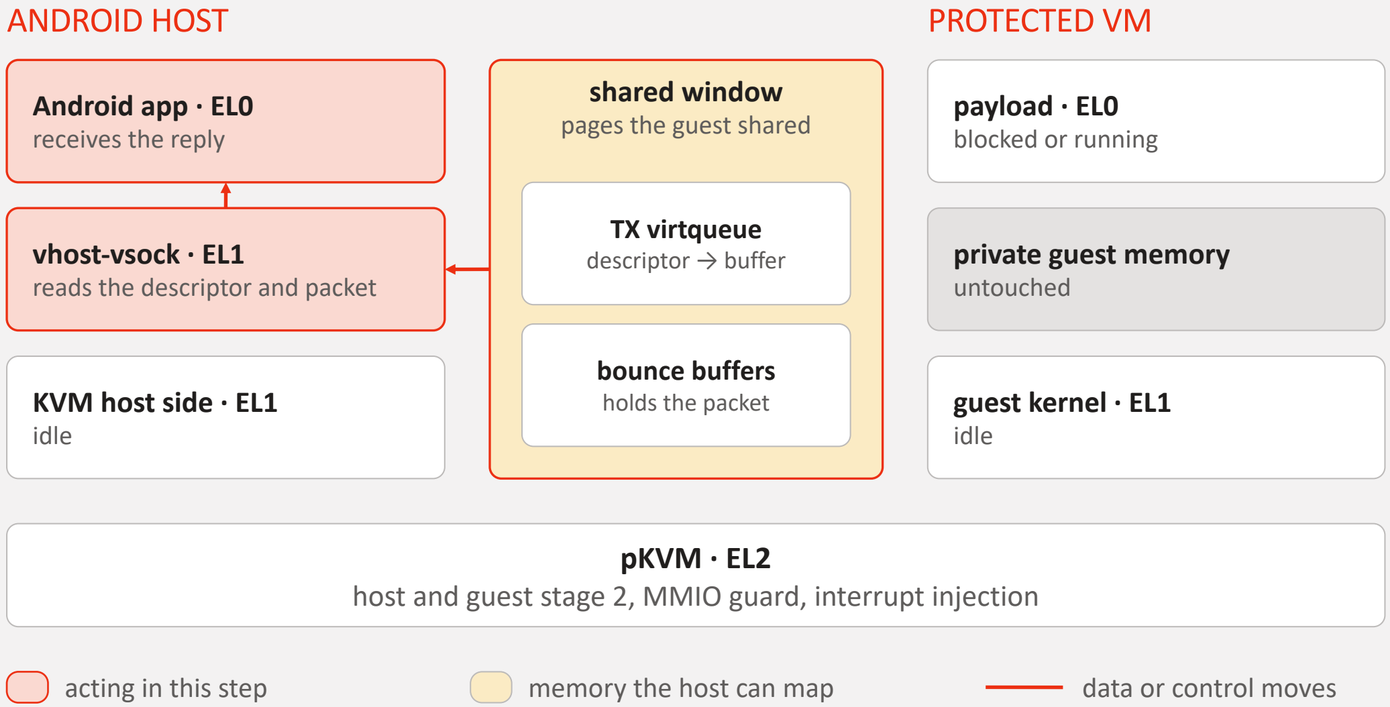
One notification, through EL2

The driver writes the queue’s notify register. The address was registered, so the exit is allowed: pKVM relays it to KVM’s host side at EL1 with the address, size and value. KVM matches an ioeventfd and wakes the vhost-vsock worker in the host kernel.

