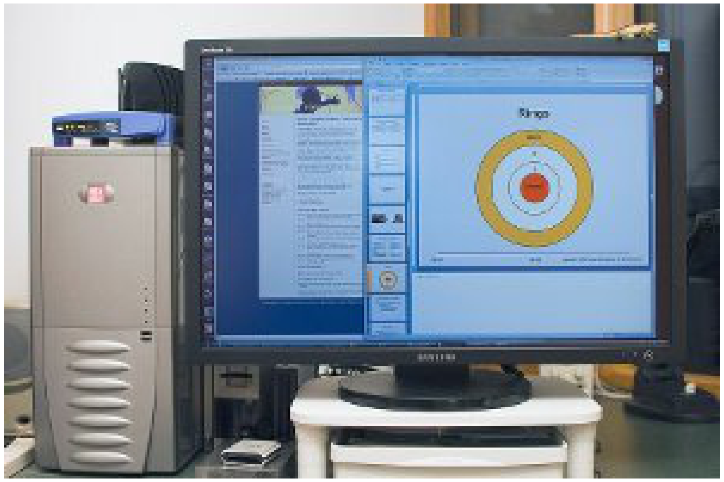
Virtual Machines
Part 2: Now

$\neq$



How They Are Different

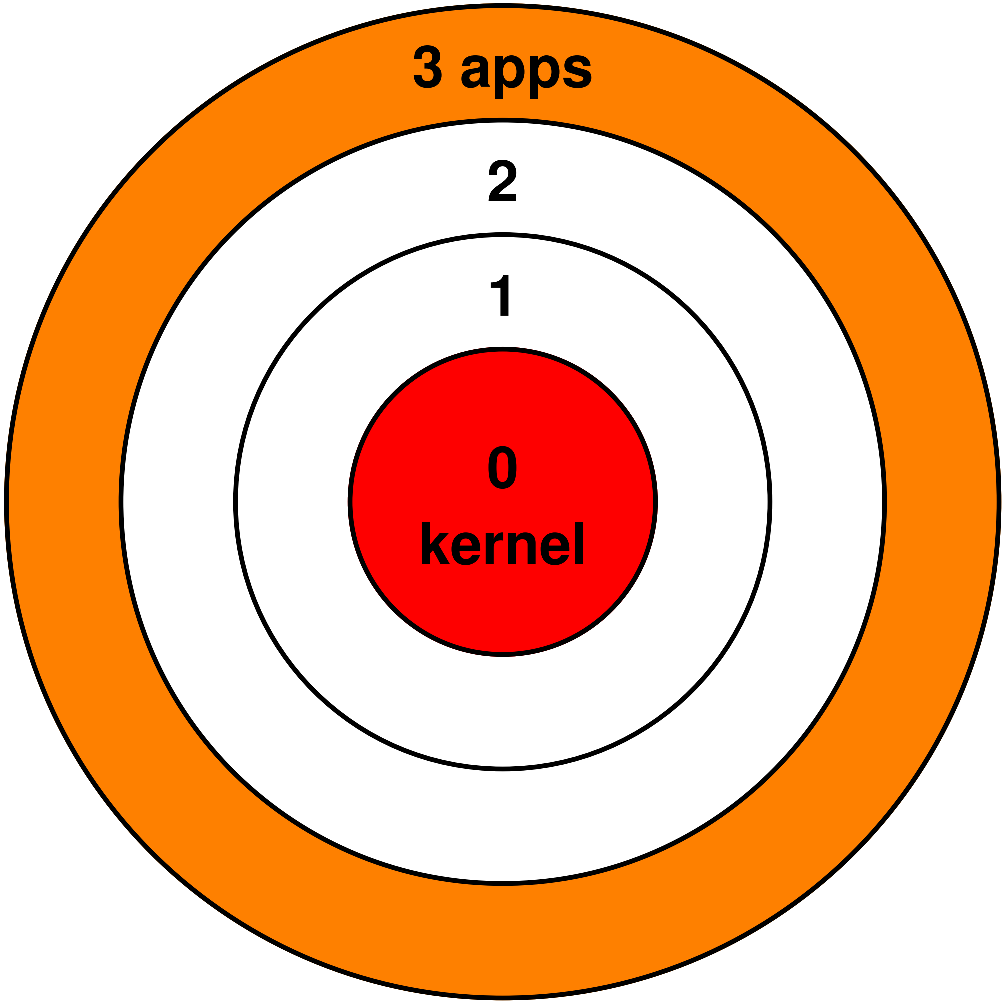
IBM 360

- Two execution modes
  - supervisor and problem (user)
  - *all* sensitive instructions are privileged instructions
- Memory is protectable: 2k-byte granularity
- All interrupt vectors and the clock are in first 512 bytes of memory
- I/O done via *channel programs* in memory, initiated with *privileged instructions*
- Dynamic address translation (virtual memory) added for Model 67

Intel x86

- Four execution modes
  - rings 0 through 3
  - *not all* sensitive instructions are privileged instructions
- Memory is protectable: segment system + virtual memory
- Special register points to interrupt table
- I/O done via *memory-mapped I/O*
  - i.e., I/O operations look like *memory accesses*
- Virtual memory is standard

Rings



- An x86 processor can be in one of 4 *modes/rings*

A Sensitive x86 Instruction

- `popf`
  - pops flags (word) off stack, setting processor flags according to word's content
    - sets all flags if in ring 0
      - including interrupt-disable flag
    - just some of them if in other rings
      - ignores interrupt-disable flag
  - bad news: if invoked in *user* mode, does *not* cause a *trap!*
    - therefore, this instruction will *execute differently* in the *guest OS* when it's running on top of a VM (as compared to running on a real machine)
      - since the OS is running in user mode under the virtual machine scheme
    - this (and a few other instructions) is one of the major problem to virtualize x86 systems
- There is another major problem related to device I/O (later)

x86 CPU Virtualization - What to Do?

- ***Binary rewriting***
  - rewrite kernel binaries of guest OSes
    - replace sensitive instructions with "*hypercalls*"
    - do so dynamically (i.e., *dynamic* binary rewriting)
      - VMware does this
    - no need to modify guest OS
- ***Hardware virtualization***
  - fix the hardware so it's virtualizable
- ***Paravirtualization***
  - virtual machine differs from real machine
    - provides more convenient interfaces for virtualization
    - *hypervisor* interface between virtual and real machines
      - we use the terms "*hypervisor*" and "*VMM*" interchangeably
    - guest OS *source code* is *modified* (and recompiled)

Binary Rewriting

- Privilege-mode code run via binary translator
  - guest OS is unmodified
  - replaces sensitive instructions with *hypercalls*
  - translated code is cached
    - usually translated just once
  - *VMWare*
  - U.S. patent 6,397,242
- VirtualBox appears to do something similar to VMWare
  - see https://www.virtualbox.org/manual/ch10.html#idp58764736 for more details

