X86-64: CPU Virtualization With VT-x

[Hardware and Software Support For Virtualization]

Chapter 4

Design Goal and Requirements (Intel's take)

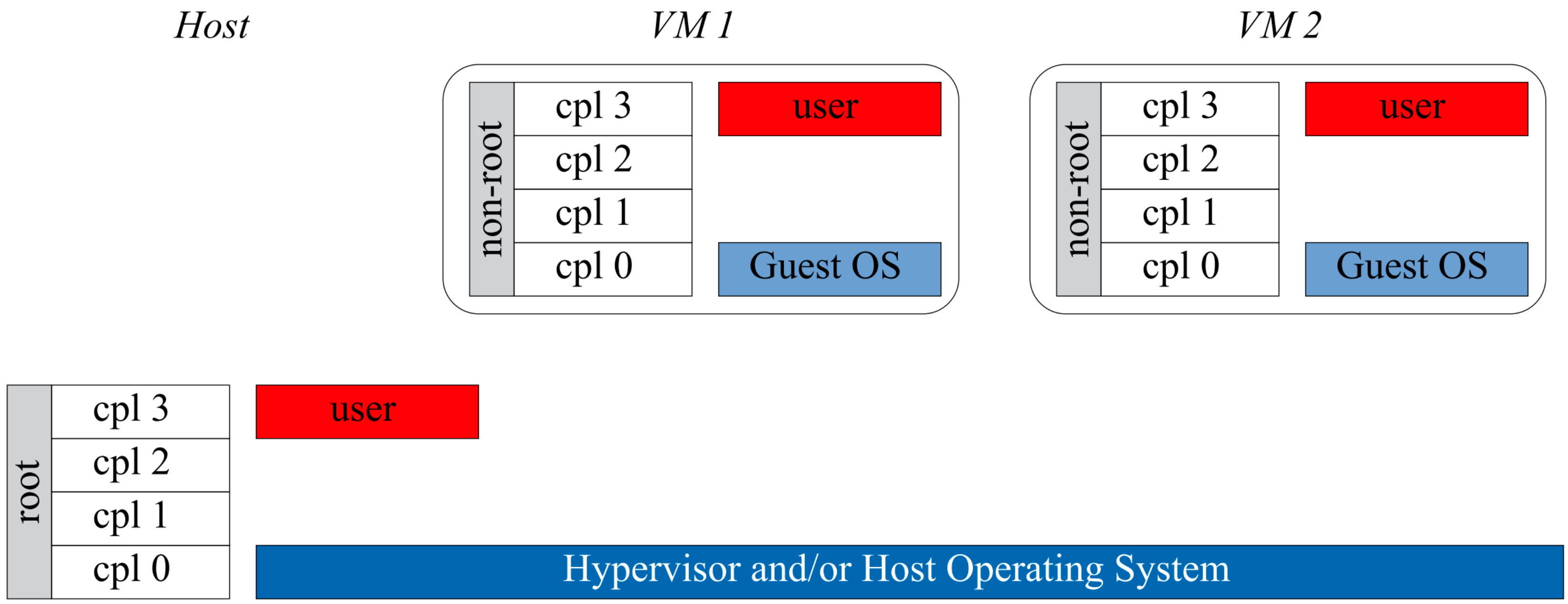
- **Objective**: Eliminate the necessity of DBT and paravirtualization
  - Enable VMM with a broader range if guest-OS
  - Improve performance
- Challenges identified with x86-32 Software techniques
  - **Ring aliasing and compression**: two levels in the guest-OS will run in the same `%cpl` (and still isolated?)
  - **Address Space Compression**: hypervisor should use a non-accessible address space in the guest-OS
  - **Non-faulting access to privileged state**: e.g., `sidt` and `sgdt` access read-mode to descriptor tables (int. and segments) in user mode
  - **Interrupt virtualization**: e.g., `%eflags.if` (pending interrupt) is visible to to user code via `pushf. popf` is control-sensitive (`%cpl =< %eflags.iopl` can control interrupts in user mode!)
  - **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**: e.g., virtualize MS-DOS!
- **Safety**
  - **Minimize hypervisor code** responsibility on security and isolation.
  - Minimize the **attack surface** in the VMM (by using simpler hypervisors)
- **Performance**
  - **Wasn't a goal** in the first release (beyond state of the art). Just setting up the roadmap for future 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 are **atomic** (not a convoluted sequence of instructions like in a context-switch or word-switch)
- Root mode can only be detected by using some specific set of instructions (**not by memory contents**). Required for **VM nesting**.
- Used only for virtualization and orthogonal to others (real mode, v8086, protected,…). Can be used in both modes. **All rings** are also available in both modes
- Each mode uses a separate 64-bit linear address space (%cr3). Only the 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 strictly follow P/G criteria
  - Doesn't take into account whether an in 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