STATE

exit relayed by pKVM · eventfd signaled

One vsock message from a protected VM, step by step



STEP 4 OF 6

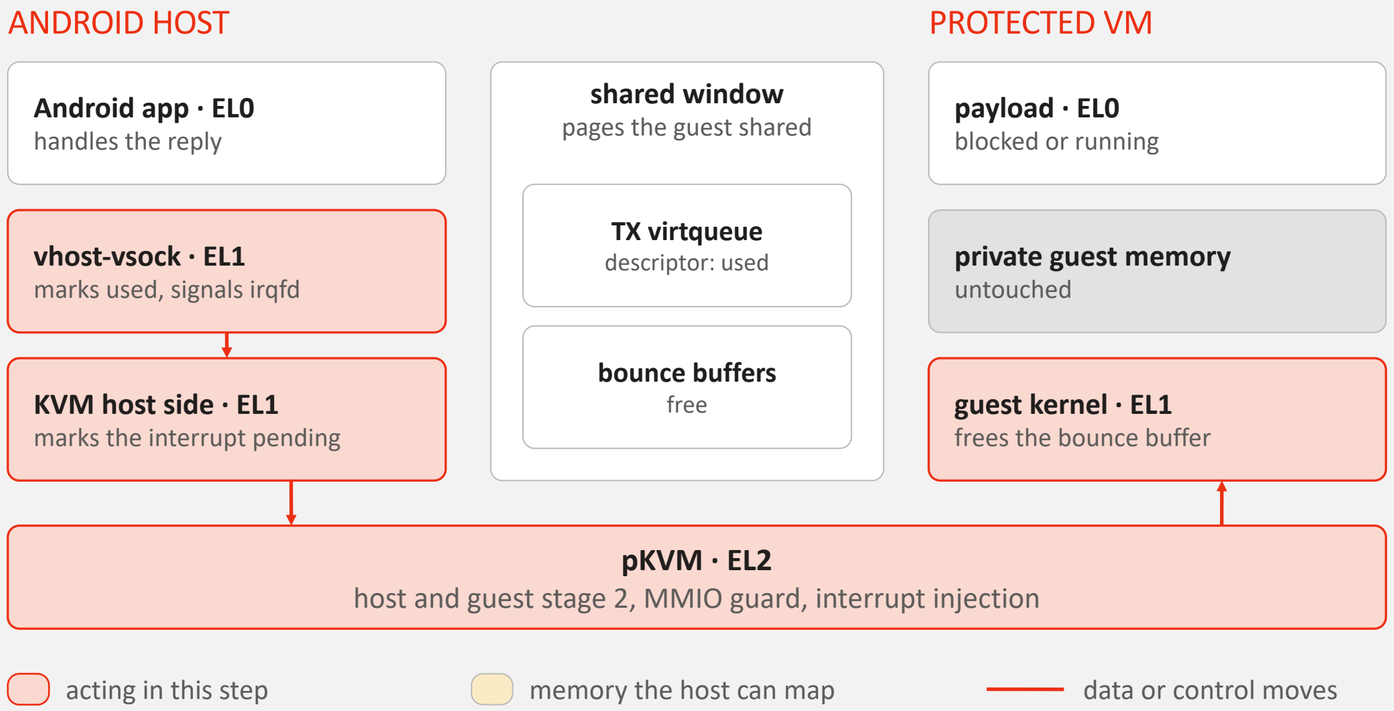
The host reads the window

vhost-vsock follows the descriptor into the shared window, the only guest memory it can map, reads the packet and delivers it to the Android app’s vsock socket. The app’s Binder RPC session sees the reply.

STATE

packet delivered to the app

One vsock message from a protected VM, step by step



STEP 5 OF 6

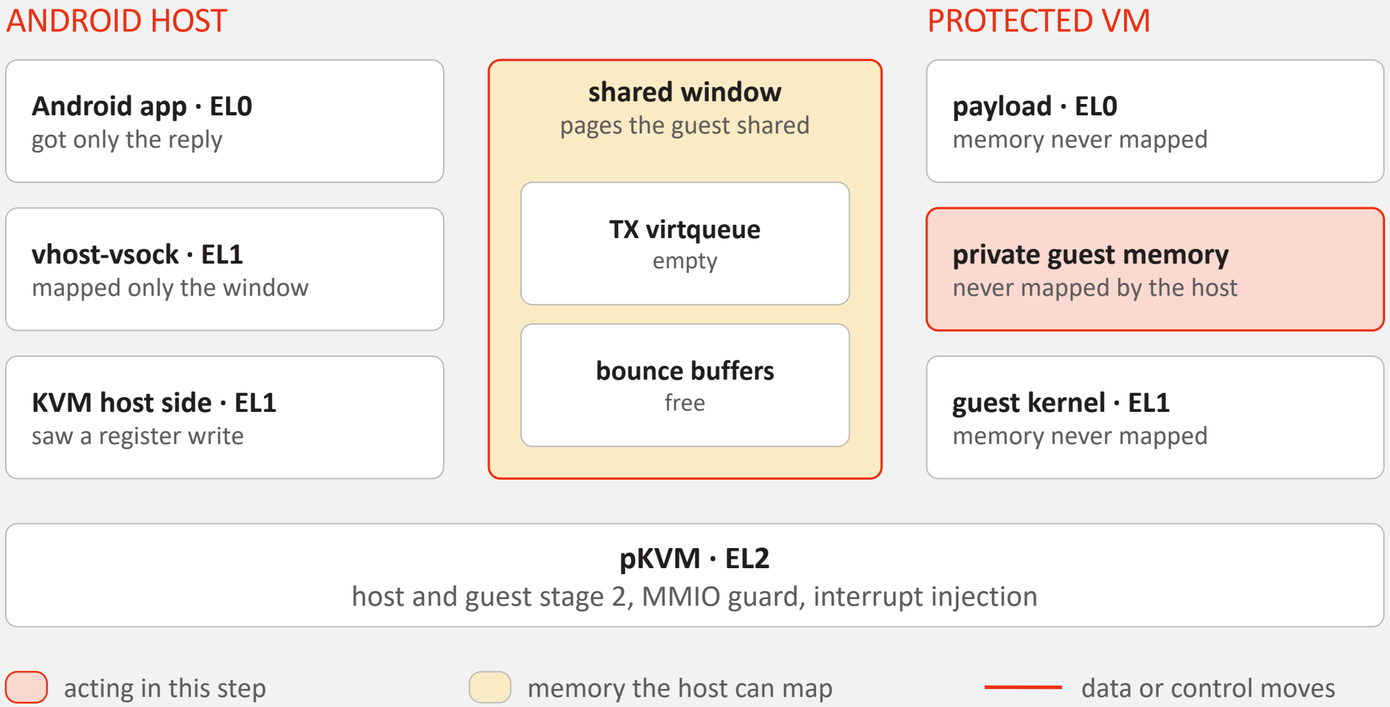
Completion and interrupt

vhost-vsock marks the buffer used in the window and signals an irqfd. KVM's host side, which emulates the interrupt controller as untrusted code, writes the pending interrupt into state it shares with EL2; pKVM loads it on the next entry, and the guest frees the bounce buffer.

