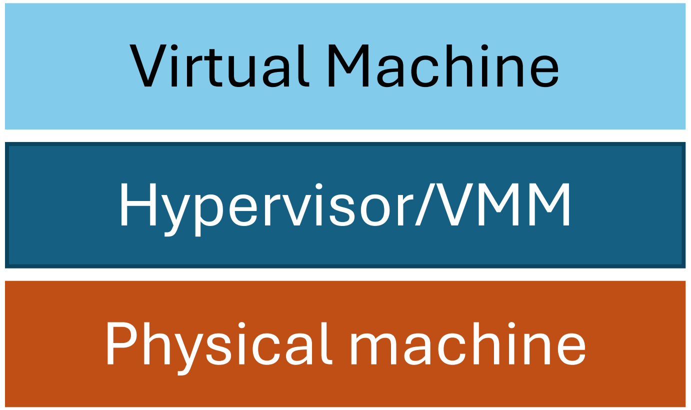
Virtualization

Based on slides borrowed from Mythili, IIT Bombay

Virtualization terminology

- Hypervisor or virtual machine monitor (VMM): a piece of software that allows multiple VMs to run on a physical machine (PM)
- Guest OS runs inside the VM, and host OS runs on the PM

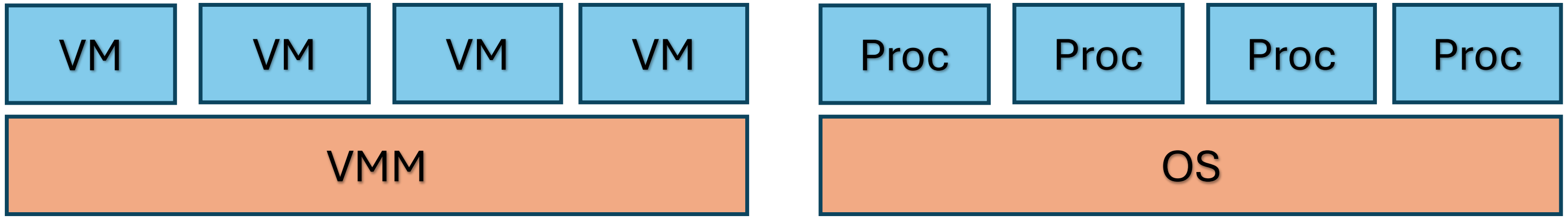


Why VMMs?

- Test and develop operating systems
- Run x86 OS on ARM hardware etc
- Backbone of cloud computing:
  - Renter perspective: I don’t want to buy and manage physical machines.
  - Vendor perspective: I can overprovision and rent out a lot more virtual machines than available physical machines
  - Run competitor virtual machines with full isolation
  - Migrate virtual machines without downtime: load balance across physical machines, shut down physical machine for maintenance/power saving

Virtual machine monitor (VMM)

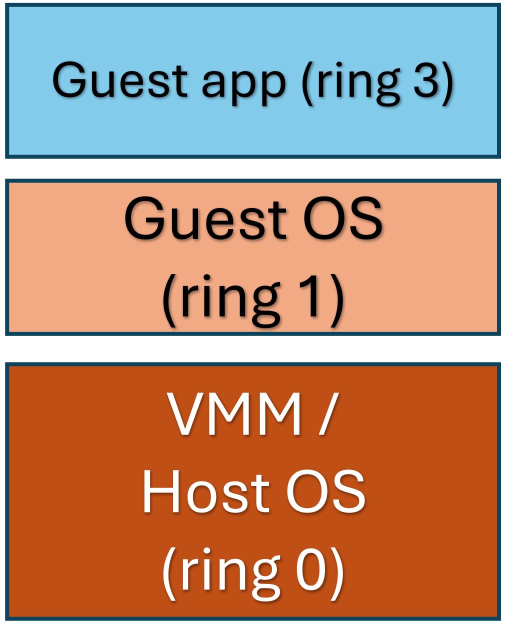
- Multiple Virtual Machines running on a Physical Machine – multiplex the underlying machine. Similar to how OS multiplexes processes on CPU



- VMM performs machine switch (much like context switch)
  - Run a VM for a bit, save context and switch to another VM, and so on…
- What is the problem?
  - Guest OS expects to have unrestricted access to hardware, runs privileged instructions, unlike user processes
  - But one guest cannot get access, must be isolated from other guests

