VMBus (Hyper-V) devices in QEMU/KVM

Roman Kagan <rkagan@virtuozzo.com>

About me

- with Virtuozzo (formerly Parallels, formerly SWSoft) since 2005
- in different roles including
  - large-scale automated testing development for container and hypervisor
  - proprietary Parallels hypervisor development
  - now: opensource QEMU/KVM-based Virtuozzo hypervisor development

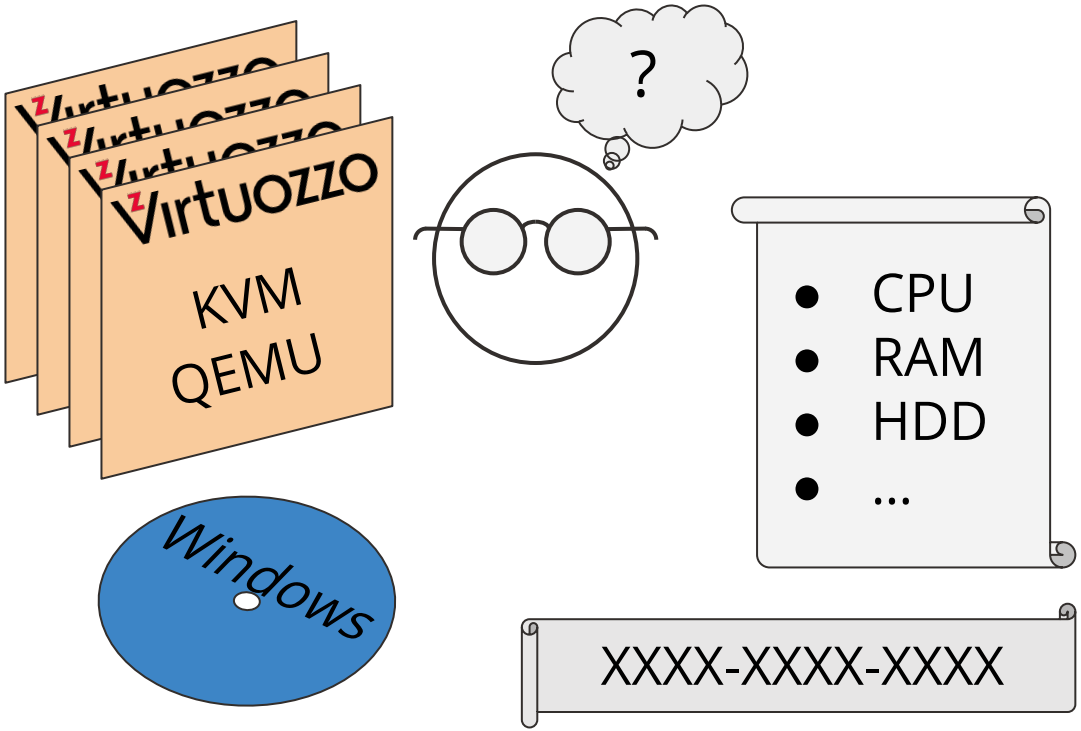
Disclaimers

- all trademarks are the property of their respective owners
- the only authoritative and up-to-date documentation is the code

Outline

1. Motivation
   a. virtual h/w choice for Windows VM
2. Hyper-V / VMBus emulation
   a. layers & components
   b. implementation details
   c. implementation status
3. Summary & outlook