STATE

buffer returned · one interrupt

One vsock message from a protected VM, step by step



STEP 6 OF 6

What the host never saw

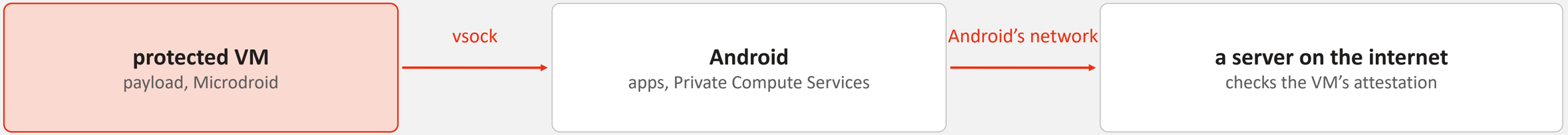
The host saw one packet that the guest chose to copy out, plus a notify write. It never mapped the payload's memory or the guest kernel. Compare ordinary KVM, where the VMM and any vhost-user back end map all of guest RAM.

STATE

visible: the window · hidden: everything else

Is a protected VM cut off from the world?

No, but every path out goes through Android, which it doesn’t trust. In AOSP’s reference setup it has no network device of its own.



end-to-end: data encrypted to a key only the VM holds

Android can still

- refuse to run the VM, or starve its vCPUs
- delay, drop or read anything the VM sends
- serve it disk images, as an untrusted file server

Android can’t

- read or change the VM’s memory
- alter its disk images without dm-verity noticing
- read its instance data, or learn the keys pvmfw derived

- AOSP: AVF architecture (crosvm devices); Microdroid; AVF use cases (“Content Safety On-Device”).

04

macOS: two frameworks

Two frameworks, two levels of control

Apple ships a low-level API much like /dev/kvm, and a high-level one that assembles whole VMs from virtio devices.

01

Hypervisor.framework

OS X 10.10, 2014. Create a VM, map memory, run vCPUs and handle exits: the VMM is entirely your code.

- one VM per process; each vCPU is a POSIX thread
- `hv_vm_map` maps the process's memory into guest-physical space
- needs the `com.apple.security.hypervisor` entitlement
- QEMU's HVF accelerator is built on it

02

Virtualization.framework

macOS 11, 2020. Describe a VM as a configuration object; Apple supplies the boot loaders and the virtio devices.

- boots a Linux kernel directly, or through EFI from macOS 13
- macOS guests on Apple silicon from macOS 12
- virtio block, network, console, file sharing, sockets, balloon, graphics
- used by UTM and by Apple's Containerization

- Apple Developer documentation: Hypervisor; Virtualization.

Hypervisor.framework on Apple silicon

```
hv_vm_create(NULL);                 // one VM for this process

void *ram = mmap(NULL, RAM_SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_PRIVATE, -1, 0);
hv_vm_map(ram, 0x80000000, RAM_SIZE,
          HV_MEMORY_READ | HV_MEMORY_WRITE | HV_MEMORY_EXEC);

hv_vcpu_t vcpu; hv_vcpu_exit_t *exit;
hv_vcpu_create(&vcpu, &exit, NULL);   // bound to this thread
hv_vcpu_set_reg(vcpu, HV_REG_PC, 0x80000000);

for (;;) {
  hv_vcpu_run(vcpu);
  if (exit->reason == HV_EXIT_REASON_EXCEPTION) {
    uint64_t esr = exit->exception.syndrome;
    /* EC 0x24: data abort -> emulate MMIO  */
    /* EC 0x16: HVC        -> hypercall     */
  }
}
```

The same shape as KVM, with Arm’s way of reporting device accesses.

1. Guest memory is the process’s own, and a vCPU belongs to the thread that created it.
2. Arm has no I/O ports. A device register is an unmapped guest-physical address, so touching it causes a data abort, which exits.
3. The VMM reads the syndrome register (ESR) for the access size, direction and register, emulates the device, and steps PC past the instruction.
4. On a timer exit the framework masks the virtual timer until the VMM unmasks it.

- Written for this lecture from Apple’s documented API (Apple Developer, Hypervisor: Apple silicon; hv_vcpu_create; hv_vcpu_run); error checks omitted.

Virtualization.framework: a Linux VM in Swift

```
let boot = VZLinuxBootLoader(kernelURL: kernelURL)
boot.initialRamdiskURL = initrdURL
boot.commandLine = "root=/dev/vda"

let config = VZVirtualMachineConfiguration()
config.bootLoader = boot
config.cpuCount = 4
config.memorySize = 4 << 30
config.storageDevices = [VZVirtioBlockDeviceConfiguration(
    attachment: try VZDiskImageStorageDeviceAttachment(
        url: diskURL, readOnly: false))]
let net = VZVirtioNetworkDeviceConfiguration()
net.attachment = VZNATNetworkDeviceAttachment()
config.networkDevices = [net]
try config.validate()

let vm = VZVirtualMachine(configuration: config)
vm.start { result in print(result) }
```

Describe the machine; the framework builds and runs it.

1. No run loop and no exit handling in the app: the framework owns both.
2. Every device is virtio, so a stock Linux kernel finds its disk at `/dev/vda` with drivers it already has.
3. `validate()` checks the whole configuration, such as CPU and memory limits, before anything runs.
4. Each VM runs in its own system service process, so a fault in device emulation can’t crash the app.

- Written for this lecture from Apple’s documented API (Apple Developer, Virtualization); Apple’s sample code project “Running Linux in a Virtual Machine” is a complete version.

What Apple added, release by release

Each macOS release since 2020 has extended the same framework rather than adding a new one.

