Speed up your kernel development cycle with QEMU

Stefan Hajnoczi <stefanha@redhat.com>

Kernel Recipes 2015

Agenda

- Kernel development cycle
- Introduction to QEMU
- Basics
  - Testing kernels inside virtual machines
  - Debugging virtual machines
- Advanced topics
  - Cross-architecture testing
  - Device bring-up
  - Error injection

About me

QEMU contributor since 2010

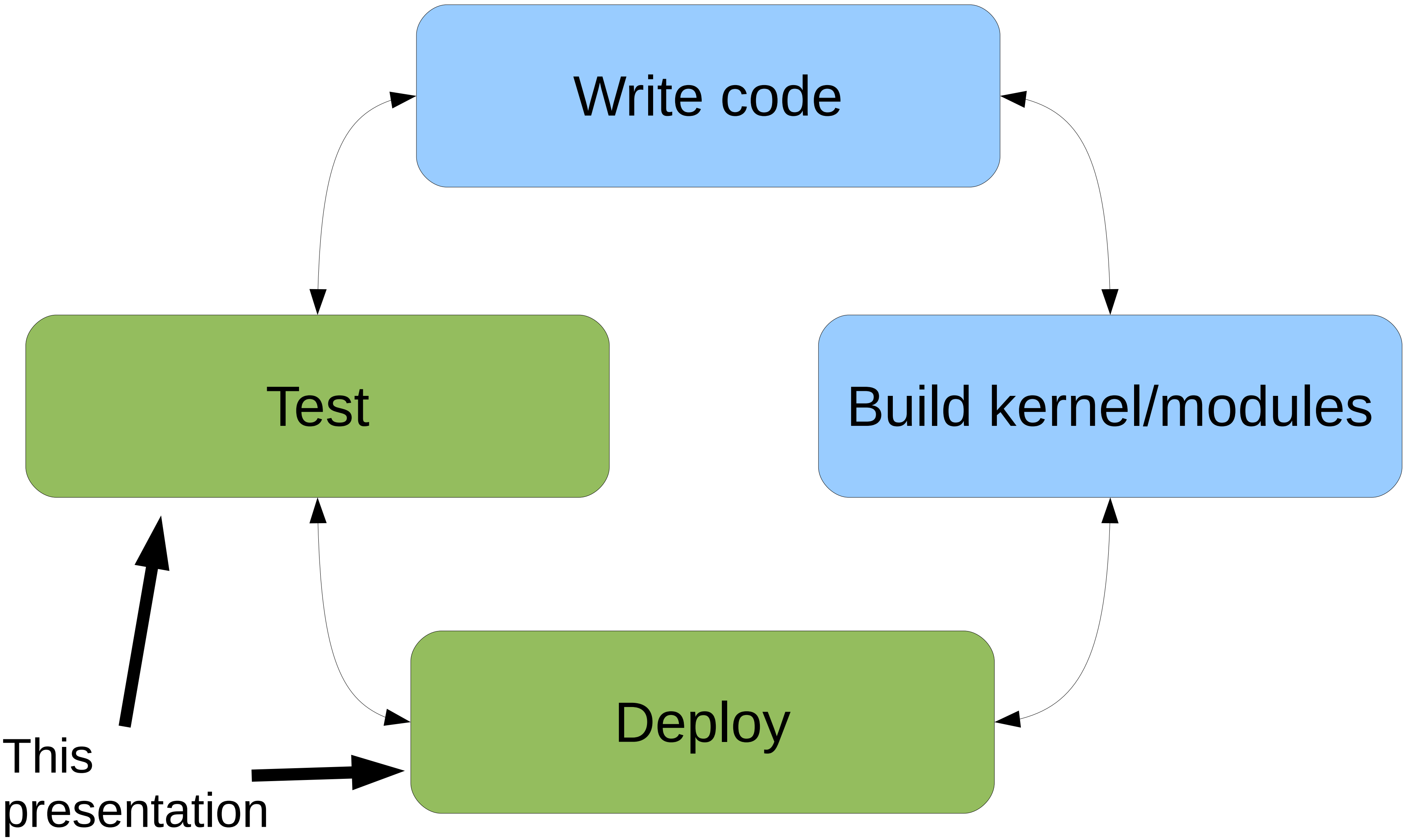
- Subsystem maintainer
- Google Summer of Code & Outreachy mentor/admin
- http://qemu-advent-calendar.org/

Occassional kernel patch contributor

- vsock, tcm_vhost, virtio_scsi, line6 staging driver

Work in Red Hat's Virtualization team

Kernel development cycle



Write code

Test

Build kernel/modules

Deploy

This presentation

If you are doing kernel development...

- Device drivers
  - USB
  - PCI
  - etc
- Tracing
  - ftrace
  - ebpf
- Storage
  - LIO SCSI target
  - File systems
  - device-mapper targets
- Networking
  - Network protocols
  - Netfilter
  - OpenVSwitch
- Resource management & security
  - Cgroups
  - Linux Security Modules
  - Namespaces

.

