Introduction to bhyve

Takuya ASADA / @syuu1228

What is bhyve?

What is bhyve?

- bhyve is a hypervisor introduced in FreeBSD
- Similar to Linux KVM, runs on host OS
- BSD License
- Developed by Peter Grehan and Neel Natu

bhyve features

- Required Intel VT-x and EPT (Nehalem or later)
  AMD support in progress
- Does not support BIOS/UEFI for now
  UEFI support in progress
- Minimal device emulation support:
  virtio-blk, virtio-net, COM port + α
- Supported guest OS:
  FreeBSD/amd64, i386, Linux/x86_64, OpenBSD/amd64

How to use it?

kldload vmm.ko

/usr/sbin/bhyveload -m ${mem} -d ${disk} ${name}

```
/usr/sbin/bhyve -c ${cpus} -m ${mem} \
-s 0,hostbridge -s 2,virtio-blk,${disk} \
-s 3,virtio-net,${tap} -s 31,lpc -l com1,stdio vm0
```

How to run Linux?

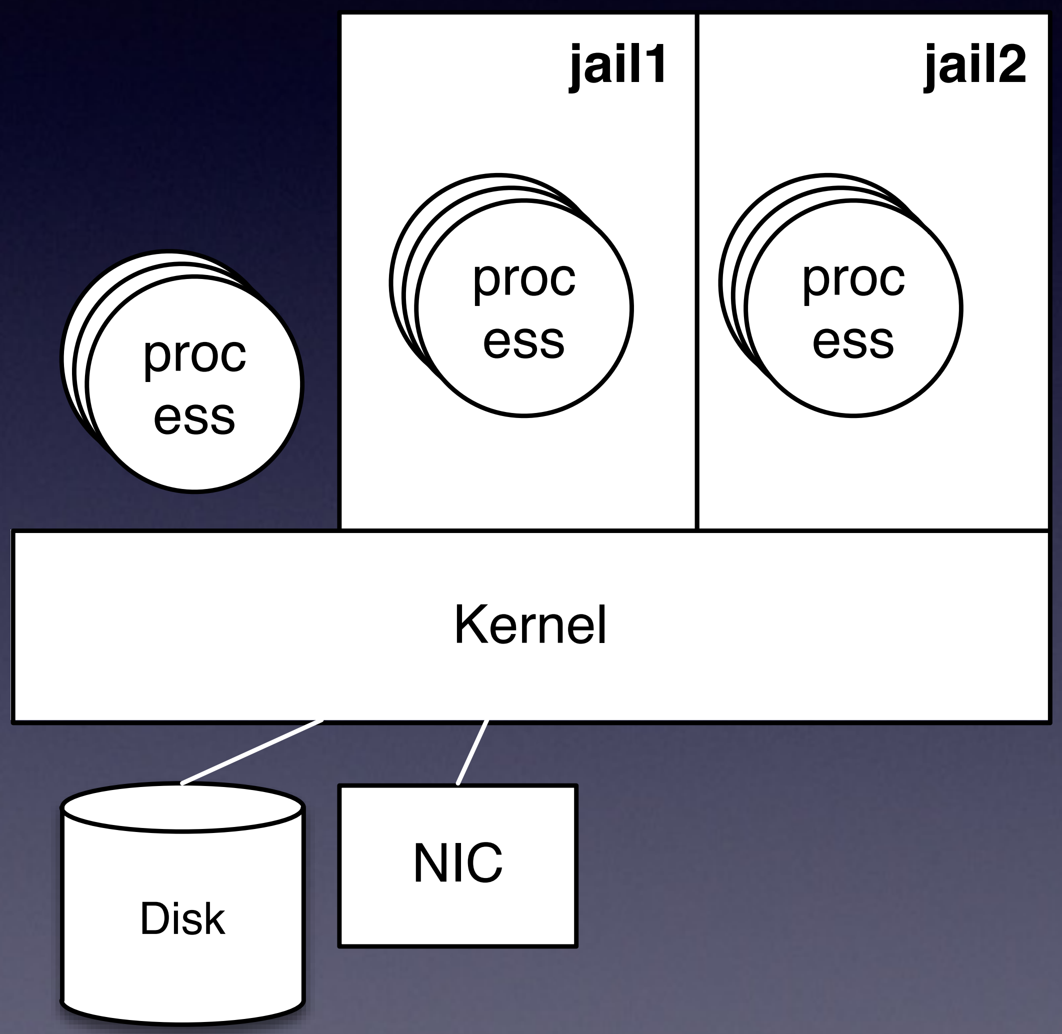
- bhyve OS Loader(/usr/sbin/bhyveload) only supports FreeBSD
  You need another OS Loader to support other OSs
- **grub2-bhyve** is the solution
  - It's modified version grub2, runs on host OS (FreeBSD)
  - Can load Linux and OpenBSD
- Available in ports & pkg!

