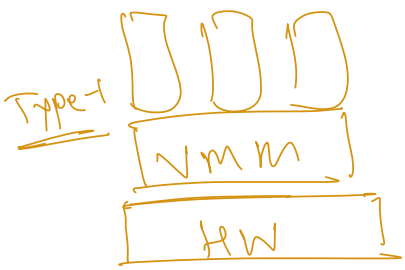
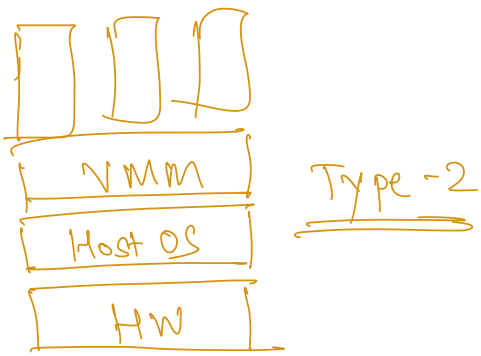
- What we are building is called VMM.

- 2 types.



⇒ Xen
⇒ VMware workstation

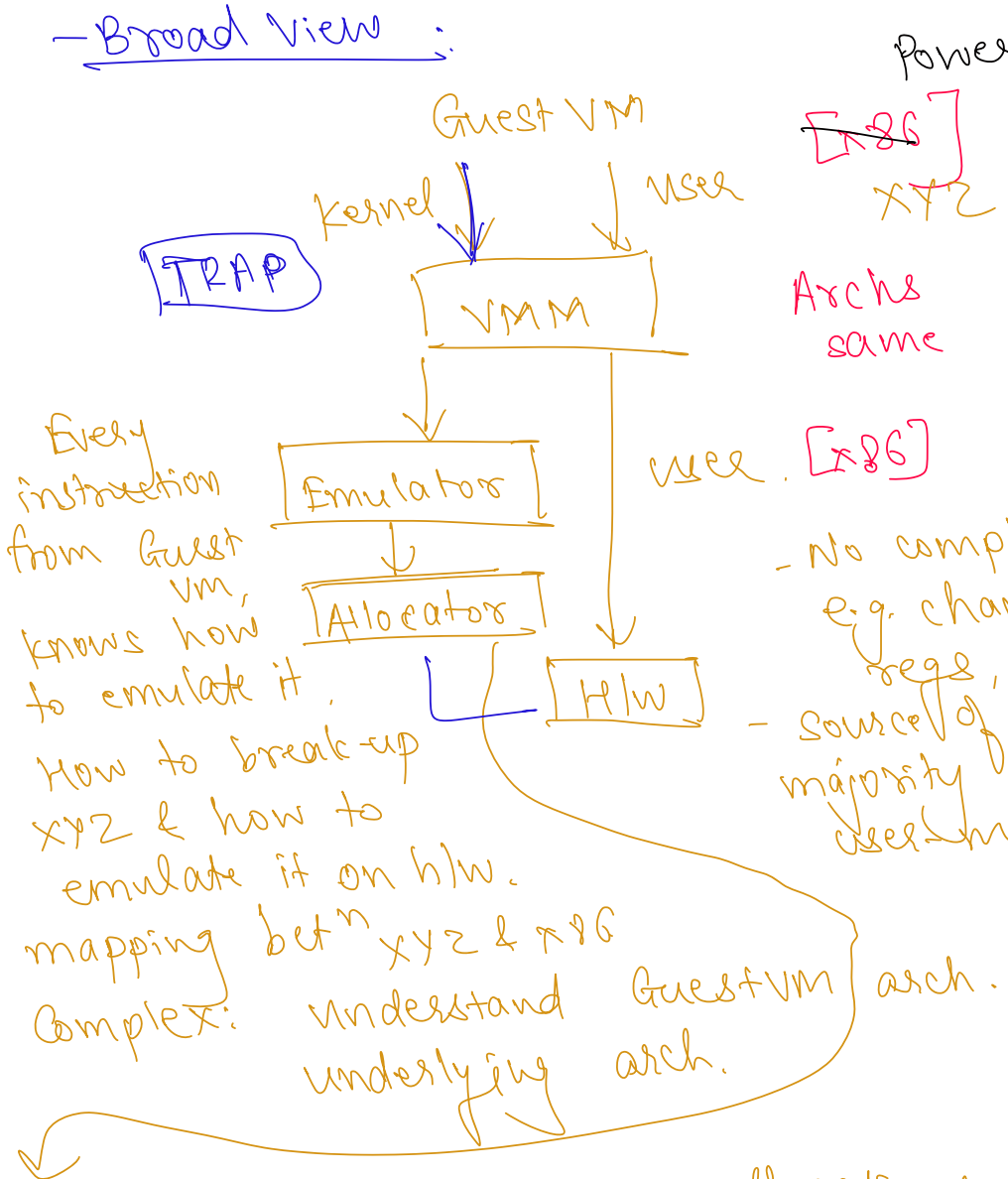
- VMM directly on h/w. No Host OS
- Responsibility of VMM to manage h/w resources & divide among diff. VMs