MACOS 11 · 2020

the Virtualization framework, and the Hypervisor framework on Apple silicon

MACOS 12 · 2021

macOS guests on Apple silicon, installed from an IPSW restore image

MACOS 13 · 2022

Rosetta inside Linux VMs runs x86-64 Linux binaries; EFI boot

MACOS 15 · 2024

nested virtualization, on M3 and later chips

MACOS 26 · 2025

Containerization: one lightweight VM per Linux container

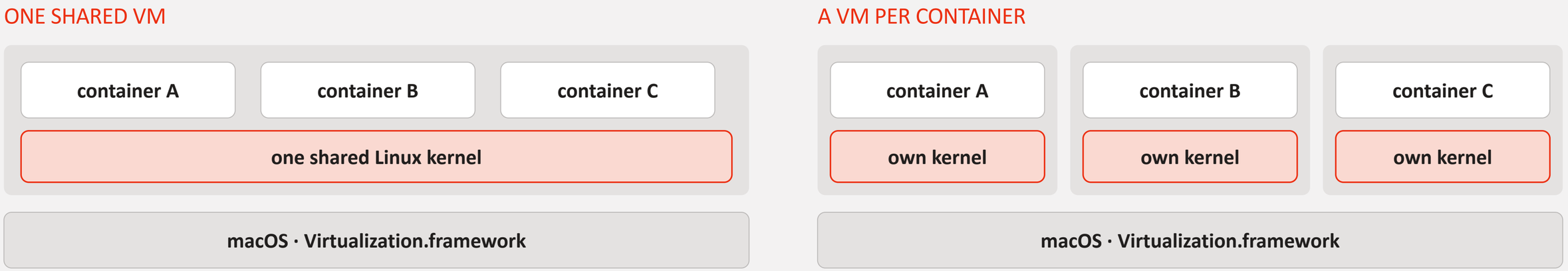
A LICENSE TERM ENFORCED IN CODE

The macOS license (section 2B(iii)) allows two additional macOS instances in VMs per Mac. The framework refuses to start a third running macOS guest with VZErrorDomain code 6. Linux guests have no such limit.

- Apple Developer documentation (Virtualization, isNestedVirtualizationSupported); macOS software license agreement 2B(iii); Apple DTS, developer forums.

Containers as VMs: Apple’s Containerization

Docker Desktop on a Mac runs every container in one shared Linux VM. Apple’s package, announced at WWDC 2025, starts a VM per container.



Isolation

Containers share no kernel, so a kernel exploit in one reaches only its own VM.

Start time

A minimal kernel and a small init, vminitd, give sub-second starts by Apple’s account.

Networking

Each container gets its own IP address, so no port forwarding is needed.

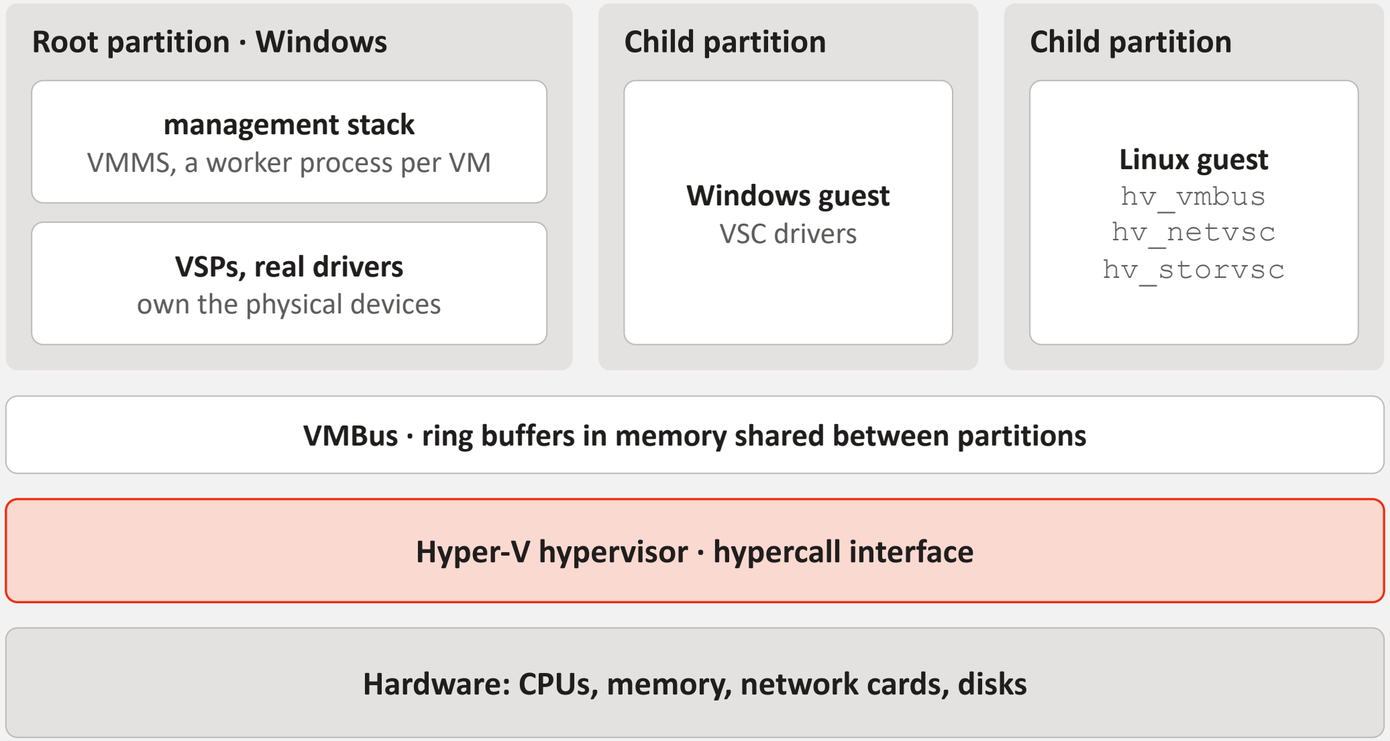
- Apple, github.com/apple/containerization (README); requires Apple silicon and macOS 26.

