KVM Architecture Overview

2015 Edition

Stefan Hajnoczi <stefanha@redhat.com>

Introducing KVM virtualization

KVM hypervisor runs virtual machines on Linux hosts

- Mature on x86, recent progress on ARM and ppc

Most popular and best supported hypervisor on OpenStack

- https://wiki.openstack.org/wiki/HypervisorSupportMatrix

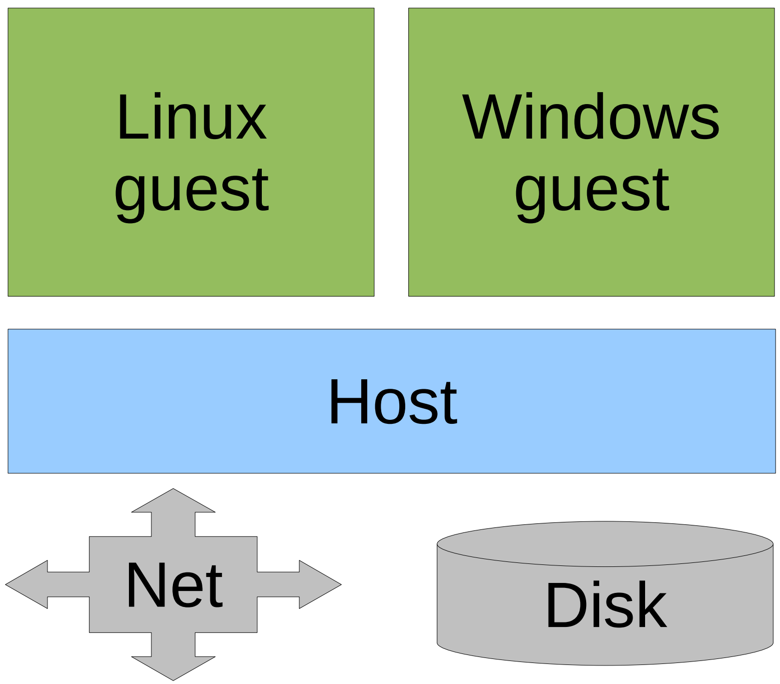
Built in to Red Hat Enterprise Linux

- Qumranet startup created KVM, joined Red Hat in 2008

Virtualization goals

Efficiently and securely running virtual machines on a Linux host

- Linux, Windows, etc guest operating systems
- Access to networking and storage in a controlled fashion



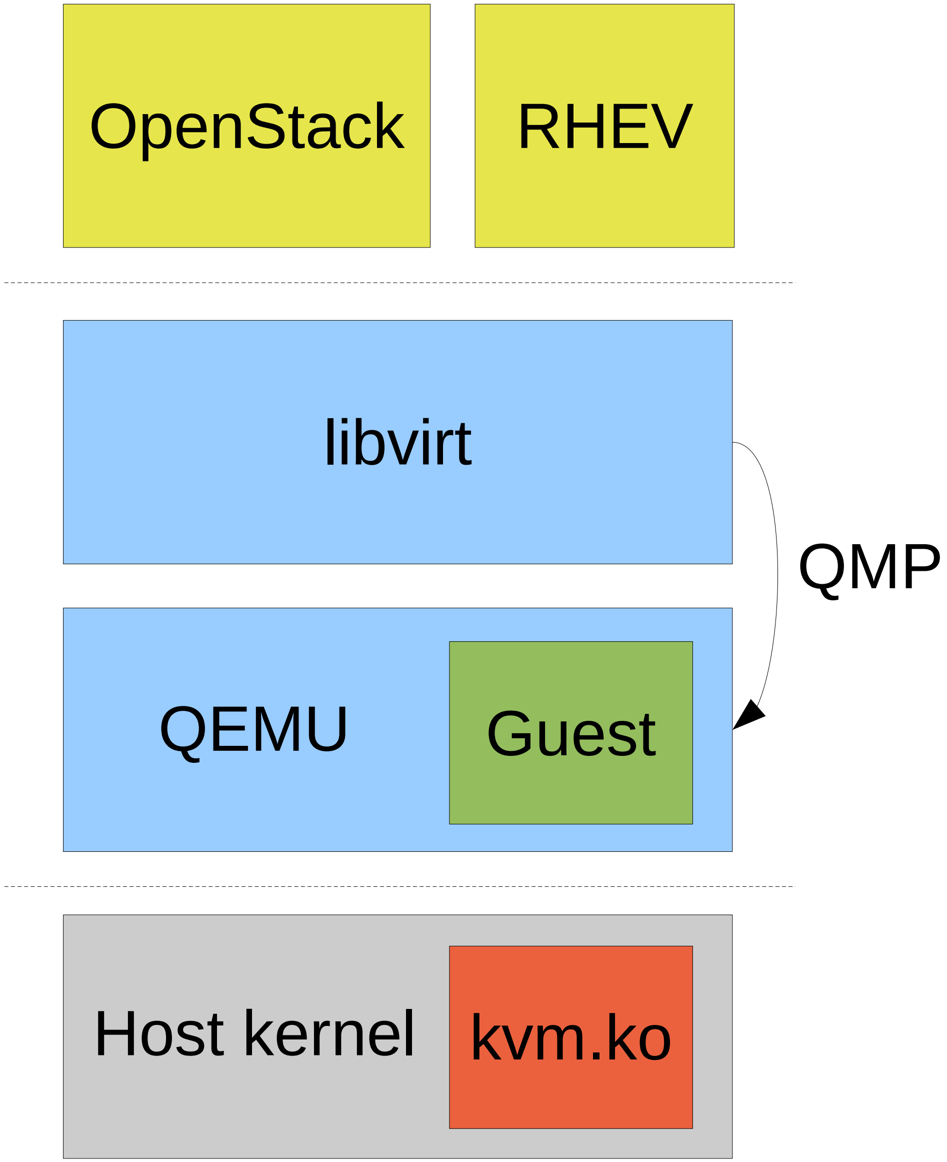
Where does KVM fit into the stack?

