OSE

virtual machines

By Dan Tsafrir 31/5/2016
Presentation partially based on slides by Muli Ben-Yehuda
What is virtualization?

- Decouple software from hardware in a way that allows running multiple OSes on same hardware
  - E.g., run both Windows and Linux on same laptop

- How is it different than dual boot?
  - Both OSes run simultaneously
  - Yet, by default, they are completely isolated from each other
  - Called: “Virtual machine” (VM)
Benefits

- **For kernel developers**
  - Easier development process (can hack OS as ordinary process)

- **For power users**
  - One computer with multiple OSes
    - My Win 7 laptop also runs Ubuntu
    - My MacBook Pro @ home also runs Win 7 (for office)
  - Allows users to use the best tool for the job
    - E.g., Windows for Office & sites that work only under IE; Linux for everything else ;-)
Benefits

- Server consolidation (probably most important, money-wise)
  - E.g., run the web server “machine” and the mail server “machine” on the same physical machine
  - (Until virtualization, different servers kept on separate machines for robustness)
  - Significant electricity savings
  - Significant room space savings
Benefits

- **Eases administration and improves robustness**
  - Untying SW from HW makes life easier to, e.g.,
    - back up server “machines”
    - restore them if HW break on a different physical machine
    - upgrading to newer machines
    - Easier provisioning of new server “machines” for new services (one can have a “new” machine ready in a few seconds)

- **Easier testing & quality assurance**
  - Products sometimes spans multiple machines
  - E.g., testing a network product (such as a firewall) might require dozens of computers
Benefits

- **Makes the IaaS cloud computing ecosystem possible**
  - Cloud providers, like Amazon, sell compute power (you pay for, e.g., 2 CPU cores for 3 hours plus 10GB of network traffic)
  - Thus, an IT organization can have a server farm, somewhere remotely, without having to allocate room, hire administrators, handle faulty machines, and all that jazz
  - The cloud is “elastic”: customers can easily grow and shrink their compute infrastructure as needed
  - Since HW and SW are decoupled, cloud providers can easily balance the load on their servers with “live migration” (moving a virtual machine from one physical machine to another, while it is running)

- ...
Definitions

- **Hypervisor or VMM (Virtual Machine Monitor)**
  - The SW layer that allows several *virtual machines* (VMs) to run on the same physical machine

- **Host**
  - The physical machine and the OS that directly controls it
  - Overload: sometimes we say “host” but we actually mean hypervisor

- **Guest (or guest OS)**
  - The virtual machine OS and all the applications it runs
Hypervisor Types

**Type 1**
(“bare-metal”)

- Guest
  - VM1
  - VM2
- Host
  - hypervisor
  - hardware

E.g., VMware ESX, Microsoft Hyper-V, Xen (typically for servers, datacenters, clouds)

**Type 2**
(“hosted”)

- Guest
  - VM1
  - VM2
- Hypervisor
  - process
  - OS
- Host
  - hypervisor
  - hardware

E.g., VMware Workstation, Microsoft Virtual PC, Sun VirtualBox, QEMU (typically user-owned)
Bare-metal vs. hosted

- **Bare-metal**
  - Has complete control over HW
  - Doesn’t have to “fight” / co-exist with OS

- **Hosted**
  - Avoid functionality/code duplication (e.g., process scheduler, memory management) – the OS already does all of that
  - Can run native processes alongside VMs
  - Familiar environment
    - How much CPU and memory does a VM take? Use top!
    - How big is the virtual disk? Use ls –l
    - Easy management: kill/stop a VM? Sure, just SIGKILL/SIGSTOP it!

- **A combination**
  - Mostly hosted, but some parts are inside the OS kernel for performance reasons and most of the reasons listed above
    - Example: KVM+QEMU (KVM ~makes the Linux kernel a hypervisor; QEMU, which is a process, is accelerated by KVM)
How it works

- **Hypervisor**
  - Is like the kernel

- **VMs**
  - Are like processes

- **Hypervisor**
  - schedules VMs,
  - allocates memory for them,
  - multiplexes their I/O, etc.

- **Just one problem...**
  - OSes think/assume they control bare-metal HW

- **Solution**
  - Hypervisor must lie and fake it:
    make it appear as if the guest controls the HW
How to run a VM?