⇒ QEMU
⇒ VirtualBox

- Specific Host OS that controls h/w
- VMM works in conjunction with host OS (e.g. KVM) works with Linux to manage h/w & multiplex
- In VirtualBox, host OS thinks it is just another application
  - VBox spawns processes for VMs
- Some cases: OS unaware
- Some cases: Tighter coupling.

- Broad View :



PowerPC.

~~x86~~] xyz

Archs same

When guestVM: memory allocations, it thinks
- it owns whole machine.
- somebody: intercept it & allocate real h/w resources so that progs can run.
e.g. Guest VM: create process that requires 4GB DRAM. Has to exist somewhere.

- If no Host O.S, everything done by Vmm.

- If Host O.S., allocator managed by Allocator. Host O.S. has access to h/w resources.

Architectural req: Kernel mode instruction must be intercepted.
Goldberg & Popek : 3 theorems.

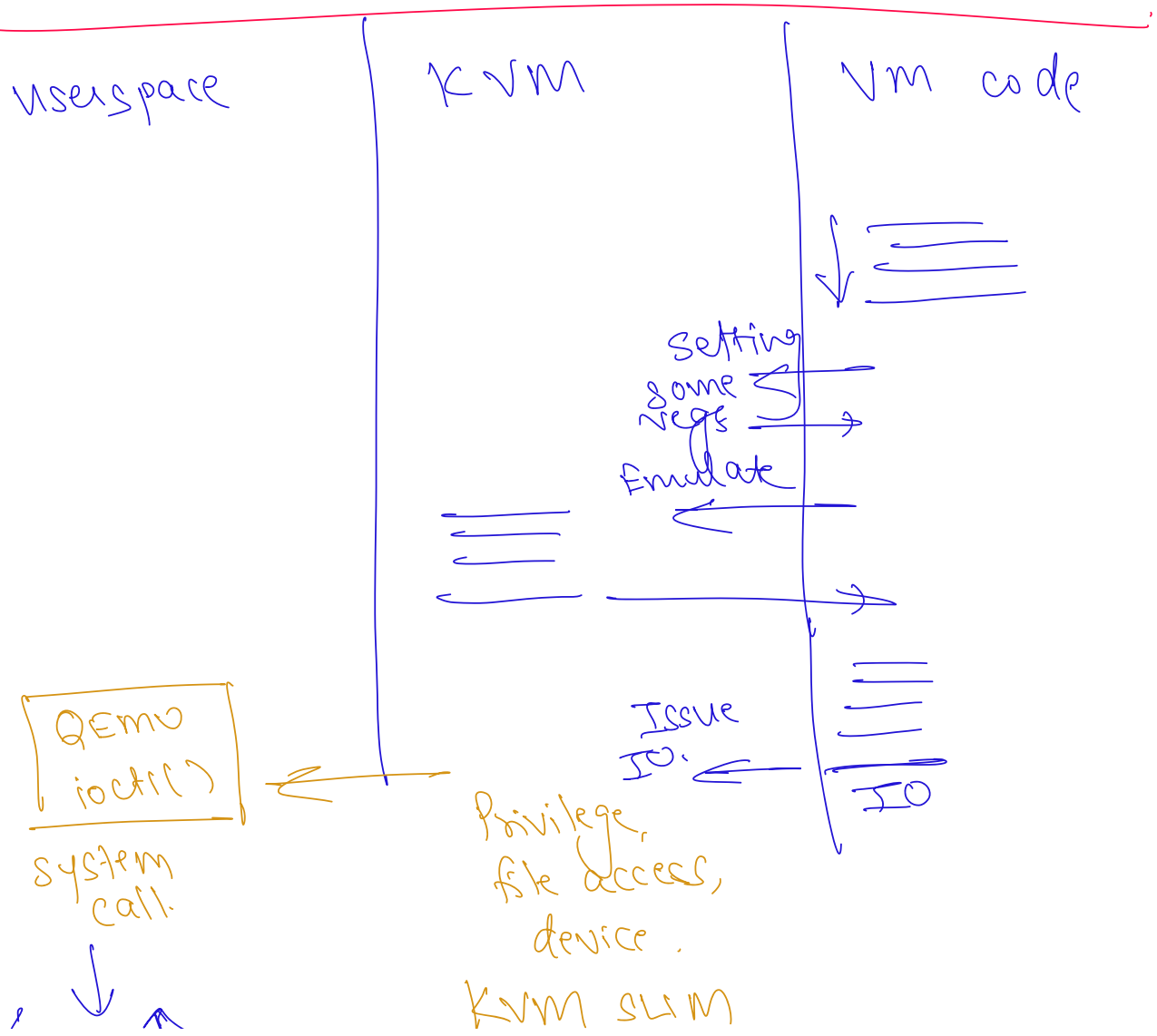
Privileged inst. : Directly manipulating h/w:
Allocating mem.,
creating I/O req.