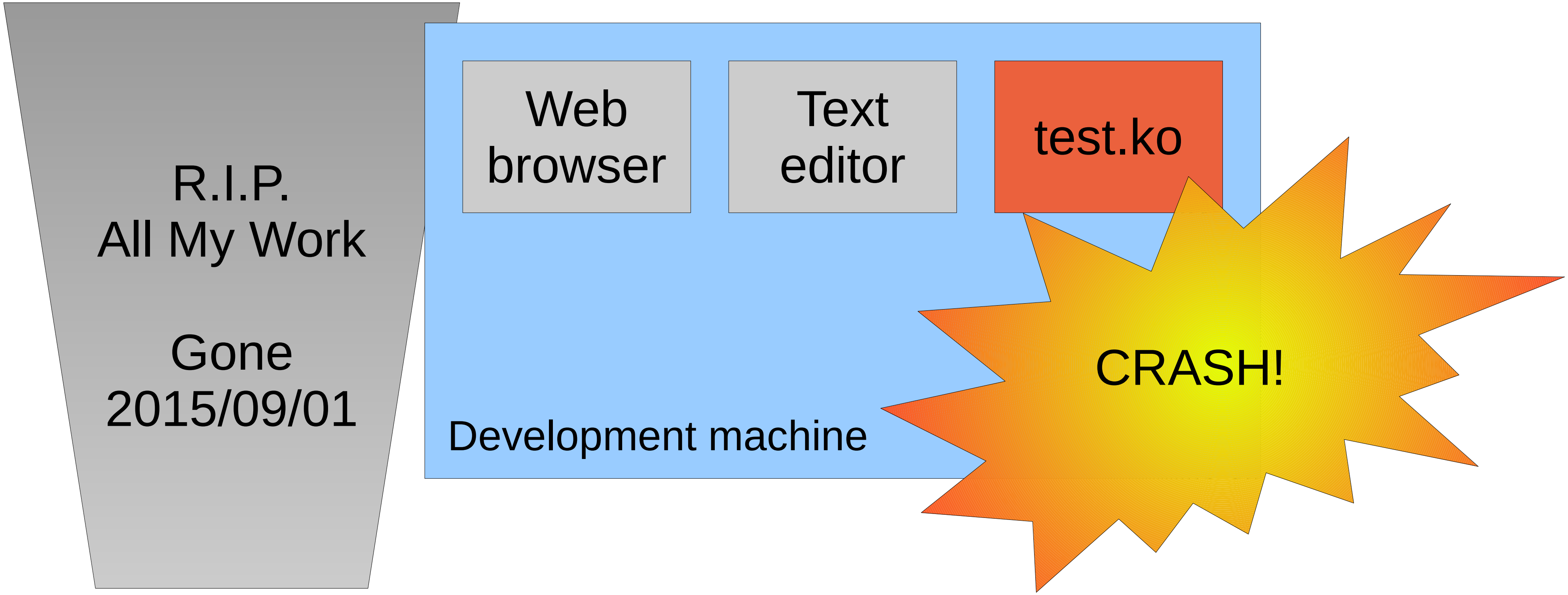
.

.

...you might have these challenges

In situ debugging mechanisms like kgdb or kdump

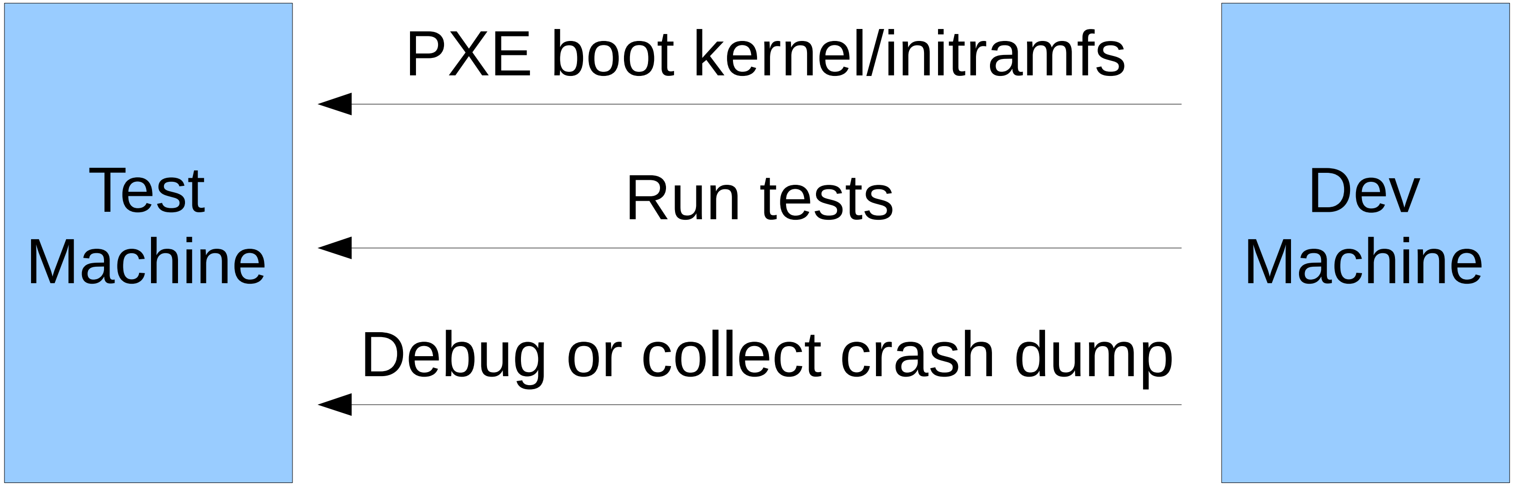
- Not 100% reliable since they share the environment
- Crashes interrupt your browser/text editor session