Virtualization in general

Difference between container and hypervisor

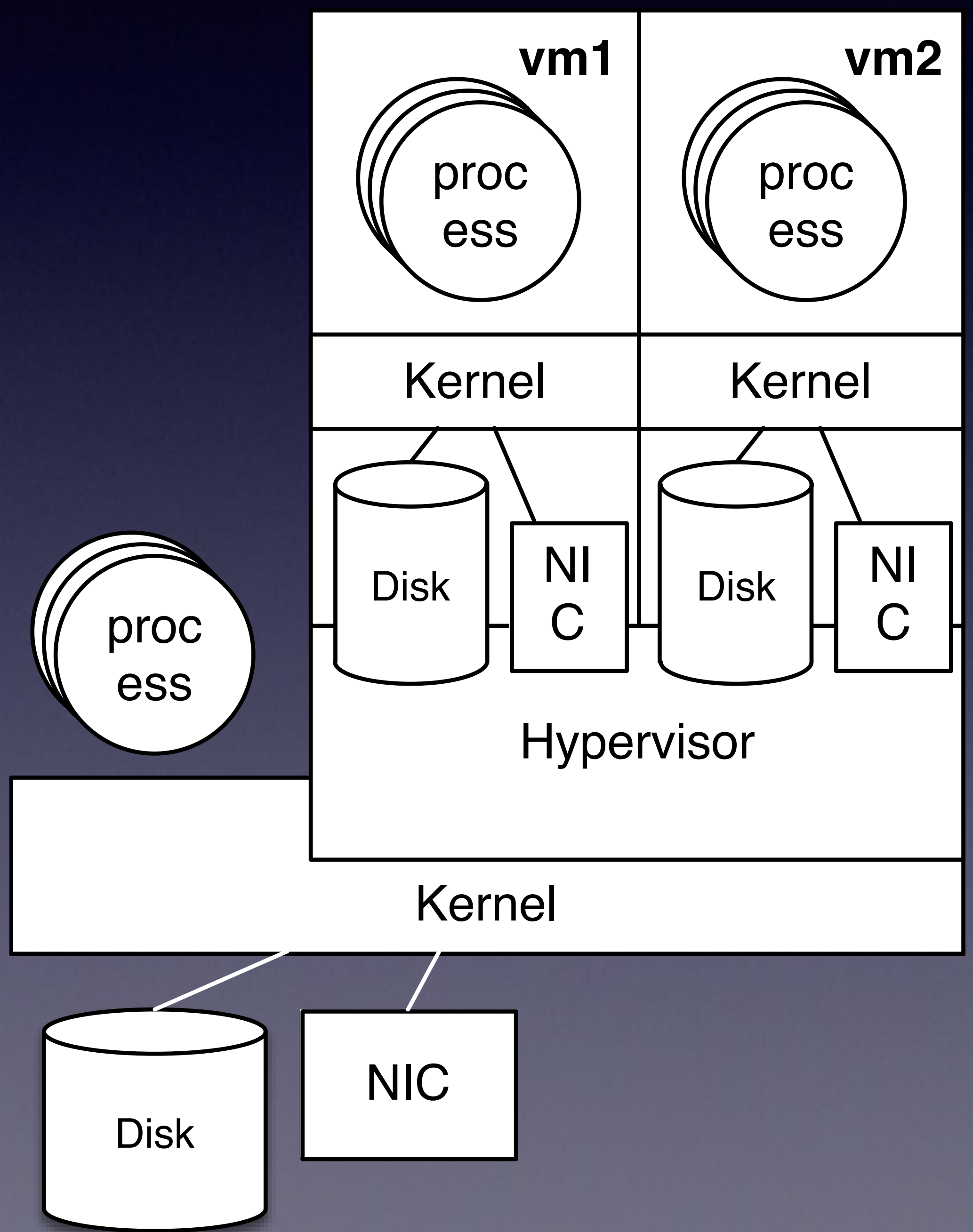
- Jail is **container**
  - It's virtualize OS environment on kernel level
- bhyve is **hypervisor**
  - It virtualizes whole machine
- Totally different approach

Container



- Process in jail is just a normal process for the kernel
- The kernel do some tricks to isolate environments between jails
- Lightweight, less-overhead
- Share one kernel with all jails
→ If the kernel panics, all jails will die
- **You cannot install another OS (No Windows, No Linux!)**

