- Last class: Popek Goldberg Theorem (3)

Sensitive Instructions ⊂ Privileged Instruction

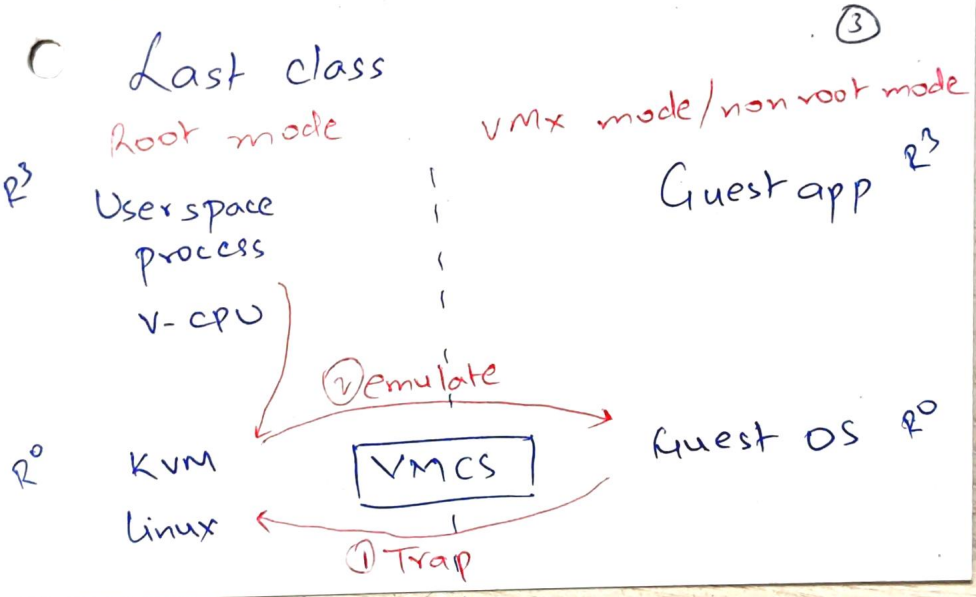
Take control of resources

↳ Trap

eg:- Change CR3
- Change IDT
- Disable Interrupts

Problem with popf

- Last class (3)



- Great performance for CPU benchmarks (2)

- 0-9% slowdown on SPEC benchmark

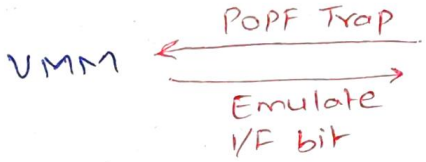
(Figure 2 ASPLOS-06. Comparison of sw hw tech for x86 virt)

- Housekeeping overhead Timer Interrupts

- Exits determine performance for trap-and-emulate hypervisors ③

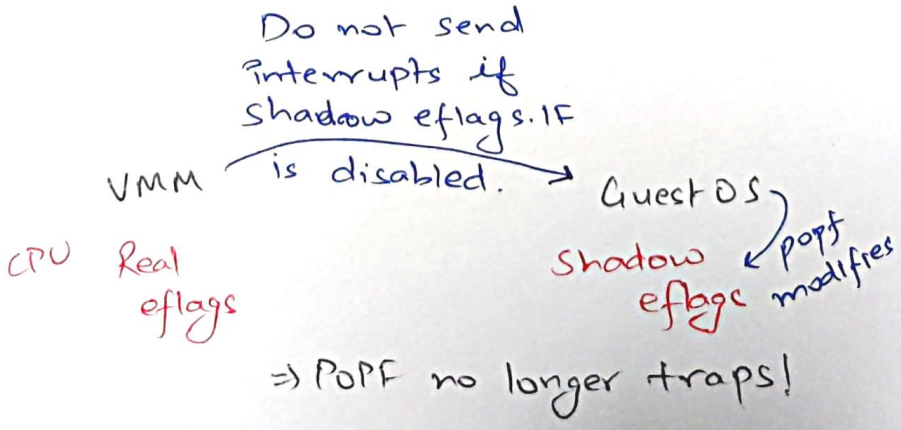
Micro Arch.	Launch date	VMM round trip cycles	Syscall round trip cycles
Nehalem	3Q09	1009	138
Sandy bridge	1Q11	784	134

- Example: guest running popf frequently ①



Solution: Avoid exits as much as possible!