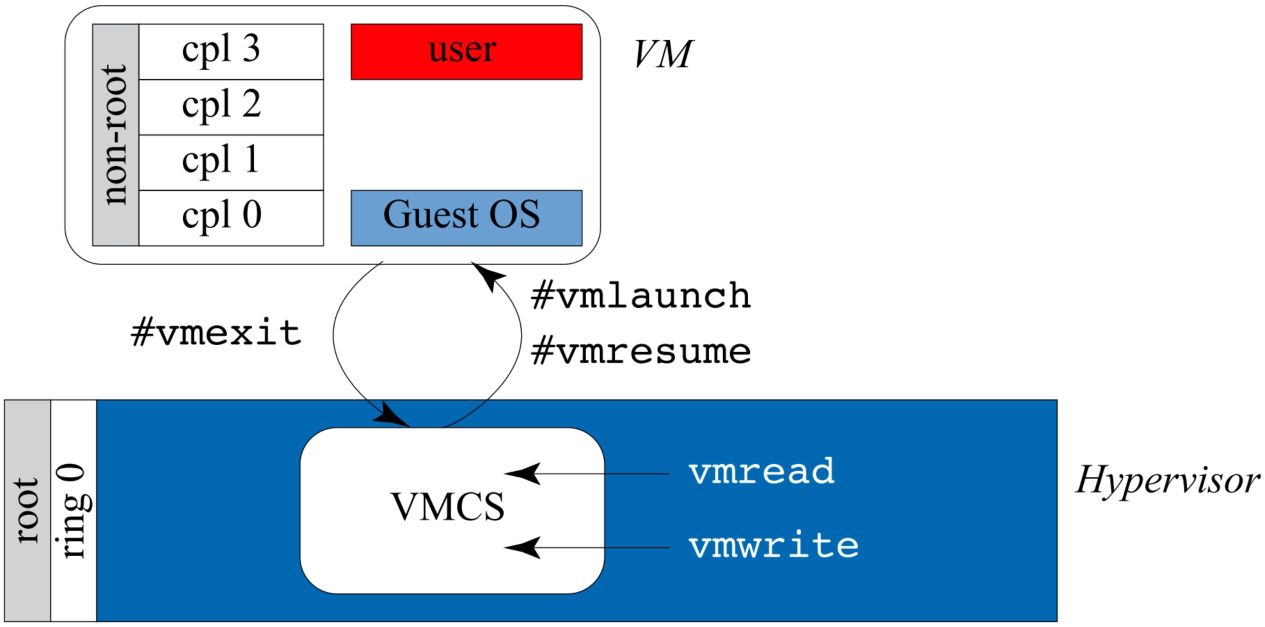
- Let's assume guest-OS in `non-root-%cpl=0` issue a privileged ins such as read control registers (in the **non-root duplicate processor** state) either:
  1) CPU performs directly the operation in the **duplicate state**
  2) **Inform** the hypervisor (via **trap**)
  - First is desired (less transitions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

- Example 2: Instructions than manipulate interrupts flag (e.g., `popf`) are sensitive
  - Are user-sensitive
  - Can be used quite frequently by modern OS
  - Specific implementation in non-root mode

- Example 3: User level read-access to int. and segm. description tables

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)



VMCS Content (Duplicate State)

VM State

GUEST STATE AREA

CR0	CR3	CR4
DR7		
RSP	RIP	RFLAGS

CS	Selector	Base Address	Segment Limit	Access Right
SS	Selector	Base Address	Segment Limit	Access Right
DS	Selector	Base Address	Segment Limit	Access Right
ES	Selector	Base Address	Segment Limit	Access Right
FS	Selector	Base Address	Segment Limit	Access Right
GS	Selector	Base Address	Segment Limit	Access Right
LDTR	Selector	Base Address	Segment Limit	Access Right
TR	Selector	Base Address	Segment Limit	Access Right
GDTR	Selector	Base Address	Segment Limit	Access Right
IDTR	Selector	Base Address	Segment Limit	Access Right

