X86-64: CPU Virtualization With VT-x

[Hardware and Software Support For Virtualization](#)

Chapter 4

Design Requirements (Intel's take)

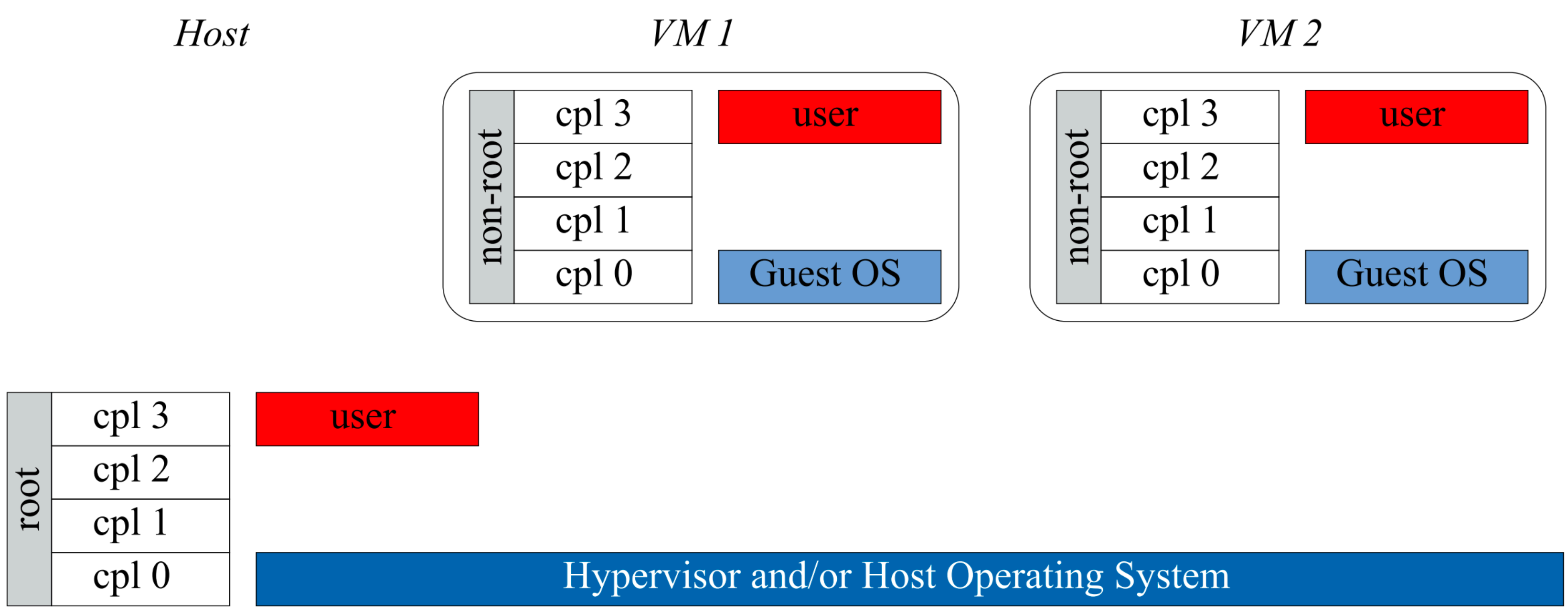
- Eliminate the necessity of DBT and paravirtualization
  - Enable VMM with a broader range if guest-OS
  - Improve performance
- Challenges identified with x86-32
  - **Ring aliasing and compression**: two levels in the guest-OS will run in the same `%cpl` (and still isolated?)
  - **Address Space Compression**: hypervisor should use a non-accessible address space in the guest-OS
  - **Non-faulting access to privileged state**: v.gr `sidt` and `sgdt` access directly to descriptor tables in user mode
  - **Interrupt virtualization**: v.gr. `%eflags.if` is visible to to user code via `pushf`
  - **Access to hidden-state**: segment resisters can't be copied in a general-purpose register or saved back into memory, unless segment table changed. Windows 95 relies in this bizarre semantics

Design Requirements

- Meet P/G requirements for a hypervisor running on top of VT-x
- **Equivalence**
  - VT-x supports architectural compatibility with both x86-32 and x86-64 ISA.
  - Enlarges the guest-OS diversity: v.gr. virtualize MS-DOS!
- **Safety**
  - Minimize hypervisor code responsibility on security and isolation.
  - Minimize attack surface in the VMM (by the means of simpler hypervisors)
- **Performance**
  - **Wasn't** a target in the first version (over state-of-the-art techniques). Just setup the roadmap for forthcoming versions of the extensions