Management for datacenters and clouds

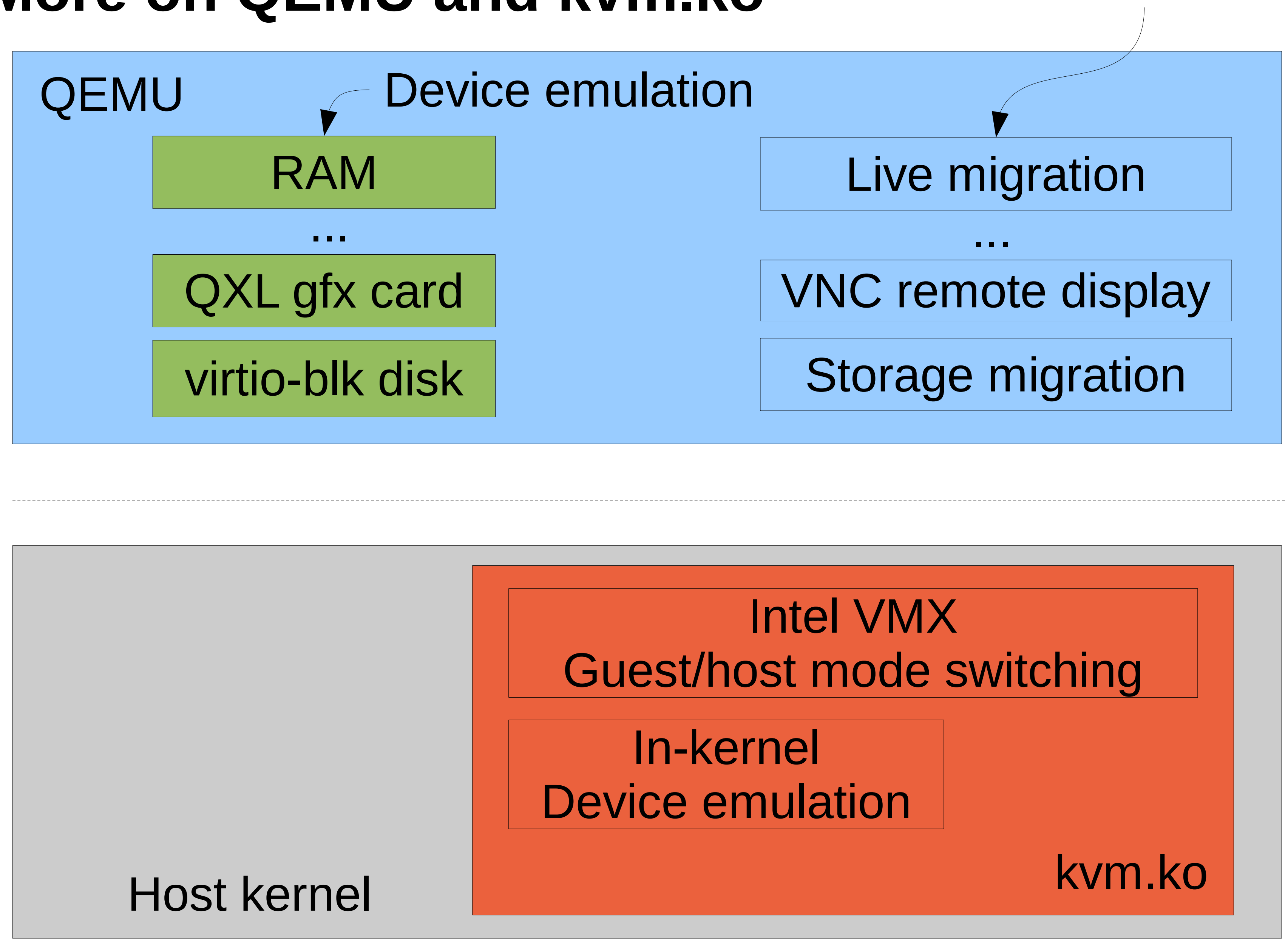
Management for one host

Emulation for one guest

Host hardware access and resource mgmt