IA32_DEBUGCTL	IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	IA32_BNDCFGS
SMBASE			

Activity state	Interruptibility state
Pending debug exceptions	
VMCS link pointer	
VMX-preemption timer value	

Page-directory-pointer-table entries	PDPTE0	PDPTE1	PDPTE2	PDPTE3
Guest interrupt status				
PML index				

Host State

HOST STATE AREA

CR0	CR3	CR4

RSP	RIP

CS	Selector	
SS	Selector	
DS	Selector	
ES	Selector	
FS	Selector	Base Address
GS	Selector	Base Address
TR	Selector	Base Address
GDTR	Base Address	
IDTR	Base Address	

IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER

- Natural-Width fields.
- 16-bits fields.
- 32-bits fields.
- 64-bits fields.

VMCS Content (Control/outs for the VMM)

Control Fields

CONTROL FIELDS

Pin-Based VM-Execution Controls	External-interrupt exiting	NMI exiting	Virtual NMIs	
	Activate VMX-preemption timer	Process posted interrupts		
Primary processor-based VM-execution controls	Interrupt-window exiting	Use TSC offsetting		
	HLT exiting	INVLPG exiting	MWAIT exiting	RDPMC exiting
	RDTSC exiting	CR3-load exiting	CR3-store exiting	CR8-load exiting
	CR8-store exiting	Use TPR shadow	NMI-window exiting	MOV-DR exiting
	Unconditional I/O exiting	Use I/O bitmaps	Monitor trap flag	Use MSR bitmaps
	MONITOR exiting	PAUSE exiting	Activate secondary controls	
Secondary processor-based VM-execution controls	Virtualize APIC accesses	Enable EPT	Descriptor-table exiting	Enable RDTSCP
	Virtualize x2APIC mode	Enable VPID	WBINVD exiting	Unrestricted guest
	APIC-register virtualization	Virtual-interrupt delivery	PAUSE-loop exiting	
	RDRAND exiting	Enable INVPCID	Enable VM functions	VMCS shadowing
	Enable ENCLS exiting	RDSEED exiting	Enable PML	EPT-violation #VE
	Conceal VMX non-root operation from Intel PT	Enable XSAVES/XRSTORS		
	Mode-based execute control for EPT	Use TSC scaling		
Exception Bitmap	I/O-Bitmap Addresses	TSC-offset		
Guest/Host Masks for CR0	Guest/Host Masks for CR4	Read Shadows for CR0	Read Shadows for CR4	
CR3-target value 0	CR3-target value 1	CR3-target value 2	CR3-target value 3	CR3-target count
APIC Virtualization	APIC-access address	Virtual-APIC address	TPR threshold	
	EOI-exit bitmap 0	EOI-exit bitmap 1	EOI-exit bitmap 2	EOI-exit bitmap 3
	Posted-interrupt notification vector	Posted-interrupt descriptor address		
Read bitmap for low MSRs	Read bitmap for high MSRs	Write bitmap for low MSRs	Write bitmap for low MSRs	
Executive-VMCS Pointer	Extended-Page-Table Pointer	Virtual-Processor Identifier		
PLE_Gap	PLE_Window	VM-function controls	VMREAD bitmap	VMWRITE bitmap
ENCLS-exiting bitmap	PML address			
Virtualization-exception information address	EPTP index	XSS-exiting bitmap		

vmexit config

VM-EXIT CONTROL FIELDS

VM-Exit Controls	Save debug controls	Host address space size	Load IA32_PERF_GLOBAL_CTRL		
	Acknowledge interrupt on exit	Save IA32_PAT	Load IA32_PAT	Save IA32_EFER	Load IA32_EFER
	Save VMX preemption timer value	Clear IA32_BNDCFGS	Conceal VM exits from Intel PT		
VM-Exit Controls for MSRs	VM-exit MSR-store count	VM-exit MSR-store address			
	VM-exit MSR-load count	VM-exit MSR-load address			

