Virtual Machine Monitors

Luca Abeni
luca.abeni@santannapisa.it

June 13, 2019
Virtual Machines

- Virtual Machine: efficient, isolated duplicate of a physical machine
  - Execution environment essentially identical to the physical machine
  - Programs only see a small decrease in speed
  - A “monitor” or “hypervisor” is in full control of physical resources
- Programs running in a VM should not see differences respect to real hw
- Virtualization should be efficient
- Programs should not be able to access resources outside of the VM
How is an OS related to Virtual Machines?
- The OS should provide support for the Virtual Machine Monitor / hypervisor
- The OS *could* be optimized to run inside a VM

OS support for virtualization (as host or as guest)
- Impact on resource management
- Impact on the exposed features
- Impact on the I/O devices support

Impact on the OS architecture?
- Host: type-I hypervisors, $\mu$-kernel systems
- Guest: library OSs, unikernels, vertically structured OSs
CPU Virtualization

- First idea: simulate the CPU hw in software
  - Software implementation of an abstract machine implementing the fetch-decode-execute-(write) cycle
  - Fails the efficiency requirement!!!

- Other idea: directly execute the virtualized instructions on the CPU
  - Virtual ISA: exact copy of the host ISA
  - Might fail the third (VMM is in control) requirement
  - Limited to unprivileged instructions (with VMM executing at a high privilege level)
  - What to do for privileged instructions?
Virtualizable CPU Architectures

- The monitor should be able to "intercept" some machine instructions
  - Some kind of trap / exception / software interrupt must be generated
  - Not always possible (think about x86 ring 0)
- The CPU must provide some support for full virtualization
  - "More than supervisor" mode → hypervisor mode
  - Introduce two operating modes: "root mode" and "non-root mode"; non-root mode can only modify a shadow copy of the CPU privileged state
- ...
OSs for Virtualizable Architectures

- Virtualizable ISA: how to use it?
  - VMM or hypervisor responsible for managing VMs and other resources
  - Re-invent an OS, or using an existing one?
- OS support for hypervisors
  - Hosted hypervisor
  - Dom0
  - ...
- Difference between a hypervisor and a $\mu$-kernel???
  - Are we reinventing an old idea?
ParaVirtualization

- So, CPU virtualization can be easy and efficient
  - Provided that the ISA is virtualizable
  - Provided host OS support / hypervisor

- What about I/O devices?
  - Virtualizing real hardware can be complex and inefficient
  - Idea: device passthrough
  - Other possibility: paravirtualization

- Paravirtualization: the guest knows that it is running in a VM
  - Memory buffers can be (securely) shared between guest and host
  - ...
Example of (Toy) CPU

- Toy CPU: just an example with many simplifications
- Modern (real) CPUs are much more complex!
  - Pipeline
  - Parallel execution

Advanced Operating Systems Virtual Machine Monitors
CPUs, Programs, & Friends

- CPU $\rightarrow$ executes programs
  - Stored in main memory
  - Use data from main memory
- Program: formal description of an algorithm
  - Using a programming language
- Sequence of machine instructions
  - Actions having effects on some objects
  - "Object": data stored in main memory
- Instance of program in execution: sequence of actions on objects
  - Example: $\text{int mcd(int a, int b)}$ and its execution
Executing a Program

- CPU: cyclical execution (fetch / decode / load / execute / save)
  - Machine instructions are executed (mainly) sequentially
- Machine designed to execute its own language!
  - Machine Language
Physical Machines...

- Computer: (physical) machine designed to execute programs
- Every machine executes programs written in its own language
- Relationship between machine and language
  - A machine has its own language (the language it can parse and execute)
  - A language can be “understood” (parsed and executed) by multiple different machines
- Program execution: (infinite) cycle fetch/decode/load/execute/save
  - CPU: hw implementation of this cycle
...And Abstract Machines!

- The fetch/decode/load/execute/save cycle can be implemented in hw or in sw...
- Software Implementation: Abstract Machine
  - Algorithms and data structures used to store and execute programs
- Once upon a time referred as “Virtual Machine”
  - Today, the term “Virtual Machine” (VM) is used with a slightly different meaning
Abstract Machines and Languages

- Similarly to physical machines (CPUs), each abstract machine has its own machine language
  - Machine language for a CPU: sequence of $0 / 1$
    - Assembly makes it more readable
  - Abstract machines generally have higher level machine languages (C, Java, etc...)
- $M_L$: abstract machine understanding language $L$
  - $L$ is the *machine language* of $M_L$
  - Program: sequence of instructions written in $L$
- $M_L$ is just a possible way to describe $L$
Abstract Machines Behaviour

- To execute a program written in $L$, $M_L$ has to:
  1. Execute some “elementary operations”
     - In hw, ALU
  2. Manage the execution flow
     - Execution is not only sequential (jumps, loops, etc...)
     - In hw, PC handling
  3. Move data from / to memory
     - Addressing modes, ...
  4. Take care of memory management
     - Dynamic allocation, stack management, etc...
Abstract Machine Example

- Execution cycle: very similar to a CPU...
- ... But it is implemented in software!
Implementing a Language

- $\mathcal{M}_\mathcal{L}$ understands 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

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

- Interpreter: program written in $\mathcal{L}_0$ (executes on $\mathcal{M}_0\mathcal{L}_0$) understanding programs written in $\mathcal{L}$
- Translates $\mathcal{L}_0$ in $\mathcal{L}$ “instruction by instruction”
Pure Compilers

- Translates the whole program from $\mathcal{L}$ to $\mathcal{L}_0$ before executing it
- Translation performed by a dedicated program, the Compiler
  - Compiler: not necessarily written in $\mathcal{L}_0$
  - Can execute on an abstract machine $\mathcal{M}_a$ different from $\mathcal{M}_{o\mathcal{L}_0}$
Hybrid Implementation

- Not a pure compiler nor a pure interpreter
- Compiler translates in an *intermediate language* $\mathcal{L}_i$
- Interpreter executes on $\mathcal{M}_{O_Lo}$ programs written in $\mathcal{L}_i$
  - Java: compiler $\rightarrow$ bytecode, then JVM
  - C: compiler *generally* produces code that needs SO 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 $\mathcal{L}_o$ 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 $\mathcal{L}_0$
  - 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) → sort of simple compiler
Qemu TCG

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

- Instead of emulating a CPU implementing $M_L$ in software, execute target instructions in the host.
  - This implies $L == L_0$!!!

- 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 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, ...)

Advanced Operating Systems
Virtual Machine Monitors
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...
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
    - $S\{GDT, IDT, LDT, MSW\}$
    - \texttt{PUSHF and POPF}
    - \texttt{LAR, LSL, VERR, VERW}
    - \texttt{PUSH, and POP with segment registers}
    - ...

Advanced Operating Systems
Virtual Machine Monitors
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
  - ...

---

**Advanced Operating Systems**

**Virtual Machine Monitors**
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)
  - ...

Advanced Operating Systems
Virtual Machine Monitors
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
Popek and Goldberg considered a very simple model of virtual memory:
- Segmented architecture with only one segment
- If $VA > \text{limit}$, memory fault (exception)
- Otherwise, $PA = VA + \text{base}$

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
1. The guest sets the page table register (example: $cr3$) to some value
   - Exception $\rightarrow$ 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, which 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”
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!!!
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