Dedicated test machines

Ex situ debugging requires an additional machine

- More cumbersome to deploy code and run tests
- May require special hardware (JTAG, etc)
- Less mobile, hard to travel with multiple machines



Test Machine

PXE boot kernel/initramfs

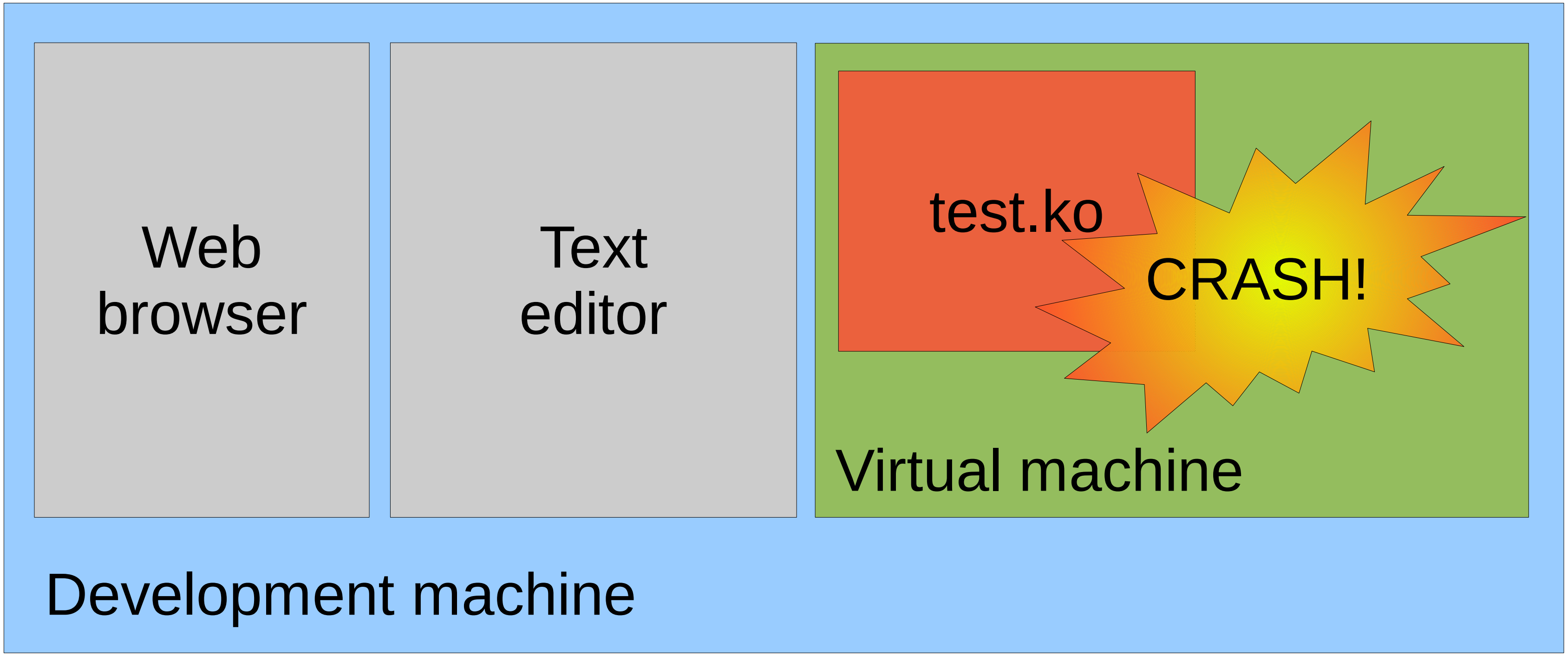
Run tests

Debug or collect crash dump

Dev Machine

Virtual machines: best of both worlds!

- Easy to start/stop
- Full access to memory & CPU state
- Cross-architecture support using emulation
- Programmable hardware (e.g. error injection)



QEMU emulator and virtualizer

Logo by Benoît Canet

Website: http://qemu-project.org/

Runs on Linux, Mac, BSD, and Windows

Emulates 17 CPU architectures (x86, arm, ppc, ...)

