Architecture of the Kernel-based Virtual Machine (KVM)

Jan Kiszka, Siemens AG, CT T DE IT 1
Corporate Competence Center Embedded Linux
jan.kiszka@siemens.com
Agenda

- Introduction
- Basic KVM model
- Memory
- API
- Optimizations
- Paravirtual devices
- Outlook
Virtualization of Commodity Computers
Virtualizing the x86 Instruction Set Architecture

**x86 originally virtualization “unfriendly”**
- No hardware provisions
- Instructions behave differently depending on privilege context
- Performance suffered on trap-and-emulate
- CISC nature complicates instruction replacements

**Early approaches to x86 virtualization**
- Binary translation (e.g. VMware)
  - Execute substitution code for privileged guest code
  - May require substantial replacements to preserve illusion
- CPU paravirtualization (e.g. Xen)
  - Guest is aware of instruction restrictions
  - Hypervisor provides replacement services (hypercalls)
  - Raised abstraction levels for better performance
Hardware-assisted x86 CPU Virtualization

Two variants
- Intel's Virtualization Technology, VT-x
- AMD-V (aka Secure Virtual Machine)

Identical core concept
Advent and Evolution of KVM

Introduced to make VT-x/AMD-V available to user space
- Exposes virtualization features securely
- Interface: /dev/kvm

Merged quickly
- Available since 2.6.20 (2006)
- From first LKML posting to merge: 3 months
- One reason: originally 100% orthogonal to core kernel

Evolved significantly since then
- Ported to further architectures (s390, PowerPC, IA64)
- Always with latest x86 virtualization features
- Became recognized & driving part of Linux
The KVM Model

Processes can create virtual machines

VMs can contain
- Memory
- Virtual CPUs
- In-kernel device models

Guest physical memory part of creating process' address space

VCPUs run in process execution contexts
- Process usually maps VCPUs on threads
Architectural Advantages of the KVM Model

Proximity of guest and user space hypervisor
- Only one address space switch: guest ↔ host
- Less rescheduling

Massive Linux kernel reuse
- Scheduler
- Memory management with swapping (though you don't what this)
- I/O stacks
- Power management
- Host CPU hot-plugging
- ...

Massive Linux user land reuse
- Network configuration
- Handling VM images
- Logging, tracing, debugging
- ...
VCPU Execution Flow (KVM View)

- Update context, raise IRQs
- Run
  - Save Host, Load Guest State
  - Update guest state

- Save Guest, Load Host State
  - Execute native guest code
  - Handle
    - In-Kernel I/O
    - [vMMU]
    - ...
  - Handle Host IRQ
  - Handle Host IRQ
  - Handle Signal
    - I/O
    - Invalid states
    - ...

- VM entry
- VM exit (with reason)
KVM Memory Model

Slot-based guest memory
- Maps guest physical to host virtual memory
- Reconfigurable
- Supports dirty tracking

In-Kernel Virtual MMU

Coalesced MMIO
- Optimizes guest access to RAM-like virtual MMIO regions

Out of scope
- Memory ballooning (guest ↔ user space hypervisor)
- Kernel Same-page Merging (not KVM-specific)
KVM API Overview

Step #1: open /dev/kvm

Three groups of IOCTLs
- System-level requests
- VM-level requests
- VCPU-level requests

Per-group file descriptors
- /dev/kvm fd for system level
- Creating a VM or VCPU returns new fd

mmap on file descriptors
- VCPU: fast kernel-user communication segment
  - Frequently read/modified part of VCPU state
  - Includes coalesced MMIO backlog
- VM: map guest physical memory (deprecated)
Basic KVM IOCTLs

KVM_CREATE_VM

KVM_SET_USER_MEMORY_REGION
KVM_CREATE_IRQCHIP / ...PIT (x86)
KVM_CREATE_VCPU

KVM_SET_REGS / ...SREGS / ...FPU / ...
KVM_SET_CPUID / ...MSRS / ...VCPU_EVENTS / ... (x86)
KVM_SET_LAPIC (x86)
KVM_RUN
Optimizations of KVM

Hardware evolves quickly
- Near-native performance in guest mode
- Decreasing costs of mode switches
- Additional features avoid software solutions, thus exits
  - Nested page tables
  - TLB tagging
  - APIC virtualization
  - ...

What will continue to consume cycles?
- Code path between VM-exit and VM-entry
- Mode switches, i.e. the need to exit at all
Lightweight vs. Heavy-weight VM-Exits

Exits cost time!
- Basic state switch in hardware
- Additional state switches in software
- Analyze exit reason

- In-kernel APIC
- In-kernel IO-APIC + PIC
- Coalescing MMIO
- In-kernel instruction interpreter (detect MMIO access)
- In-kernel network stub (vhost-net)

- Software-managed state switch
- Hardware state switch

>10.000 cycles
Optimizing Lightweight Exits

Let's get lazy!
- Perform only partial state switches
- Complete at latest possible point
- Late restoring for guest and host state

Candidates (x86)
- FPU
- Debug registers
- Model-specific registers (MSRs)

Requirements
- Usage detection when in guest mode
  - Depends on hardware support
- Demand detection while in host mode
  - Preemption notifiers
  - User-return notifier
Lazy MSR Switching

Why is this possible?
- Some MSRs unused by Linux
- Some MSRs only relevant when in user space
- Some are identical for host & guest

Approach
- Keep guest values of certain MSRs until...
  - sched-out fires
  - KVM_RUN IOCTL returns
- Keep others until user-return fires (Intel only)

Optimizations are vendor-specific

Exemplary saving:
- 2000 cycles for guest → idle thread → guest
Paravirtual Devices

Advantages
- Reduce VM exits or make them lightweight
- Improve I/O throughput & latency (less emulation)
- Compensates virtualization effects
- Enable direct host-guest interaction

Available interfaces & implementations
- virtio (PCI or alternative transports)
  - Network
  - Block
  - Serial I/O (console, host-guest channel, …)
  - Memory balloon
  - File system (9P)
- Clock (x86 only)
  - Via shared page + MSRs
  - Enables safe™ TSC guest usage

user space
business
(primarily)

KVM
business
An Almost-In-Kernel Device – vhost-net

Goal: high throughput / low latency guest networking
- Avoid heavy exits
- Reduce packet copying
- No in-kernel QEMU, please!

The vhost-net model
- Host user space opens and configures kernel helper
- virtio as guest-host interface
- KVM interface: eventfd
  - TX trigger → ioeventfd
  - RX signal → irqfd
- Linux interface vie tap or macvtap

Enables multi-gigabit throughput
What's next?

**Generic Linux improvements**
- Transparent huge pages (mm topic)
- NUMA optimizations (scheduler topic)

**Improve spin-lock-holder preemption effects**

**Zero-copy & multi-queue vhost-net**

**Further optimize exits**
- Instruction interpretation (hardware may help)
- Faster in-kernel device dispatching

**Nested virtualization as standard feature**
- AMD-V bits already merged and working
- VT-x more complex but likely solvable

**Hardware-assisted virtualization on non-x86**
- PowerPC ISA 2.06
- ARMv7-A “Eagle” extensions

…
Thanks you for listening!

Questions?