Hypervisor

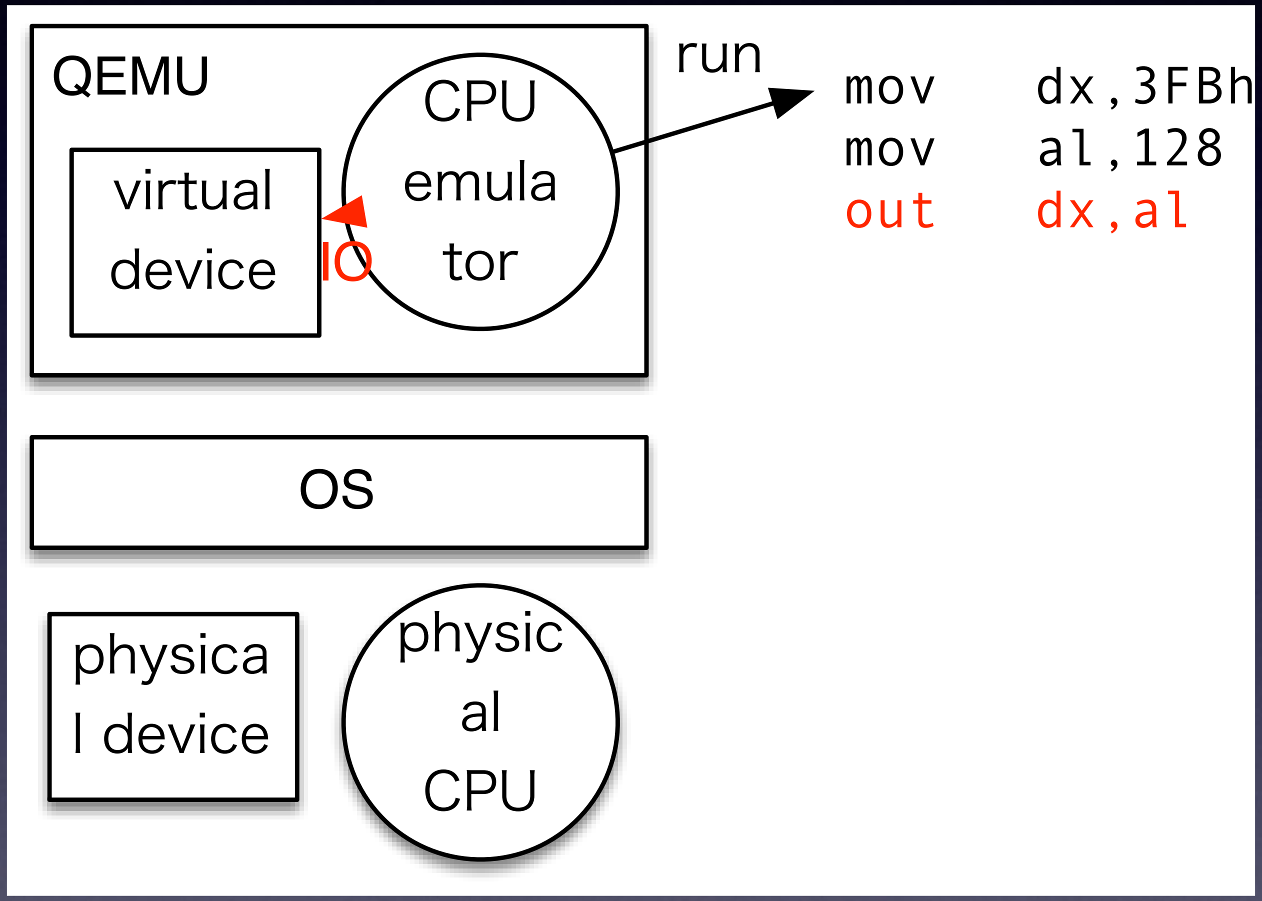


- Hypervisor virtualizes a machine
- From guest OS, it looks like real hardware
- Virtual machine is a normal process for host OS
- Does not share kernel, it is completely isolated
- You can run Full OS inside of the VM → **Windows! Linux!**

How hypervisor virtualize machine?

- To make complete virtual machine, you need to virtualize following things:
  - CPU
  - Memory (Address Space)
  - I/O

CPU Virtualization: Emulate entire CPU?

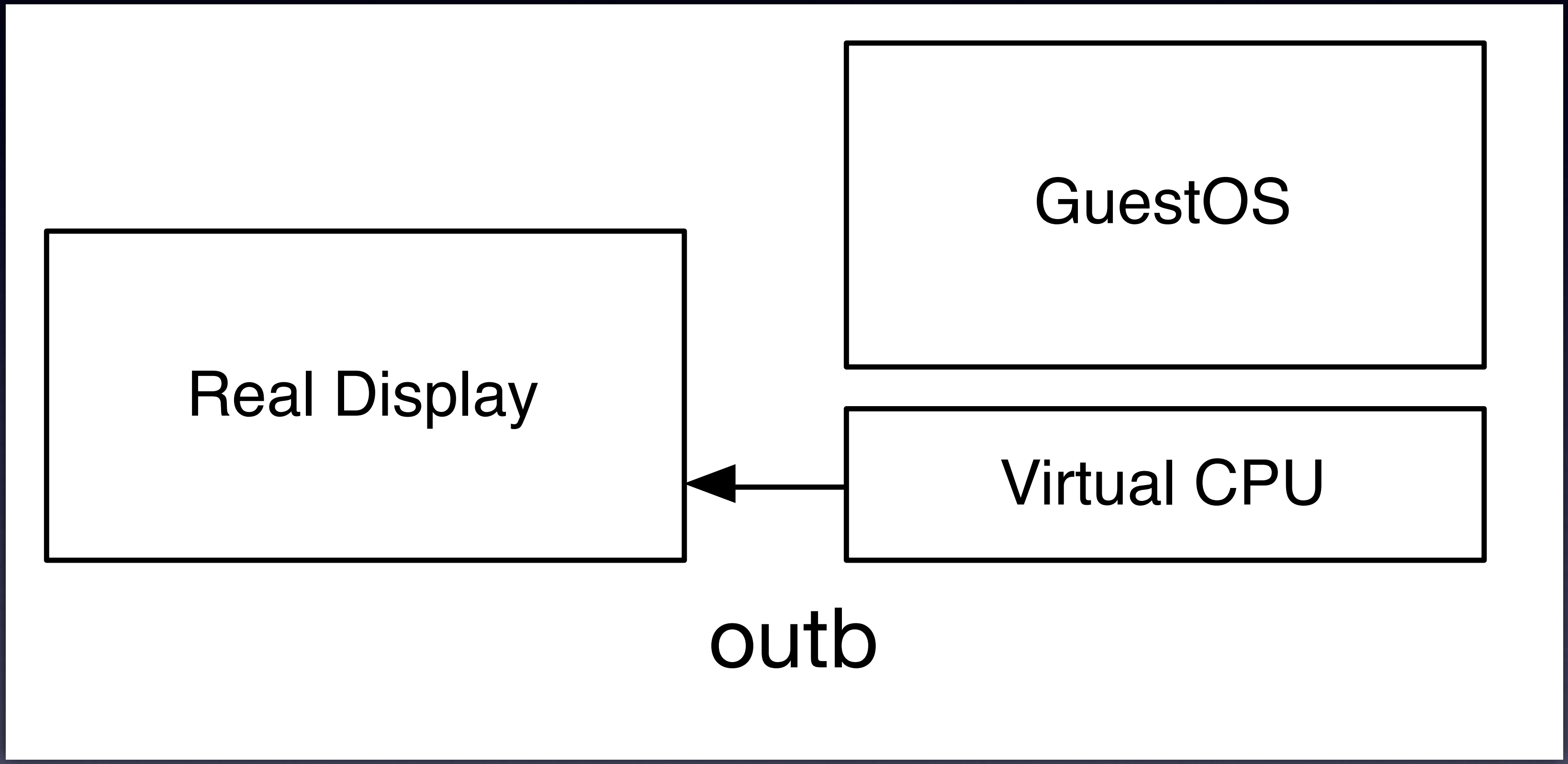


- Like QEMU
- You can emulate the entire CPU operation on a normal process
- Very slow, not a really useful choice for virtualization

CPU Virtualization: Direct execution?