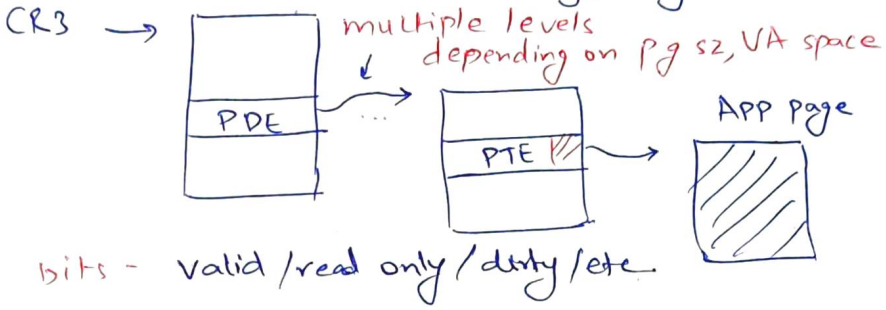
- Dealing with popf: Shadow EFLAGS register in VMCB ②



⇒ POPF no longer traps!

Virtualize memory! Paging ③

Typically 4K



bits - valid / read only / dirty / etc.

Metrics of success ②

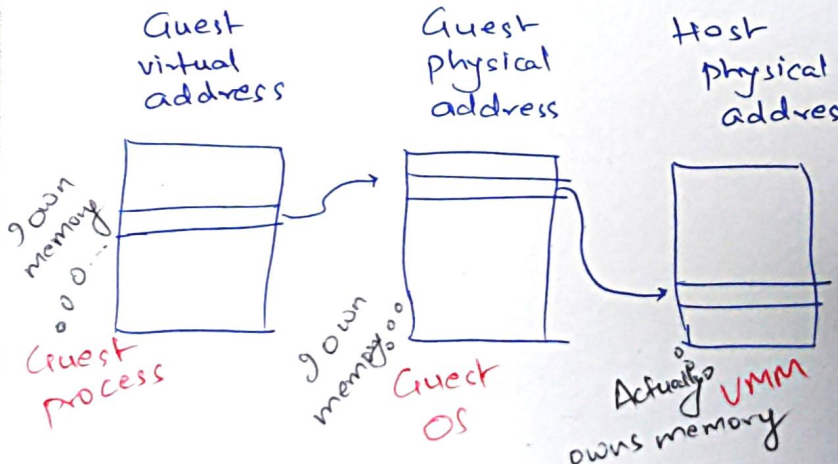
Transparency: Process (→ Guest OS, Guest process) thinks it owns memory

Isolation: One process (→ Guest OS, Guest process) can not read another's memory

Performance: Once mapped, address translation need not involve OS (Guest OS, VMM) Fast load/stores

General idea: ③

Two levels of address translation



Idea 1: Interpretive Execution (Lab) ②

30xx → mem[xx] = AC

Virtual address

array owned by Interpreter

✓ Isolation

✓ Transparency

✗ Performance

eg: IBM System 370
SIE instruction → start Interpretive Exec

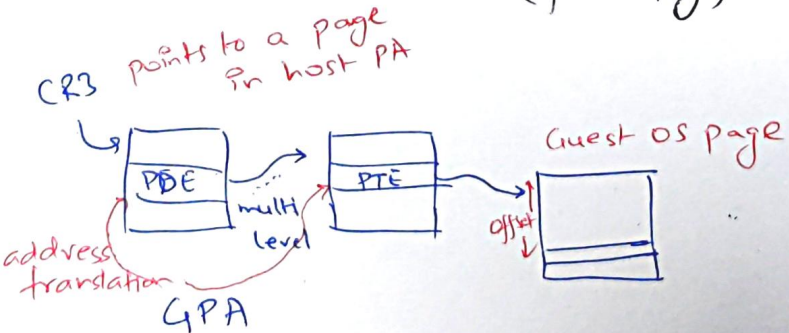
Idea 2: Reuse paging hardware ②

Shadow Page table:

map directly from ⇒ GVA → HPA

⇒ Most memory ops LD/ST run at native speed

Step1: Prepare page table for GPA (for booting) ②



- Now guest OS can boot. (3)
  - Initially, address translation is disabled on real hardware
- CR0, bit 31 can enable-disable paging.