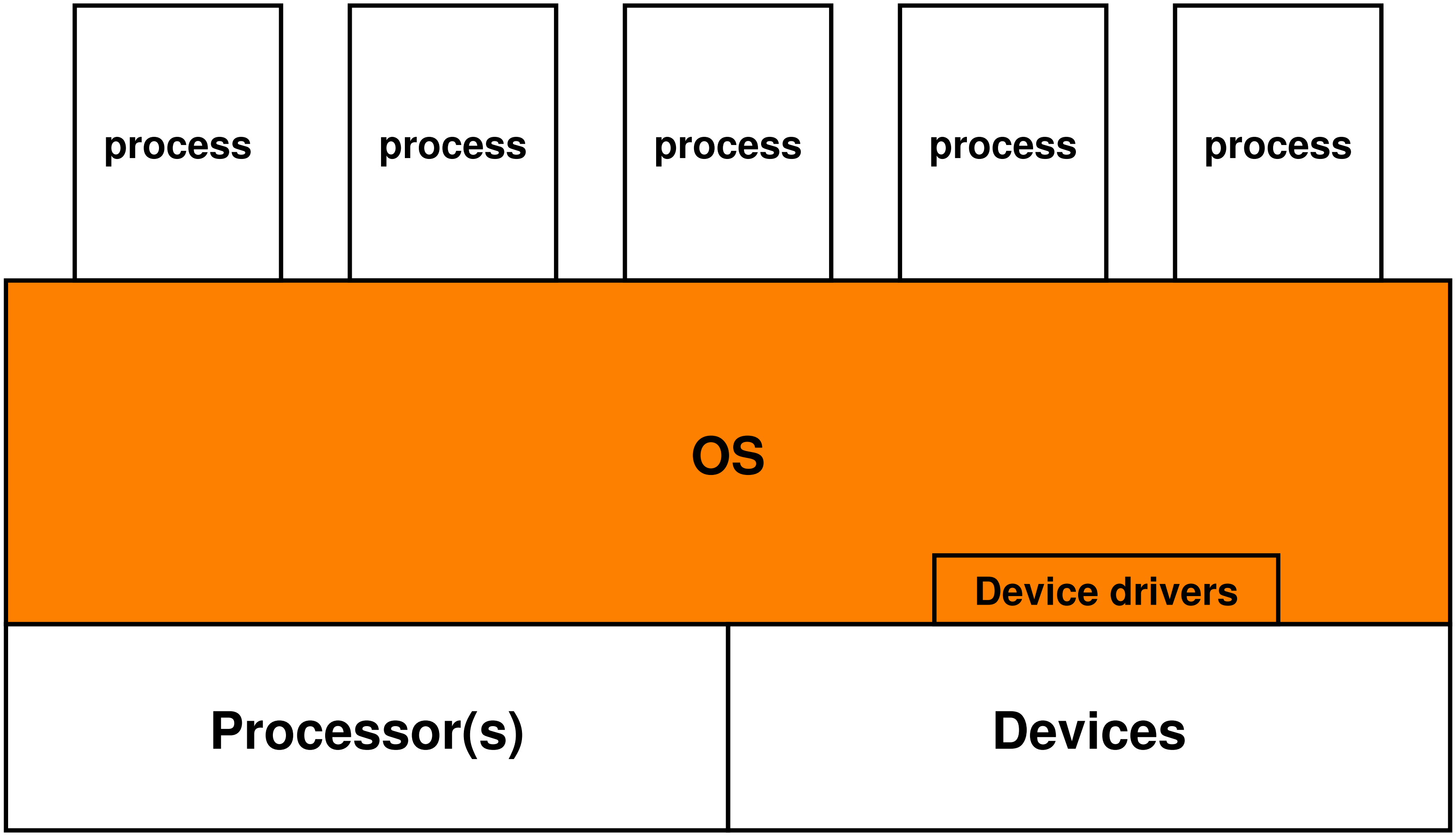
Fixing the Hardware

- Intel Vanderpool technology: VT-x
  - new processor mode
    - "ring -1"
      - *root* mode
      - other modes are *non-root*
  - certain operations and events in non-root mode cause *VM-exit* to root mode
    - essentially a hypercall
    - code in root mode specifies which operations and events cause VM-exits
      - e.g., `popf`, page fault
  - non-VMM OSes must not be written to use root mode!