Sensitive inst. : They read data.
e.g. read current addr of interrupt table. (confidential)
user-mode inst. that does this w/o trap.
must cause a TRAP

QEMU: Full system simulator.
- can simulate x86, PowerPC.
- w/o any support is fully capable of simulating it.
- Does the loop: Every instruction: emulate using h/w ...

- no hardware support needed.
- Not efficient. Slow
- KVM: Accelerate this. use hlw for some things.
  - Helps get some hlw resources of machine
  - KVM represents host O.S.

WHAT IS THE USE OF KVM IN QEMU?



I/O Device

↳ very heavy orders of magnitude slower than instructions.

Inst. 0.5 ns

HDD: 10 ms.

Handle I/O out of critical path.

Instantiate I/O. with QEMU, KVM Simmer.

VT-X (2005)

⇒ Hardware support for virt.. Intel 2005.

⇒ Vmware: binary translation

Xen: paravirtualization.

Most popular approach

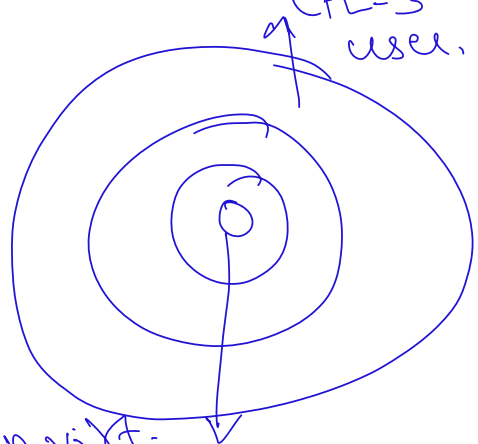
Goal: hardware support for virt. such that unmodified x86-based o.s directly on h/w.