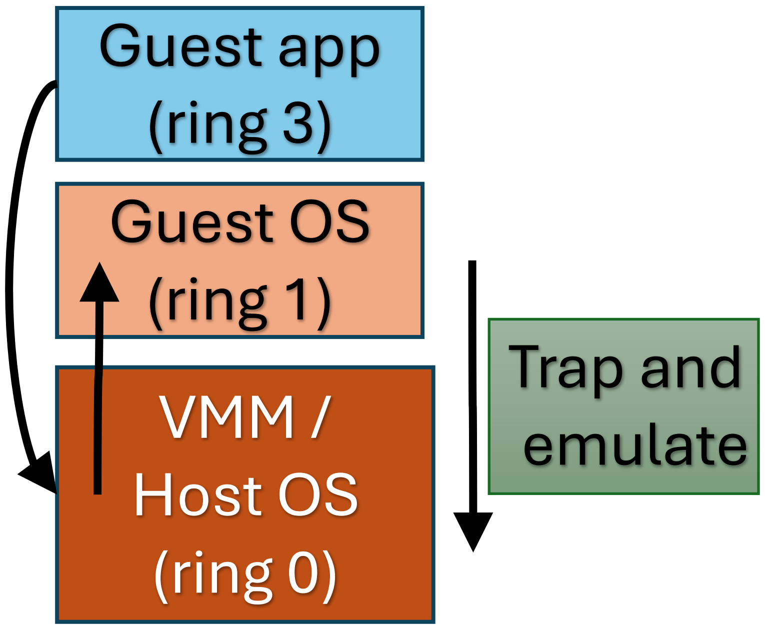
Trap and emulate VMM (1)

- All CPUs have multiple privilege levels
  - Ring 0,1,2,3 in x86 CPUs
- Normally, user process in ring 3, OS in ring 0
  - Privileged instructions only run in ring 0
- Now, user process in ring 3, VMM/host OS in ring 0
  - Guest OS must be protected from guest apps
  - But not fully privileged like host OS/VMM
  - Can run in ring 1?
- Trap-and-emulate VMM: guest OS runs at lower privilege level than VMM, traps to VMM for privileged operation



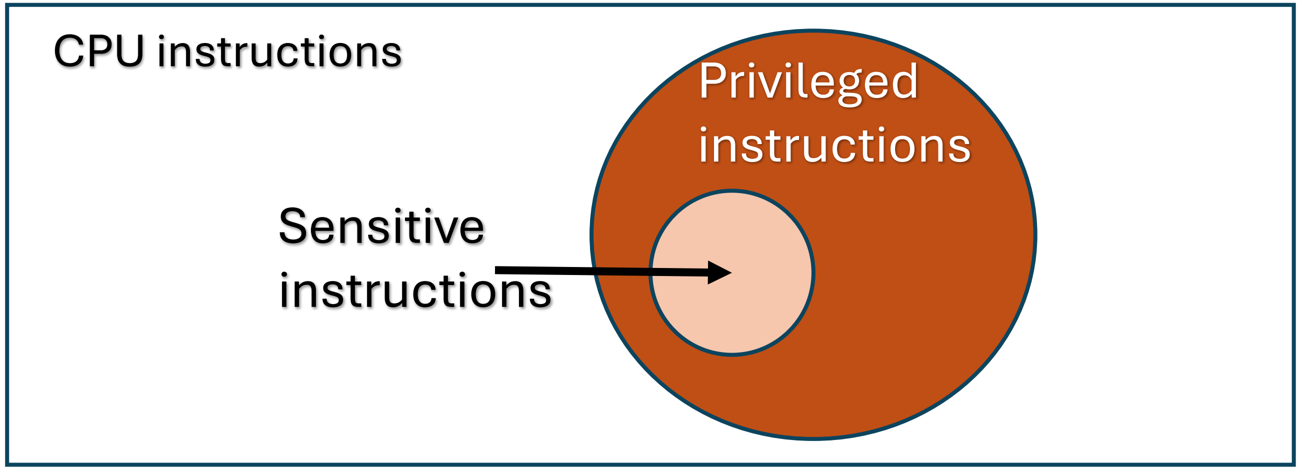
Trap and emulate VMM (2)