The VT-x Architecture

- Address the problems **without changing** the original **semantics** of each instruction
  - Introduce a new mode called **root mode**



Properties of root mode

- Transitions between modes is **atomic** (not convoluted sequence of instructions like in a context-switch or word-swich)
- Root mode can be only detected using some specific set of instructions (not by memory content). Required for VM nesting.
- Only used for virtualization and orthogonal to others (real mode, v8086, protected,…). Can be used in both modes. All rings are available also in both modes
- Each mode uses a separate 64-bit linear address space. Only current mode is active in the TLB (TLB changes atomically between modes)
- Each mode has it own interrupt flag. Hence, software in non-root can manipulate `%eflags.if`

VT-x and P/G

In an architecture with root and non-root modes of execution and a full duplicate of processor state, a hypervisor may be constructed if all sensitive instructions (according to the non-virtualizable legacy architecture) are root-mode privileged.

When executing in non-root mode, all root-mode-privileged instructions are either (i) implemented by the processor, with the requirement that they operate exclusively on the non-root duplicate of the processor or (ii) cause a trap.

- Doesn't take into account when a ins is privileged or not
- These traps are sufficient for safety and equivalence criteria
- Implement certain instructions in hardware (performance critical) to meet performance criteria

Sensitivity is orthogonal to privilege: examples

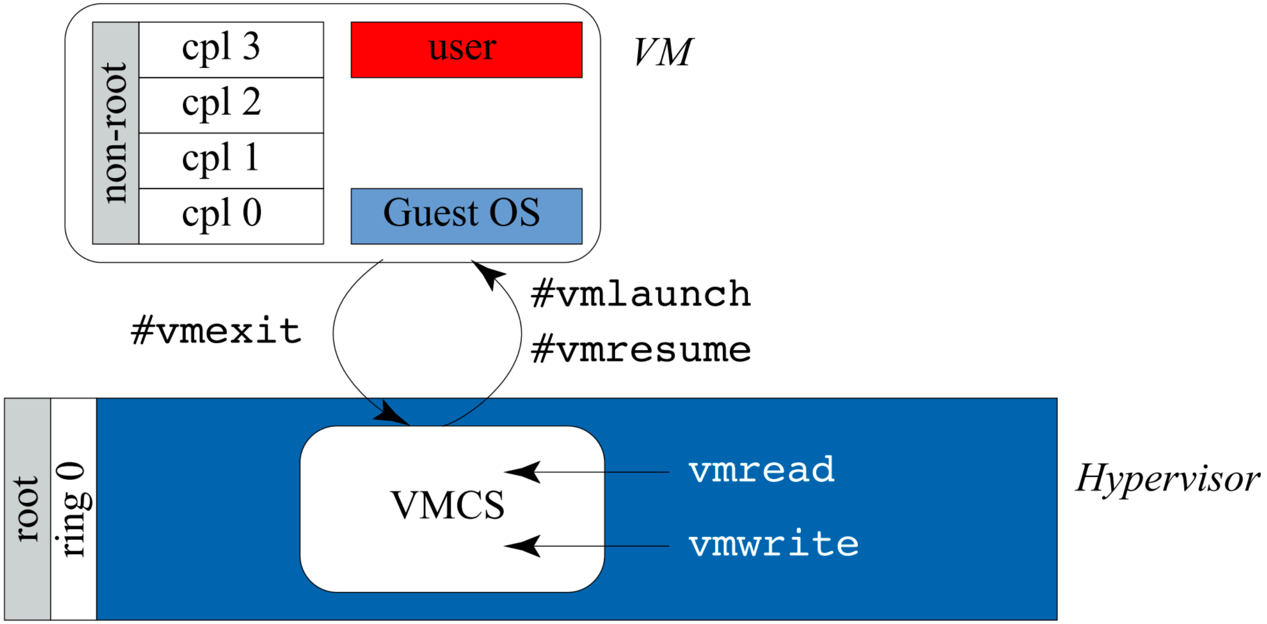
- Let's assume guest-OS in non-root-%cpl=0 issue a privileged ins such as read control registers (in the non-root duplicate processor state)
  1) CPU performs directly the operation in the duplicate state
  2) Inform the hypervisor (via trap)
- First is desired by requires changes in the hardware

