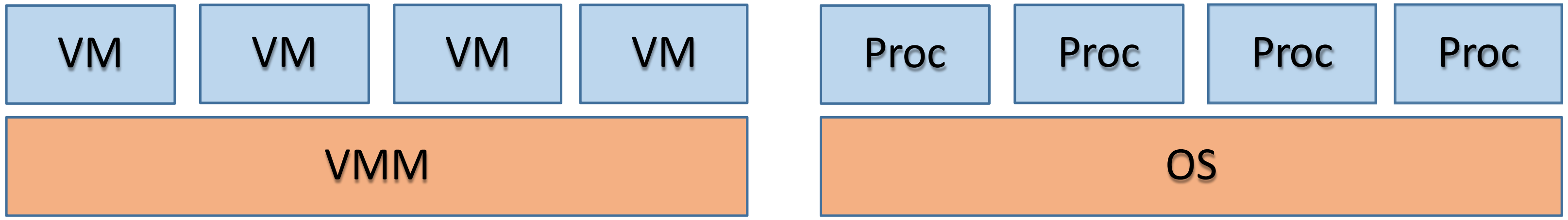
Virtualization

What does VMM do?

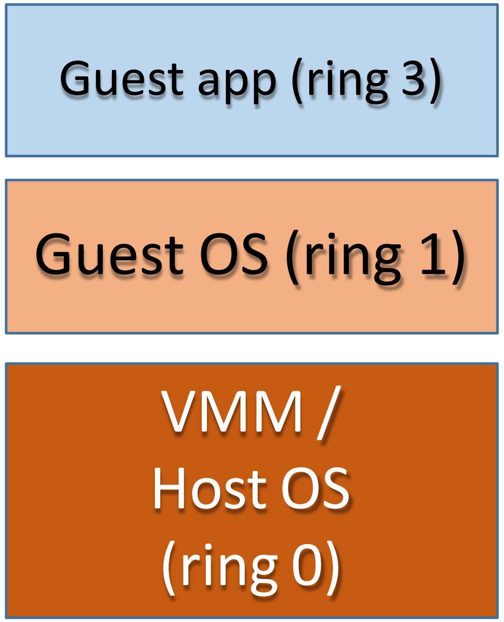
- Multiple VMs running on a PM – multiplex the underlying machine
  - Similar to how OS multiplexes processes on CPU



- 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

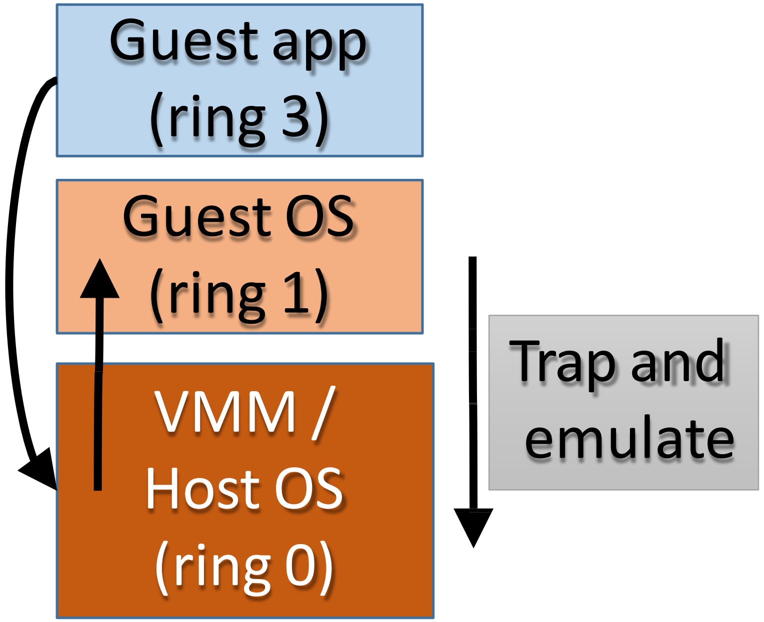
Trap and emulate VMM (1)

- 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 instr, traps to VMM
  - VMM jumps to corresponding user process