- first version 2005: focus not perf. Just get it functional
- over time: faster.
- In 2005, perf. was inferior than

other approaches

- Since 2005: instructions that VT-x introduces ~5x faster.

CPU Virtualization - Introduce new mode root mode.



x86.

This is how non-virt. O.S. work.

VT-x introduces new mode called root mode

HW → Root mode → 4 rings [ARCH STATE]

HW → non-root mode → 4 rings. [ARCH STATE]

- Before: physical machine had register which VM has access to but only written by hypervisor

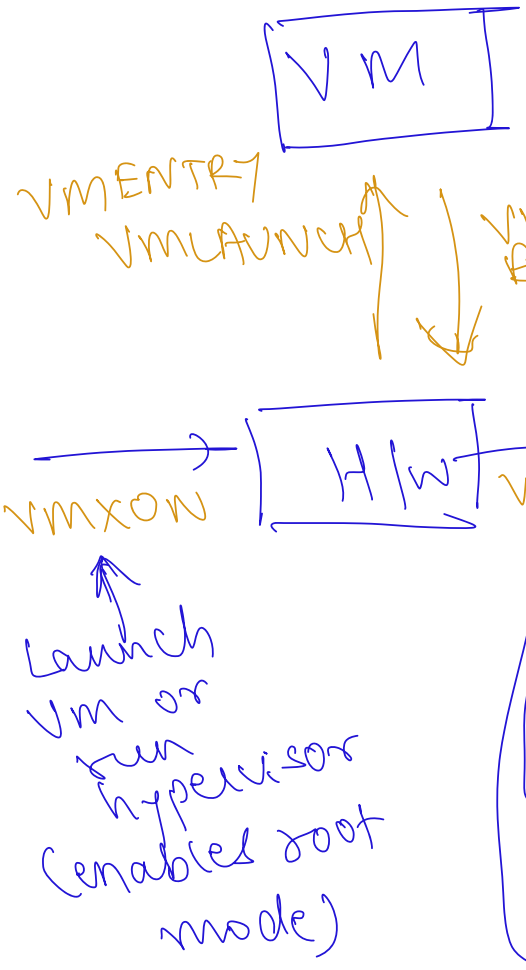
2 sets of resources:

- One set just for root mode
- Another set just for non-root mode.
- Makes it easy. Even if we have sensitive inst. that is going to read state, it is going to be non-root state

[never see root mode state]

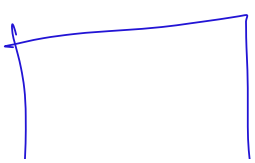
- Fact of root & non-root mode orthogonal to inst. that Intel already had.
- whenever you read reg.: you had no idea whether root or non-root mode
  - Assumed that it is the only privileged reg. in system, but actually 2 copies.
  - unless in root mode, cannot access state associated with root mode.

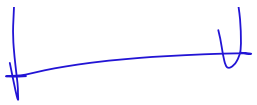
→ Doubled resources on the chip
kept one for hypervisor, other
for V.M



- new instructions Intel made for us:
- from hypervisor, use VMLAUNCH to launch V.M.
- Event from VM that requires hypervisors attention: control returned back to vmm. After this, if you do VMLAUNCH, it fails.

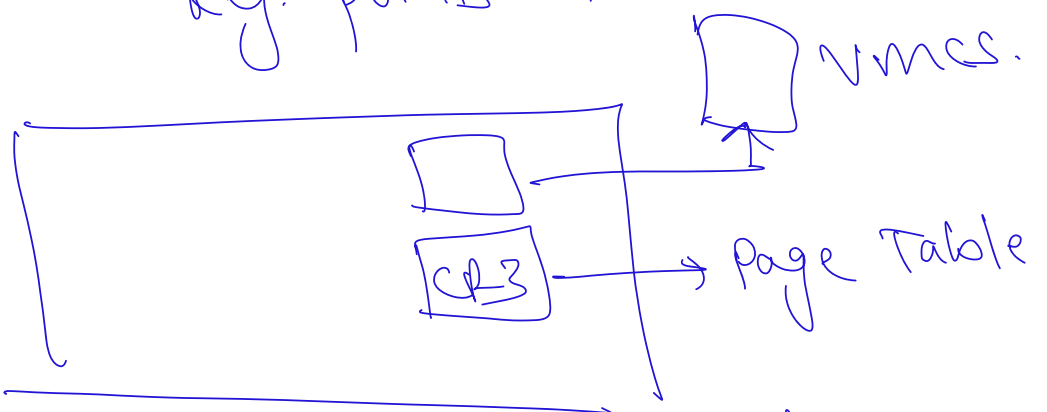
(VMCS)
VM Control Structure → one VMCS with each VM