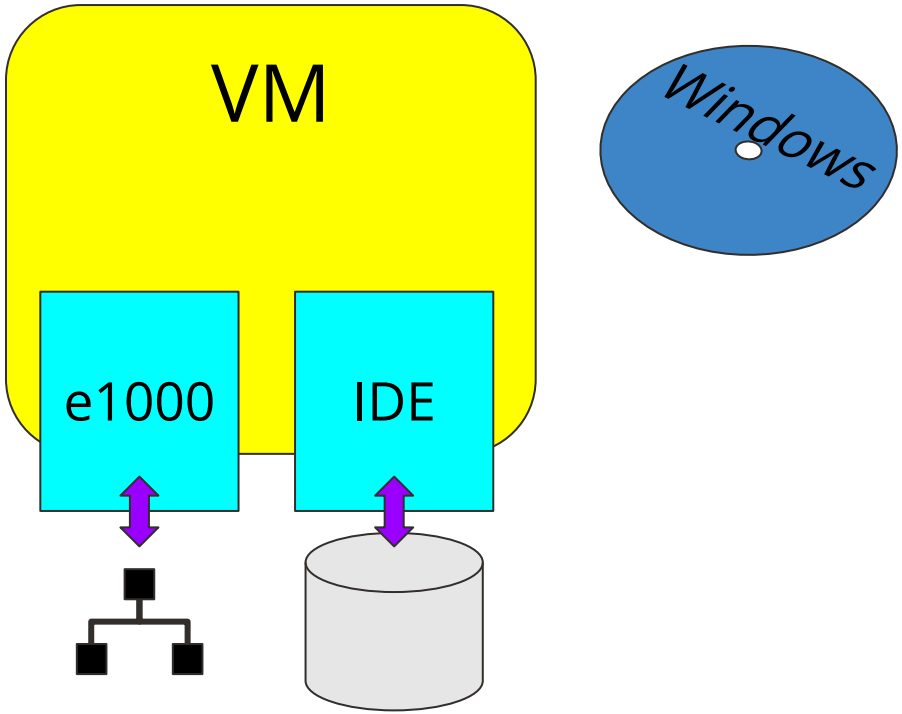
Motivation



wanted:

- performance
- easy to deploy
- support

Choice #1: h/w emulation



- ✔ easy to deploy
- ✔ support
- ✘ performance

Virtual machine ≠ physical machine

physical machine:

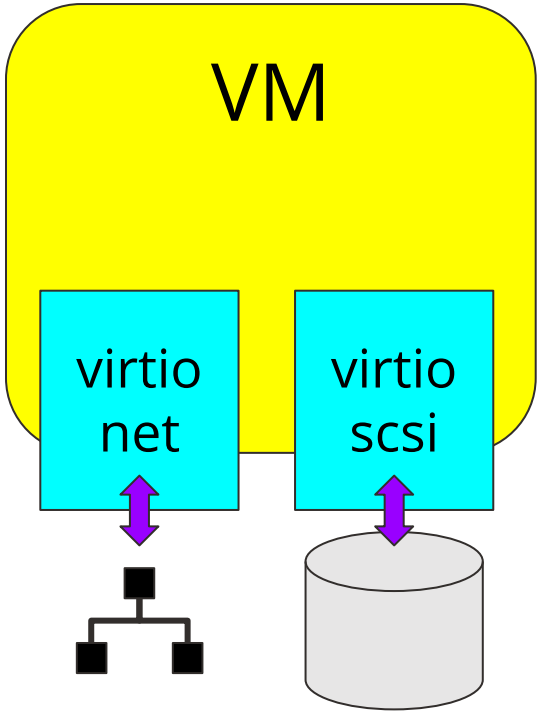
- all CPU and RAM is yours
- timing is (somewhat) predictable

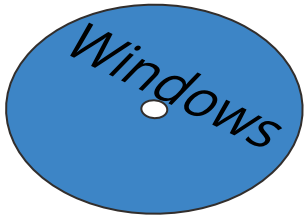
virtual machine:

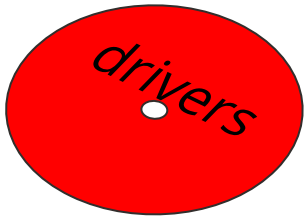
- can be preempted
- can be swapped out
- many things become expensive (APIC, I/O, MSRs, *etc*)

answer: paravirtualization

Choice #2: VirtIO







WindowsGuestDrivers

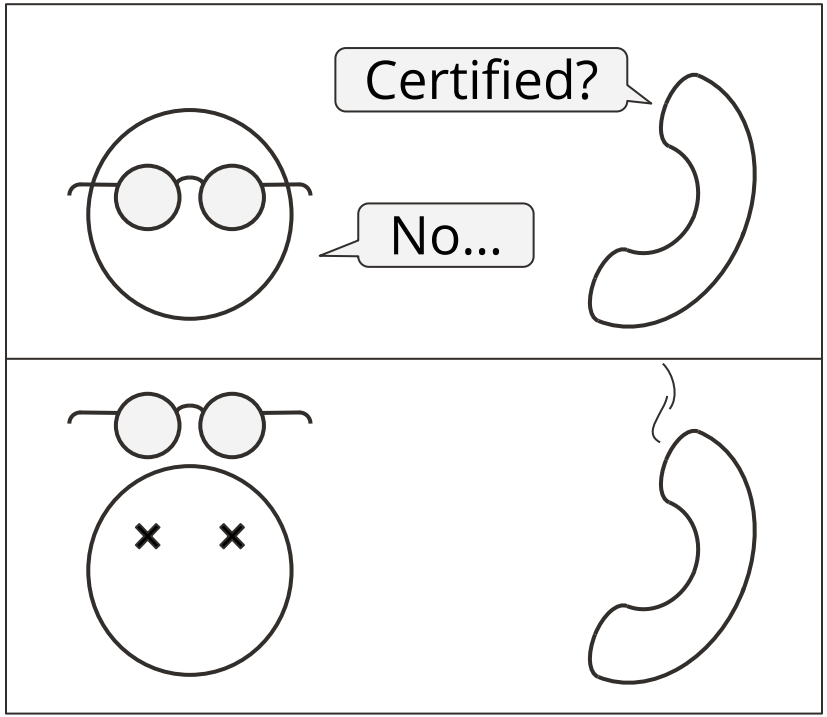
(aka virtio-win)

✔ performance

✘ easy to deploy

✘ support

What's wrong with virtio-win?