- **How? A few alternatives**
  - Software emulation
  - Trap-and-emulate
  - Dynamic binary translation
  - Paravirtualization

- **Need to virtualize:**
  1. CPU;
  2. memory;
  3. I/O

- **Let us first focus on the CPU...**
How to run a VM? – SW emulation

- Do whatever CPU does but ourselves, in software
  - Fetch the next instruction
  - Decode (is it an ADD, a XOR, a MOV?)
  - Execute (using the SW emulated registers and memory)

- For example:
  - `addl %ebx, %eax /* eax += ebx */`

**Is emulated as:**
  - `enum {EAX=0, EBX=1, ECX=2, EDX=3, ...};`
  - `unsigned long regs[8];`
  - `regs[EAX] += regs[EBX];`

- **Pro: Simple!**
- **Con: Sloooooow...**

- **Example: BOCHS**
How to run a VM? – trap & emulate

- Actually, most VM code can execute directly on CPU just fine
  - E.g., addl %ebx, %eax
- So instead of emulating this code
  - Let it run directly on the CPU
- But some operations are sensitive and require the hypervisor to lie, e.g.,
  - int $0x80 (generates system call interrupt; hypervisor knows that from now on the guest thinks it’s in privileged mode; guest can’t really run in privileged mode, of course, because otherwise it’d be able to mess stuff up for the host / other guests)
  - movel <something>, %cr3 (switch virtual memory address spaces; once again, hypervisor can’t allow the guest to actually manipulate address spaces on its own, but it can do it for the guest)
  - I/O ops (I/O channels are multiplexed by the host so as to allow all the guests to use them, which once again means the hypervisor can’t allow direct access; also, I/O devices handling will not be able to tolerate multiple OSes performing uncoordinated ops)
How to run a VM? – trap & emulate

- **Idea**
  - Trap-and-emulate all these “sensitive” instructions
  - E.g., if guest runs INT $0x80, trap it and execute guest’s handler of interrupt 0x80
  - We are leveraging the fact that many sensitive operations trigger an interrupt when performed by unprivileged user-mode SW

- **Pro**
  - Performance!

- **Problem (32bit machine olden days)**
  - Not all sensitive ops trigger a trap when executed in user-mode
  - Example for x86/32bit
    - POPF, which may be used to set/clear interrupt flag (IF)
    - Will silently fail!
    - Namely, it will (1) not trap, and it will (2) not change the IF value
How to run a VM? – trap & emulate

- **Solution #1**
  - HW support for virtualization (modern chips rectify the problem)
  - Hypervisors can, e.g., configure which ops would generate traps
  - Intel calls such support “VMX”
  - AMD calls such support “SVM”

- **Example hypervisor**
  - As opposed to some other, earlier hypervisors, KVM was originally implemented by making use of HW support for virtualization

- **Problem: hypervisors that predated HW support**
  - Had to solve the problem in some other way... (next slides)
How to run a VM? – dynamic binary translation

- **Solution #2: binary translation – idea**
  - Block of (VM) ops encountered for 1st time?
  - Translate it, on-the-fly, to “safe” code
    - Similarly to JIT-ing
    - Put it in the “code cache” (indexed by address)
  - From now on
    - Safe code would be executed directly on CPU

- **BTW**
  - You can do above yourselves:
  - Download Intel’s pin
How to run a VM? – dynamic binary translation

- Translation rules?

  - Most code translates similarly
    - E.g., movl %eax, %ebx

  - Sensitive ops are translated into "hypercalls"
    - = Calls into hypervisor
      - (to ask for service)
    - Implemented as trapping ops
      - (unlike, e.g., POPF)
    - Similar to syscall
      - (call into hypervisor to request service)
How to run a VM? – dynamic binary translation

- **Pros**
  - No hardware support required
  - Performance is much better than full SW emulation

- **Cons**
  - Performance may be worse than native-HW trap-and-emulate
  - Hard (!) to implement
    - Hypervisor needs on-the-fly x86-to-x86 binary compiler
    - Consider the challenge of getting branch target addresses right

- **Example hypervisors**
  - VMWare (x86 32bit), QEMU
How to run a VM? – paravirtualization

- So far
  - Guest OS was unmodified

- Conversely, paravirtualization
  - Requires guest OS to “know” it is being virtualized
  - And to explicitly use hypervisor services through a hypercall
  - E.g., instead of doing “cli” to turn off interrupts, guest OS should do: `hypercall(DISABLE_INTERRUPTS)`

- Pros
  - No hardware support required

- Cons
  - Requires specifically modified guest
  - Same guest cannot run in the VM and on bare-metal

- Example hypervisor
  - Xen
Prevailing trend

- Trap & emulate with HW support (VMX, SVM, ...)
- Paravirtualization for device drivers
How to run a VM?

- **The problem**
  - OSes think/assume they control bare-metal HW

- **The solution**
  - Hypervisor must lie to the guest and fake it: make it appear as though the guest controls the HW

- **How? A few alternatives**
  - Emulation
  - Trap-and-emulate
  - Dynamic binary translation
  - Paravirtualization

- **Need to virtualize:** (1) CPU; (2) memory; (3) I/O
  - Let us first focus on the CPU...
Reminder: x86 paging

- **Need to translate**
  - from: virtual addresses
  - to: physical addresses

- **Translation is cached on-chip TLB**
  - (Translation Lookaside Buffer)

- **Page table is read & modified by HW**
  - (Access/dirty bit)

- **Each process has its own virtual address space**
  - Page table pointed to by CR3 register
  - During context switch the OS updates the value of CR3.