I/O Virtualization

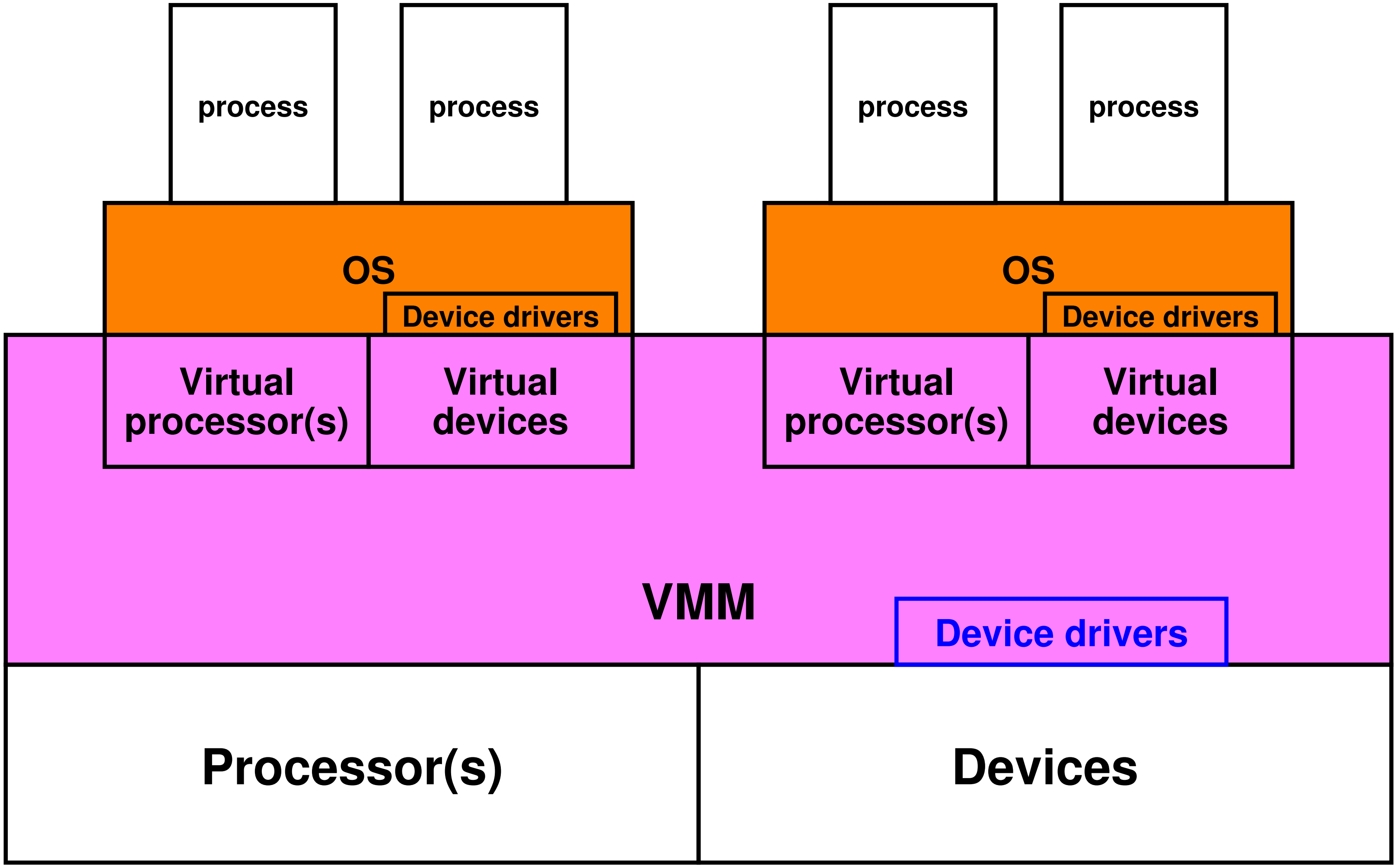
- Channel programs were generic for IBM 360
  - can be emmulated in the VMM
- I/O via memory-mapped registers is not
  - lots and lots and lots of device drivers
  - must VMM handle all of them?
    - problem: *scalability*