Supports fast hardware virtualization using KVM

Open source GPLv2 license

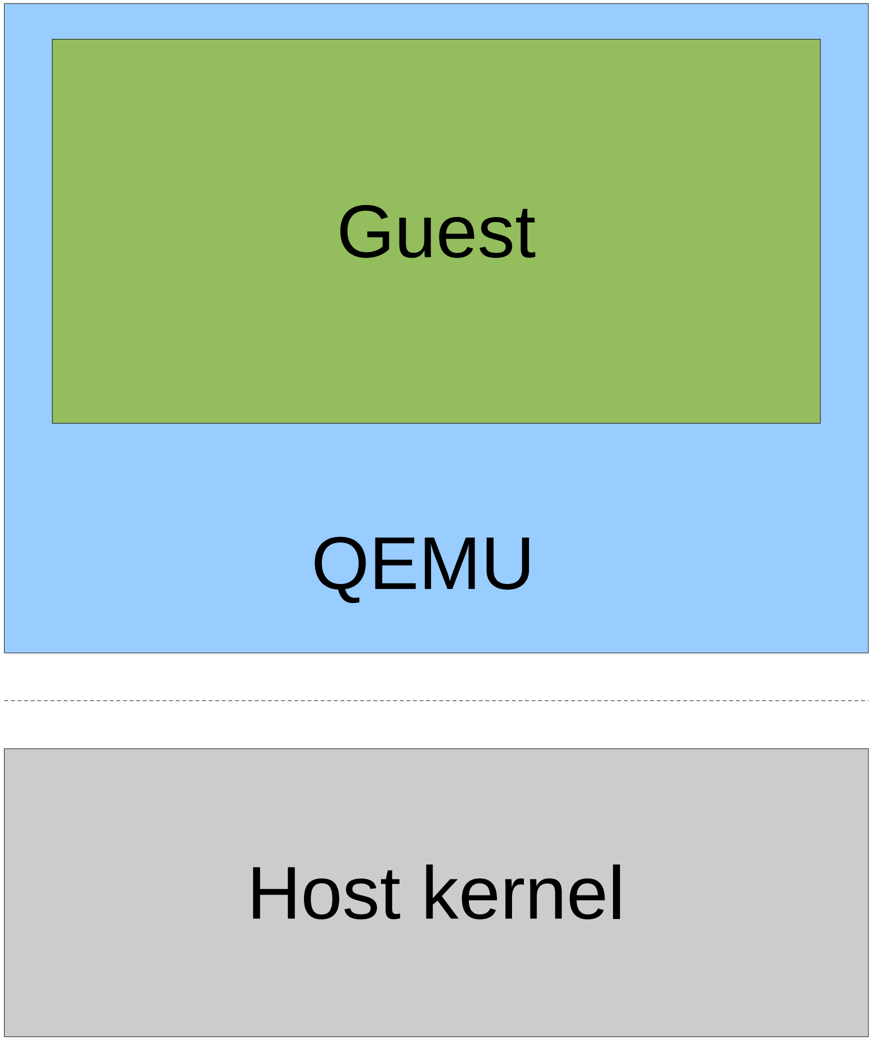
QEMU overview

Guest code runs in a virtual machine

Hardware devices are emulated

QEMU performs I/O on behalf of guest

QEMU appears as a normal userspace process on the host



Cross-architecture emulation

Run another type of machine on your laptop

- `qemu-system-arm`
- `qemu-system-ppc`
- ...

