KVM,
QEMU
Yiying Zhang
Outline

- Recap on hardware-assisted virtualization
- KVM
- QEMU
Discussion

• Comparing VMware ESX, Xen, KVM (+QEMU), what are their pros and cons? Why do you think AWS went from Xen to KVM?

• The initial goal of virtualization (allowing one type of OS/architecture to run on another type) seems to be less and less relevant nowadays. Do you agree? Can you think of some use cases where there’s still such a need?
Two new modes of execution (orthogonal to protection rings)

- VMX root mode: same as x86 without VT-x
- VMX non-root mode: runs VM, sensitive instructions cause transition to root mode, even in Ring 0

New hardware structure: VMCS (virtual machine control structure)

- One VMCS for one virtual processor
- Configured by VMM to determine which sensitive instructions cause VM exit
- Specifies guest OS state
[Lec3] Comparison of Pre VT-x and Post VT-x

- **Hardware w/o VT-x**
  - **Hypervisor** (Ring 0)
  - **Guest OS** (Ring 1)
  - **Guest Applications** (Ring 3)

- **Hardware w/ VT-x**
  - **Hypervisor** (Ring 0)
  - **Host Applications** (Ring 3)
  - **VMX root** (Ring 0)
  - **VMX non-root** (Ring 3)

- **Hypervisor** (Ring 0)
  - **Guest OS** (Ring 1)
  - **Guest Applications** (Ring 3)
[Lec4] Hardware-Assisted Memory Virtualization

- VPN
- PPN
- EPT
- MPN
- PT
- VA
- PA

Guest VM

VMM

Non-root mode

Root mode
KVM: Linux-based Virtualization

Avi Kivity
avi@qumranet.com

Columbia University Advanced OS/Virtualization course
At a glance

- KVM – the Kernel-based Virtual Machine – is a Linux kernel module that turns Linux into a hypervisor
- Requires hardware virtualization extensions
- Supports multiple architectures: x86 (32- and 64- bit) s390 (mainframes), PowerPC, ia64 (Itanium)
- Competitive performance and feature set
- Advanced memory management
- Tightly integrated into Linux
The KVM approach

- Reuse Linux code as much as possible
- Focus on virtualization, leave other things to respective developers
- Integrate well into existing infrastructure, codebase, and mindset
- Benefit from semi-related advances in Linux
KVM model benefits

- Reuse scheduler, memory management, bringup
- Reuse Linux driver portfolio
- Reuse I/O stack
- Reuse management stack
KVM Process Model

task

task
guest

task
guest

kernel
KVM Execution Model

- Three modes for thread execution instead of the traditional two:
  - User mode
  - Kernel mode
  - Guest mode

- A virtual CPU is implemented using a Linux thread

- The Linux scheduler is responsible for scheduling a virtual cpu, as it is a normal thread
KVM Execution Model

- **Native Guest Execution**
- **Kernel exit handler**
- **Switch to Guest Mode**
- **ioctl()**
- **Userspace exit handler**
- **Guest**
- **Kernel**
- **Userspace**
KVM Execution Model

- Guest code executes natively
  - Apart from trap'n'emulate instructions
- Performance critical or security critical operations handled in kernel
  - Mode transitions
  - Shadow MMU
- I/O emulation and management handled in userspace
  - Qemu-derived code base
  - Other users welcome
KVM Memory Model

- User Address Space
- Kernel Address Space
- VMM userspace code and data
- Guest physical address space
KVM Memory Model

- Guest physical memory is just a chunk of host virtual memory, so it can be
  - Swapped
  - Shared
  - Backed by large pages
  - Backed by a disk file
  - COW'ed
- The rest of the host virtual memory is free for use by the VMM
  - Low bandwidth device emulation
  - Management code
Linux Integration

- Preemption (and voluntary sleep) hooks: preempt notifier
- Swapping and other virtual memory management: mmu notifiers
Preempt Notifiers

- Linux may choose to suspend a vcpu's execution
- KVM runs with some guest state loaded while in kernel mode (FPU, etc.)
- Need to restore state when switching back to user mode
- Solution: Linux notifies KVM whenever it preempts a process that has guest state loaded
  - ... and when the process is scheduled back in
- Allows the best of both worlds
  - Low vmexit latency
  - Preemptibility, sleeping when paging in
Preempt notifiers

- External interrupt or trap
- Guest
- VMM process in host kernel
- Scheduler
- Other process
- Context switch
- Restore host state
- Restore guest state
- Context switch
- Restore host state
- Restore guest state
- Context switch
- Other process
MMU Notifiers

- Linux doesn't know about the KVM MMU
- So it can't
  - Flush shadow page table entries when it swaps out a page (or migrates it, or ...)
  - Query the pte accessed bit when determines the recency of a page
- Solution: add a notifier
  - for tlb flushes
  - for accessed/dirty bit checks
- With MMU notifiers, the KVM shadow MMU follows changes to the Linux view of the process memory map
Paravirtualization

- Yesterday's hot topic
  - Needed for decent MMU performance without two-dimensional paging
  - Intrusive

- KVM has modular paravirtualization support
  - Turn on and off as needed by hardware
  - Still needs hardware virtualization extensions