- Guest app has to handle syscall/interrupt
  - Special trap instr (int n), traps to VMM
  - VMM doesn't know how to handle trap
  - VMM jumps to guest OS trap handler
  - Trap handled by guest OS normally
- Guest OS performs return from trap
  - Privileged instructions trap to VMM
  - VMM jumps to corresponding user process
- Any privileged action by guest OS traps to VMM, emulated by VMM
  - Example: set IDT, set CR3, enable/disable interrupts
  - Sensitive data structures like IDT must be managed by VMM, not guest OS



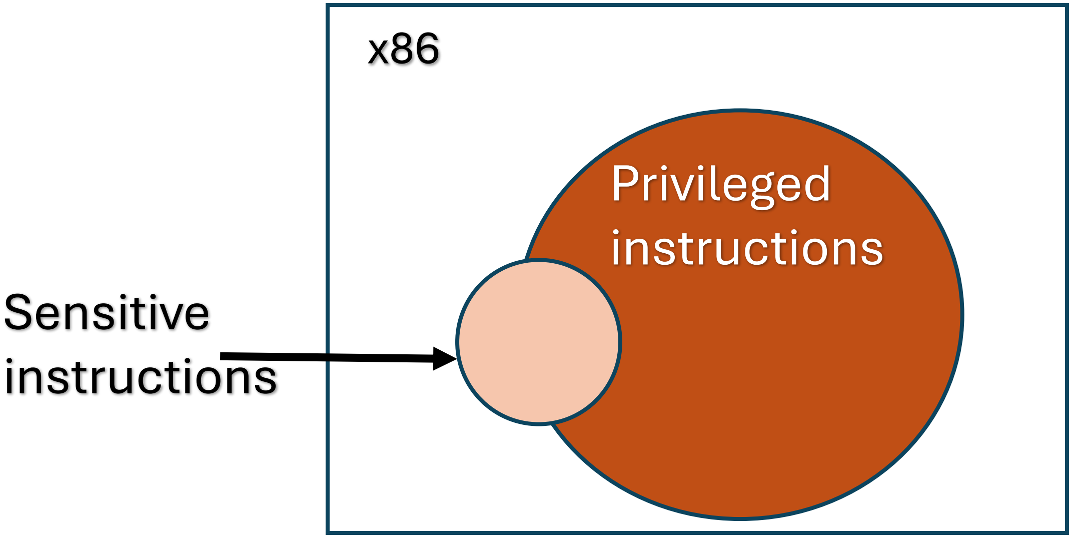
Popek Goldberg theorem

- Sensitive instruction = changes hardware state, e.g. disable interrupts
- Privileged instruction = runs only in privileged mode
  - Traps to ring 0 if executed from unprivileged rings
- In order to build a VMM efficiently via trap-and-emulate method, sensitive instructions should be a subset of privileged instructions



Problems with trap and emulate

- Some x86 instructions which change hardware state (sensitive instructions) do not trap in less privileged ring 1!
- Why these problems?
  - OSes not developed to run at a lower privilege level
  - Instruction set architecture of x86 is not easily virtualizable (x86 wasn't designed with virtualization in mind)



x86 does not follow Popek-Goldberg theorem

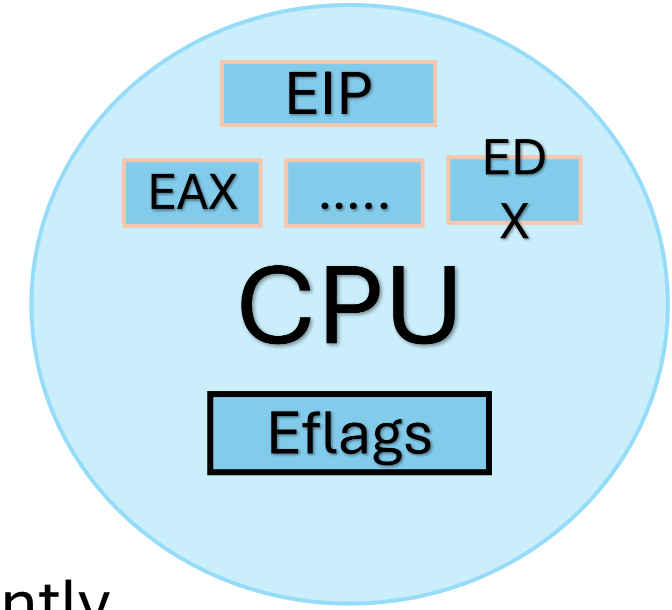
Table 2.2: List of sensitive, unprivileged x86 instructions