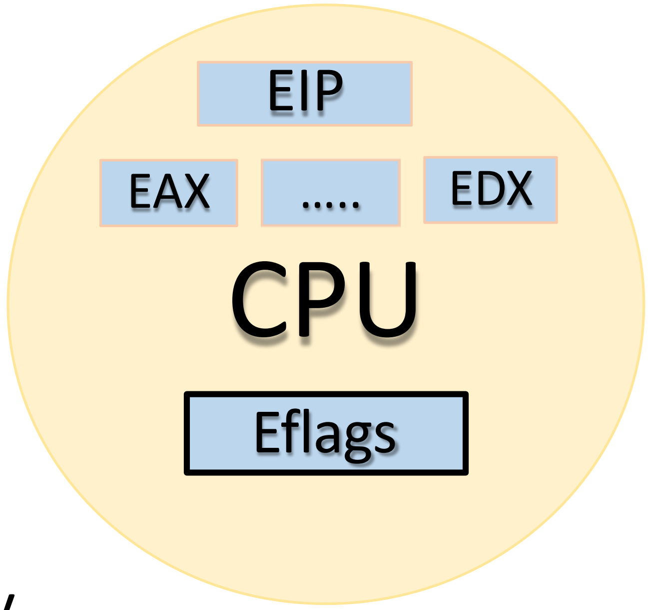
- Any privileged action by guest OS traps to VMM, emulated by VMM
  - Example: set IDT, set CR3, access hardware
  - Sensitive data structures like IDT must be managed by VMM, not guest OS

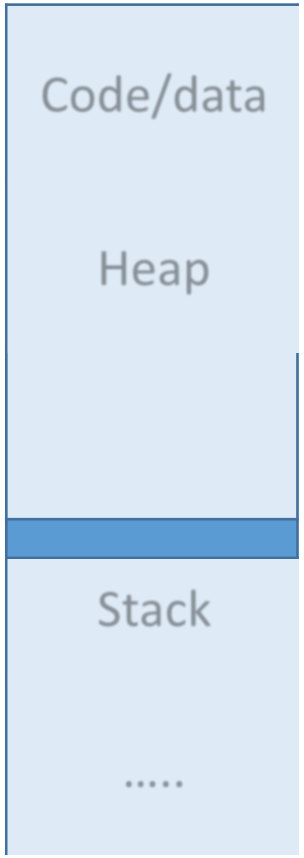
Problems with trap and emulate

- Guest OS may realize it is running at lower privilege level
  - Some registers in x86 reflect CPU privilege level (code segment/CS)
  - Guest OS can read these values and get offended!
- Some x86 instructions which change hardware state (sensitive instructions) run in both privileged and unprivileged modes
  - Will behave differently when guest OS is in ring 0 vs in less privileged ring 1
  - OS behaves incorrectly in ring1, will not trap to VMM
- 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)

Example: Problems with trap and emulate

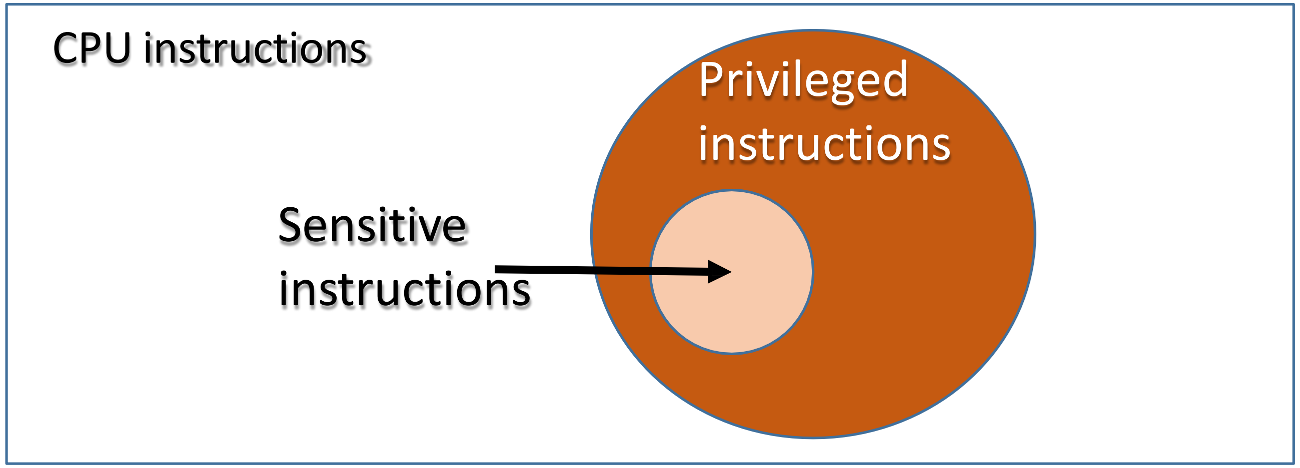
- Eflags register is a set of CPU flags
  - IF (interrupt flag) indicates if interrupts on/off
- Consider the popf instruction in x86
  - Pops values on top of stack and sets eflags
- Executed in ring 0, all flags set normally
- Executed in ring 1, only some flags set
  - IF is not set as it is privileged 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