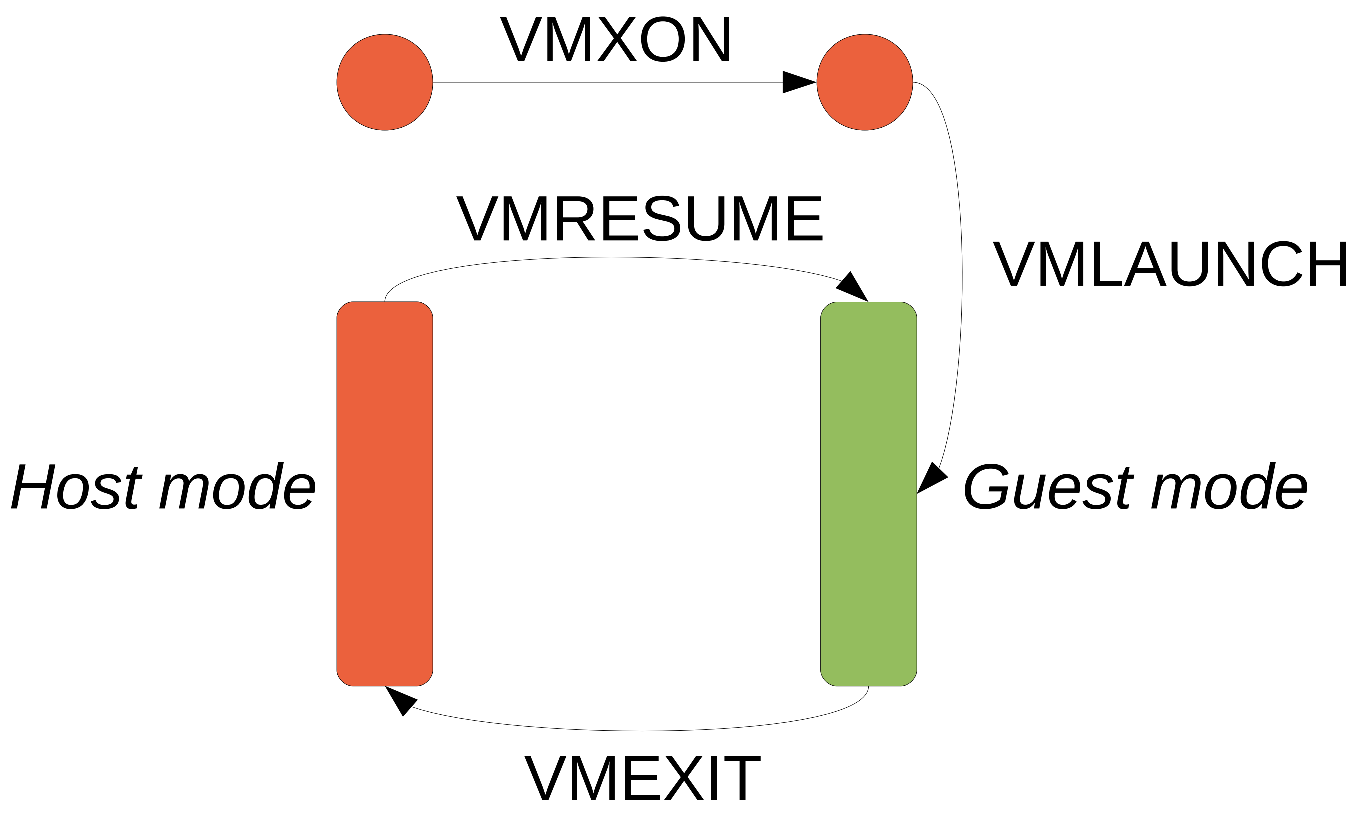
More on QEMU and kvm.ko



Hardware virtualization support with Intel VMX

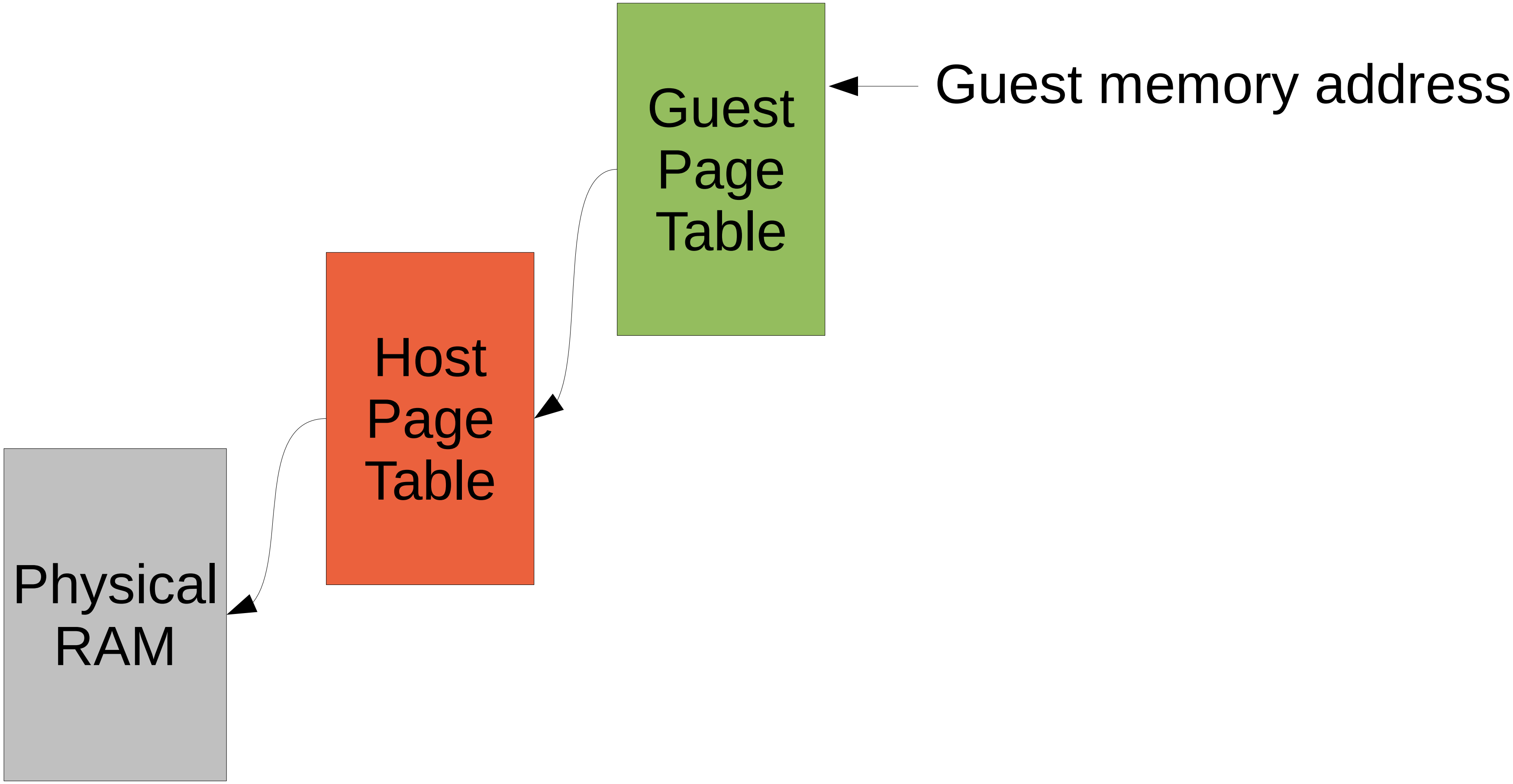
Allows safe guest code execution at native speed