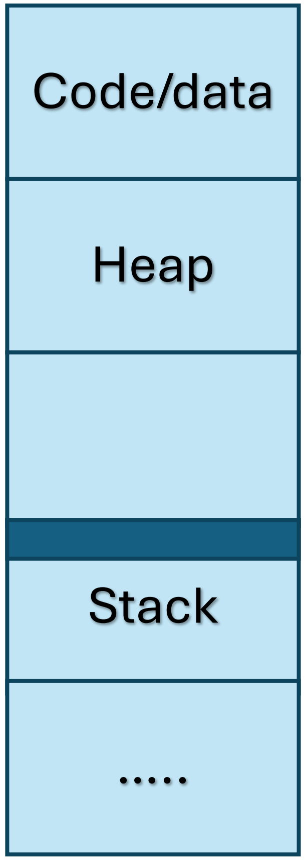
Group	Instructions
Access to interrupt flag	pushf, popf, iret
Visibility into segment descriptors	lar, verr, verw, lsl
Segment manipulation instructions	pop <seg>, push <seg>, mov <seg>
Read-only access to privileged state	sgdt, sldt, sidt, smsw
Interrupt and gate instructions	fcall, longjump, retfar, str, int <n>

Robin et.al. USENIX Security, 2000

Example: Problems with trap and emulate

- Eflags register is a set of CPU flags
  - IF (interrupt flag) indicates if interrupts on/off
- Consider the popf instruction in x86
  - Pops values on top of stack and sets eflags
- Executed in ring 0, all flags set normally
- Executed in ring 1, only some flags set
  - IF is not set as it is privileged flag
- So, popf is a sensitive instruction, not privileged, does not trap, behaves differently when executed in different privilege levels
  - Guest OS is buggy in ring 1





Techniques to virtualize x86 (1)

- Paravirtualization: rewrite guest OS code to be virtualizable
    - Guest OS won't invoke privileged operations, makes "hypercalls" to VMM
    - Needs OS source code changes, cannot work with unmodified OS
    - Example: Xen hypervisor
- Full virtualization: CPU instructions of guest OS are translated to be virtualizable
    - Sensitive instructions translated to trap to VMM
    - Dynamic (on the fly) binary translation, so works with unmodified OS
    - Higher overhead than paravirtualization
    - Example: VMWare workstation

Dynamic Binary Translation-- Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

C program

```
89 f9 be 02 00 00 00 39 ce 7d ...
```

Binary representation

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov    %esi, $2   ; i = 2
           cmp    %esi, %ecx ; is i >= a?
           jge    prime      ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
           cdq               ; sign-extend
           idiv   %esi       ; a % i
           test   %edx, %edx ; is remainder zero?
           jz     notPrime   ; jump if yes
           inc    %esi       ; i++
           cmp    %esi, %ecx ; is i >= a?
           jl     nexti      ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
           ret
notPrime:  xor    %eax, %eax ; %eax = 0
           ret
```

Assembly instructions

Dynamic Binary Translation– Example (2)