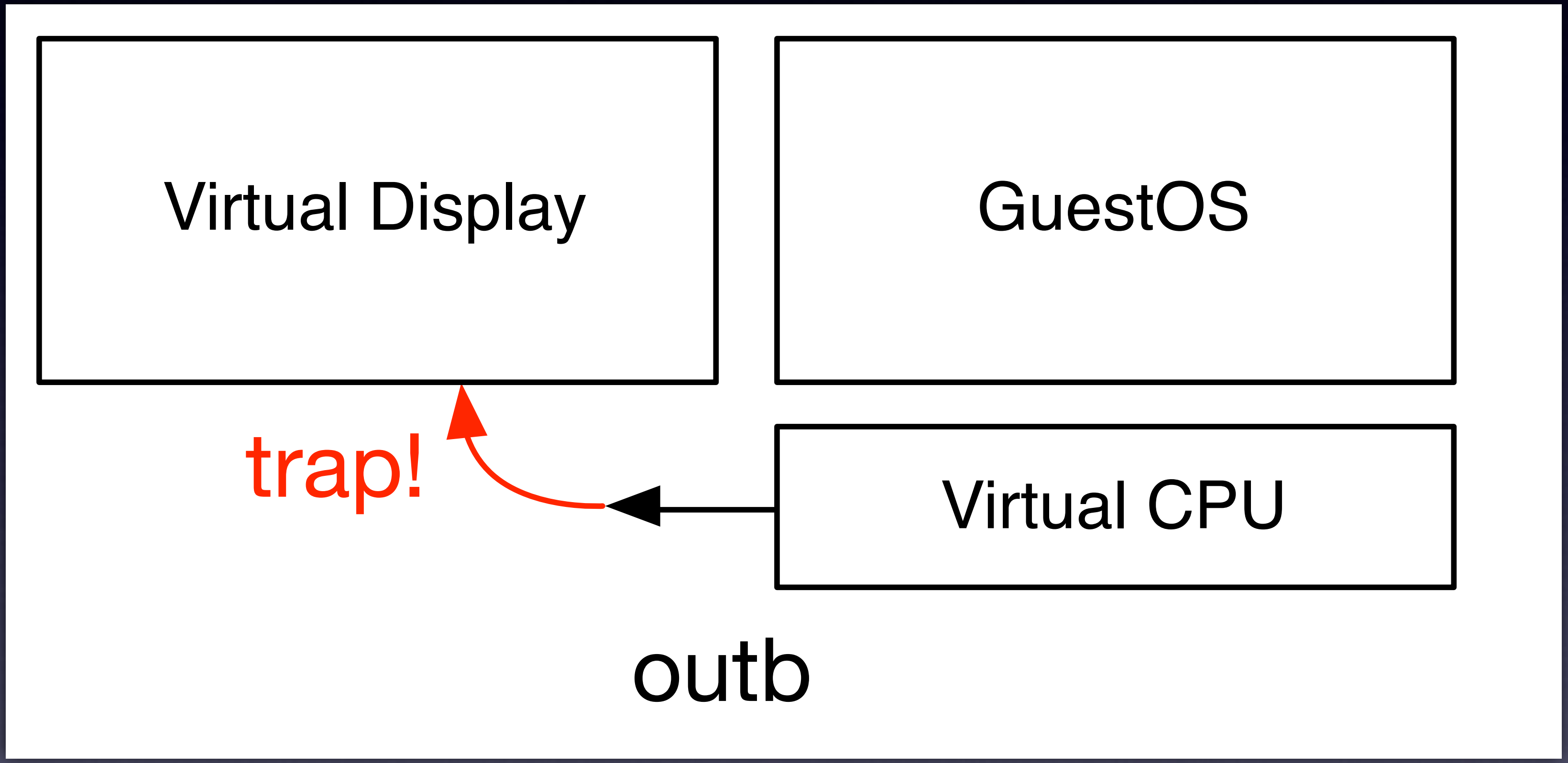
- You want run guest instructions directly on a real CPU since you are virtualizing x86 on x86
- You need to avoid executing some instructions which modify system global state, or perform I/O (called sensitive instructions)
  - If you execute these instructions on a real CPU, it may break host OS state such as directly accessing a HW device

Perform I/O on VM



- You need to avoid access to real HW from VM
- Need to prevent execution of the instruction

Perform IO on VM



- You can trap them by executing in lower privileged mode
- However, on x86, there are some instructions which are impossible to trap because these are nonprivileged instructions

