Virtual Machines

Luca Abeni
`luca.abeni@santannapisa.it`

November 24, 2021

Virtual Machine Abstraction

- Remember? Virtual Machine == efficient, isolated duplicate of a physical machine
  - Virtual devices, some virtual CPUs, some amount of (virtual!) memory, ...
  - Supports the execution of OS kernel or bare-metal applications
    - Users can (and have to!) install their own kernel, etc...
- Execution environment essentially identical to the physical machine
- A “virtual machine monitor” or “hypervisor” is in full control of physical resources

Virtual Machine Implementation

- Software (maybe hardware-assisted) implementation of an abstract machine
  - Understands the hardware machine language and implements some devices
  - Plus, eventually some additional machine instuctions: *hypercalls*
- Requirements

1. Programs running in a VM should not see differences respect to real hw
2. Virtualization should be efficient
3. Programs should not be able to access resources outside of the VM

Implemented Abstraction

- Abstract machine $\mathcal{M}_{\mathcal{L}}$: understand language $\mathcal{L}$ (same as the physical machine language, plus hypercalls)
  - Can execute sequences of instructions written in $\mathcal{L}$
- So, $\mathcal{M}_{\mathcal{L}}$ has to:
  - Execute some “elementary operations”
  - Manage the execution flow
  - Move data from / to (virtual) memory and devices
  - Take care of memory management
- All respecting the three requirements mentioned above