- Supported areas
  - Hypercall-based, batched mmu operations
  - Clock
Most devices emulated in userspace
  - With fairly low performance
Paravirtualized I/O is the traditional way to accelerate I/O
Virtio is a framework and set of drivers:
  - A hypervisor-independent, domain-independent, bus-independent protocol for transferring buffers
  - A binding layer for attaching virtio to a bus (e.g. pci)
  - Domain specific guest drivers (networking, storage, etc.)
  - Hypervisor specific host support
KVM Conclusion

• Tight integration with Linux

• The KVM module is relatively simple, with most of the functionalities already implemented in the Linux kernel

• KVM relies on hardware virtualization support (which is prevalent now)

• KVM performance is generally good

• Increasing popularity with AWS making the big move from Xen to KVM
Outline

• Recap on hardware-assisted virtualization
• KVM
• QEMU
QEMU

• Open-source Type-2 hypervisor

• Originally written by Fabrice Bellard (author of the paper you read)

• Full virtualization that supports cross-architecture conversion

• Using dynamic binary translation

• Supports two modes
  • User-mode: runs Linux process in one architecture on another (host) arch
  • System emulation: runs full guest OS
QEMU Binary Translation

- Functional simulation
  - Simulate what a processor does, not how it does it
  - Supports many devices (serial, Ethernet, etc.) and many architectures

- Dynamic binary translation
  - Not an interpreter (interpreter executes one inst at a time and very slow)
  - QEMU converts code as needed and stores converted code in a translation cache
  - Code translated one block at a time

- A lot of similarities to VMware’s dynamic binary translation
Converting Code across Architectures
QEMU Tiny Code Generator (TCG)
QEMU Binary Translation

- Tiny Code Generation
- Micro-operations
- Fixed register allocation

Source: https://lugatgt.org/content/qemu_internals/downloads/slides.pdf
QEMU Dynamic Binary Translation Stage 1

```
push %ebp
mov %esp,%ebp
not %eax
add %eax,%edx
mov %edx,%eax
xor $0x55555555,%eax
pop %ebp
ret
```
QEMU Dynamic Binary Translation Stage 2

Guest Code

\[ \text{gen\_intermediate\_code()}} \]

TCG Operations

\[ \text{tcg\_gen\_code()}} \]

Host Code

\[
\ld_{i32} \text{ tmp2, env, } 0\times10 \\
\text{qemu\_ld32u tmp0, tmp2, } 0\timesffffffff \\
\ld_{i32} \text{ tmp4, env, } 0\times10 \\
\text{movi}_{i32} \text{ tmp14, } 0\times4 \\
\text{add}_{i32} \text{ tmp4, tmp4, tmp14} \\
\text{st}_{i32} \text{ tmp4, env, } 0\times10 \\
\text{st}_{i32} \text{ tmp0, env, } 0\times20 \\
\text{movi}_{i32} \text{ cc\_op, } 0\times18 \\
\text{exit\_tb } 0\times0
\]
QEMU Dynamic Binary Translation Stage 3

Guest Code

\text{gen\_intermediate\_code()}

TCG Operations

\text{tcg\_gen\_code()}

Host Code

\ldots

\begin{verbatim}
mov 0x10(%ebp),%eax
mov %eax,%ecx
mov (%ecx),%eax
mov 0x10(%ebp),%edx
add $0x4,%edx
mov %edx,0x10(%ebp)
mov %eax,0x20(%ebp)
mov $0x18,%eax
mov %eax,0x30(%ebp)
xor %eax,%eax
jmp 0xba0db428
\end{verbatim}

/*This represents just the ret instruction!*/
QEMU Binary Translation

- Translation block and trans cache
  - cpu exec() called each time around main loop. Program executes until an unchained block is encountered. Returns to cpu exec() through epilogue.
- Translation Block chaining
- Condition code optimization
- Async interrupt

Source: https://lugatgt.org/content/qemu_internals/downloads/slides.pdf
Memory Virtualization

- Software MMU translates virtual memory address to physical one at every memory access
- Caches the translation
- Translation blocks indexed by physical addresses
Storage Virtualization

- Application
- File system & block layer
- Driver
- Hardware emulation
- Image format (optional)

- Application and guest kernel work similar to bare metal.
- Guest talks to QEMU via emulated hardware.

- QEMU performs I/O to an image file on behalf of the guest.
- Host kernel treats guest I/O like any userspace application.

[ source: Stefan Hajnoczi, IBM Linux Technology Center, 2011 ]
QEMU+KVM

- When the guest architecture is the same as the physical host architecture (and on a hardware-assisted virtualization platform)

- Can use KVM as the “main” hypervisor (allows native execution of guest code (in guest mode) etc.)

- Calls into QEMU when KVM cannot handle certain things (basically hardware device emulation)
Discussion

• Comparing VMware ESX, Xen, KVM (+QEMU), what are their pros and cons? Why do you think AWS went from Xen to KVM?

• The initial goal of virtualization (allowing one type of OS/architecture to run on another type) seems to be less and less relevant nowadays. Do you agree? Can you think of some use cases where there’s still such a need?