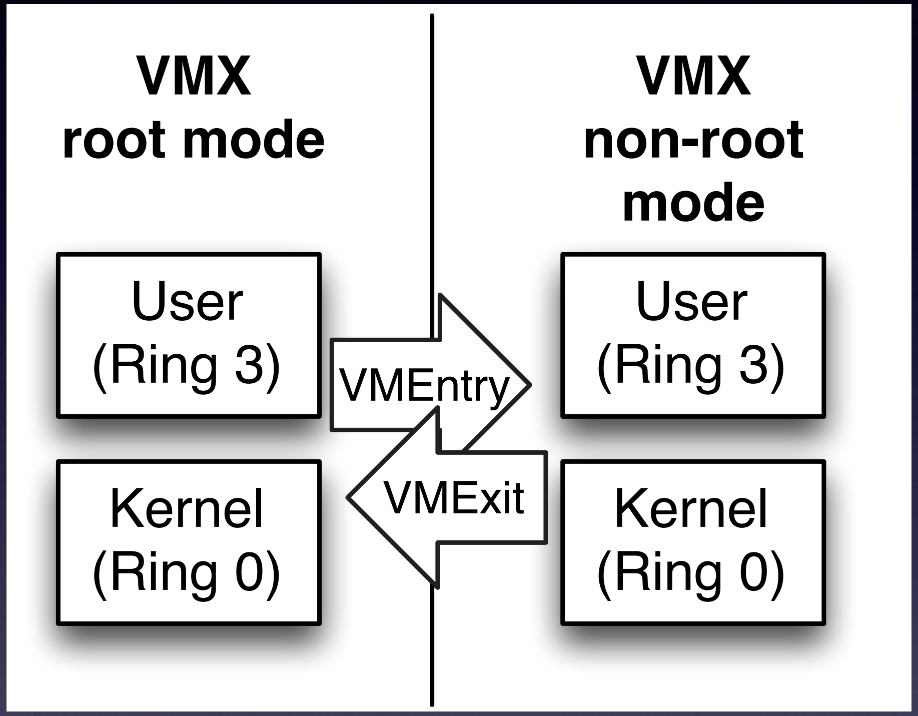
Software techniques to virtualize x86

- Binary translation (old VMware): interpret & modify guest OS's instructions on-the-fly
  → Runs fast, but implementation is very complex

- Paravirtualization (old Xen): Modify guest OS for the hypervisor
  → Runs fast, but is impossible to run unmodified OS's

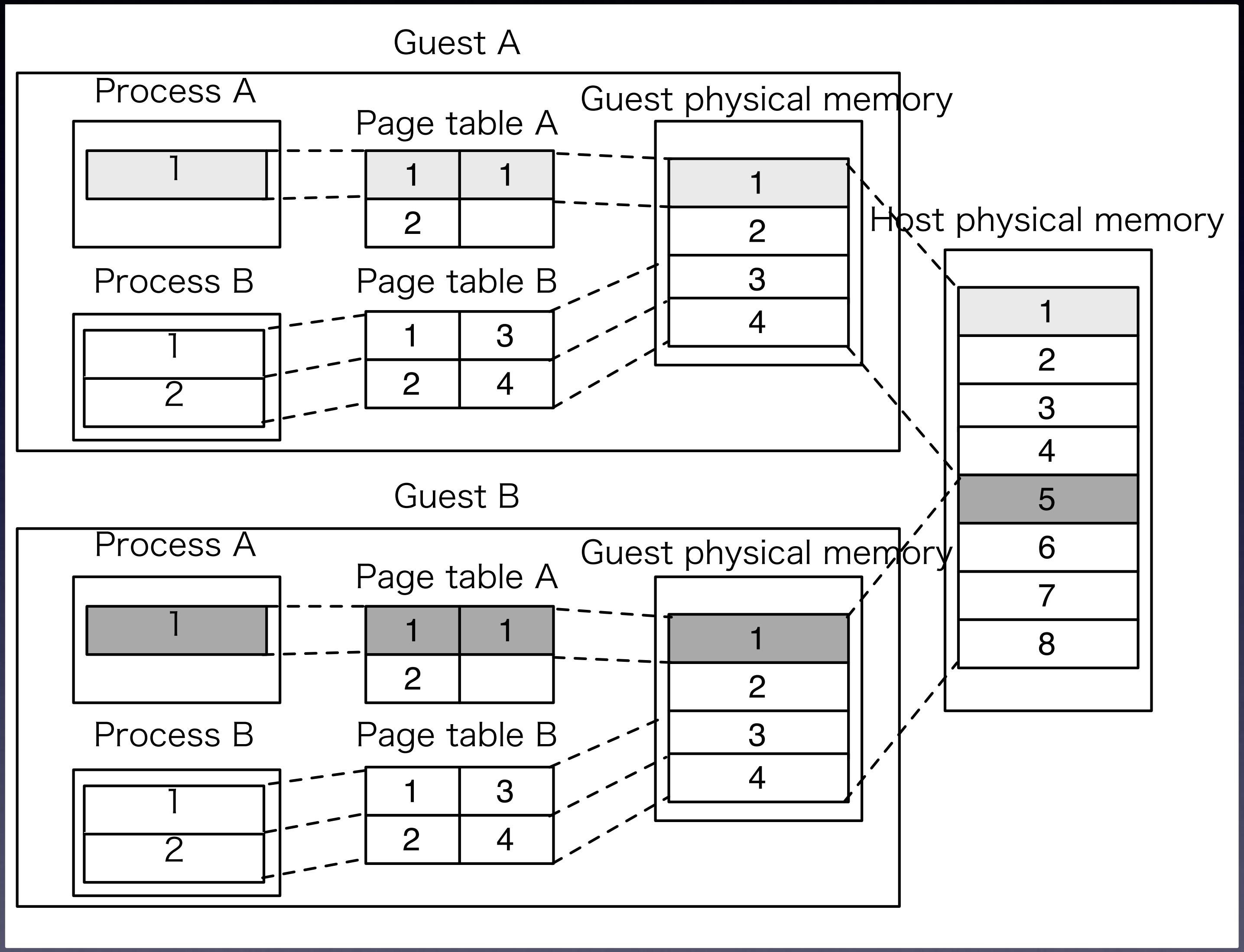
- We want an easier & better solution
  → **HW assisted virtualization!**

Hardware assisted virtualization(Intel VT-x)