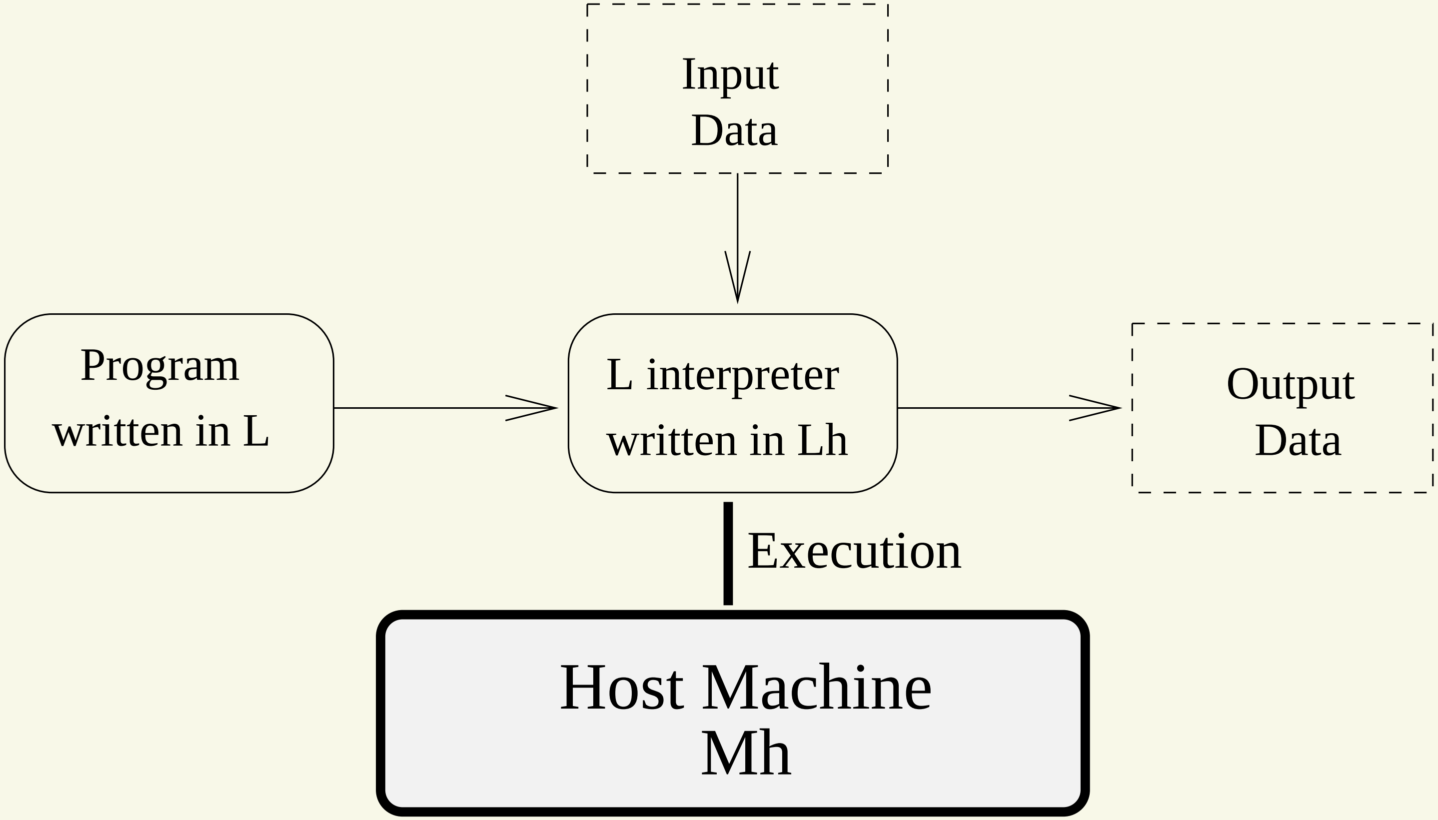
Implementing a Language

- $\mathcal{M}_{\mathcal{L}}$ undestands its machine language $\mathcal{L}$
    - One single machine language per abstract machine
- $\mathcal{L}$ can be executed by multiple different abstract machines
    - Might differ in implementation, data structures, ...
- Implementation of language $\mathcal{L}$: abstract machine $\mathcal{M}_{\mathcal{L}}$ that understands programs written in language $\mathcal{L}$
    - Implementation in hw, sw, firmware, ...

Software Implementation

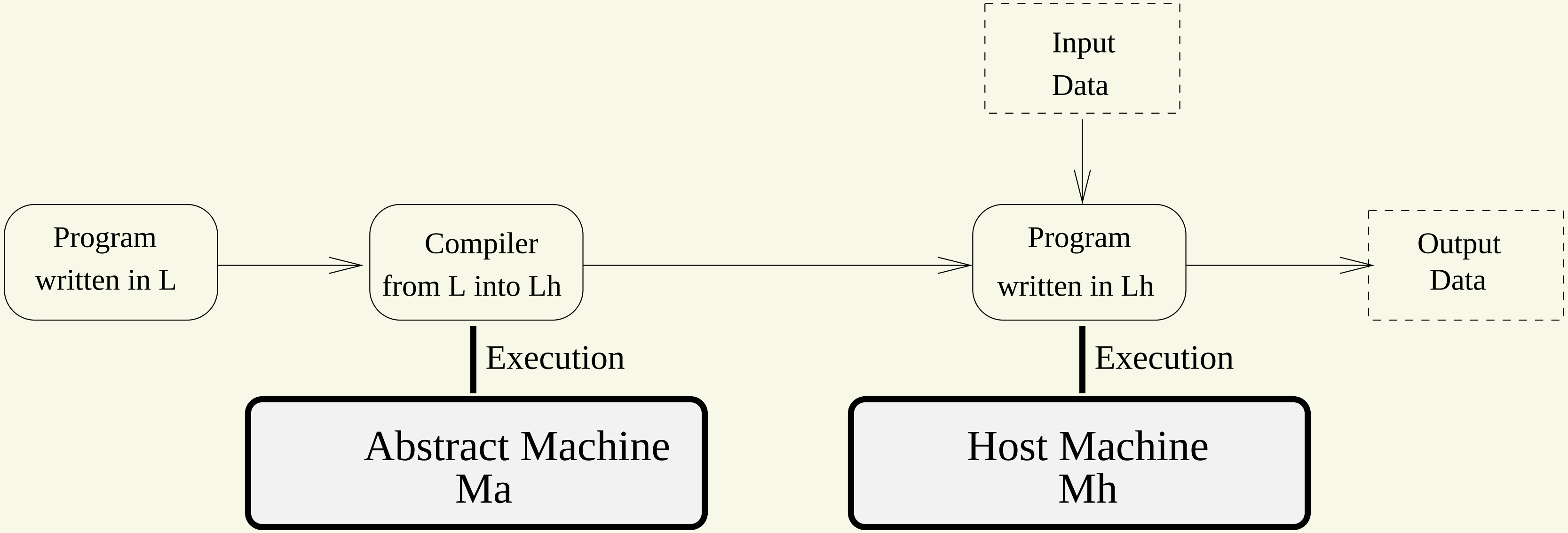
- $\mathcal{M}_{\mathcal{L}}$ in software (can execute programs written in $\mathcal{L}$)
- Executes on a Host Machine $\mathcal{M}h_{\mathcal{L}h}$ (having machine language $\mathcal{L}h$)
- Two possible implementations: *interpreter* or *compiler*
  - Interpreter: program written in $\mathcal{L}h$ that understands and executes $\mathcal{L}$
    - Implements the fetch/decode/load/exec/save cycle
  - Compiler: program translating other programs from $\mathcal{L}$ to $\mathcal{L}h$