vmexit helpers

VM-EXIT INFORMATION FIELDS

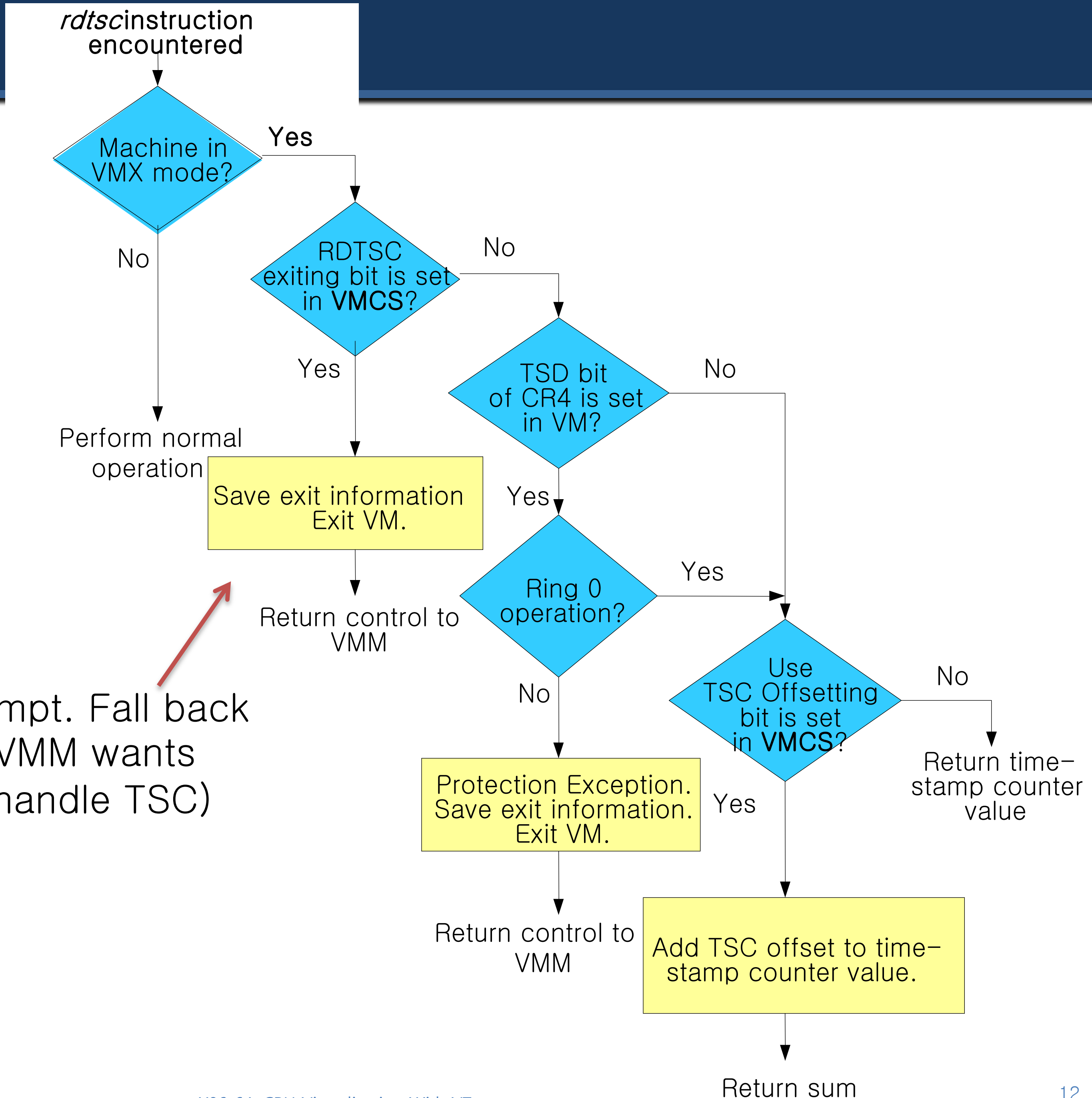
Basic VM-Exit Information	Exit reason	Exit qualification		
	Guest-linear address	Guest-physical address		
VM Exits Due to Vectored Events	VM-exit interruption information	VM-exit interruption error code		
VM Exits That Occur During Event Delivery	IDT-vectoring information	IDT-vectoring error code		
VM Exits Due to Instruction Execution	VM-exit instruction length	VM-exit instruction information		
	I/O RCX	I/O RSI	I/O RDI	I/O RIP
VM-instruction error field				

Hardware Emulation

- Programmable VM exit conditions given in VMCS
  - E.g., which instructions should cause exit to VMM

- **Example**: Read Time Stamp Counter (`rdtsc`)
  - Reads *Time-stamp register* IA32_TIME_STAMP_COUNTER (a MSR o Model-specific Register) in a GP registers
  - If rdtsc exiting is **off**, compatibility fall back
  - If rdtsc exiting is **on**, use hardware assist
    - Hardware emulation if **TSD** bit in control register 4 (CR4) is **off**
    - If TSD **is** on, RDTSC does:
      - If ring == 0, ignores TSD
      - If ring !=0, traps (*protection mode exception*)

RDST



Compt. Fall back (if VMM wants to handle TSC)

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 (i.e., via shadow page table)

- This choice make to fail performance criteria
  - Over 90% of the `vmexit` were due to shadow paging, which is the worst overall performance when using VT-X.