- Certain operations trap out to the hypervisor



Memory virtualization with Intel EPT

Extended Page Tables (EPT) add a level of address translation for guest physical memory.

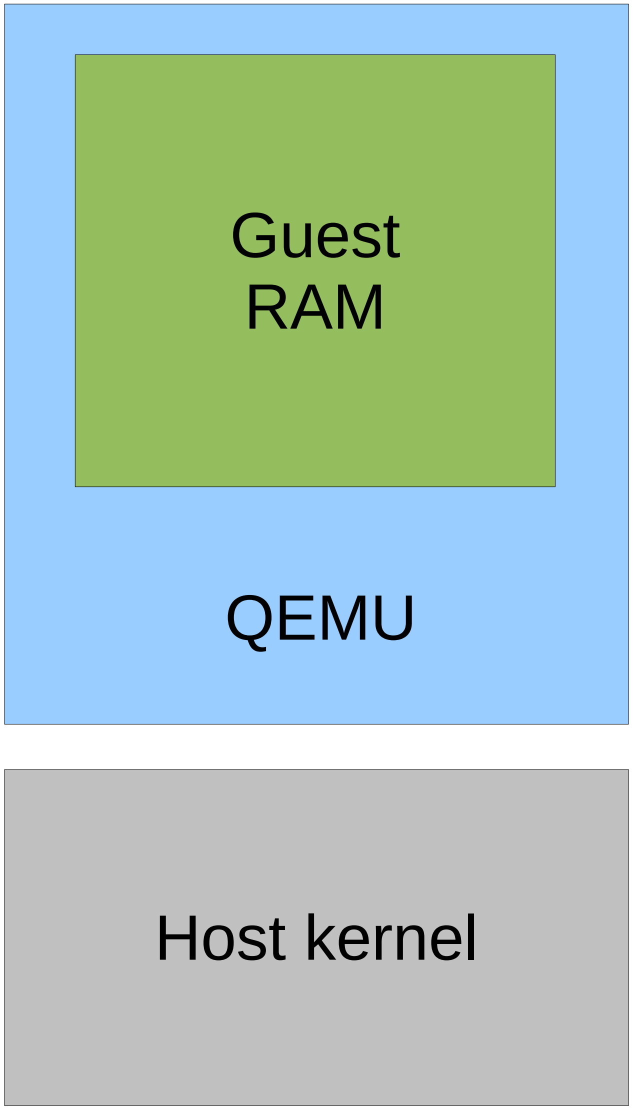


How QEMU uses kvm.ko

QEMU userspace process uses kvm.ko driver to execute guest code:

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
     ioctl(KVM_RUN)
     switch (exit_reason) {
     case KVM_EXIT_IO:  /* ... */
     case KVM_EXIT_HLT: /* ... */
     }
}
```

QEMU process model



QEMU is a userspace process

Unprivileged and isolated using SELinux for security

Each KVM vCPU is a thread

Host kernel scheduler decides when vCPUs run

Linux concepts apply to QEMU/KVM

Since QEMU is a userspace process, the usual Linux tools work:

ps(1), top(1), etc see QEMU processes and threads

tcpdump(8) sees tap network traffic

blktrace(8) sees disk I/O requests

SystemTap and perf see QEMU activity

etc

Architecture: Event-driven multi-threaded

Event loops are used for timers, file descriptor monitoring, etc

- Non-blocking I/O
- Callbacks or coroutines

Multi-threaded architecture but with big lock

- VCPU threads execute in parallel
- Specific tasks that would block event loop are done in threads, e.g. remote display encoding, RAM live migration work, virtio-blk dataplane, etc
- Rest of QEMU code runs under global mutex

Architecture: Emulated and pass-through devices

Guest sees CPU, RAM, disk, etc like on real machines

- Unmodified operating systems can run
- Paravirtualized devices for better performance

Most devices are emulated and not real

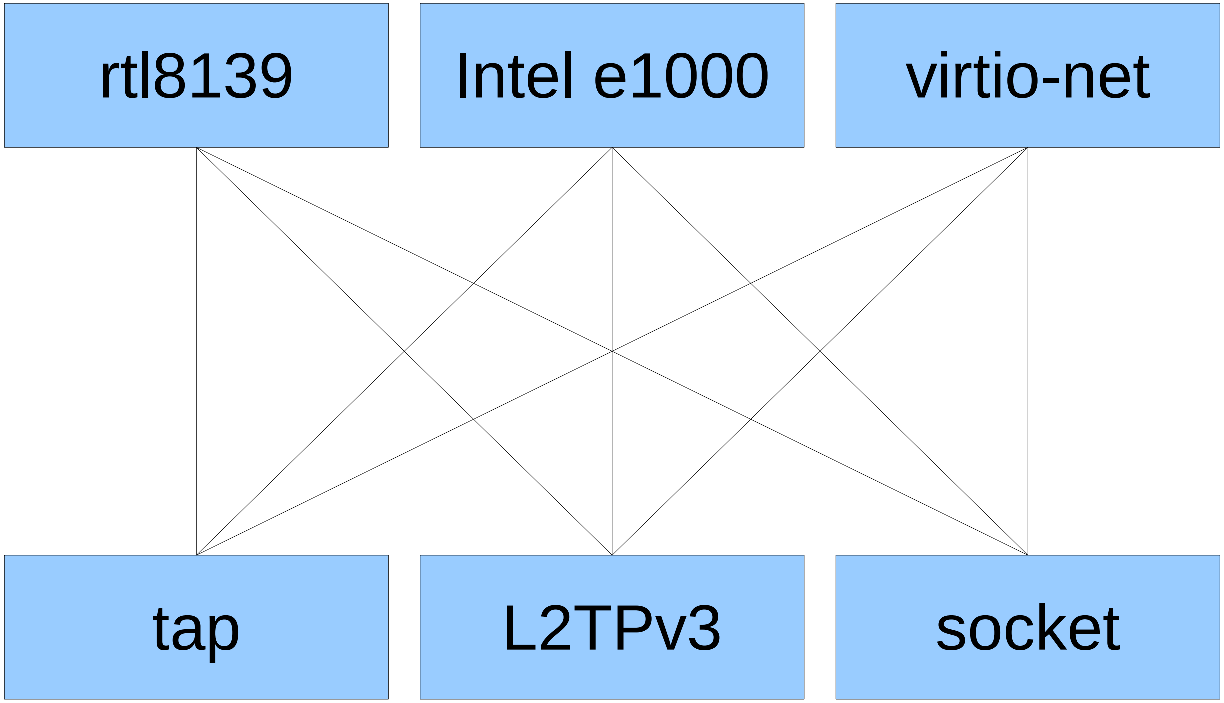
- Isolation from host for security
- Sharing of resources between guests

Pass-through PCI adapters, disks, etc also possible

- Dedicated hardware

Architecture: Host/guest device emulation split

Guest device – device model visible to guest



Decouples hardware emulation from I/O mechanism

Host device – performs I/O on behalf of guest

Architecture: virtio devices

KVM implements virtio device models

- net, blk, scsi, serial, rng, balloon
- See http://docs.oasis-open.org/virtio/ for specs

Open standard for paravirtualized I/O devices

Red Hat contributes to Linux and Windows guest drivers