Uses just-in-time compilation to achieve reasonable speed

- Overhead can still be noticable

Launching a virtual machine

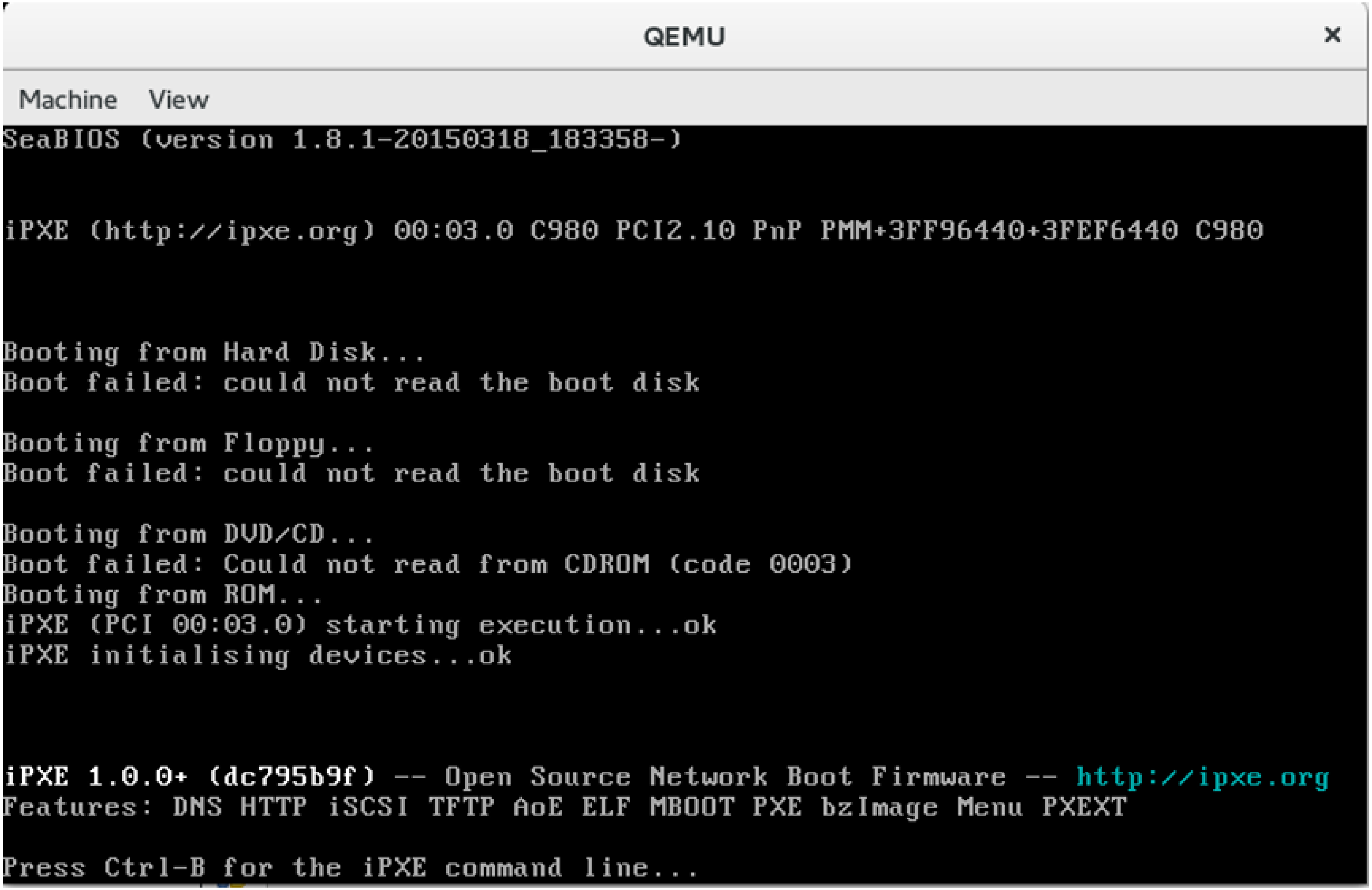
Example with 1024 MB RAM and 2 CPUs:

```
qemu-system-x86_64 -m 1024 \
                   -smp 2 \
                   -enable-kvm
```

Drop `-enable-kvm` for emulation (e.g. ARM on x86)

Boots up to BIOS but there are no bootable drives...

QEMU virtual machine in BIOS/PXE



```
SeaBIOS (version 1.8.1-20150318_183358-)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+3FF96440+3FEF6440 C980



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Boot failed: could not read the boot disk

Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok



iPXE 1.0.0+ (dc795b9f) -- Open Source Network Boot Firmware -- http://ipxe.org
Features: DNS HTTP iSCSI TFTP AoE ELF MBOOT PXE bzImage Menu PXEXT

Press Ctrl-B for the iPXE command line...
```

How to boot a development kernel

```
qemu-system-x86_64 -enable-kvm -m 1024 \
    -kernel /boot/vmlinuz \
    -initrd /boot/initramfs.img \
    -append param1=value1
```

These options are similar to bootloader (GRUB, etc) options.