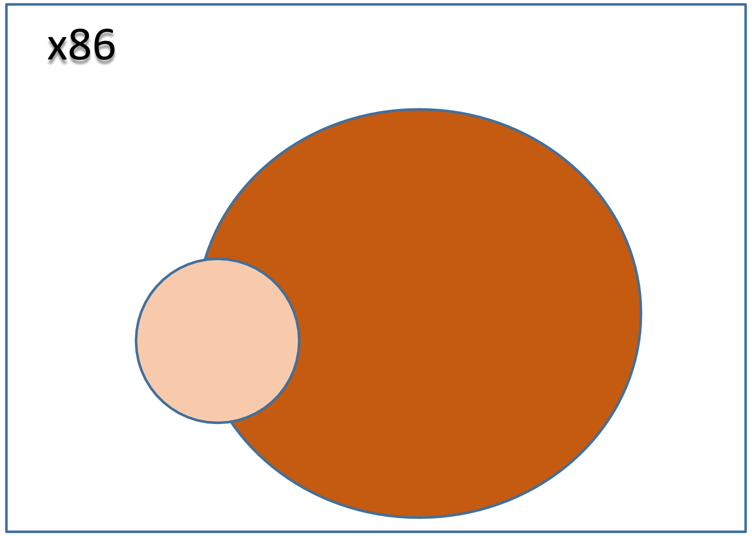




Popek Goldberg theorem

- Sensitive instruction = changes hardware state
- 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
  - x86 does not satisfy this criteria, so trap and emulate VMM is not possible



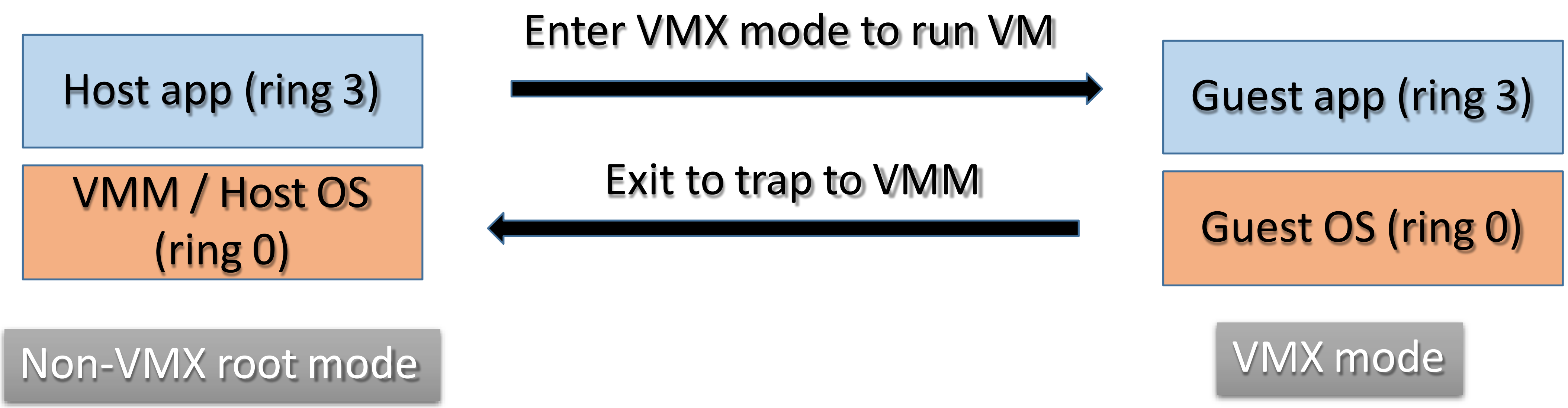


Techniques to virtualize x86 (1)