05

Windows: Hyper-V

With Hyper-V on, Windows is a guest

The hypervisor boots first. The Windows you log in to runs in the root partition, much like Xen’s Domain 0.



HYPERVISOR

Schedules virtual processors, manages second-level address translation and routes interrupts. It has no device drivers.

ROOT PARTITION

Owns the real devices and creates child partitions through hypercalls. Runs the management stack.

CHILD PARTITION

Reaches devices through VMBus: synthetic devices when its kernel is enlightened, emulated ones otherwise.

- Microsoft Learn, Hyper-V architecture (Windows Server documentation).

Xen’s vocabulary, Microsoft’s names

Hyper-V reached the same design as Xen. Most lecture 11 concepts have a direct counterpart.

XEN (LECTURE 11)	HYPER-V	NOTES
Domain 0	root partition	owns devices; creates the other partitions
Domain U	child partition	no direct hardware access
hypercall page	hypercall page	the guest writes an MSR and the hypervisor fills the page in
paravirtualized kernel	enlightened kernel	calls the hypervisor where that beats trapping, such as TLB flushes
event channel	synthetic interrupt (SynIC)	an extension of the virtual interrupt controller
front end, back end	VSC, VSP	one pair per synthetic device class
I/O ring	VMBus channel	a pair of ring buffers in shared guest memory
XenStore	VMBus channel offers	the root offers channels; the guest opens them

- Microsoft Learn: Hyper-V architecture; Hypervisor Top Level Functional Specification (TLFS).

The user-mode virtualization stack

Hyper-V emulates each VM’s devices in that VM’s own process in the root, and opens the same run loop to other VMMs.

01 **VMMS: Virtual Machine Management Service**
one Windows service tracks the state of every VM

02 **VMWP: one worker process per VM**
emulated devices and management for that VM only

03 **VID: Virtualization Infrastructure Driver**
partition, virtual processor and memory management in the kernel

04 **VSPs**
kernel drivers serving synthetic disks, networks and video over VMBus

WINDOWS HYPERVISOR PLATFORM

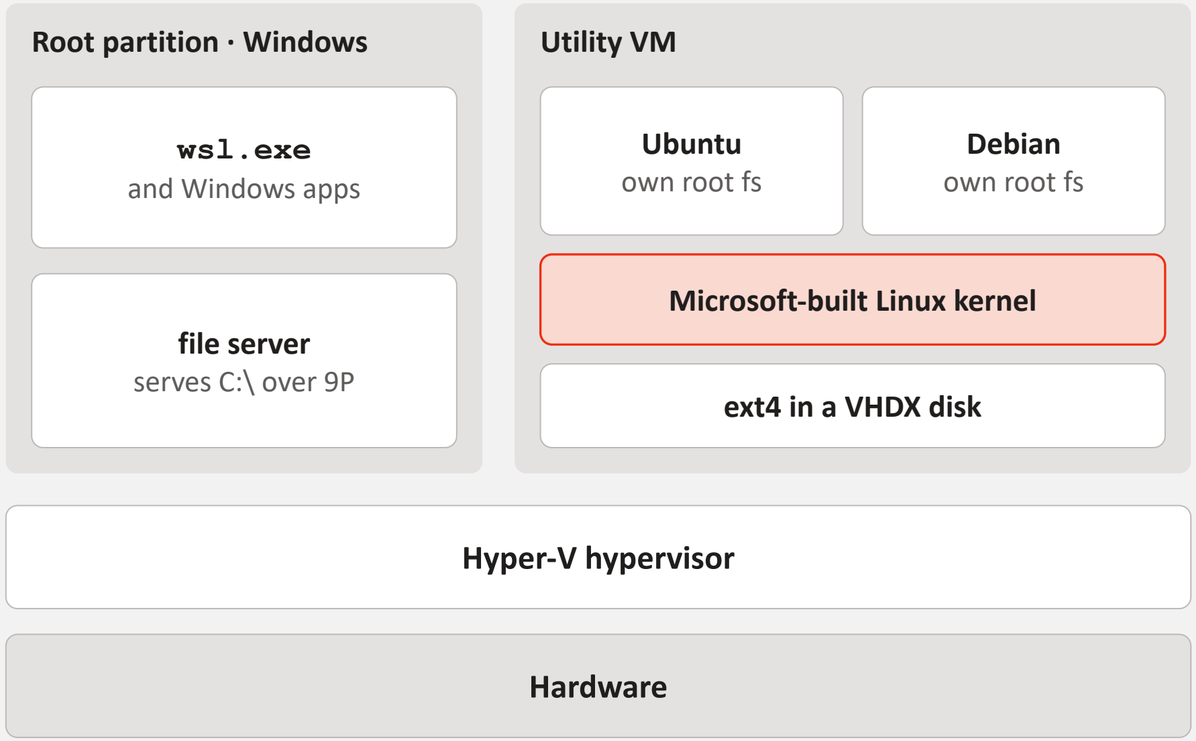
Since Windows 10 version 1803, `WinHvPlatform.dll` lets any process create a partition and run it with `WHvRunVirtualProcessor`: the loop from section 01.