Guest OS tries to disable paging → Trap → VMM changes CR3 to the prev. page table
← emulate

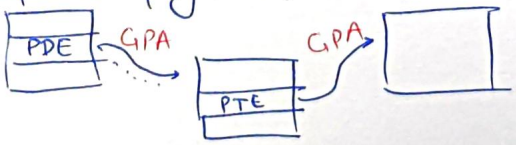
- Guest "thinks" it has disabled paging. (3)

Reading CR0.31 shall { trap or return false { shadow register

But address translation is active.
Addresses treated as GPA

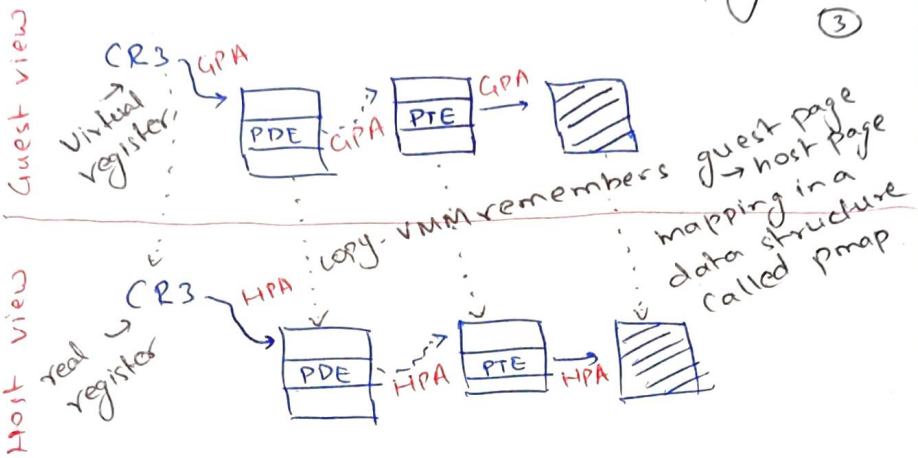
- Guest tries to run a process (2)

① Prepares page table

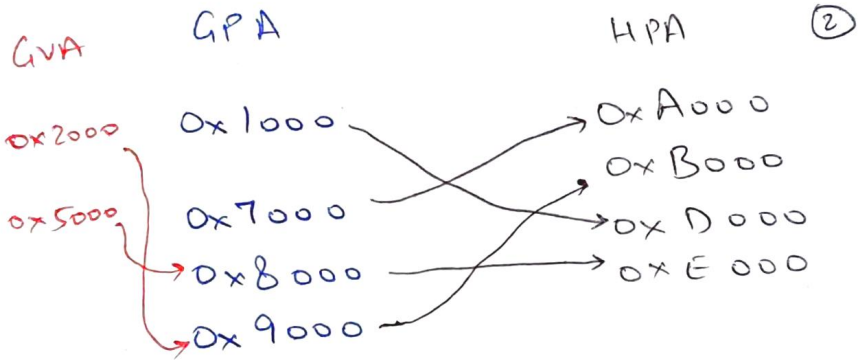


② Changes CR3 to top-level GPA → Trap

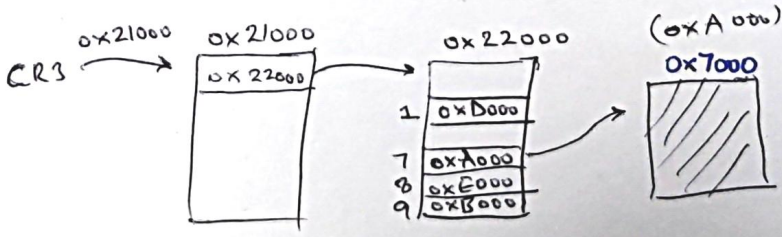
VMM constructs shadow page table (3)



Example: memory mapping setup for VM (2)

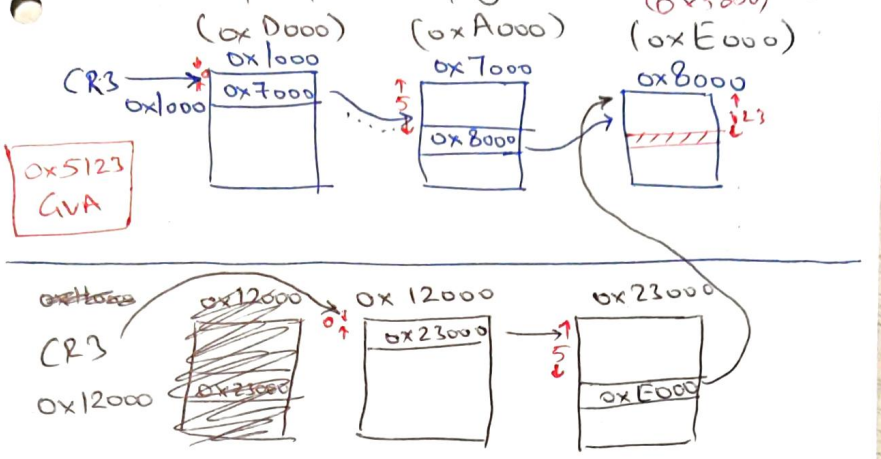


Example: memory setup for VM



When CR0.31 bit is disabled by guest, hypervisor converts GPA → HPA

Guest prepares page table (3)