- Instructions than manipulate interrupts flag (eg. `popf`) are sensitive
  - Are user-sensitive
  - Can be used quite frequently by modern OS
  - Specific implementation in non-root mode

- **Transistors to the rescue**

Transitions between root and non-root

- Transition is atomic via specific instructions in the ISA: `vmlaunch`, `vmresume`, `vmexit`
- Virtual Machine Control Structure (VMCS) is the key conduit to communicate hypervisor with the hardware
- Hypervisor should be adjusted to the processor specification (AMD!=Intel)



Reasons to `vmexit`

Category	Exit Reason	Description
Exception	0	Any guest instruction that causes an exception
Interrupt	1	The exit is due to an external I/O interrupt
Triple fault	2	Reset condition (bad)
Interrupt window	7	The guest can now handle a pending guest interrupt
Legacy emulation	9	Instruction is not implemented in non-root mode; software expected to provide backward compatibility, e.g., `task switch`
Root-mode Sensitive	11-17, 28-29, 31-32, 46-47:	x86 privileged or sensitive instructions: `getsec, hlt, invd, invlpg, rdpmc, rdtsc, rsm, mov-cr, mov-dr, rdmsr, wrmsr, monitor, pause, lgdt, lidt, sgdt, sidt, lldt, ltr, sldt`
Hypercall	18	`vmcall` : Explicit transition from non-root to root mode
VT-x new	19-27, 50, 53	ISA extensions to control non-root execution: `invept, invvpid, vmclear, vmlaunch, vmptrld, vmptrst, vmreas, vmresume, vmwrite, vmxoff, vmxon`
I/O	30	Legacy I/O instructions
EPT	48-49	EPT violations and misconfigurations

What about the MMU? --- A cautionary Tale

- In early version of VT-x, basically nothing (just `%cr3` duplication)
  - Allows disjoint address space without the address compression of prev. solutions
  - Everything else: software (v.gr. via shadow page table)

- This choice make to fail performance criteria
  - Over 90% of `vmexit` where due to shadow paging, being the global resulting performance worst using VT-x

- Purely software was faster!
  - VMWare, via DBT, avoids most guest-OS page table modifications!
  - Xen memory paravirtualization does not have Shadow Page Tables

KVM: a hypervisor for VT-x

- Most relevant FOSS Type-2 Hypervisor
- KVM relies on **QEMU** to emulate I/O
  - QEMU is a complete machine emulator with cross-architectural binary translation (v.gr. RISC-V on x86). *Equivalent to VMX on VMWare*
- CPU and Memory virtualization is closely integrated with Linux mainline (as module) **avoiding any redundancy**
  - *No VMM or VMMonitor like in VMWare*
- Unlike Xen or VMWare, was designed from the ground up **assuming** hardware support for virtualization (originally VT-x or AMD-v, now also RISC-V, ARM, etc…)
  - Good candidate to explore the intricacies of using VT-x

Leveraging VT-x in KVM

- **Equivalence**
  - KVM should be able to run x86-32 and x86-64 guest-OS **without** modifications
- **Safety**
  - All resources exposed to the VM (CPU, Mem, I/O buses and devices, BIOS) should be virtualized
  - KVM will retain control of the real resources under any circumstance
- **Performance**
  - Performant with **production workloads**
  - Linux Kernel takes the performance critical decision (seamlessly with other regular processes).
  - KVM module handles x86 emulation, MMU, interrupt subsystem,…
- Leveraged two previous FOSS projects:
  - **QEMU**: I/O emulation (still tightly integrated today)
  - **Xen**: X86 initial emulation come from early Xen versions (divergent paths)

The KVM Kernel Module (`kvm.ko`)