- Whenever there is VM EXIT for eg., VMCS populated & stored in mem.
- on VMLAUNCH & VMENTRY, data from VMCS used to populate regs in h/w.
  - stored in physical DRAM.

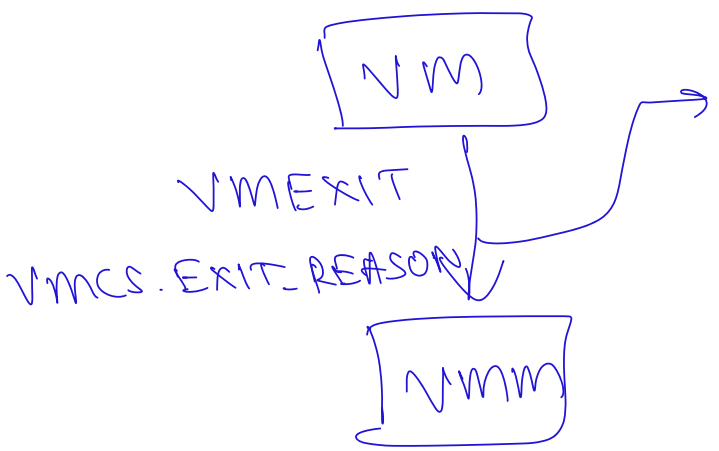
Reg. points to VMCS structure.



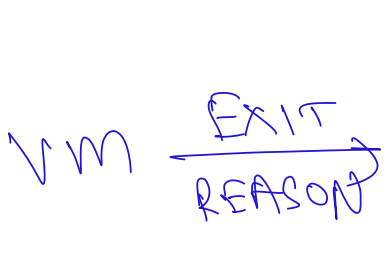
Every time context switch betn V.M, reg. points to diff. VMCS structure.

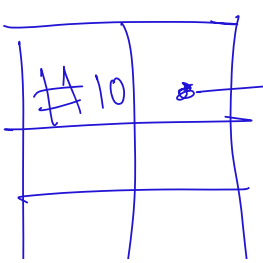
- VMCS struct can only be read using VMREAD (from Vmm)
  - write: VMWRITE (from vmm)

- Structure of VMCS not specified by INTEL.
- only readable & writable by VMREAD & VMWRITE.
- Allows Intel to change how VMCS is stored.
  - Not any guarantee provided by Intel.
- Whenever VMEXIT.



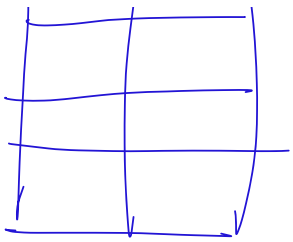
Populated by hw when trapped from VM.
Can be read by VMM
e.g. page fault, div by 0





handled.
EMULATION ALLOCA-

Total 55 diff. reasons for TRAP.
e.g. exception, pte fault.