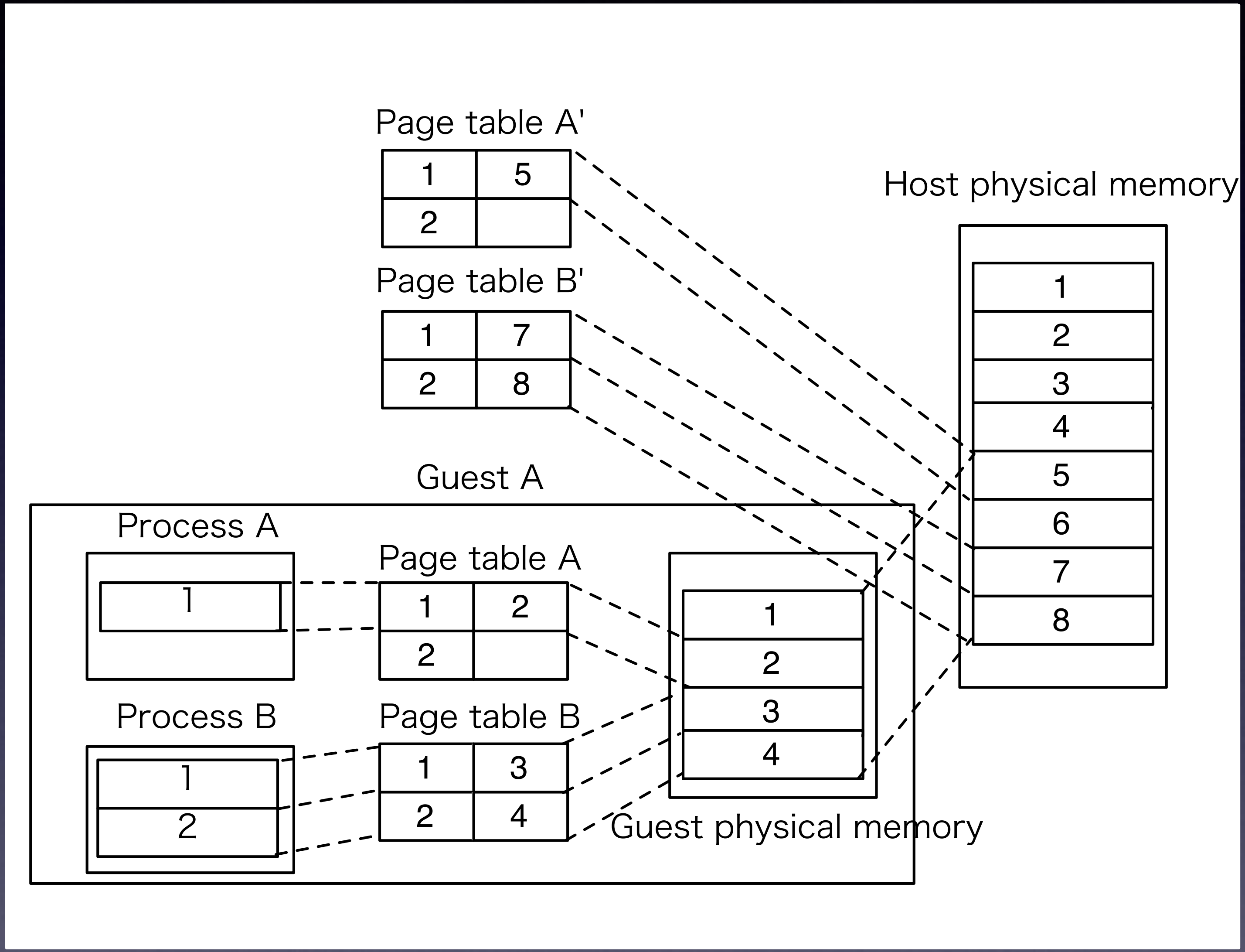
- New CPU mode:
  VMX root mode (hypervisor) / VMX non-root mode (guest)
- If some event needs to emulate in the hypervisor,
  CPU stops guest, exit to hypervisor → VMExit
- You don't need complex software techniques
  You don't have to modify the guest OS

Memory Virtualization



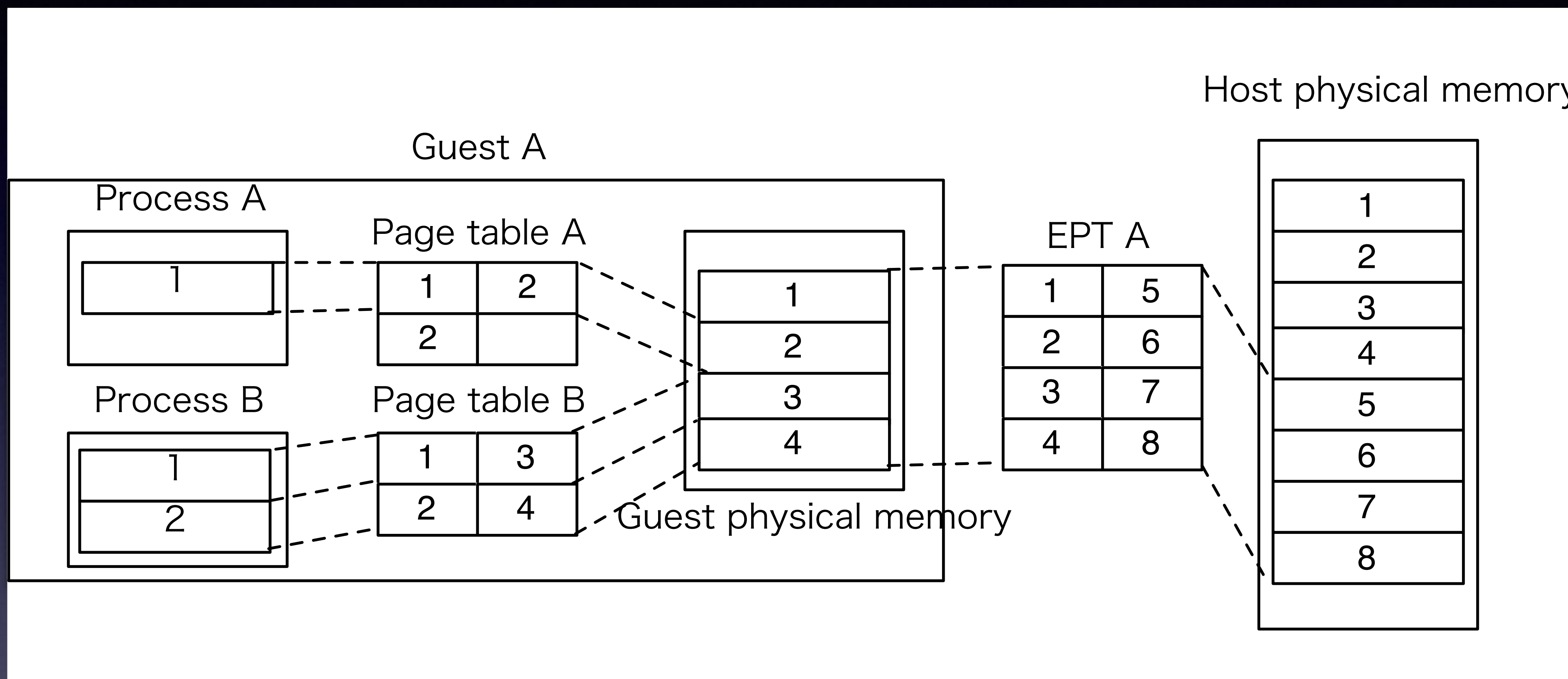
- If you run guest OS natively, memory address translation become problematic
- If GuestB loads Page table A, virtual page 1 translate to Host physical page 1 but you meant Host physical page 5