QEMU’s WHPX accelerator and VirtualBox use it so they can run while Hyper-V is on.

- Microsoft Learn: Hyper-V architecture; Windows Hypervisor Platform; WHvRunVirtualProcessor.

WSL 2: a Linux kernel in a utility VM

WSL 1 translated Linux system calls into Windows ones. WSL 2, announced in May 2019, runs a real Linux kernel instead.



FULL SYSTEM CALL COMPATIBILITY

It is Linux, so programs WSL 1 couldn’t run, such as Docker, now work.

ONE VM, MANY DISTRIBUTIONS

Each distribution runs as a container inside one shared utility VM, with its own root file system.

FILES ACROSS THE BOUNDARY

Linux sees Windows drives under `/mnt/c` through a 9P file server; Windows reaches Linux files at `\\wsl$`.

SERVICED BY WINDOWS UPDATE

Microsoft builds the kernel from kernel.org stable sources, starting with 4.19, and ships the updates.

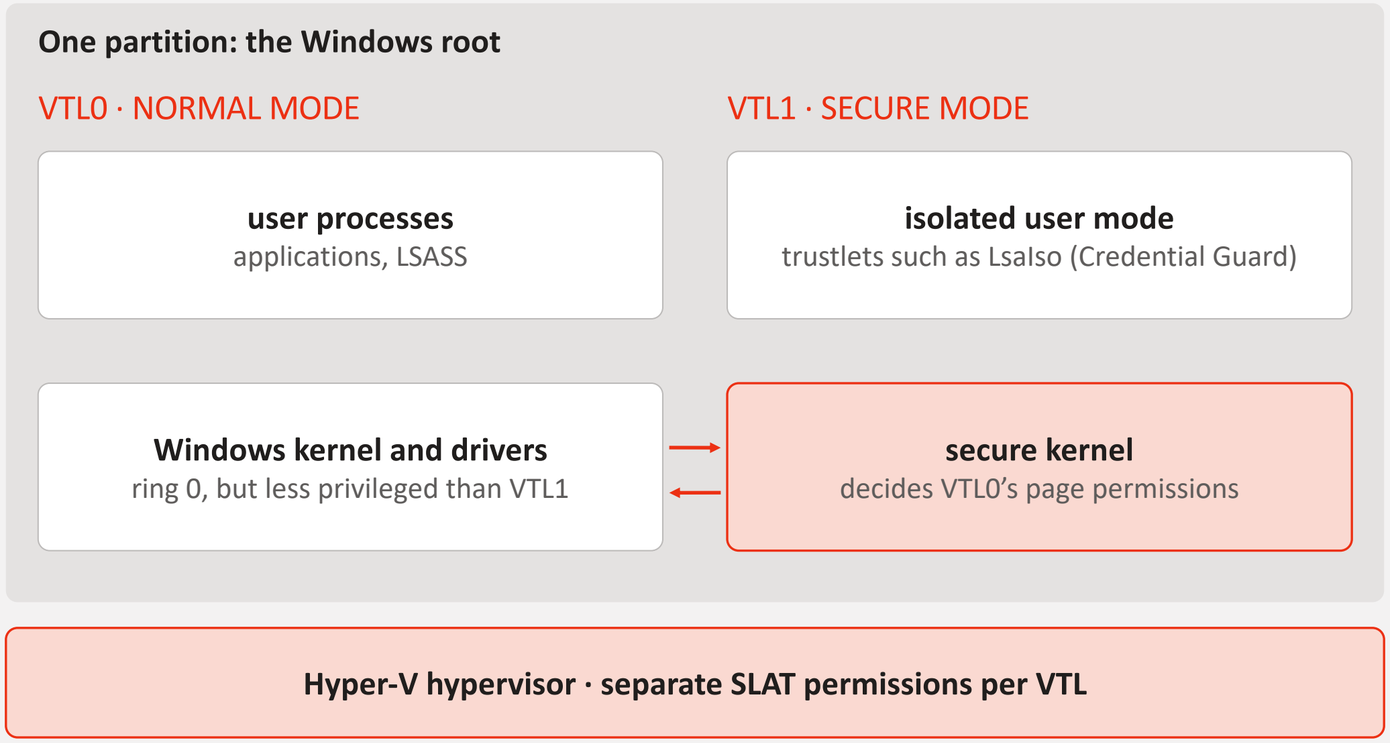
- Microsoft, Announcing WSL 2, Windows Command Line blog, May 6, 2019; Microsoft Learn, Comparing WSL versions.

06

Windows: virtualization-based security

Virtual Trust Levels: two worlds in one partition

Virtual Secure Mode lets one partition hold a second, more privileged world that its normal kernel can’t touch.



- Microsoft Learn, Hypervisor Top Level Functional Specification: Virtual Secure Mode.

ORDERED BY PRIVILEGE

Higher VTLs are more privileged. The specification allows up to 16; Windows’ VBS uses VTL0 and VTL1.

PER-VTL PAGE PERMISSIONS

A higher VTL sets read, write, kernel-execute and user-execute rights on a lower VTL’s pages, enforced in the SLAT tables.

SEPARATE CPU STATE

Each VTL has private registers and interrupt state, which a lower VTL can’t read.

Memory integrity: loading a driver

With memory integrity (HVCI) on, the Windows kernel can no longer decide alone which pages may run as kernel code.