Small tests can run from initramfs

Initramfs can be customized to contain test programs

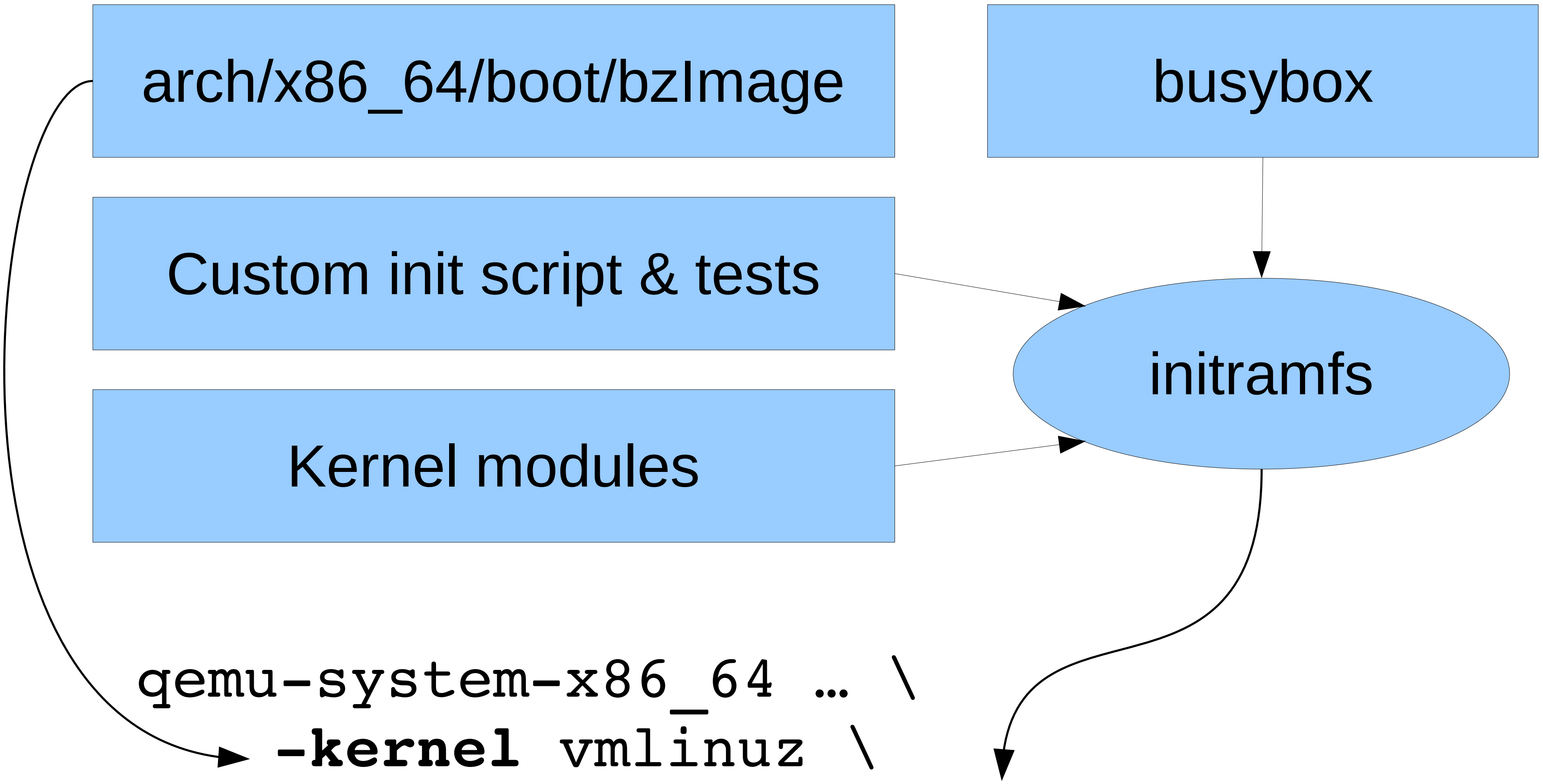
No need for full root file system

- Kick off tests from /init executable

Rebuild initramfs when kernel or test code changes

Result: Fast deployment & test

Deploying kernel build products



arch/x86_64/boot/bzImage

busybox

Custom init script & tests

initramfs

Kernel modules

```
qemu-system-x86_64 … \
    -kernel vmlinuz \
    -initrd initramfs.img \
    -append param1=value1
```

Building initramfs with gen_init_cpio

gen_init_cpio takes description file as input:

```
file /init my-init.sh 0755 0 0
dir /bin 0755 0 0
nod /dev/zero 0666 0 0 c 1 5
file /sbin/busybox /sbin/busybox 0755 0 0
slink /bin/sh /sbin/busybox 0755 0 0
```

Produces cpio archive as output:

```
$ usr/gen_init_cpio input | gzip >initramfs.img
```

Included in Linux source tree (usr/gen_init_cpio)

Build process

Compile your kernel modules:

```
$ make M=drivers/virtio \
       CONFIG_VIRTIO_PCI=m modules
```

Build initramfs:

```
$ usr/gen_init_cpio input | gzip >initramfs.img
```

Run virtual machine:

```
$ qemu-system-x86_64 -m 1024 -enable-kvm \
          -kernel arch/x86_64/boot/bzImage \
          -initrd initramfs.img \
          -append 'console=ttyS0' \
          -nographic
```

Using QEMU serial port for testing

I snuck in the QEMU `–nographic` option

- Disables GUI
- Puts serial port onto stdin/stdout
- Perfect for running tests from terminal
- Easy to copy-paste error messages from output

Tell kernel to use `console=ttyS0`

Challenges with manually built initramfs

Shared library dependencies must be found with ldd(1) and added

Paths on the host may change across package upgrades, breaking your initramfs build process

Rebuilding large initramfs every time is wasteful

Maybe it's time for a real root file system?

Persistent root file system

Two options:

1) *Share directory with host* using virtfs or NFS
Pro: Easy to manipulate and inspect on host

2) *Use disk image file* with partition table and file systems
Pro: Easy to install full Linux distro

Kernel can still be provided by `–kernel` option.

Kernel modules need to be in initramfs and/or root file system.

Debugging a virtual machine

How do I inspect CPU registers and memory?

How do I set breakpoints on kernel code inside the virtual machine?

QEMU supports **GDB remote debugging** to attach to the virtual machine.

kgdb is not required inside virtual machine.