- 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

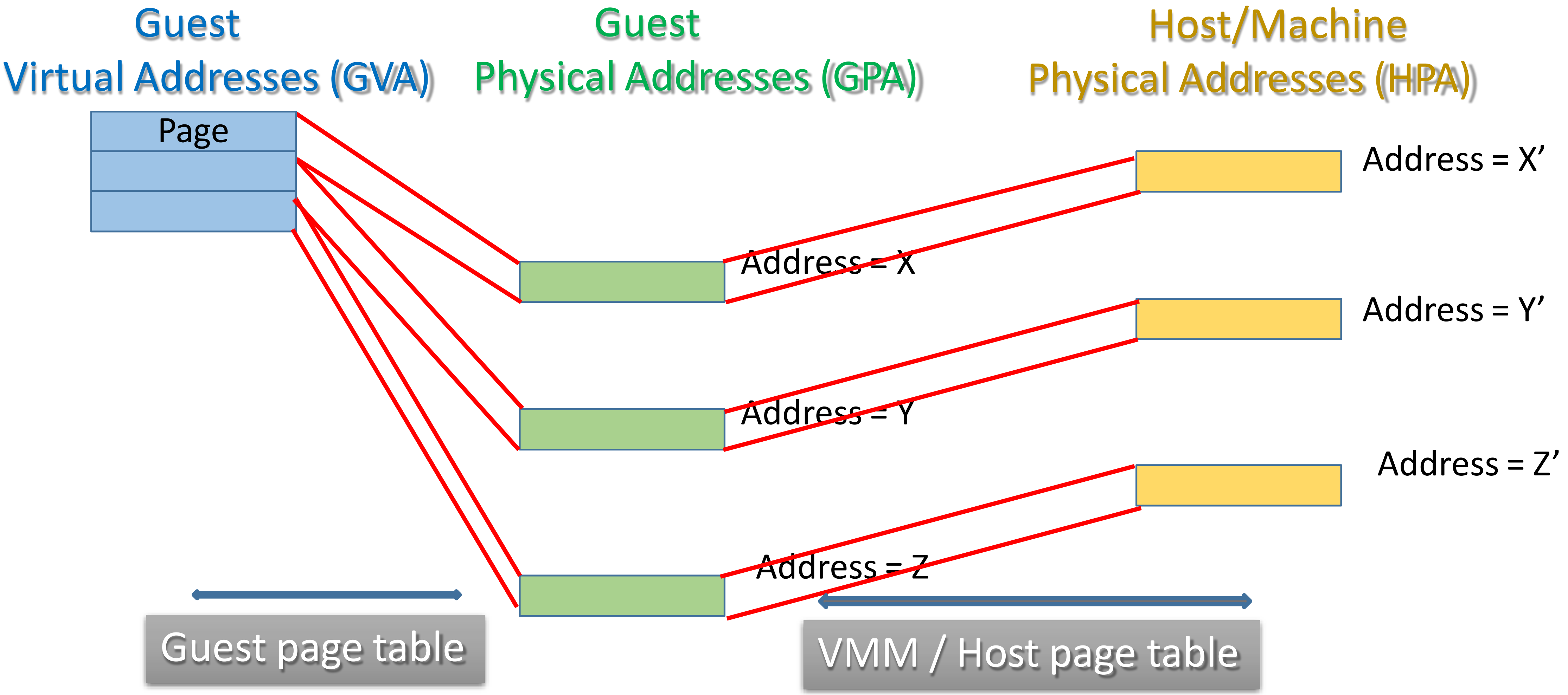
Techniques to virtualize x86 (2)

- 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?



Techniques for memory virtualization

- Guest page table has GVA→GPA mapping
  - Each guest OS thinks it has access to all RAM starting at address 0
- VMM / Host OS has GPA→HPA mapping
  - Guest "RAM" pages are distributed across host memory
- Which page table should MMU use?
- Shadow paging: VMM creates a combined mapping GVA→HPA and MMU is given a pointer to this page table
  - VMM tracks changes to guest page table and updates shadow page table
- Extended page tables (EPT): MMU hardware is aware of virtualization, takes pointers to two separate page tables
  - Address translation walks both page tables
- EPT is more efficient but requires hardware support

I/O Virtualization

- Guest OS needs to access I/O devices, but cannot give full control of I/O to any one guest OS
- Two main techniques for I/O virtualization:
  - Emulation: guest OS I/O operations trap to VMM, emulated by doing I/O in VMM/host OS
  - Direct I/O or device passthrough: assign a slice of a device directly to each VM
- Many optimizations exist, active area of research

Summary

- Techniques for CPU virtualization
  - Paravirtualization: rewrite guest OS source code
  - Full virtualization: dynamic binary translation
  - Hardware-assisted virtualization: CPU has special virtualization mode
- Techniques for memory virtualization:
  - Shadow page tables: combined GVA→HPA mappings
  - Extended page tables: MMU is given separate GVA→GPA and GPA→HPA mappings
- I/O virtualization: emulation, device passthrough
- VMMs use a combination of above techniques
  - We will study all of the above techniques in detail