Real-Machine OS Structure



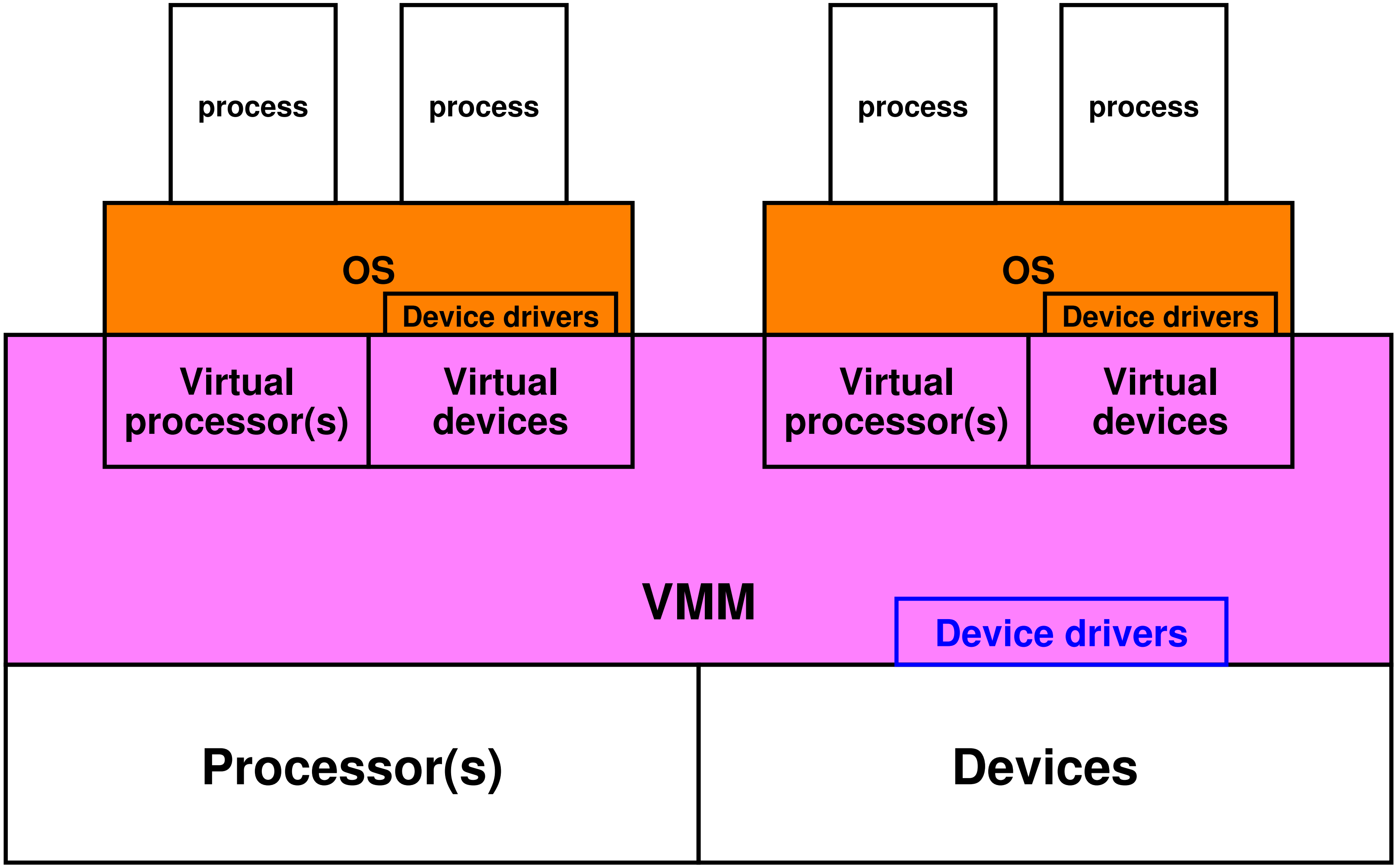
- Lots of devices need to be supported by desktop OSes (such as Windows and Mac OS X)

On a Virtual Machine ...