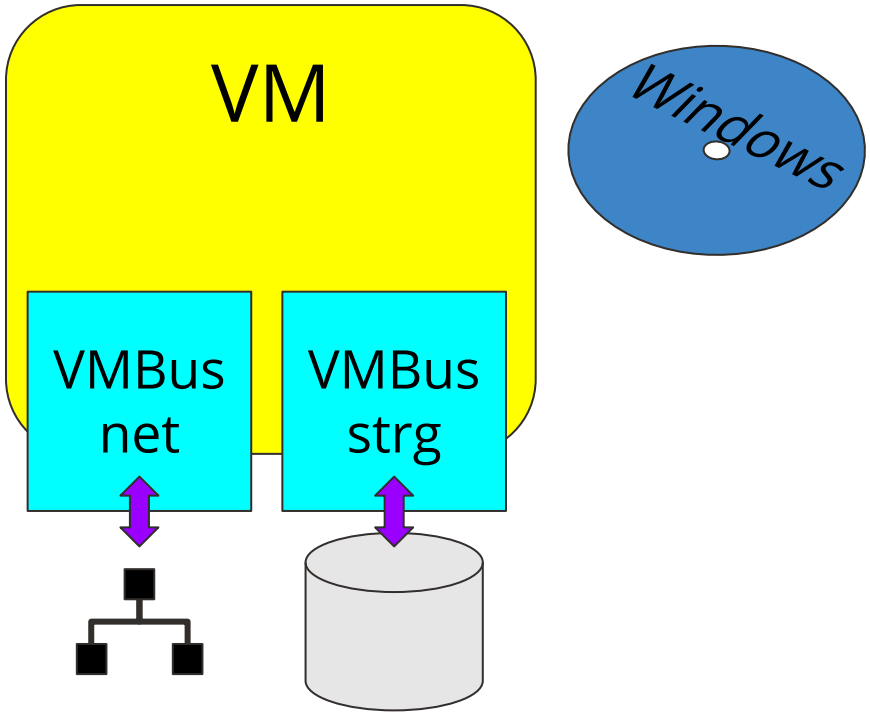


WHQL ⇒ SVVP ⇒ support

GPL 🔥 WHQL

in order to ship it, you need to own it

Choice #3: Hyper-V emulation



- ✔ performance
- ✔ easy to deploy
- ✔ support

👍 sounds like a plan! 👍

Hyper-V: how to?

1. Microsoft docs on GitHub
2. Linux guest code for Hyper-V (everything under `CONFIG_HYPERV`)
3. trial & error
   - e.g. things work with Linux hyperv guest but break with Windows guest

Hyper-V paravirtualization

- previously implemented enlightenments
- management MSRs
- synthetic interrupt controller
- timers
- hypercalls
- VMBus
- devices

Hyper-V preexisting enlightenments

- management MSRs
  - `GUEST_OS_ID`
  - `VP_INDEX`
- hypercall infrastructure
- scheduler
  - `NOTIFY_LONG_SPIN_WAIT` hypercall
- LAPIC
  - MSR access to EOI / ICR / TPR
  - APIC assist page (aka pvEOI)

Hyper-V management MSRs

- reset
- panic
  - `CRASH_CTL`, `CRASH_P0`…`P3` — BSOD info
- `VP_RUNTIME`

Hyper-V clocks

partition reference time: monotonic clock in 100ns ticks since boot

- time reference counter:
  `rdmsr HV_X64_MSR_TIME_REF_COUNT`
  - 1 vmexit / clock read
  - no hardware requirements

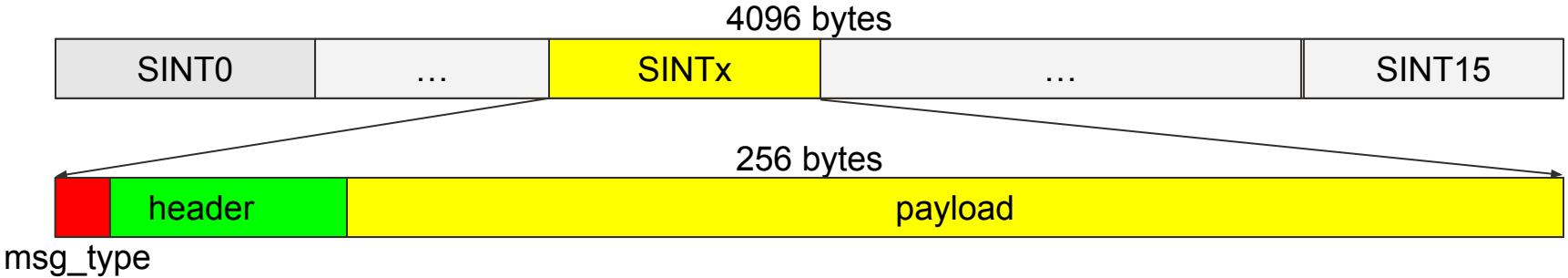
Hyper-V clocks (cont'd)