- Purely software might be faster!
  - VMWare, via DBT, **avoids** most guestOS 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 (userspace)*

- CPU and Memory virtualization is closely integrated with Linux Kernel 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,…
- Leveraging of two previous FOSS projects:
  - **QEMU**: I/O emulation (KVM is the main driver of QEMU evolution)
  - **Xen**: X86 initial emulation come from early Xen versions (divergent paths)

The KVM Kernel Module (`kvm.ko`)

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

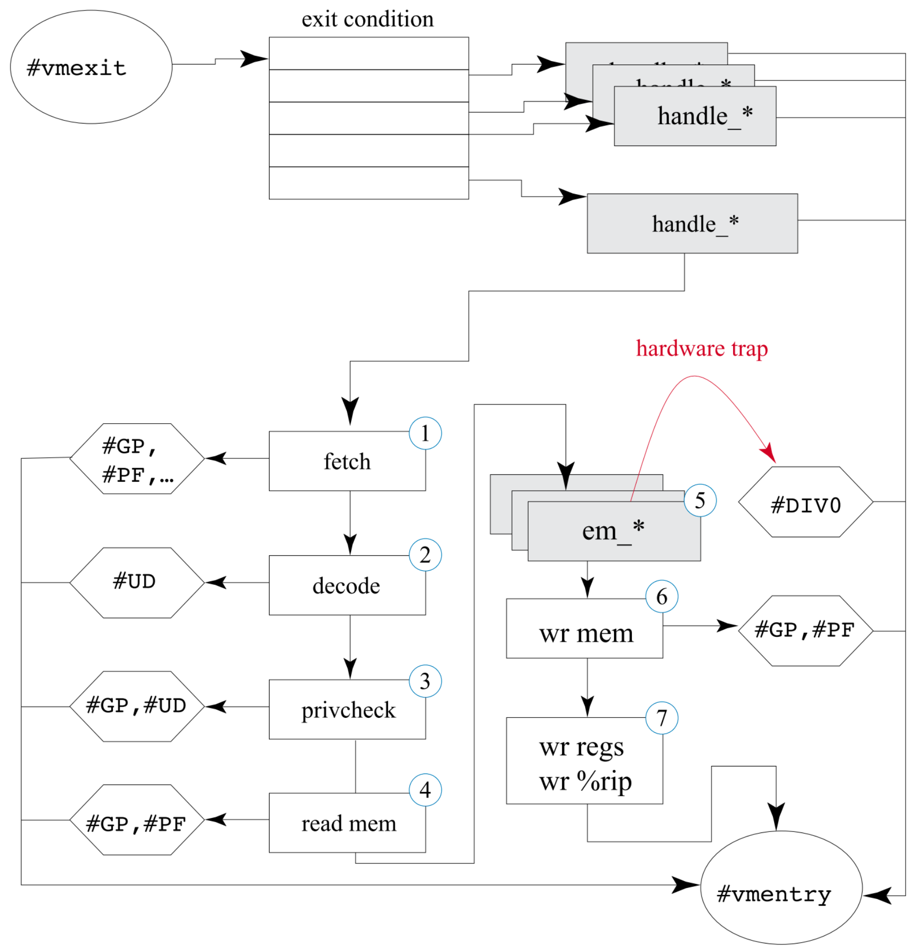
- One might think that since hardware is P/G compliant, KVM should be straightforward

- In reality it is **complex** (e.g., 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. Cases:
  1. Emulate the instruction semantics (often via VMCS) and PC+1
  2. If fault or interrupt, forward to the guestOS, preparing the **trap-frame**
  3. Change the underlying env and retry (EPT violations)
  4. Do nothing. External interrupt.
  - `handle_*` on `arch/x86/kvm/vmx.c`

- Unfortunately, VMCS information does not suffice (at times) to handle exit cause (1)
  - Requires emulating the “offending” instruction to achieve equivalence
  - 5+ KLOC in a general-purpose emulator on `em_*` `arch/x86/kvm/emulate.c`)