Architectural exception: vhost in-kernel devices

Most device emulation is best done in userspace

- Some APIs or performance features only available in host kernel

vhost drivers emulate virtio devices in host kernel

- vhost_net.ko high-performance virtio-net emulation takes advantage of kernel-only zero-copy and interrupt handling features
- Other devices could be developed in theory, but usually userspace is a better choice

Storage in QEMU

Block drivers fall in two categories:

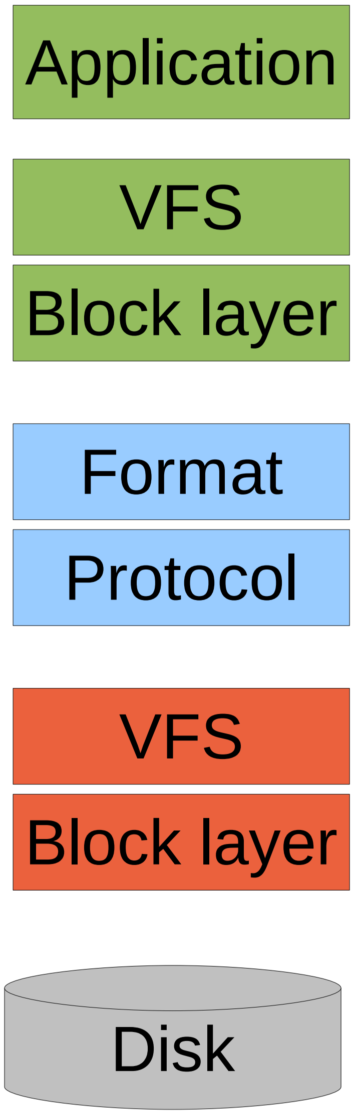
Formats – image file formats (qcow2, vmdk, etc)

qcow2	raw	…
raw-posix	rbd (Ceph)	…

Protocols – I/O transports (POSIX file, rbd/Ceph, etc)

Plus additional block drivers that interpose like quorum, blkdebug, blkverify

Storage stack



Guest – application plus full file system and block layer

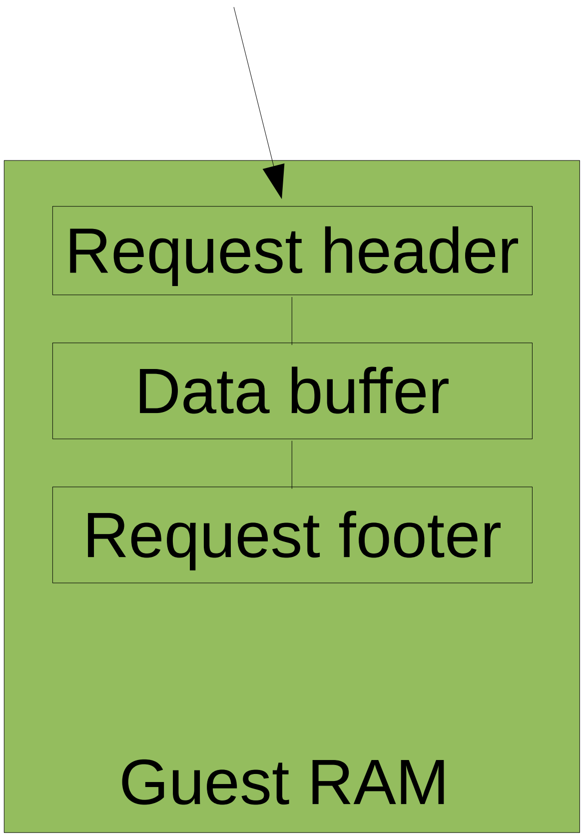
QEMU – image format, storage migration, I/O throttling

Host – full file system and block layer