- TSC reference page: similar to kvm_clock
  `time = (scale * tsc) >> 64 + offset`
  - no vmexits
  - invariant TSC req'd
  - one per VM
  - read consistency via `seqcount`
  - `seqcount == 0` ⇒ fall-back to time ref count
  - no `seqlock` semantics ⇒ use fall-back on updates ⇒ *monotonicity with time ref count* req'd

Hyper-V SynIC (synthetic interrupt controller)

- LAPIC extension managed via MSRs
- 16 SINT's per vCPU
- AutoEOI support
  - *incompatible with APICv*
- `KVM_IRQ_ROUTING_HV_SINT`
  - `GSI → vCPU#, SINT#`
- `irqfd` support
- `KVM_EXIT_HYPERV(SYNIC)` on MSR access

Hyper-V SynIC – message page



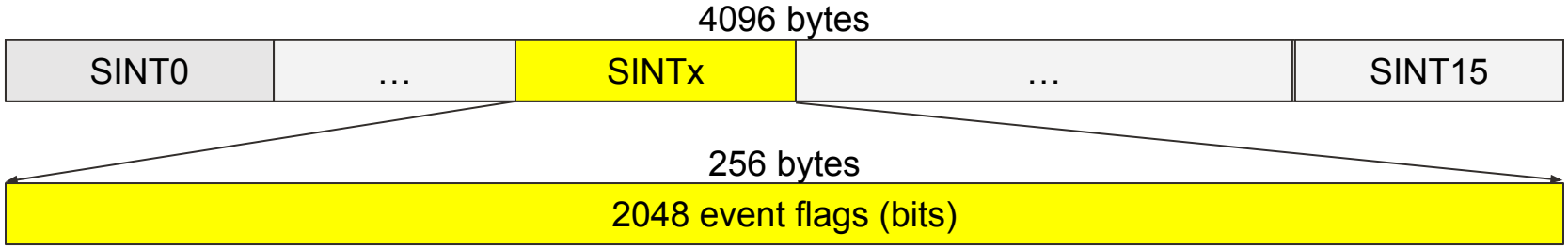
hypervisor post:

- msg_type: *CAS* `TYPE_NONE→TYPE_NNN`
- write payload
- deliver SINTx

guest receive:

- read payload
- msg_type: *atomic* `TYPE_NNN→TYPE_NONE`
- EOI or EOM ⇒ eventfd

Hyper-V SynIC – event flags page



hypervisor signal:

- event flag: *CAS* `0→1`
- deliver SINTx

guest receive:

- event flag: *atomic* `1→0`
- EOI or EOM ⇒ eventfd

Hyper-V timers

- per vCPU: 4 timers × 2 MSRs (config, count)
- in *partition reference time*
- SynIC messages `HVMSG_TIMER_EXPIRED`
  - expiration time
  - delivery time
- in KVM ⇒ first to take message slot
- periodic / one-shot
- lazy (= *discard*) / period modulation (= *slew*)

Hyper-V hypercalls

extend existing implementation in KVM:

- new hypercalls
  - `HVCALL_POST_MESSAGE`
  - `HVCALL_SIGNAL_EVENT`
- pass-through to userspace
  - `KVM_EXIT_HYPERV(HCALL)`
- stub implementation in QEMU

Hyper-V VMBus

- announced via ACPI
- host–guest messaging connection
  - host → guest: SINT & message page
  - guest → host: `POST_MESSAGE` hypercall
- used to
  - negotiate version and parameters
  - discover & setup devices
  - setup *channels*

Hyper-V VMBus channel

entity similar to VirtIO virtqueue

- descriptor rings akin to VirtIO vrings
- 1+ per device
- signaling:
  - host → guest: SINT & event flags page
  - guest → host: `SIGNAL_EVENT` hypercall
- used for data transfer

Hyper-V VMBus devices

- util (shutdown, heartbeat, timesync, VSS, *etc*)
- storage
- net
- balloon

Firmware support

needed to boot off Hyper-V storage or network

- SeaBios
- OVMF

⇒ port over from kernel

Summary

- Hyper-V / VMBus emulation is a viable solution to make Windows guests' life on QEMU/KVM easier
- we have the groundwork in KVM and QEMU mostly complete
- the actual VMBus devices implementation is being worked on

Outlook

- performance measurement & tuning
- `vhost` integration
- `AF_VSOCK` transport
- event logging
- debugging
- more devices
  - input
  - video