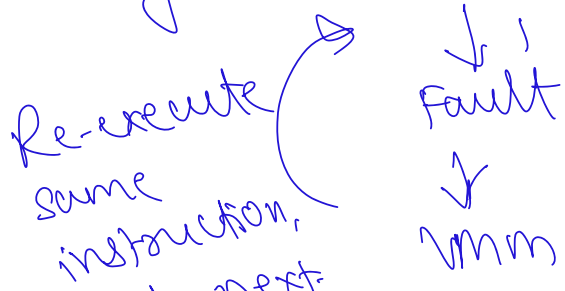


TION

- EXIT REASON: what happened,
= handler invoked & handle it
e.g. if req emulation, can emulate,
if requires allocation resources

- subtle thing in how we handle exit from vmm

1) Retry instruction.
e.g. READ 2000

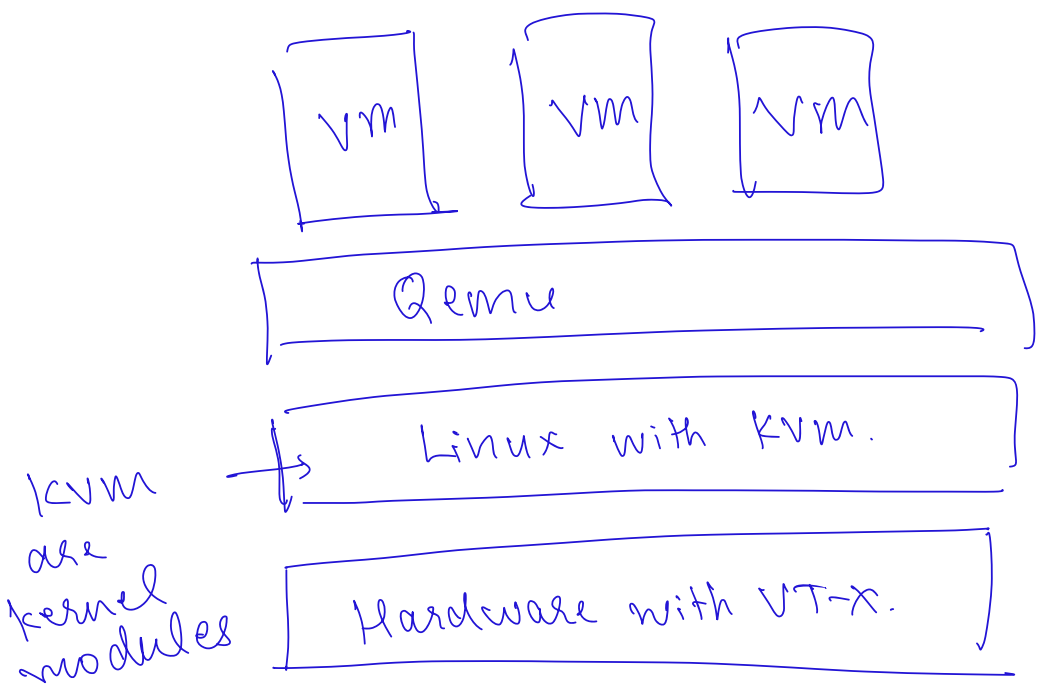


Do not increment instruction ptr.
e.g. I/pointer.

2) Increment i.p.
e.g. h/w doesn't support inst.,
emulate it & increment

ptr by 1.

- Confusion: if trapped always next instruction.



/dev/kvm are kernel modules that turn Linux into a hypervisor.

2 main modules:

kvm.ko → core module

kvm-intel.ko → machine-specific module.

kvm-amd.ko.

KVM: Launches VMLAUNCH | VMRESUME

set up VMCS, deal with VMEXIT.

QEMU talks to kvm using system calls. | ioctl() calls.

QEMU :

user-space component. Can run Guest using PowerPC & host using x86

2-modes: Emulator & Virtualizer

Emulator → Fake hardware; binary translation. No KVM.

Virtualizer → Gets help from KVM

Because QEMU supports emulation, it can emulate I/O:

e.g. assume that VM on QEMU emulates certain serial port that doesn't exist

on host.

When VM tries I/O on port, VM exits to KVM. KVM looks at reason & passes to QEMU.

QEMU emulates the I/O device & passes control to KVM.

KVM executes VMRESUME to let VM proceed.