Pure Interpreters



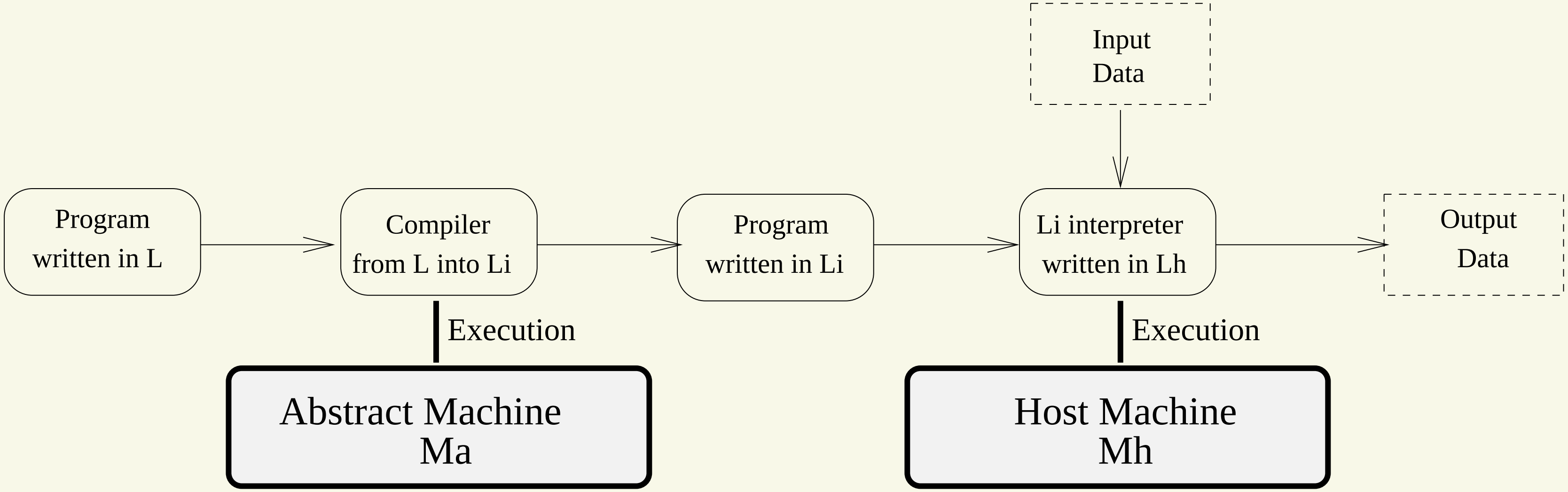
- Interpreter: program written in $\mathcal{L}h$ (executes on $\mathcal{M}h_{\mathcal{L}h}$) understanding programs written in $\mathcal{L}$
- Translates $\mathcal{L}h$ in $\mathcal{L}$ “instruction by instruction”

Pure Compilers



- Translates the whole program from $\mathcal{L}$ to $\mathcal{L}h$ *before executing it*
- Translation performed by a dedicated program, the Compiler
  - Compiler: not necessarely written in $\mathcal{L}h$
  - Can execute on an abstract machine $\mathcal{M}a$ different from $\mathcal{M}h_{\mathcal{L}h}$

Hybrid Implementation



Input Data

Program written in L

Compiler from L into Li

Program written in Li

Li interpreter written in Lh

Output Data

Execution

Execution

Abstract Machine Ma

Host Machine Mh

- Not a pure compiler nor a pure interpreter
- Compiler translate in an *intermediate language* $\mathcal{L}i$
- Interpreter executes on $\mathcal{M}h_{\mathcal{L}h}$ programs written in $\mathcal{L}i$
    - Java: compiler $\rightarrow$ bytecode, then JVM
    - C: compiler generally produces code that needs OS and runtime to execute

CPU Emulators

- CPU Emulator: software implementation of the fetch/decode/load/exec/save cycle
  - Can be an interpreter, some sort of compiler, or a hybrid implementation
  - Different complexity / performance / flexibility trade-offs depending on the implementation strategy
- Performance penalty respect to direct execution on the emulated CPU
- Allows to emulate target CPU architectures different from the host CPU architecture
  - $\mathcal{L}$ and $\mathcal{L}h$ can be different
  - No constraints on the emulated or host ISA