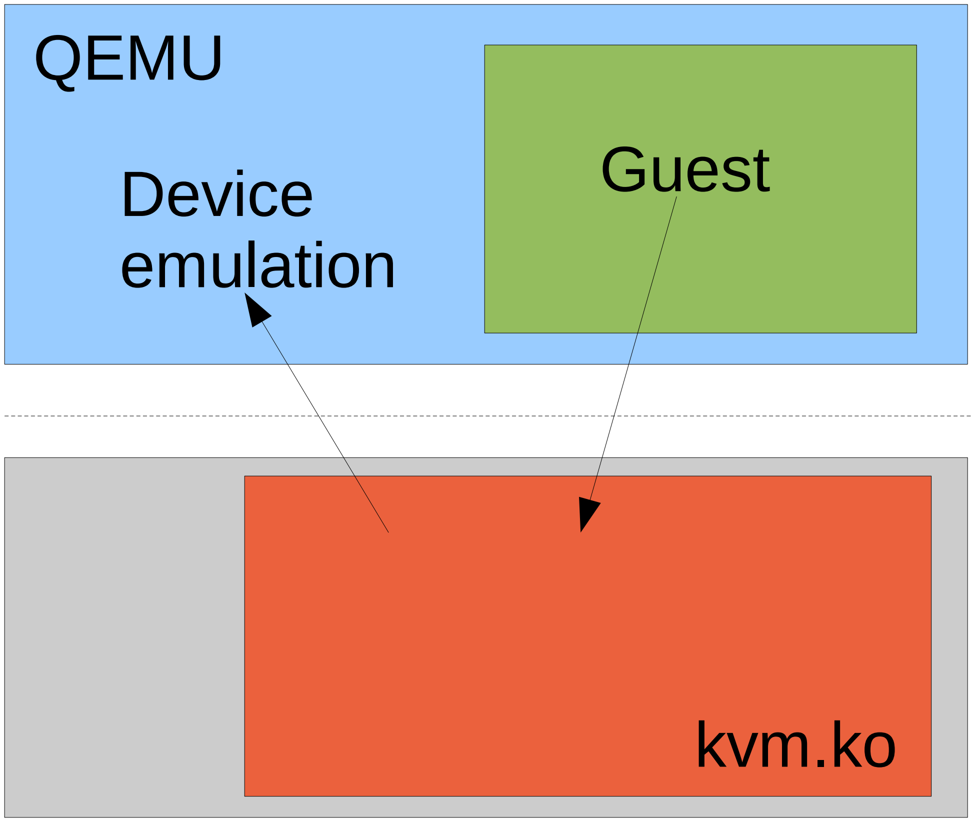
Beware double caching and anticipatory scheduling delays!

Walkthrough: virtio-blk disk read request (Part 1)

1. Guest fills in request descriptors

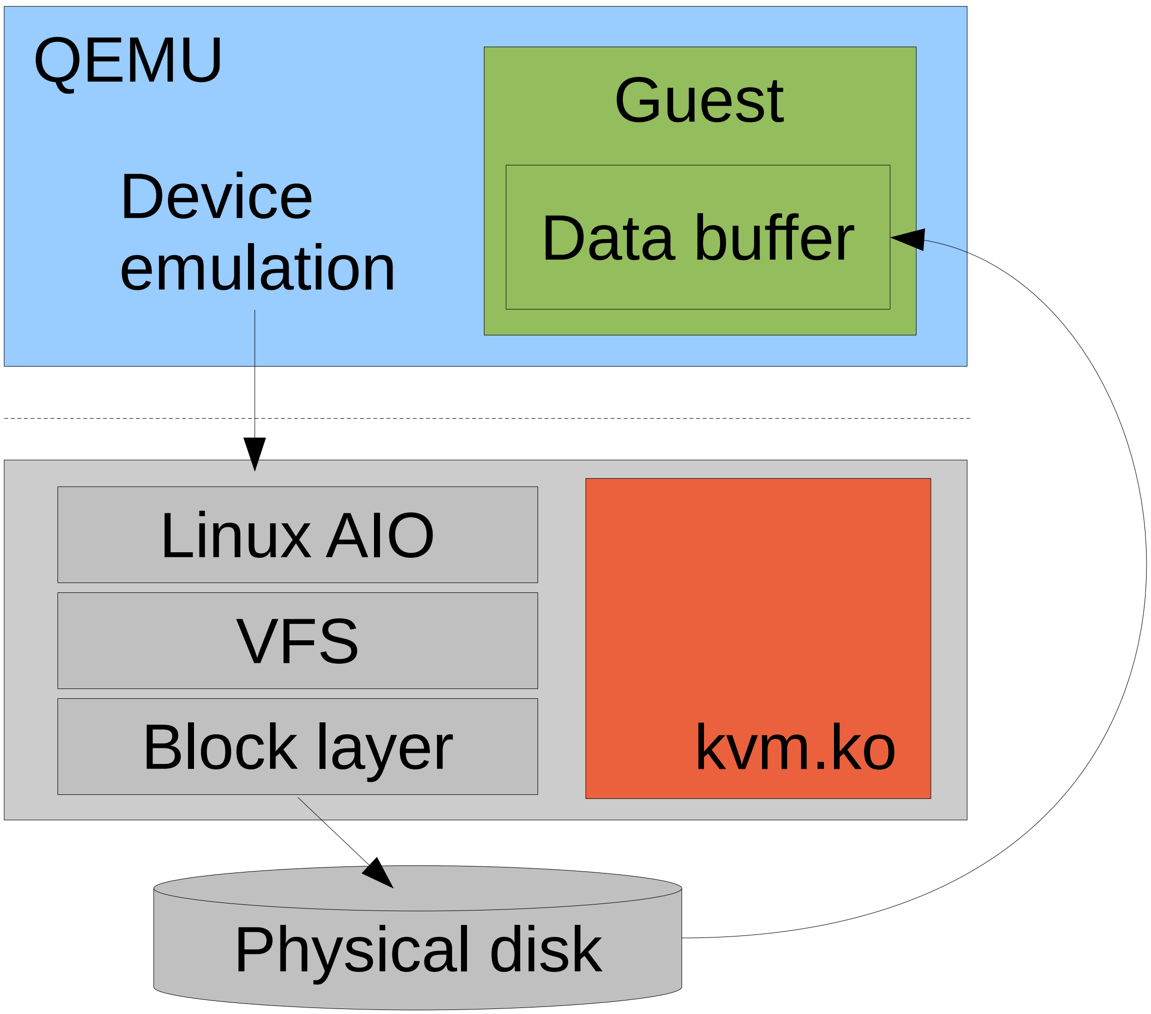


2. Guest writes to virtio-blk virtqueue notify register



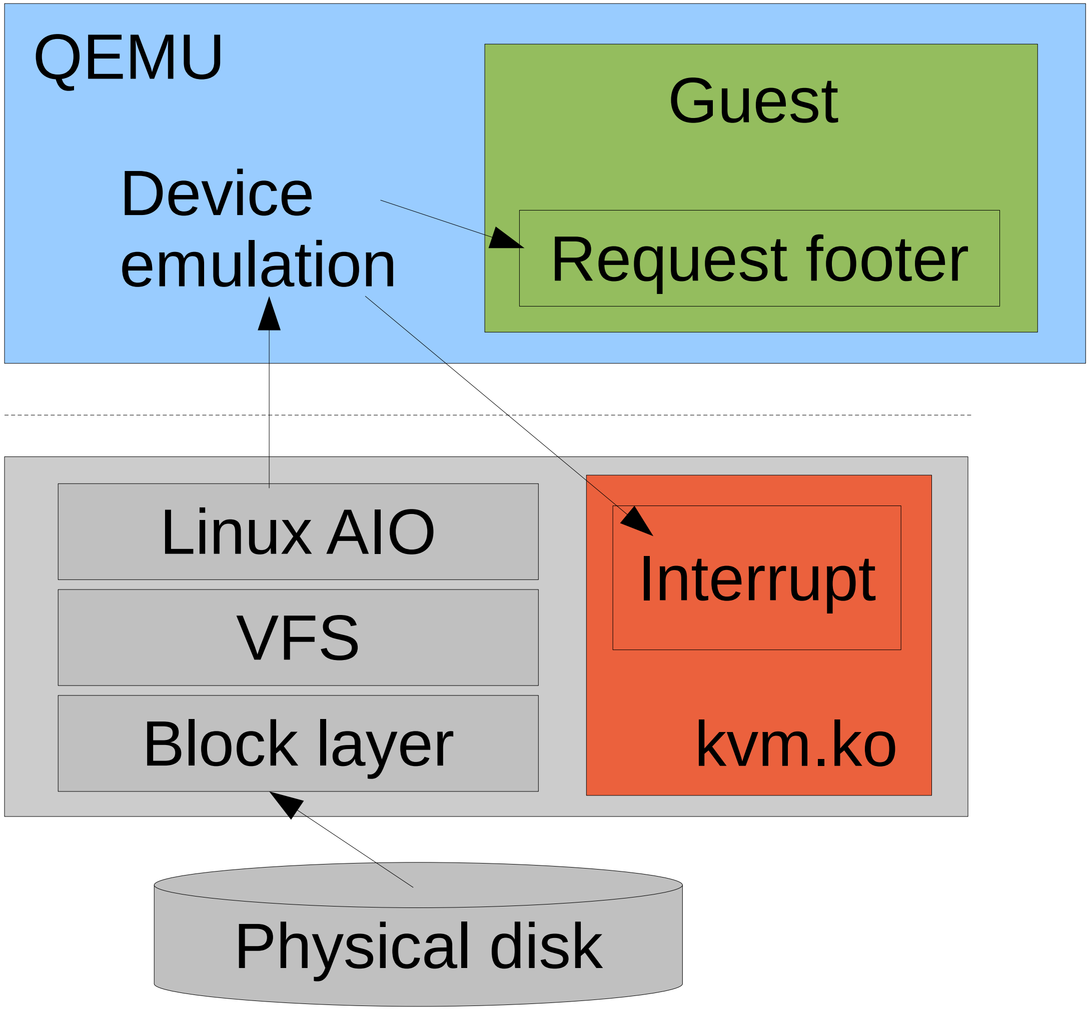
Walkthrough: virtio-blk disk read request (Part 2)

3. QEMU issues I/O request on behalf of guest



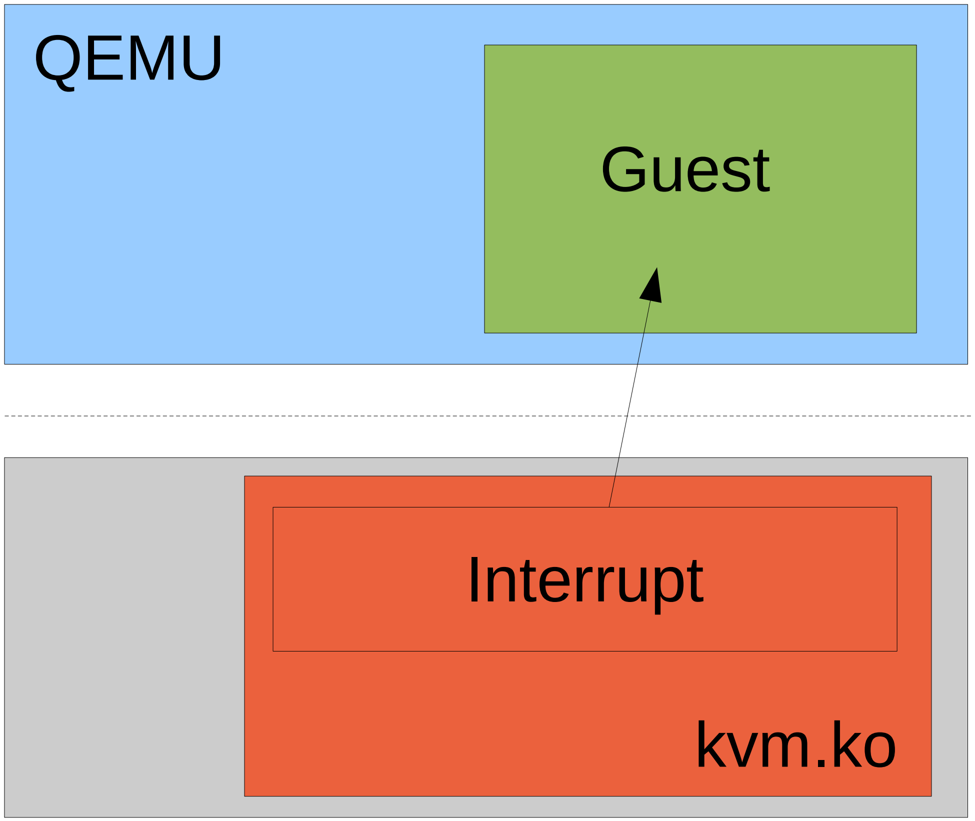
Walkthrough: virtio-blk disk read request (Part 3)

4. QEMU fills in request footer and injects completion interrupt

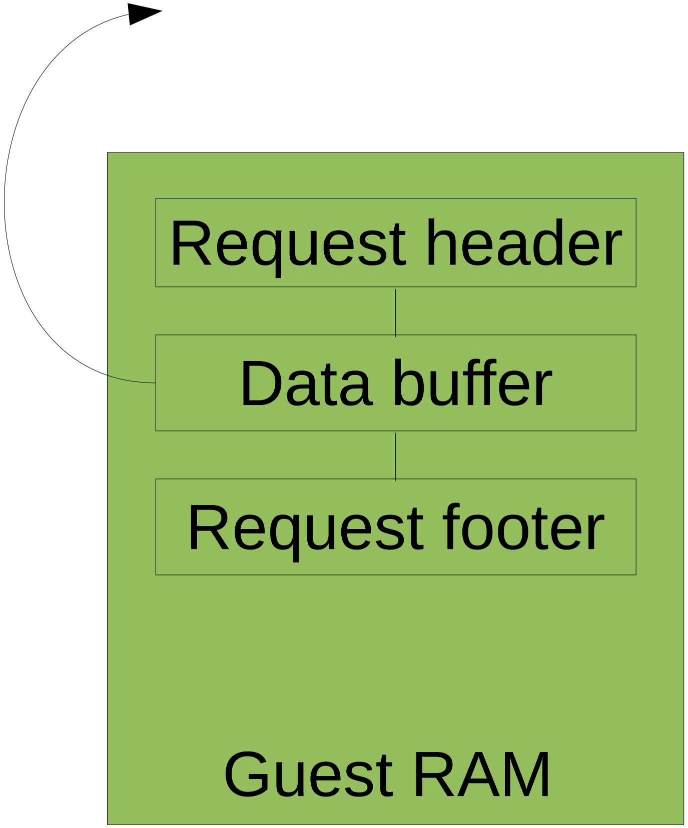


Walkthrough: virtio-blk disk read request (Part 4)

5. Guest receives interrupt and executes handler



6. Guest reads data from buffer



Thank you!

Technical discussion: qemu-devel@nongnu.org

IRC

- #qemu on irc.oftc.net
- #kvm on chat.freenode.net

http://qemu-project.org/

http://linux-kvm.org/

More on my blog: http://blog.vmsplice.net/