Shadow Paging



- Trap page table loading/modifying, create "Shadow Page Table", tell physical page number to the MMU
- A software trick that works well, but is slow

Nested Paging (Intel EPT)



- HW assisted memory virtualization!
- You will have Guest physical : Host physical translation table
- MMU translates address by two step (Nested)

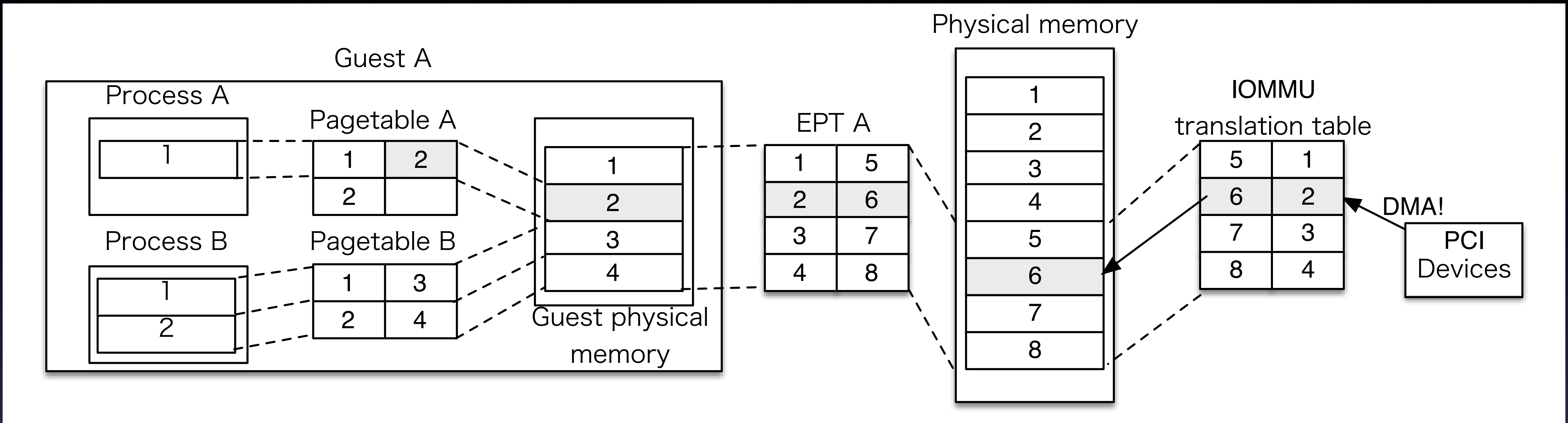
I/O Virtualization

- To run unmodified OSs, you'll need to emulate all devices what you have on the real hardware
  - SATA, NIC(e1000), USB(ehci), VGA(Cirrus), Interrupt controller(LAPIC, IO-APIC), Clock(HPET), COM port…
- Emulating real devices is not very fast because it causes lot of VMExits, not ideal for for virtualization

Paravirtual I/O

- Virtual I/O device is designed for VM use
- Much faster than emulating real devices
- Required device driver on guest OS
- De-facto standard: virtio-blk, virtio-net