Interpreting CPU Instructions

- Simplest CPU emulator: software cycle interpreting CPU instructions
  - Read CPU instructions one by one ← according to the syntax defined in ISA manuals
    - Machine language instructions can have fixed size (RISC) or variable size (x86, ...)
  - Decode and execute (eventually loading or saving data) modifying the emulator's state
- Can be easily implemented reading the CPU documentation
- Example: Bochs (`http://bochs.sf.net`)

Compiling Blocks of CPU Instructions

- Compiler-based approach: just-in-time translation of CPU instructions from $\mathcal{L}$ to $\mathcal{L}h$
  - More complex than a CPU interpreter, but can provide better performance
  - Example: loop translated 1 time and then execute multiple times at near-native speed
- Additional issues with self-modifying code and similar...
- Example: qemu
  - Contains a “Tiny Code Generator” (TCG) $\rightarrow$ sort of simple compiler

Qemu TCG

- Compile a “Translation Block” (TB) when needed, and then execute compiled instructions
- Different “frontends” for each supported target (language $\mathcal{L}$)
  - Convert machine instructions of $\mathcal{L}$ into “TCG instructions”
- Different “backends” for each supported host architecture (host language $\mathcal{L}h$)
  - Convert TCG instructions into machine instructions of $\mathcal{L}h$
- Issues: identify TBs, invalidate them when needed, etc...

CPU Virtualization

- Instead of emulating a CPU implementing $\mathcal{M}_{\mathcal{L}}$ in software, execute target instructions in the host
  - This implies $\mathcal{L} == \mathcal{L}h$!!!
- How can the monitor be in control of physical resources?
  - If the guest has control of the virtual machine...
  - ...It risks to have full control of the physical machine too!!!
- Only some of the guest instructions can be directly executed on the host CPU
  - Which ones? User application (low privilege level) for sure...

The Monitor / Hypervisor

- The Virtual Machine Monitor (VMM) must be in control of physical resources (requirement 3)
  - It manages Virtual Machines like an OS kernel manages processes
  - Virtual Machine: contains user code (unprivileged instructions) and (guest) OS kernel
- OS Kernel: runs in supervisor mode → supervisor for user code (user processes)
- VMM: supervises both user code and OS kernels → supervisor of supervisors ⇒ Hypervisor!!!
  - How does it work?
  - Mechanisms to control the execution of OS kernel code (privileged instructions)?

Direct Execution of Untrusted Guest Code

- Some instructions cannot be executed
  - Which ones? We need a formal definition...
  - When the guest tries to execute these instructions, the hypervisor / VMM must intercept them
- OS kernels have similar issues
  - When user code tries to execute a privileged instruction, an exception fires → the kernel handles it
  - Simple concept: user code cannot execute privileged instructions
- Can something similar be done for CPU virtualization?

Guest Code at Low Privilege Level

- Idea: execute the guest with a low privilege level
  - Intel x86: ring 3
- Hypervisor / VMM at high privilege level
  - When the guest tries to execute privileged instructions, exception / trap!
  - The VMM can handle it
- Will this work?
  - Thinking about x86, we can immediately see some issues...
  - Example: some unprivileged instructions can read some parts of the "CPU state" (AKA machine status word) without generating exceptions

More Formal Definitions: Popek and Goldberg

- Paper from 1974!!!
    - *Formal Requirements for Virtualizable Third Generation Architectures*
- Provides formal definitions for VMM (the term "hypervisor" is only used in the keywords)
- Uses the formal definitions to determine a set of requirements for easily and efficiently virtualize the CPU
    - If the requirements are satisfied, it is possible to execute guest code in the host intercepting the relevant instructions
- Distinction between sensitive instructions and privileged instructions

Privileged and Sensitive Instructions

- Privileged instructions (we already know)
  - Can be executed when the CPU is at high privilege level
  - Generate an exception when the CPU is at low privilege level
- Sensitive instructions (these are the “problematic ones”)
  - Change the “CPU configuration” / CPU state
  - Reveal something about the CPU state
- Popek and Goldberg provide formal definitions (for a simplified system: only memory, no interrupts, no paging, ...)

Sensitive Instructions

- These are the instructions relevant when virtualizing the CPU!!!
- Control Sensitive Instructions: change the CPU state
  - In Popek and Goldberg's model, privilege level or accessible memory - memory is the only considered resource
  - In real systems, interrupt table, paging table, ...
- Behavior Sensitive Instructions: effects depend on the CPU state
  - In Popek and Goldberg's model, privilege level or accessible memory
  - In real systems, things are more complex...