(3)

- Transparency: If guest reads CR3, it is emulated or kept as Shadow CR3 — 0x1000

Performance: Directly use MMU once page table is setup

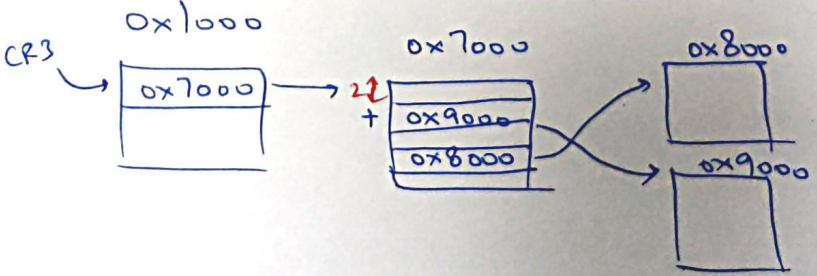
Safety / Isolation: Guest can't directly touch shadow page table. 0x12000, 0x23000 are NOT in guest's address space

(2)

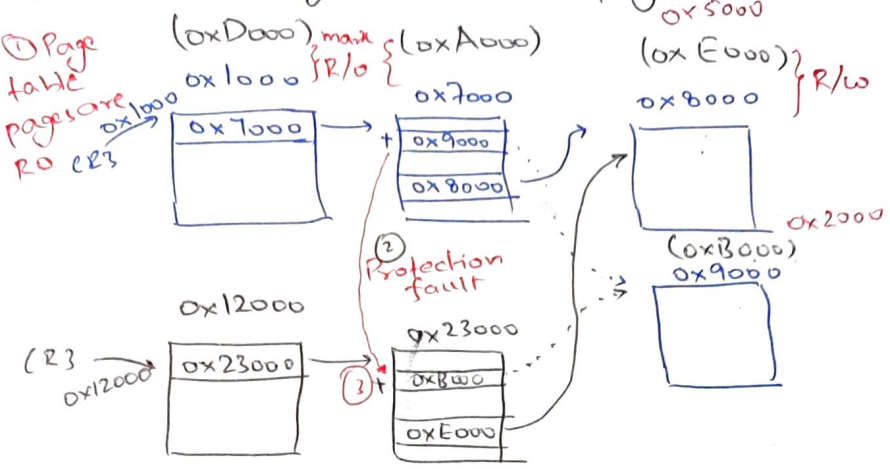
- Guest app mmaped a GVA.

Guest OS did not allocate (lazy)

LD 0x2123 → PF in VMM → emulate → PF in guest OS



Maintaining shadow page table ③



Memory tracing- ②

- Trace writes- mark R/O
- Trace read/writes- mark invalid

Another use- memory mapped I/O

→Memory / performance optimizations ②

Need 1 shadow page table per guest page table.

① VMM deletes a shadow page table

Ok- Rebuild when CR3 is updated back to 0x1000.

Hidden Page fault → Transparent to guest

② Move 0xE000 page to disk. Set "Not Present" in shadow page table. ③

App runs LD 0x5123

↳ Page fault

Bring from disk to 0xF000.

Update ALL shadow page tables pointing to 0xE000

* Maintain backward mappings in "pmap"

③ Move 0xA000 to disk. When ② guest OS tries to update page table.

→ Page fault

Bring to 0xC000

Update GPA → ~~GVA~~ HPA mapping

0x7000 → 0xC000

Summary: ②

① Trap on mv CR3

② Copy PT to shadow PT

③ Give shadow PT directly to h/w for address translation

④ Memory tracing for maintaining consistency

⑤ Pagefault, hidden page faults → backward mapping

- Transparency ✓ Isolation ✓ (2)
- Performance: Native LD/ST in common case ✓
  - Overhead on process creation
  - " " new PTE
  - " " page faults
  - Additional overhead - hidden page fault
- Resource utilization: maintain copy of page table for every guest virtual address space