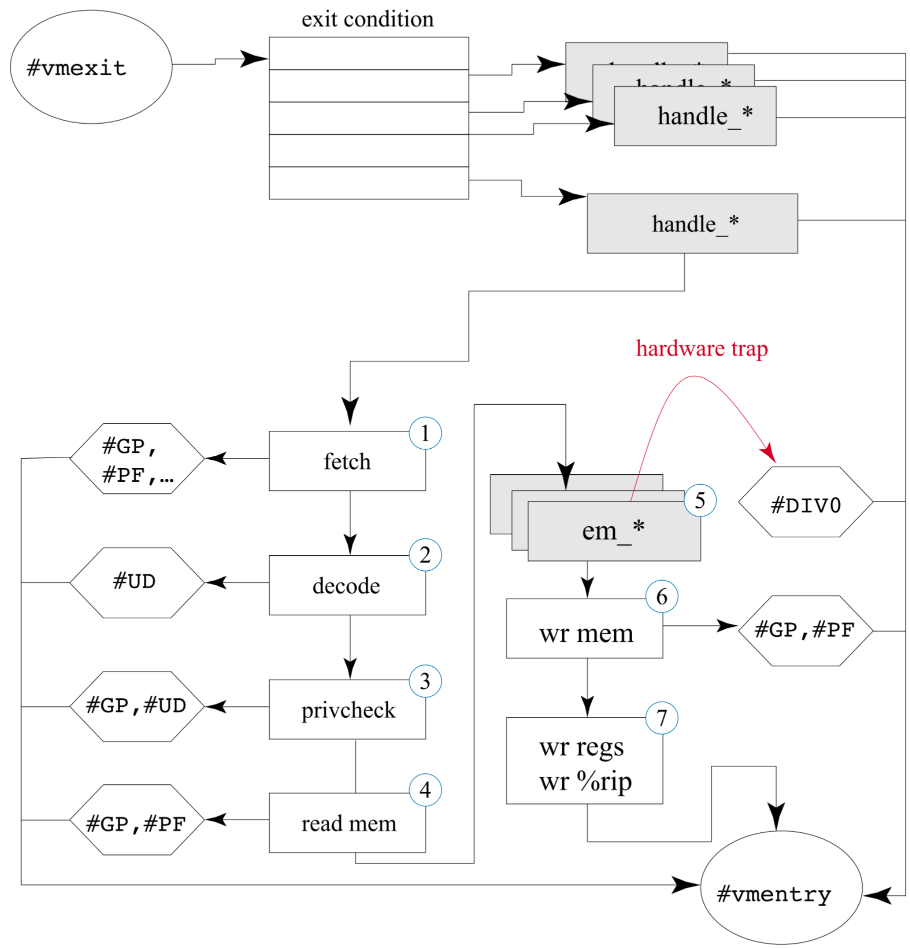
- `kvm.ko` handles CPU and platform emulation issues
  - CPU, memory management and MMU, interrupt and some chipset emulation (APIC, IOAPIC)
  - Excludes all I/O emulation

- One might think that since hardware is P/G compliant, KVM should be straightforward

- In reality it is complex (v.gr. kernel-4.9 25KLOC), due to:
  - Support for multiple iterations of the extensions and models (AMD-v, VT-x, versions, etc…)
  - The inherent complexity of the x86 ISA
  - The incomplete support of the hardware (allegedly infrequent) instructions

Trap and Emulate in KVM

- 54 exit reasons, each one with its own handler. Most are straightforward
  - Emulate the instruction semantics (via VMCS) and PC+1
  - If fault or interrupt, forward to the guestOS, preparing the **trap-frame**
  - Change the underlying env and retry (EPT violations)
  - Do nothing. External interrupt.
  - `Handle_*` on `arch/x86/kvm/vmx.c`
- Unfortunately VMCS information does not suffice (at times) to handle exit cause
  - Requires emulating the “offending” instruction to achieve equivalence
  - 5+ KLOC in a general-purpose emulator
  - Fech ins from guestOS → decode the operands and opcode → verify that is correct to execute → read-operands from memory → emulate (can be any ins of the ISA) → write results in memory → update guestOS registers