Popek & Goldberg Requirements

A VMM can be easily and efficiently implemented if the set of sensitive instructions is a subset of the privileged instructions

- Intuition: all the “problematic” instructions cause an exception if executed with low privilege level
    - Hence a privileged VMM can intercept them by executing the guest as unprivileged!!!
- More formally, instructions executed in user mode either:
    - Generate a result that does not depend on the “CPU state”...
    - ...Or generate an exception!

Real CPUs vs Popek & Goldberg

- Do real CPUs satisfy Popek & Goldberg requirements?
  - Some of them do... Mainly by IBM
- Other CPUs did not initially comply with the virtualization requirements
  - Motorola 68000: unprivileged instruction able to read the whole status register
    - Fixed in 68010
  - ARM: some sensitive unprivileged instructions
  - Intel x86: plenty of sensitive unprivileged instructions
  - MIPS had issue too... Fixed in Release 5 (2012)

Intel x86 vs Popek & Goldberg

- Original x86 architecture: plenty of sensitive unprivileged instructions
  - Mainly related to the accessibility of status flags and to the privilege levels bits in segment registers
- `S{GDT, IDT, LDT, MSW}`
- `PUSHF` and `POPF`
- `LAR`, `LSL`, `VERR`, `VERW`
- `PUSH`, and `POP` with segment registers
- ...

Instructions Accessing Special Registers

- `GDTR`, `LDTR` and `IDTR`: registers pointing to *descriptor tables* (data structures controlling the CPU operation
- `SGDT`, `SLDT` and `SIDT` allow to read the content of these registers
  - Sensitive instructions!
  - A guest OS can use them to know the host descriptor tables...
- Allowed in user mode (ring 3 - low privilege level) without raising exceptions!
- `SMSW` allows to read the machine status word (part of `cr0`)
  - Sensitive too... And still not privileged!

`PUSHF` and `POPF`

- Flags register: contains sensitive information, such as the interrupt flag
- `PUSHF`: pushes the flags register on the stack
  - Can be used to know the state of the interrupt flag
  - Does not generate exceptions...
- `POPF`: pops the flags register from the stack
  - Could be used to set / reset the interrupt flag???
  - If executed from ring 3, the state of `if` is not changed, but no exception is generated!!!

Instructions Accessing the Privilege Level

- `LAR`, `LSL`, `VERR` and `VERW` play with the privilege level of a segment (least significant 2 bits of the segment descriptor)
  - Allow to read the privilege level of a segment
  - Allow to check if a segment can be accessed from current privilege level
  - ...
- Again, no exception is generated
  - A guest OS can easily know the host segments
  - A guest OS kernel can know that it is not running in ring 0
  - ...

`PUSH` / `POP` with Segment Registers

- `PUSH` and `POP` can be used with segment registers
- Segment register: contain a segment descriptor
  - Two rightmost bits: protection level for the segment
  - Can easily leak from host to guest!!!
- Similar issues with segment registers in other instructions
  - `STR`
  - `MOVE`
  - `CALL FAR` / `INT FAR`
  - ...

Example: `POPF`

```
    movl $0, %eax
    pushl %eax
    popf
```

- Tries to load “0” in the flags register
- The flags register contains the interrupt flag ⇒ clear the interrupt flag!
    - Clearly not possible at low privilege level (ring 3)
    - The interrupt flag (and other flags) is not affected by `POPF` at ring 3
- No exception is generated ⇒ the VMM cannot know that the guest is trying to clear `if`

A Dirty Workaround

- Does this mean that VMM / hypervisors could not be implemented on x86?
  - VMWare proved the opposite...
- Notice: Popek and Goldberg say that a VMM cannot be *easily and efficiently* implemented
  - If we accept complications and performance loss, we can work around the issue...
- Idea: replace all the sensitive unprivileged instructions with something that generate an interrupt / exception!!!
  - VMWare & friends used variations of this idea...
  - Possibly patented?

The ARM Architecture

- ARM: RISC CPU (32-bit instructions, 16 registers, ...) with pragmatic design
  - Currently one of the major players in embedded systems
- Many different versions of the ARM core
  - Let's consider ARM v7
- Multiple privilege levels: user (`USR`), system (`SYS`), supervisor (`SVC`), interrupt (`IRQ`), fast interrupt (`FIQ`), abort (`ABT`) and undefined (`UND`)

ARM vs Popek & Goldberg

- Original ARM: some sensitive unprivileged instructions
  - As for x86, mainly related to accessibility of the CPU state (status flags and other)
- CPU state:
  - Currently Active Processor Status Register (`CPSR`), saved in SPSR when switching from user mode to a privileged mode
  - Some *coprocessors* (example: `CP15` - system coprocessor - controlling caches and similar)
  - ...

