Lecture 7: I/O Virtualization Techniques

Mythili Vutukuru
IIT Bombay
Spring 2021
Techniques for I/O virtualization

• Guest OS cannot get full access to I/O devices
  • VMM must share I/O device access across guests
• Two ways to virtualize I/O devices:
  • **Emulation**: I/O access in guest traps to VMM, which performs I/O
  • **Direct I/O** or **device passthrough**: a slice of device is assigned directly to guest
• Many optimizations exist, only basics discussed here
Communication between OS and device

- Device memory exposed as registers (command, status, data etc.)
  - I/O happens by reading/writing this memory
  - E.g., write command into device register to begin I/O

- OS can read/write device registers in two ways:
  - Explicit I/O: in/out instructions in x86 can write to device memory
  - Memory mapped I/O: Some memory addresses are assigned to device memory and are not to RAM. I/O happens by reading/writing this memory.

- Accessing device memory (via explicit I/O or memory mapped I/O) can be configured to trap to VMM

- Device raises interrupt when I/O completes (alternative to polling)
  - Modern I/O devices perform DMA (Direct Memory Access) and copy data from device memory to RAM before raising interrupt
  - Device driver provides physical address of DMA buffers to device
QEMU/KVM I/O handling

1. User application makes system call to guest OS

2. Guest OS exits when it attempts I/O

3. KVM returns to QEMU VCPU thread (exit info in kvm_run)

4. QEMU I/O thread initiates I/O in host OS (may block)

5. VCPU thread resumes VM

6. KVM resumes guest VM

7. Root mode

8. Return to user process

Guest OS

Guest application

KVM (kernel module)
Host OS

QEMU (Userspace process)

Guest VM physical memory

VCPU-0

I/O thread

Separate I/O threads to handle I/O ops

VCPU threads do not block for I/O

Guest VM resumes and can context switch to another process

Ring 0

Ring 3
QEMU/KVM interrupt handling

1. Interrupt occurs, user process goes to kernel mode

2. Guest OS configured in VMCS to exit on interrupts

3. Interrupts for host are handled by host OS

4. Interrupt for guest OS are injected into guest via VMCS

5. Guest OS handles its interrupts normally

QEMU (Userspace process)

- QEMU/KVM interrupt handling
- QEMU (Userspace process)
- KVM (kernel module)
- Guest VM physical memory
- VCPU-0
- I/O thread

Guest OS

- Guest application
- Interrupt for guest OS are injected into guest via VMCS
- Interrupt occurs, user process goes to kernel mode
- Guest OS handles its interrupts normally
- 3. Interrupts for host are handled by host OS
Full virtualization VMM architecture

1. Guest user app makes system call to perform I/O
2. Privileged action traps to VMM
3. VMM exits to host OS to handle I/O
   - Some traps can be handled by VMM without world switch, e.g., exit only once per batch of I/O requests
4. VMM kernel driver or userspace process handle I/O requests via emulation
5. Interrupts handled by host and injected into guest

Host OS context
- VMM userspace process
- Guest VM physical memory

VMM context
- Guest application
- Guest OS
- VMM (guest OS traps here)
- VMM kernel driver (Host OS)
QEMU/KVM virtio optimization

1. User application makes system call to guest OS
2. Special virtio “frontend” device driver places requests in shared memory
3. Guest OS exits after a batch of requests accumulate
4. KVM and QEMU “backend” access requests from shared ring. Virtual interrupt raised in guest after batch of responses.

QEMU (Userspace process)
- VCPU-0
- I/O thread
- Guest VM physical memory

KVM (kernel module)
- Host OS

Shared ring
- Memory copy avoided
- Batching of requests, interrupts
- Standardized across devices
- High performance

Guest application

Guest OS
Device passthrough or Direct I/O

• More efficient than device emulation

• Example: SR-IOV (Single Root IO Virtualization) in network devices
  • Network card has one physical function (PF) and many virtual functions (VFs)
  • PF managed by host OS, each VF assigned to one guest VM
  • Each VF is like a separate NIC, and is bound to a guest VM
  • Packets destined to the MAC address of VM are switched to corresponding VF
SR-IOV

- SR-IOV NIC communicates directly with device driver in guest OS
  - Packets do not go to the host OS stack at all
  - Packets switched at Layer-2 using VM virtual device’s MAC address
  - Packets DMA’ed directly into guest VM memory, host OS not involved
  - But, interrupts may still cause VM exit (interrupt can be for host too)

- Challenge: when guest device driver provides DMA buffers to VF, it can only provide guest physical addresses (GPA) of the buffer
  - NIC cannot access the DMA buffer memory using GPA alone

- SR-IOV capable NICs have an inbuilt MMU (IOMMU) to translate from GPA to HPA
Summary

• Techniques for I/O virtualization
  • Device emulation
  • Virtio optimization
  • Device passthrough or direct I/O (SR-IOV)