Remote debugging != debugging QEMU

Often causes confusion:

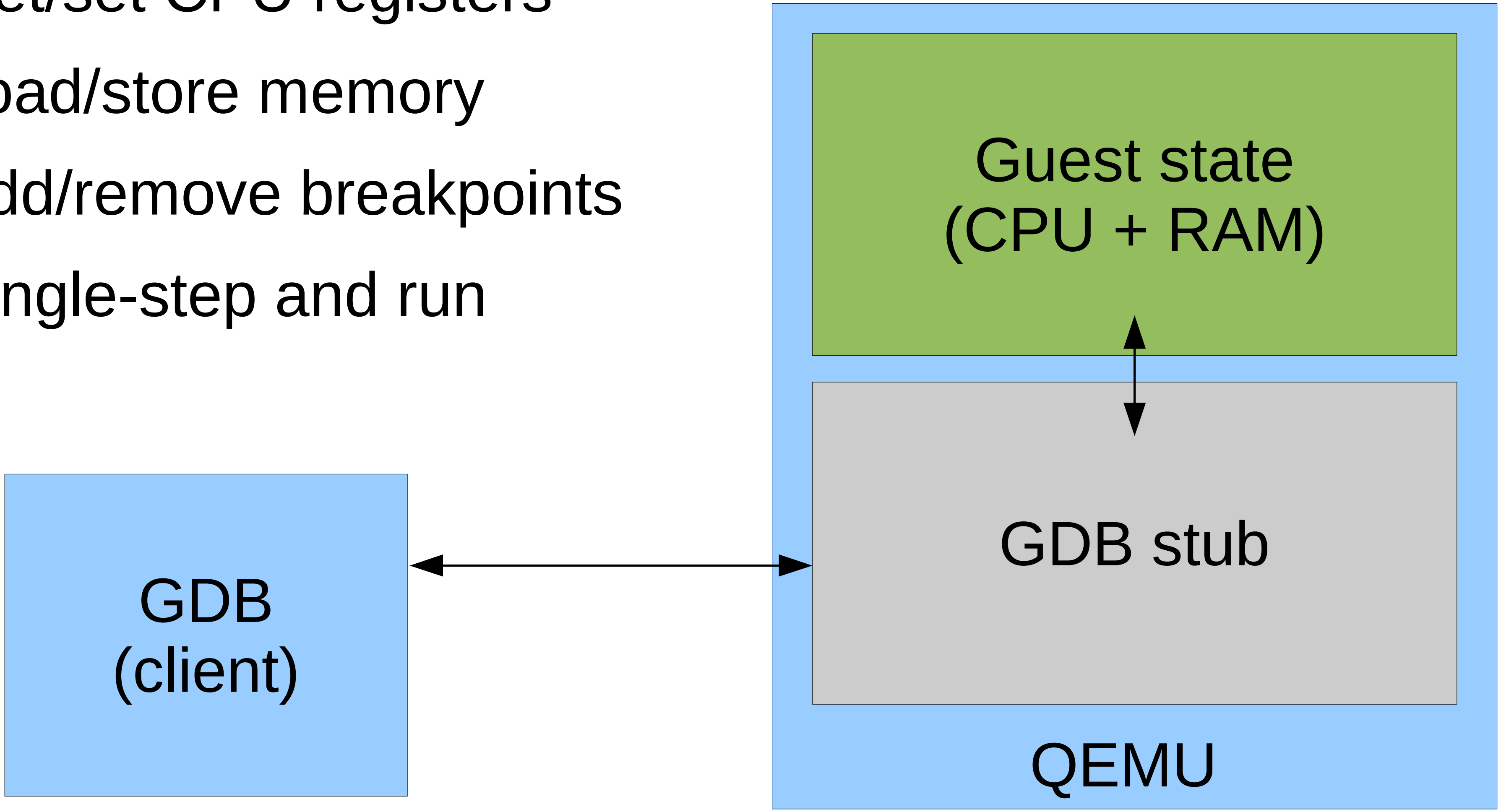
If you want to **debug what the virtual machine sees**, use remote debugging (gdbstub).

If you want to **debug device emulation or QEMU internals**, use gdb -p $QEMU_PID.

GDB remote debugging

Protocol for remote debugging:

- Get/set CPU registers
- Load/store memory
- Add/remove breakpoints
- Single-step and run



QEMU gdbstub example

```
qemu-system-x86_64 -s -enable-kvm -m 1024 \
           -drive if=virtio,file=test.img
```

```
(gdb) set architecture i386:x86-64

(gdb) file vmlinux

(gdb) target remote 127.0.0.1:1234

(gdb) backtrace

#0  native_safe_halt () at
./arch/x86/include/asm/irqflags.h:50

#1  0xffffffff8101efae in arch_safe_halt ()
...
```

Things to remember with remote debugging

Tell GDB which **(sub-)architecture** to use

- x86: 16-bit vs 32-bit vs 64-bit mode, check RFLAGS register
- Careful with guest programs that switch modes!

Memory addresses are generally **virtual addresses** (i.e. memory translation applies)

GDB doesn't know much about current userspace process or swapped out pages!

Device bring-up

Challenges for driver developers:

- Real hardware is not available yet
- Hardware is expensive
- Hardware/software co-development

How to develop & test drivers under these conditions?

1) Implement device emulation in QEMU
2) Develop driver against emulated device
3) Verify against real hardware when available

QEMU for device bring-up

Write C code in QEMU to emulate your device

- Out-of-tree solutions for hw *simulation* exist too!

