Advanced CPU Virtualization

Luca Abeni

`luca.abeni@santannapisa.it`

December 21, 2022

Popek and Goldberg's Virtualization

- Basically, trap and emulate
  - Execute guest code at low privilege level
  - Execution of privileged instructions causes exceptions / faults
  - The hypervisor running at high privilege level can emulate such instructions (exeption handler)
- Works if all sensitive instructions are privileged
  - For some architectures (x86, ARM, ...) this requirement is not satisfiled
  - Hardware extensions for virtualization
- Do not consider devices (interrupts), paging, etc...

Hardware Assisted Virtualization

- Needed if the original hw architecture is not virtualizable...
  - ...Or to improve performance
  - Paging support, interrupt virtualization, ...
- Must somehow keep compatibility with the original hw architecture
- First idea: introduce a new privilege level
  - Hypervisor privilege level, more privileged than system (kernel) privilege level
  - All sensitive instruction trap to hypervisor level (even if they do not trap to kernel privilege level)

Hypervisor Privilege Level

- Privilege level -1 (privilege level 0 is kernel)
- Designed to comply with Popek and Goldberg's requirements
- Advantage: trap and emulate can be implemented!
  - Writing simple hypervisors is easy
- But there are some disadvantages...
  - The hypervisor execution environment is different from the kernel's one
    - Difficult to re-use existing kernel code, problem for hosted hypervisors
  - Every sensitive instruction is emulated
    - Exception / trap / VM exit $\rightarrow$ overhead!

Beyond Popek and Goldberg

- Should we emulate in software every sensitive instruction?
  - If the hardware "just complies" with Popek and Goldberg requirements, yes!
  - But the hardware can do better...
- Idea: keep a copy of the CPU state, and allow the guest instructions to access the copy
  - So, we do not need to emulate all of them!
  - The CPU in a "special execution mode" will not access the real state, but only the shadow copy! Without the hypervisor intervention
- Two modes of operation: one for the host and one for the guests

Shadow CPU State

- Host execution mode: the “real CPU state” is accessed
  - Can be identical to a CPU without virtualization
- Guest execution mode: the “shadow copy” is accessed (one copy per guest)
  - Data structure in memory, containing a private copy of the CPU state
  - The guest can access it without compromising security and performance
  - The hypervisor can access / modify / control all of the copies
- Advantage: performance
- Disadvantage: much more complex to use / program

Intel VT-x

- Intel VT-x technology follows the second approach for hw assisted virtualization (shadow guest state)
  - Distinction between “root mode” and “non-root mode”
  - Both the two execution modes have the traditional intel privilege levels
  - In root mode, the CPU is almost identical to a “traditional” intel CPU
- In non-root mode, the shadow guest state is stored in a Virtual Machine Control Structure
  - The VMCS actually also contains configuration data and other things

Using Intel VT-x

- First, check if the CPU supports it
  - Use the `cpuid` instruction to check for VT-x
  - Access a machine specific register to check if VT-x is enabled
    - If it is not, try to enable it - if the BIOS did not lock it
- Then, initialize VT-x and enter root mode
  - Set a bit in `cr4`
  - Assign a VMCS region to root mode
  - Execute `vmxon`
- Now, the difficult part begins...

Creating VT-x VMs

- Once in root mode, it is possible to create VMs...
  - Allocate a VMCS for the VM
  - Assign it to the VM (`vmptrld` instruction)
  - Configure the VMCS
  - Start the VM (`vmlaunch` instruction)
- VMCS configuration: host / guest state and control information)
  - Guest state: initialization of the “shadow state” for the guest
  - Host state: CPU state after VM exit
  - Control: configure which instructions cause VM exit, the behaviour of some control registers, ...

VMCS Setup - I

- Configuring the guest state, it is possible to execute real-mode, 32bit or 64bit guests, controlling paging, etc...
  - It is possible to configure an inconsistent guest state
  - `vmlaunch` will fail
- Control information: VM exits (which instructions to trap), some “shadow control registers”, ...
  - Example: guest access to `cr0`
  - Possible to decide if the guest “sees” the host `cr0`, the guest `cr0`, or some “fake value” configured by the hypervisor
  - This is configurable bit-per-bit

VMCS Setup - II

- VMCS configuration and setup is not easy
  - Also, requires to know a lot of details about the CPU architecture
- Starting a VM (even a “simple” one) requires some work!
  - I skipped the details about nested page tables...
- On the other hand, it is easier to build hosted hypervisors

The Kernel Virtual Machine

- Kernel Virtual Machine (`kvm`): Linux driver for VT-x
  - Actually, it also supports AMD's `SVM`
- Hides most of the dirty details in setting up a hardware-assisted VM
  - Also checks for consistency of the guest state, etc...
- Started as an x86-only driver, now supports more architectures
  - With some "tricks", for example for ARM
- Accessible through a `/dev/kvm` device file
  - Allows to use the "standard" UNIX permission management

Using kvm

- First, check if the CPU is supported by kvm
  - Open `/dev/kvm`
  - This also checks for permissions
- Then, check the kvm version
  - Use the `KVM_GET_API_VERSION` ioctl
  - Compare the result with `KVM_API_VERSION`
- Now, create a VM (`KVM_CREATE_VM` ioctl)
  - Without memory and virtual CPUs
  - Memory must be added later
    - `KVM_SET_USER_MEMORY_REGION` ioctl
  - Virtual CPUs must be created later
  - `KVM_CREATE_VCPU` ioctl

kvm Virtual CPUs

- Created after creating a VM, and associated to it
  - Allow to create multi-(v)CPU VMs
- After creating a virtual CPU, its state must be initialized
  - Allow to start VMs in real-mode, protected mode, long mode, etc...
  - Done by setting registers and system registers (`KVM_{GET,SET}_REGS` and `KVM_{GET,SET}_SREGS` ioctls)
- Interaction through memory region shared between kernel and application (`mmap()`)

Virtual CPU Setup

- Before starting a VM, the state of each virtual CPU must be properly initialized
- RM, 32bit PM (with or without paging), 64bit “long mode” (paging is mandatory), ...
  - Properly initialize some control registers (`cr0`, `cr3` and `cr4`, ...)
  - In PM, setup segments
    - No need to setup a GDT, kvm can do it for us!!!
  - Page tables configuration
- kvm checks the consistency of this configuration
  - Example: if we configures segments, PM must be enabled in `cr0`

Running the VM

- A thread for each virtual CPU
- Loop on the `KVM_RUN` ioctl
  - The ioctl can return because of error
    - Check for `EINTR` or `EAGAIN`
  - Or because of a VM exit (`KVM_EXIT`)
    - Check the exit reason (`KVM_EXIT_xxx`)...
    - ...And properly serve it!
- Virtual CPU execution can be interrupted by signals
- Virtual devices implemented serving I/O exits or accesses to unmapped memory