1. The VTL0 kernel loads a driver image into memory, as it always has.
2. Before the code may run, the kernel asks the VTL1 secure kernel to check the image’s signature.
3. If the signature is valid, VTL1 has the hypervisor mark the code pages kernel-executable and read-only in VTL0’s SLAT entries.
4. Data pages stay writable but not executable. No page is both.
5. An attacker in the VTL0 kernel who writes to code, or jumps into data, now faults, even though the attacker controls ring 0.

WHY THE HARDWARE MATTERS

SLAT needs separate kernel-execute and user-execute bits, which Intel calls mode-based execute control. Microsoft notes a larger performance cost on older processors.

WHAT THE ATTACKER NEEDS NOW

A bug in the secure kernel or the hypervisor, not just in a driver.

- Microsoft Learn: Virtual Secure Mode (TLFS); Memory integrity and VBS enablement.

On by default

Each Windows release has widened where VBS runs, so on most new PCs the hypervisor is always on.

2015 · WINDOWS 10

VBS, Credential Guard and memory integrity introduced; off by default outside S mode

2021 · WINDOWS 11

memory integrity on by default for clean installs on compatible hardware and Secured-core PCs

2022 · WINDOWS 11 22H2

Credential Guard on by default in Enterprise editions

OCTOBER 2026

quality updates begin turning memory integrity, and VBS if needed, on for eligible upgraded PCs

WHAT THIS MEANS

Even with no VM in sight, the hypervisor runs underneath, and its first guest is Windows itself.

- Microsoft Learn, Memory integrity and VBS enablement; Microsoft Windows IT Pro blog (September 2026), as reported by Windows Central and Tom’s Hardware.

07

Side by side

Three systems, compared

Same hardware features, three different places for the pieces.

	LINUX	MACOS	WINDOWS
Hypervisor	KVM, in the host kernel	in the XNU kernel	Hyper-V, below Windows
VMM API	`/dev/kvm` ioctls	Hypervisor.framework	Windows Hypervisor Platform
High-level API	libvirt, QEMU	Virtualization.framework	Hyper-V Manager, PowerShell
Paravirtual I/O	virtio	virtio	VMBus synthetic devices
Running Linux tools	native	Linux VMs, Containerization	WSL 2
Protects the host OS	pKVM, on Android	not with the hypervisor	VBS and memory integrity
In the cloud	AWS Nitro, Google Cloud	none: EC2 Mac rents whole Macs	Azure

- Sources as on the preceding slides; Google Cloud: “7 ways we harden our KVM hypervisor at Google Cloud”, 2017.

The 2003 mechanisms are still here

Hardware assist removed the need to port the kernel, but not the need to share memory cheaply.

LECTURE 11	LINUX KVM	MACOS	WINDOWS HYPER-V
hypercall page	Linux patches in VMCALL or VMMCALL at boot	HVC instruction	hypercall page, set up through an MSR
I/O rings	virtqueues	virtqueues	VMBus ring buffers
ballooning	virtio-balloon	virtio balloon device	Dynamic Memory
shadow page tables	EPT or NPT; shadows for nested guests	Arm stage-2 translation	SLAT
paravirtual time	kvm-clock	Arm generic timer, virtualized in hardware	reference TSC page

- Linux arch/x86/include/asm/kvm_para.h; Hyper-V TLFS; Apple Developer, Virtualization (VZVirtioTraditionalMemoryBalloonDevice).

Most escapes start in a device model

The hypervisor core is small. The code that parses guest-controlled device state is large, and that is where escapes usually begin.

VENOM, CVE-2015-3456

A buffer overflow in QEMU’s floppy disk controller emulation.

A guest kernel could reach it even with no floppy drive configured, on Xen, KVM and VirtualBox hosts that used QEMU’s code.

A device nobody had used in years was part of every VM’s attack surface.

01

Shrink the device model

Firecracker has no floppy, no PCI and no legacy devices at all.

02

Sandbox the device model

QEMU under seccomp and SELinux; Firecracker’s jailer; Hyper-V’s per-VM worker process; Apple’s per-VM service.

03

Move it off the host CPU

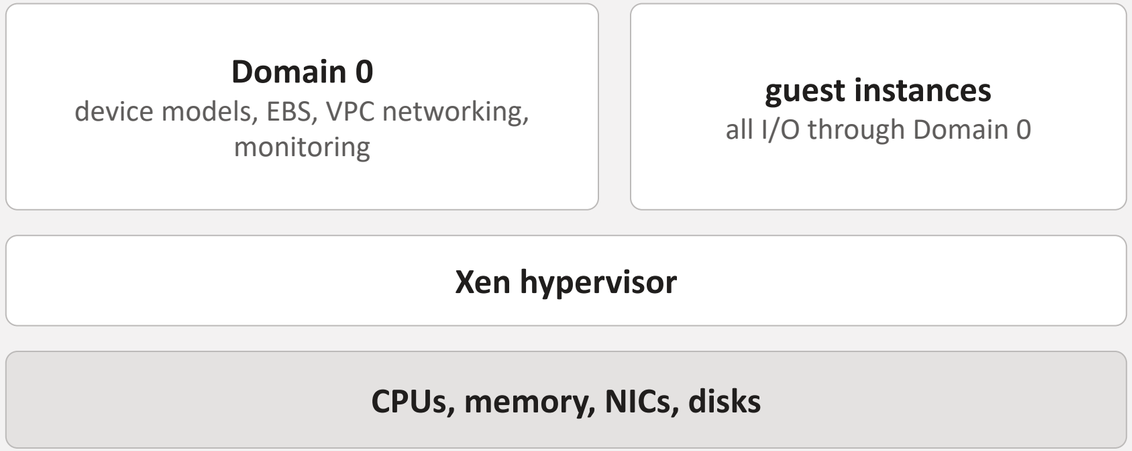
Nitro cards implement network and storage devices in dedicated hardware.