```
isPrime:   mov    %ecx, %edi ; %ecx = %edi (a)
           mov    %esi, $2   ; i = 2
           cmp    %esi, %ecx ; is i >= a?
           jge    prime      ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
           cdq               ; sign-extend
           idiv   %esi       ; a % i
           test   %edx, %edx ; is remainder zero?
           jz     notPrime   ; jump if yes
           inc    %esi       ; i++
           cmp    %esi, %ecx ; is i >= a?
           jl     nexti      ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
           ret
notPrime:  xor    %eax, %eax ; %eax = 0
           ret
```

Assembly instructions

isPrime(49)
Translator

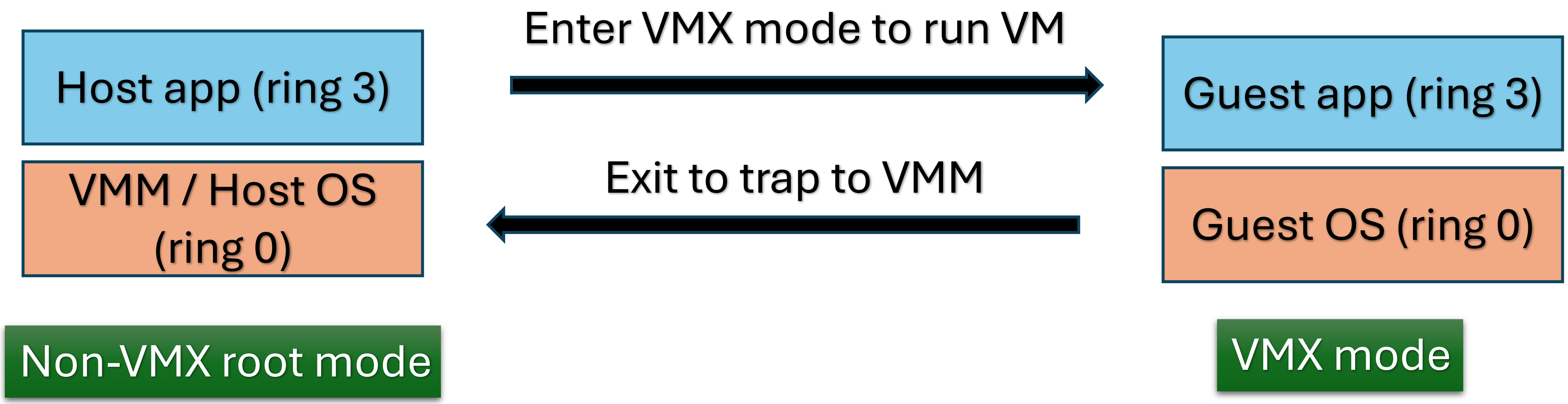
```
isPrime':    mov %ecx, %edi    ; IDENT
             mov %esi, $2
             cmp %esi, %ecx
             jge [takenAddr]   ; JCC
             jmp [fallthrAddr]
```

Compiled Code Fragment