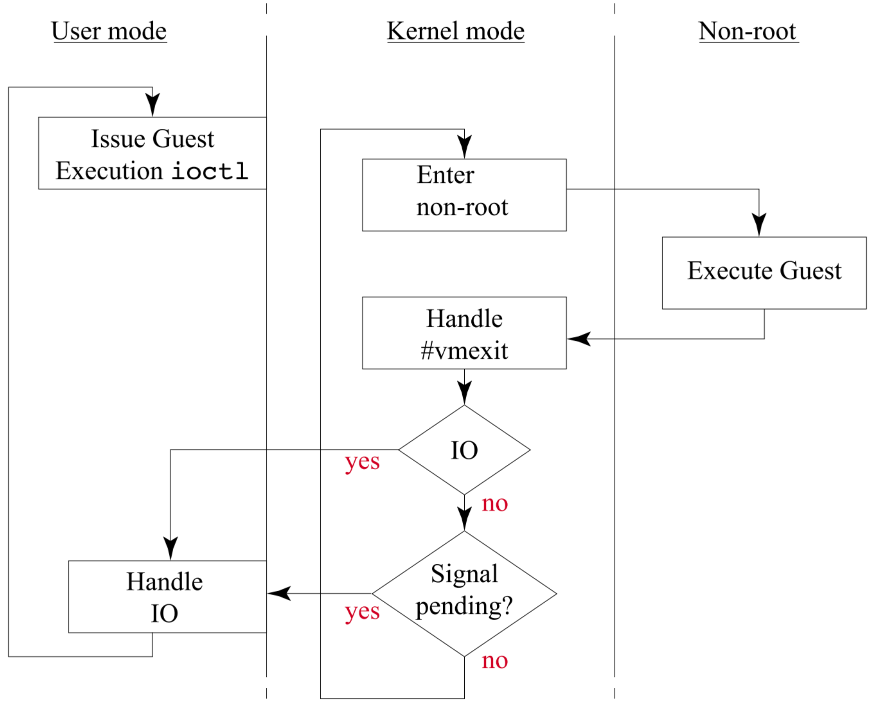
Emulation

- **Fetch**
  - Determine with `%cs:%eip` the offending instruction address in the VM
  - Guest-physical to host-physical and access memory to grab the instruction
- **Decode**
  - Read the opcode from memory: in CISC is non-trivial task (variable legth)
- **Verify**
  - Check if the VM `%cpl` allows the execution of the instruction
- **Read**
  - Using the similar approach of fetch, load instruction operands from memory (if needed)
- **Emulate**
  - A specific emulation routine is executed for the decoded instruction (can be anyone in x86 ISA) (`em_*` on `arch/x86/kvm/emulate.c`) [famous hardcore piece of code]
- **Write**
  - Any write of the resulting emulation will be transferred back to address space of the VM
- **Update**
  - Guest registers and instruction pointer is updated (in the corresponding VMCS)

- In summary: complex, full of intricacies and corner cases. In some cases `vmexit` must generate a trap within the guest-OS, or even in the own emulation routine to the real hardware (v.gr. DIV0)
- Still brittle piece of code: early bugs in the ISA extensions. Hard ISA. 2015 paper identified 72 bugs here.

The role of Host OS in KVM

- Unlike VMWare or VirtualBox, designed for Linux
- From user mode launch guest via syscall `ioctl` to `/dev/kvm`
- Outer Loop
  - KVM executes guestOS until
    - The guestOS issues I/O
    - The host receives a I/O inter (v.gr. timer)
  - QEMU device emulator emulates I/O request (if needed) in user space
  - Host I/O interrupt go back to first step when host decide (host controls sched.)



- Inner Loop
  - Restore current state of the vCPU
  - Enters non-root with `vmresume` (util VM pe
  - Handles `vmexit`
  - If the exit reason was IO (via interrupt or I/O memory mapped access) break the loop and go back to user space

Performance Considerations

- **Atomic** transitions between modes do not imply high speed (and by any means single-cycle execution time!)
- Might **stall** the execution pipeline
- Might require **substantial code** in the processor microcode firmware (in some cases directly wired in the pipeline (?))
- E.g. `vmexit` with a NULL handler in the hypervisor

Microarchitecture	Launch Date	Cycles
Prescott	3Q05	3963
Merom	2Q06	1579
Penryn	1Q08	1266
Nehalem	3Q09	1009
Westmere	1Q10	761
Sandy Bridge	1Q11	784

Other instructions

Processor	Prescott	Merom	Penryn	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell
VMXON	243	162	146	302	108	98	108	116
VMXOFF	175	99	89	54	84	76	73	81
VMCLEAR	277	70	63	93	56	50	101	107
VMPTRLD	255	66	62	91	62	57	99	109
VMPTRST	61	22	9	17	5	4	43	44
VMREAD	178	53	26	6	5	4	5	5
VMWRITE	171	43	26	5	4	3	4	4
VMLAUNCH	2478	948	688	678	619	573	486	528
VMRESUME	2333	944	643	402	460	452	318	348
vmexit/vmcall	1630	727	638	344	365	334	253	265
vmexit/cpuid	1599	764	611	389	434	398	327	332
vmexit/#PF	1926	1156	858	569	507	466	512	531
vmexit/IOb	1942	858	708	427	472	436	383	397
vmexit/EPT	N/A	N/A	N/A	546	588		604	656