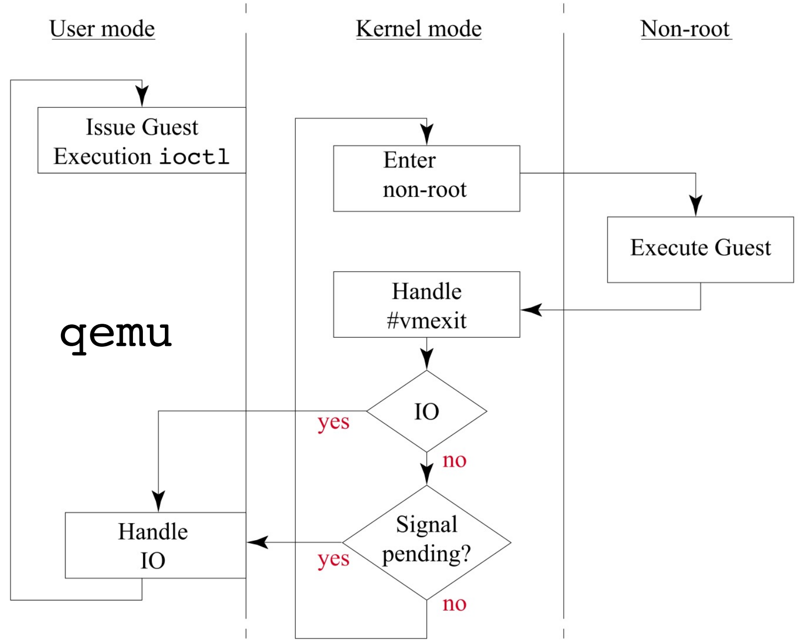
Emulation

- **Fetch**
  - Determine with `%cs:%eip` the offending instruction address in the VM
  - Guest-physical to host-physical prior access memory to grab the instruction
- **Decode**
  - Read the opcode from memory: in CISC is non-trivial task (variable length)
- **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 anything 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 (e.g., 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 (e.g. specific `perf` mode… hard in Xen)
- Outer Loop
  - From user mode launch guest via syscall `ioctl` to `/dev/kvm`
  - KVM executes guestOS until
    - The guestOS issues I/O
    - The host receives an extern. I/O inter or timer
  - QEMU device emulator emulates I/O request (if needed) in user space
  - Timer interrupt go back to kernel where host decide (host controls scheduler)



- Inner Loop
  - Restore current state of the vCPU
  - Enters non-root with `vmresume` (util VM performs `vmexit`)
  - 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

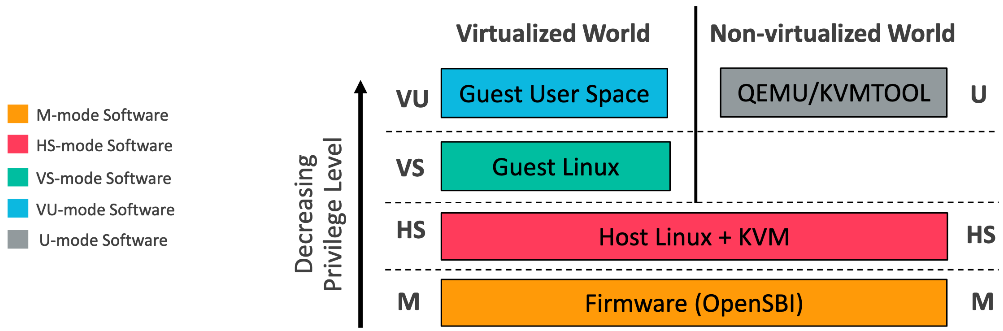
Microarchitecture	Launch Date	Cycles
Prescott	3Q05	3963
Merom	2Q06	1579
Penryn	1Q08	1266
Nehalem	3Q09	1009
Westmere	1Q10	761
Sandy Bridge	1Q11	784

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

RISC-V Virtualization Support

- ISA is P/G virtualizable (CS and BS are privileged)
- Hypervisor Extension "H" added for performance reasons



- Similar idea to VT-x (VMCS) but way simpler (CSRs)
- Support in QEMU, kvm,.. (also emulated)
  - https://github.com/torvalds/linux/tree/master/arch/riscv/kvm
- Chapter 9 on privileged instructions (~38 pages) ratified
  - 15 instructions
- Real hardware will start to support it (e.g., SiFive P6X0) [~ARM A78 performance (2020)].