Example: Accessing the PSR

- `CPS` modifies the `CPSR`
  - Similar to x86 flags register: can disable interrupts, etc...
  - Obviously, can be done from a privileged mode only!
- If executed with low privilege level (user mode), does nothing!
  - Does not trap!!!
- So it is control sensitive (can disable interrupts), behaviour sensitive (its behaviour depends on the privilege level) and unprivileged!

ARM Sensitive Unprivileged Instructions

- ARM handling of the PSR → very similar to x86 handling of flags register
    - Unprivileged instructions can read it
        - Access to interrupt flag and other sensible information (behaviour sensitive)
        - Access to the privilege level (that is part of PSR) ← similar to x86 issues with segment registers
    - Unprivileged instructions can try to write it without generating exceptions!
- Looks like ARM “inherited” from x86 some of the issues that make it non-compliant with Popek & Goldberg requirements

Virtual Memory

- Popek and Goldberg considered a very simple model of virtual memory
  - Segmented architecture with only one segment
  - If $VA >$ limit, memory fault (exception)
  - Otherwise, $PA = VA +$ base
- Paging can also be supported, if P&G requirements are met and the VMM can intercept page faults
  - The VMM knows when the guest accesses the page table register
  - The VMM knows when the guest causes a page fault
  - The VMM can know when the guest accesses the page table

Virtualized Paging

- The guest page table is not the “real” (host) page table
  - The VMM can intercept accesses to the page table register...
- The guest can freely modify its “virtualized page table”
  - Without even knowing that it is not the real page table!
- When the guest tries to use some of the mappings it created, a host page fault is generated!
  - The VMM can handle it adding a proper mapping in the host page table

Example - 1

1. The guest sets the page table register (example: `cr3`) to some value
    - Exception → the VMM intercepts the write
    - Now the VMM knows where the guest page table is
    - If the guest tries to read the page table register, the read is intercepted by the VMM, that returns this value
    - The host page table is not affected
2. The guest modifies its page table mapping address VA_1 into PA_1
    - Nothing happens in the VMM / host

Example - 2

3. The guest accesses VA_1
   - VA_1 is not mapped in the “real” page table $\Rightarrow$ page fault!
4. The VMM handles the page fault
   - Look at the guest page table
   - Find mapping for VA_1
   - Create appropriate mapping in the host page table
5. The guest access to VA_1 completes without issues

- Technique sometimes known as “shadow paging”

Shadow Paging - 1

- A “shadow page table” is used for converting guest VA into host PA
  - The guest page table is not really used by the MMU!!!
  - Used only by the VMM to update the shadow page table
- The VMM handles page faults
  - If a VA is not mapped in the guest page table, page fault forwarded to the guest
  - Otherwise, used to update the shadow page table
- A guest memory access can result in 2 page faults!!!

Shadow Paging - 2

- The VMM can detect accesses to the guest page table, and update the shadow page table immediately
  - Avoid “lazy behaviour”
  - Can avoid the double page fault...
  - ...At the cost of introducing other page faults!
  - More complex code
- In any case, huge overhead!!!
  - Can we do better?
  - Not without paravirtualization or hardware support!

Hardware Support for Page Table Virtualization

- In non-virtualized CPUs, the MMU translates VAs to PAs
  - Translation performed in hw → fast, efficient
  - TLB-like caching tricks to improve performance
- What to do in virtualized CPUs?
  - Additional level of indirection: VA → PA → MA (Machine Address)
  - VA and PA are guest addresses, MA is a host address
- The MMU uses two page tables: guest page table (VA → PA) and host page table (PA → MA)
  - Can use TLB-like caches and trickery, etc...

Extended / Nested Page Tables

- Hardware feature provided by the major CPU manufacturers
  - Intel: Extended Page Tables (EPT)
  - AMD: Nested Page Tables (NPT)
  - ARM has a similar thing, too...
- Different naming, small differences, similar concepts
  - The VMM can setup a Nested / Extended page table to convert guest PAs in host MAs
  - The guest can handle its page table (no need to intercept accesses to the guest page table!)
  - The VMM just needs to update its extended page table when a guest tries to access a PA not mapped in MA

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)
  - All sensitive instruction trap to hypervisor 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 → 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 Structute
  - 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