- NVD, CVE-2015-3456; CrowdStrike, VENOM vulnerability advisory, May 2015.

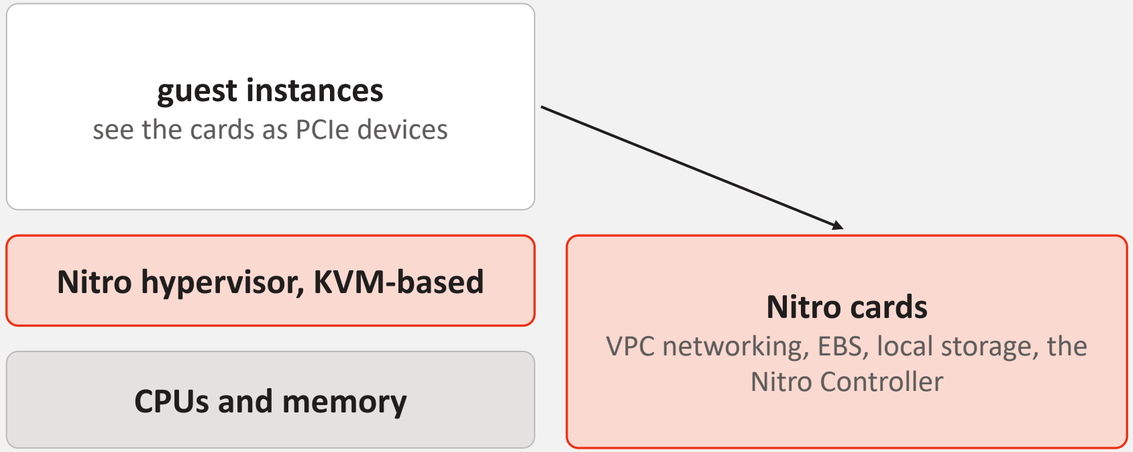
AWS: from Domain 0 to Nitro

EC2 began on Xen. Starting with C5 instances in 2017, AWS replaced Domain 0 with dedicated cards and a minimized KVM-based hypervisor.

XEN, BEFORE NITRO



NITRO, C5 AND LATER



WHAT THE NITRO HYPERVISOR LEAVES OUT

No networking stack, general-purpose file system, peripheral drivers, SSH server or shell. It takes VM commands from the Nitro Controller and partitions CPUs and memory.

- AWS, The Security Design of the AWS Nitro System (whitepaper, 2024): “The Nitro System journey” and “The Nitro Hypervisor”.

Firecracker: a VMM built for one job

AWS Lambda needed VM isolation at close to container cost, so AWS wrote a minimal VMM in Rust on top of KVM.

< 125 ms
boot to application code
with a minimal guest kernel configuration

< 5 MB
memory overhead per microVM
the VMM process, excluding guest RAM

150
microVMs started per second
per host

WHAT IT LEAVES OUT

- no BIOS: it loads a Linux kernel directly
- no PCI and no legacy devices beyond a serial port and a reset controller
- no live migration, and no Windows guests without major changes

WHY SMALLER MATTERS

About 50,000 lines of Rust, against more than 1.4 million lines of C in QEMU. A separate jailer applies seccomp filters, namespaces and cgroups before the guest boots.

A. Agache et al., Firecracker: Lightweight Virtualization for Serverless Applications, NSDI 2020; LWN, The Firecracker virtual machine monitor, January 2019.

08

Takeaways

Four things to remember

01

The hypervisor became an OS feature

Hardware assist made it small enough to ship in Linux, Windows and macOS. The VMM became an ordinary process.

02

One run loop everywhere

`KVM_RUN`, `hv_vcpu_run` and `WHvRunVirtualProcessor` all block until an exit the VMM must handle.

03

Xen’s I/O design won

virtio and VMBus are shared rings with notification suppression, as in 2003.

04

Hypervisors now guard the host

VBS and pKVM use the hypervisor to limit what the host kernel itself can do.

Check your understanding (1 of 2)

01

KVM has no scheduler or memory allocator of its own. What does it use instead, and what does a VM gain from that?

02

Walk through a KVM guest writing one byte to the serial port. Which steps would an ioeventfd remove, and why can’t it serve the UART?

03

Firecracker and Apple’s Containerization both put one workload in one VM. What did each leave out to make that cheap?

Check your understanding (2 of 2)

04

Compare Hyper-V’s root partition, Xen’s Domain 0 and pKVM’s host. Which of them can read a guest’s memory?

05

How does memory integrity stop a ring-0 attacker from running injected code? What must the attacker compromise instead?

06

virtio passes guest-physical addresses where Xen passed grant references. What does that assume, and which system here breaks it?

For discussion

01 On most new PCs, Windows runs as a guest of its own hypervisor. Is Windows still the operating system on that machine?

02 AWS moved device emulation into cards; Apple moved it into a separate process. Which boundary is stronger, and against what?

03 Apple enforces a license term inside the hypervisor. Should a hypervisor enforce policies like that?

04 Linux, macOS and Windows converged on the same run loop and the same rings. Did the hardware force that, or was it a choice?