```
isPrime':   *mov    %ecx, %edi    ; IDENT
             mov    %esi, $2
             cmp    %esi, %ecx
             jge    [takenAddr]   ; JCC
                                  ; fall-thru into next CCF
nexti':     *mov    %eax, %ecx    ; IDENT
             cdq
             idiv   %esi
             test   %edx, %edx
             jz     notPrime'     ; JCC
                                  ; fall-thru into next CCF
```

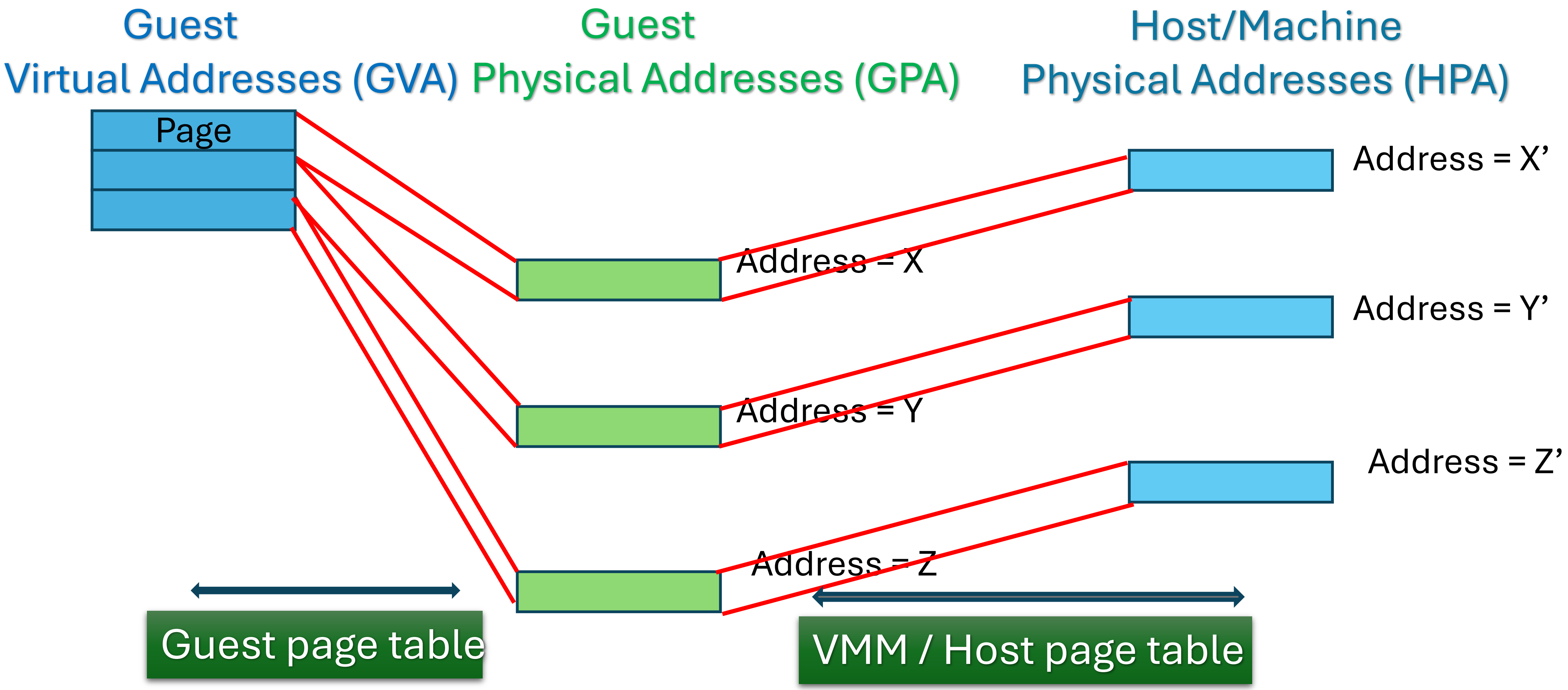
Techniques to virtualize x86 (2)

- Hardware assisted virtualization: KVM/QEMU in Linux
  - CPU has a special VMX mode of execution
  - X86 has 4 rings on non-VMX root mode, another 4 rings in VMX mode
- VMM enters VMX mode to run guest OS in (special) ring 0
- Exit back to VMM on triggers (VMM retains control)

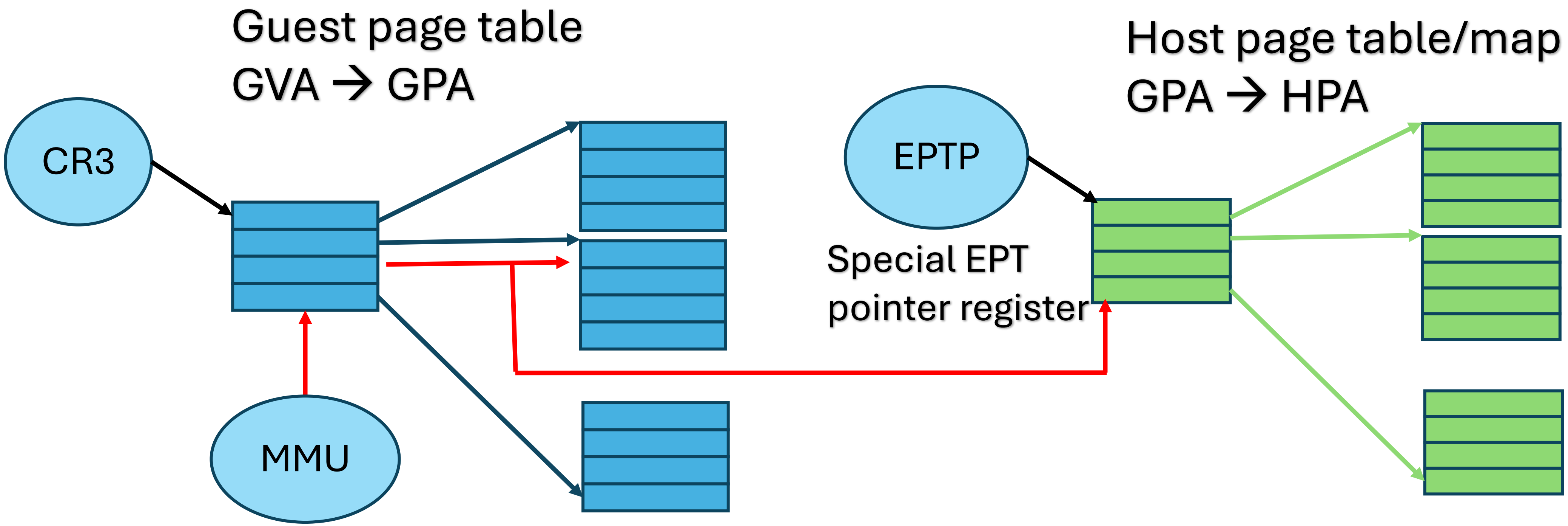


Memory virtualization

- What about address translation in virtual machines?



Extended page tables



- Page table walk by MMU: Start walking guest page table using GVA
- Guest PTE (for every level page table walk) gives GPA (cannot use GPA to access memory)
- Use GPA, walk host page table to find HPA, then access memory page, then next level access