- **Page table is a hierarchical structure**
Reminder: x86 paging

Linear Address Space (4K Page)

- 6: sign extension
- 4: PML4
- 3: PDP
- 3: DIR
- 2: TABLE
- 1: OFFSET

- 9: 512 entry Page Directory Pointer Table
- 9: 512 entry Page Directory
- 9: 512 entry Page Table
- 2: 4KByte Page
- 8: data

- 40 (4KB aligned)
- CR3 (PDPTOR)

- CR3 (PDPTR)

- 40 (4KB aligned)
Virtualizing the virtual memory

- So we previously had to translate
  - from: virtual addresses
  - to: physical addresses

- But the above is actually
  - from: guest virtual addresses (GVA)
  - to: guest physical addresses (GPA)

- Both GVA & GPA aren’t real
  - Do not correspond to the physical memory

- Virtualization therefore requires another level of translation
  - from: guest physical (GPA)
  - to: host physical (HPA)
Virtualizing the virtual memory

- There are two ways to accomplish this additional level
  - With HW support (EPT/NPT)
  - With “shadow page table”
    - Which requires no HW support
**Shadow page table**

- **Hypervisor computes the double translation GVA to HPA**,  
  - Storing them in a new set of page tables (called shadow page tables)

- **To build/maintain shadow page table**
  - All page faults are trapped (hypervisor handles interrupts)
  - Hypervisor walks guest page table
    - If it’s a “guest page fault” (=no translation in guest page table):  
      “inject” (=emulate) page fault to guest
    - Otherwise, we found a guest page table translation
      => Build missing entries in shadow page table using hypervisor’s internal SW data structure that maps guest’s GPA to HPA
  - Hypervisor traps-and-emulates all changes made by the guest to its page tables by write-protecting them
Shadow page table

- **Challenges**
  - Complex....
  - Hypervisor must maintain access/dirty bits within guest PTEs
  - Hypervisor needs to support all x86 paging modes
    - real mode, 32bit, PAE, and 64bit
    - (modes have different hierarchies, PTE sizes, and huge page sizes)
Shadow page table

- To simplify, can start building shadow page from scratch on every cr3 change (= every context switch)
  - Caching is challenging because, e.g., the guest may start using the pages for other purposes (recall that they are write-protected)

- Pro
  - As noted, requires no HW support

- Cons
  - Overwhelmingly complex
  - Can be slow due to all the overheads involved
2D/nested/extended page table (EPT/NPT)

- Since shadow page tables are complex and expensive
  - => HW support for 2nd translation table

- Processor support two level page tables:
  - Regular guest page table (GVA => GPA) maintained by guest OS
  - New second translation table (EPT) from guest physical address (GPA) to host physical address (HPA) maintained hypervisor

- Schematically, translations looks as follows

  ![Diagram of page table translations]

  - In reality a bit more complex...
how many mem refs upon TLB miss?
Shadow PT vs. EPT

- **Tradeoffs discussed to far**
  - EPT requires HW support but
    - It makes things much simpler relative to shadow PT
    - And it eliminates much of the shadow PT overheads

- **Question**
  - Is it possible that using shadow PT will yield performance superior to EPT?

- **Answer**
  - Yes! (Think of why)
2D/nested/extended page table (EPT/NPT)

- Guest has full control over its page table
  - No need to trap changes in CR3, page faults, modification to guest PTs

- EPT’s structure is similar to the x86 page table structure
  - One issue originally missing and recently rectified: access/dirty bits

- EPT translation are cached on-chip
  - Similarly to TLB; eliminates the need to walk the table in the common case

- Note that
  - The EPT table changes rarely

- Interrupts to hypervisor
  - EPT violation – no translation for the guest physical address
    - How can we utilize such violations?
  - EPT misconfiguration
How to run a VM?

- The problem
  - OSes think/assume they control bare-metal HW

- The solution
  - Hypervisor must lie to the guest and fake it: make it appear as though the guest controls the HW

- How? A few alternatives
  - Emulation
  - Trap-and-emulate
  - Dynamic binary translation
  - Paravirtualization

- Need to virtualize: (1) CPU; (2) memory; (3) I/O
  - Let us first focus on the CPU...
I/O virtualization

- **Types of I/O**
  - Block (e.g., HDD, SSD, NVMe)
  - Network (NIC = network interface card)
  - Input (e.g., keyboard, mouse)
  - Sound
  - Video

- **Most performance-critical to servers**
  - Network
  - Block
Pseudo code of a physical NIC driver