QEMU device emulation covers common busses:

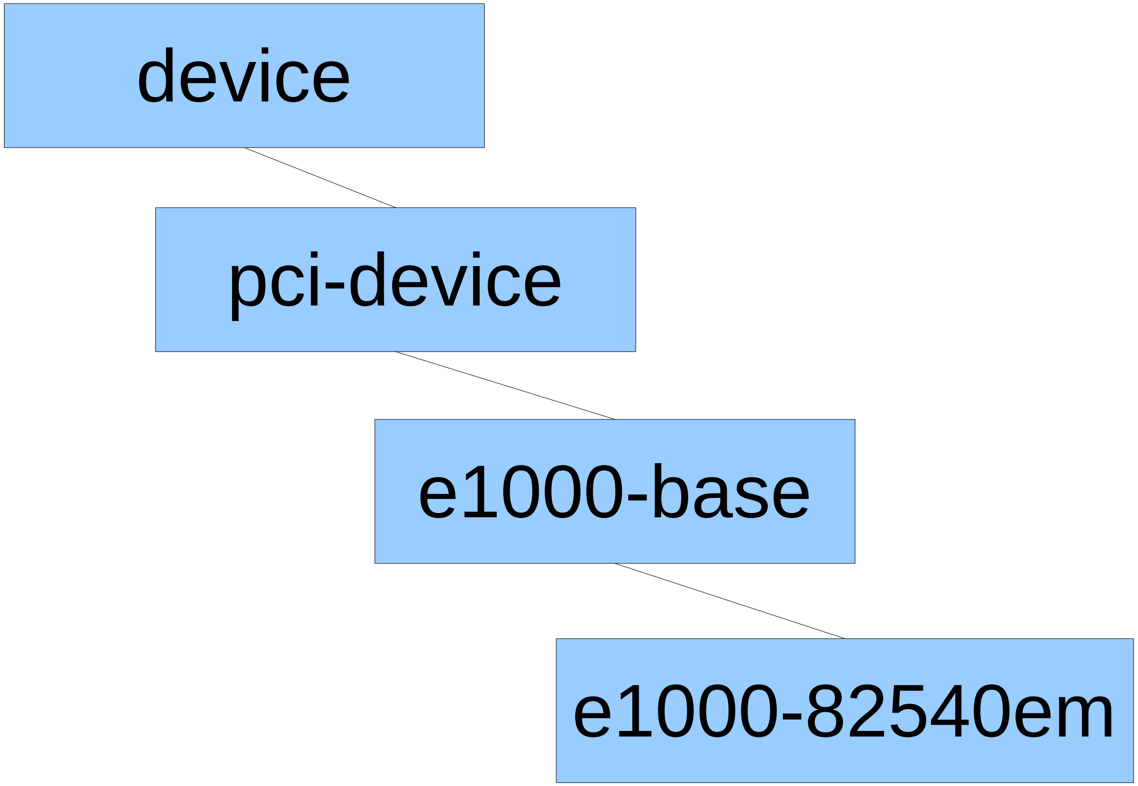
- PCI, USB, I2C

Examples where this approach was used:

- Rocker OpenFlow network switch
- NVDIMM persistent memory
- NVMe PCI flash storage controller

QEMU device model

Object-oriented device model:



device

pci-device

e1000-base

e1000-82540em

Allows you to focus on unique device functionality instead of common behavior.

Memory API

Device register **memory regions** for PIO and MMIO hardware register access:

```
static const MemoryRegionOps vmport_ops = {
    .read = vmport_ioport_read,
    .write = vmport_ioport_write,
    .impl = {
        .min_access_size = 4,
        .max_access_size = 4,
    },
    .endianness = DEVICE_LITTLE_ENDIAN,
};
```

```
memory_region_init_io(&s->io, OBJECT(s),
         &vmport_ops, s, "vmport", 1);
isa_register_ioport(isadev, &s->io, 0x5658);
```

Interrupts

Devices use bus-specific methods to raise interrupts:

```
void pci_set_irq(PCIDevice *pci_dev, int level)
```

QEMU emulates interrupt controllers and injecting interrupts

- Interrupt controller state is updated
- Guest CPU interrupt vector is taken

More information on device emulation

Plenty of examples in QEMU `hw/` directory

- Learn from existing devices
- Documentation is sparse

Post patches to qemu-devel@nongnu.org for feedback

- Guidelines for submitting patches: http://qemu-project.org/Contribute/SubmitAPatch

Error injection

How do I exercise rare error code paths in kernel?

QEMU can simulate error conditions

- Without overheating or damaging real hardware
- Without reaching into a box to pull cables

Simple scenarios:

- Test hot unplug while device is in use

```
(qemu) device_del e1000.0
```

Advanced error injection

QEMU's block layer has an error injection engine:

```
[set-state]
state = "1"
event = "write_aio"
new_state = "2"

[inject-error]
state = "2"
event = "read_aio"
errno = "5"
```

This script fails disk reads after the first write.

Documentation: `docs/blkdebug.txt`

Questions?

Email: stefanha@redhat.com

IRC: stefanha on #qemu irc.oftc.net

Blog: http://blog.vmsplice.net/

QEMU: http://qemu-project.org/

Slides available on my website: http://vmsplice.net/