PCI Device passthrough



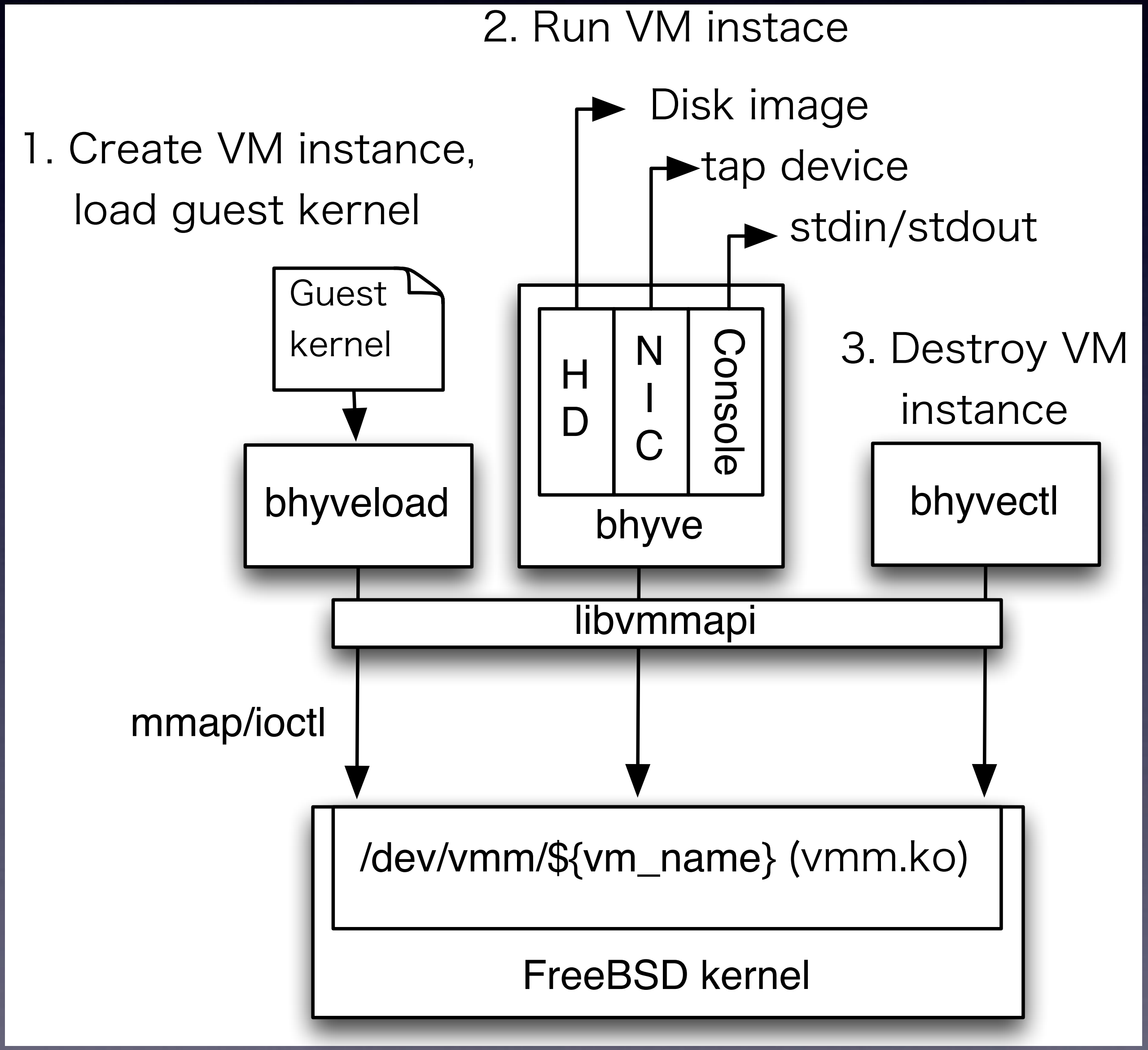
- If you attach a real HW device on a VM, you will have a problem with DMA
- Because the device requires physical address for DMA but the guest OS doesn't know the Host physical address
- **Address translator for the devices: IOMMU(Intel VT-d)**
- **Translates guest physical to host physical using a translation table**

bhyve internals

How bhyve virtualize machine?

- CPU: HW-assisted virtualization (Intel VT-x)
- Memory: HW-assisted memory virtualization (Intel EPT)
- IO: virtio, PCI passthrough, +α
- Uses HW assisted features

bhyve overview



- bhyveload: loads guest OS
- bhyve: userland part of Hypervisor, emulates devices
- bhyvectl: a management tool
- libvmmapi: userland API
- vmm.ko: kernel part of Hypervisor

vmm.ko

- All VT-x features only accessible in kernel mode, vmm.ko handles it
- Most important work of vmm.ko is CPU mode switching between hypervisor/guest
- Provides interface for userland via /dev/vmm/${vmname}
  - Each vmm device file contains each VM instance state

/dev/vmm/${vmname} interfaces

- create/destroy
  Can create/destroy device file via sysctl hw.vmm.create, hw.vmm.destroy
- read/write/mmap
  Can access guest memory area by standard syscall (Which means you even can dump guest memory by *dd* command)
- ioctl
  Provides various operations to VM

/dev/vmm/${vmname} ioctls

- VM_MAP_MEMORY: Maps guest memory area at requested size
- VM_SET/GET_REGISTER: Access registers
- VM_RUN: Run guest machine, until virtual devices accessed (or some other trap happened)

libvmmapi

- wrapper library of /dev/vmm operations
  - vm_create(name)→ sysctl(“hw.vmm.create”, name)
  - vm_set_register(reg, val) → ioctl(VM_SET_REGISTER, reg, val)

bhyveload

- bhyve uses OS loader instead of BIOS/UEFI, to load guest OS
- FreeBSD bootloader ported to userland: userboot
- bhyveload runs host OS, to initialize guest OS
- Once it called, it does following things:
  - Parse UFS on diskimage, find kernel
  - Load kernel to guest memory area
  - Initialize Page Table
  - Create GDT, IDT, LDT
  - Initialize special registers to get ready for 64bit mode
- Guest machine can starts from kernel entry point, with 64bit mode

bhyve

- bhyve command is the userland part of the hypervisor
- It invokes ioctl(VM_RUN) to run GuestOS
- Emulates virtual devices
- Provides user interface(no GUI for now)

main loop in bhyve

```
while (1) {
    ioctl(VM_RUN, &vmexit);
    switch (vmexit.exit_code) {
    case IOPORT_ACCESS:
        emulate_device(vmexit.ioport);
        …
    }
}
```

Q&A?