- **Transmit path:**
  - OS prepares packet to transmit in a buffer in memory
  - Driver writes start address of buffer to register X of the NIC
  - Driver writes length of buffer to register Y
  - Driver writes ‘1’ (GO!) into register T
  - NIC reads packet from memory addresses [X,X+Y) and sends it on the wire
  - NIC sends interrupt to host (TX complete, can free buffer)

- **Receive path:**
  - Driver prepares buffer to receive packet into
  - Driver writes start address of buffer to register X
  - Driver writes length of buffer to register Y
  - Driver writes ‘1’ (READY-TO-RECEIVE) into register R
  - When packet arrives, NIC copies it into memory at [X,X+Y)
  - NIC interrupts host (RX)
  - OS processes packet (e.g., wake the waiting process up)
I/O virtualization – emulation

- **Emulation**
  - Emulate some physical NIC in SW (all hypervisors emulate e1000, all guests have an e1000 driver)
  - NIC’s registers are variables in Hypervisor’s memory
  - Memory is write protected (Hypervisor reacts according to values being written)
  - Interrupts are injected by hypervisor to guest

- **Pros**
  - Unmodified guests (all OSes already have a driver for e1000)
  - Use only one device => robust
  - Portable across HW & hypervisors (and hence clouds)

- **Cons**
  - Slow (traps on every register access)
  - Hypervisor needs to emulate overly complex HW (can be simpler)
I/O virtualization – paravirtualization

- **Paravirtualization**
  - Emulate a “new” device, which isn’t physical in any sense
  - Guest installs a host-specific device driver
    - Denoted: paravirtual device driver
  - Protocol between frontend (driver installed in guest) and backend (hypervisor) is optimized for efficiency

- **Protocol in emulation case**
  - Guest writes registers X, Y, waits a bit & writes to register T
  - => Hypervisor **infers** guest wants to transmit packet

- **Protocol in paravirtual case**
  - Guest does a hypercall, passes it start address and length as arguments; hypervisor *knows* what it should do
I/O virtualization – paravirtualization

- **Pros & cons**
  - Its exactly like emulation
  - Except that it is faster 😊
  - But it requires guest modification, making it less portable 😞
    - Harder to move between cloud providers
    - Every SW modification is a risk
  - Still not as fast as using the actual HW

- **Difference between paravirtual I/O and paravirtual Guest?**
  - Guest requires to modify whole OS (try do that to windows...)
  - I/O requires an addition of a single driver (much, much easier)
I/O virtualization – direct assignment

- **Direct device assignment**
  - Pull NIC out of host and plug it into the guest for its exclusive use
  - Guest accesses device directly without hypervisor intervention

- **Pro:**
  - Much more performant than paravirtual I/O (which still induces many context switches)

- **Cons**
  - Need device per guest
  - Plus one for host
  - Can’t do I/O interposition
I/O virtualization – HW support

- **IOMMU (I/O memory management unit)**
  - I/O devices (like our NIC) perform DMA ops
    - Access memory on their own
  - Traditionally, devices used physical addresses to do so
  - This is seriously problematic in a setup where multiple untrusted guests are simultaneously running, sharing the same machine
    - What if a guest is malicious?
    - What if the device driver is buggy?
    - => Negates direct device assignment
    - (Also, what if device is legacy and can use only 32bit addresses, yet the physical memory is much bigger)
I/O virtualization – HW support

- **IOMMU (I/O memory management unit)**
  - The HW IOMMU solves this problem:
    - It allows hypervisor to arrange things such that devices use IOVAs (I/O virtual addresses) instead of PAs for their DMA ops
    - Like the MMU, the IOMMU knows how to walk the table
    - Like the MMU (which has a TLB), the IOMMU has an IOTLB
    - Unlike the MMU (which allows the OS to recover from page faults), an I/O page fault (generated as a result of a DMA) is not tolerated (=> causes a “blue screen”); therefore, DMA-related memory must be pinned to physical memory
  - Recently, AMD & Intel support nested IOMMU page walk
I/O virtualization – HW support

- **SR-IOV**
  - The ability of a device to appear to SW as multiple devices
  - Single root I/O virtualization
  - Contains a physical function controlled by the host, used to create virtual functions
  - Each virtual function is assigned to a guest (like in direct assignment)
  - Each guest thinks it has full control of NIC, accesses registers directly
  - NIC does multiplexing/demultiplexing of traffic

- **Pro:**
  - Nearly **fast** as device assignment
  - And need only one NIC (as opposed to direct assignment)

- **Cons**
  - Emerging standard (few hypervisors/clouds fully support it)
  - Requires newer hardware
  - Can’t do I/O interposition
x86 Virtualization Performance

VM performance

bare-metal performance

HW supports CPU virt
HW supports MMU virt
HW supports I/O virt

Exitless Interrupts (ELI)

HW support for virtualization

CPU intensive
memory intensive
I/O intensive