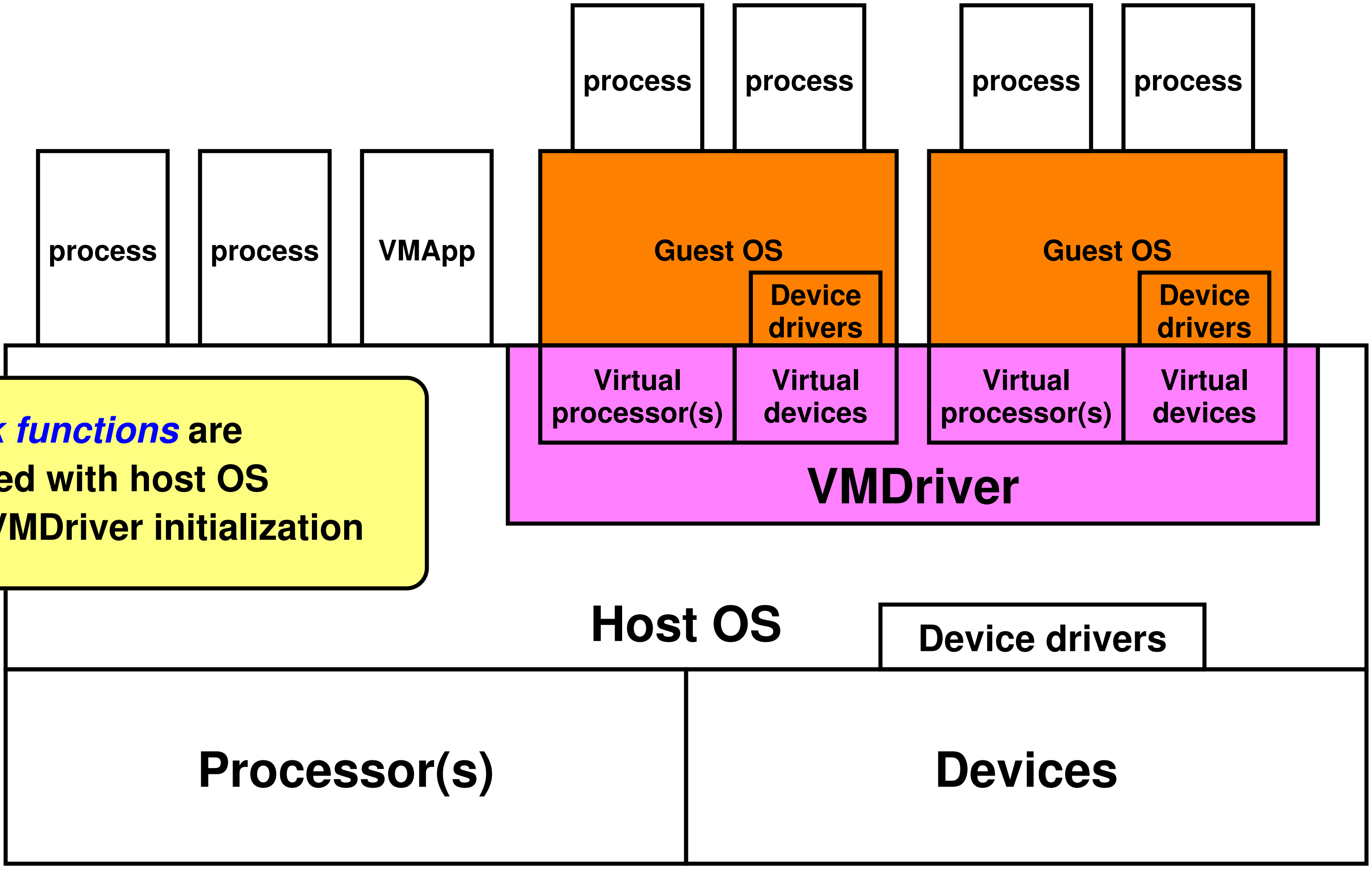
- Who is going to write all the device drives for the new OS?

On a Virtual Machine ...



- This is more suitable for server machines (higher performance)
  - scalability problem: who is going to write device drivers for VMM in lower-end machines?

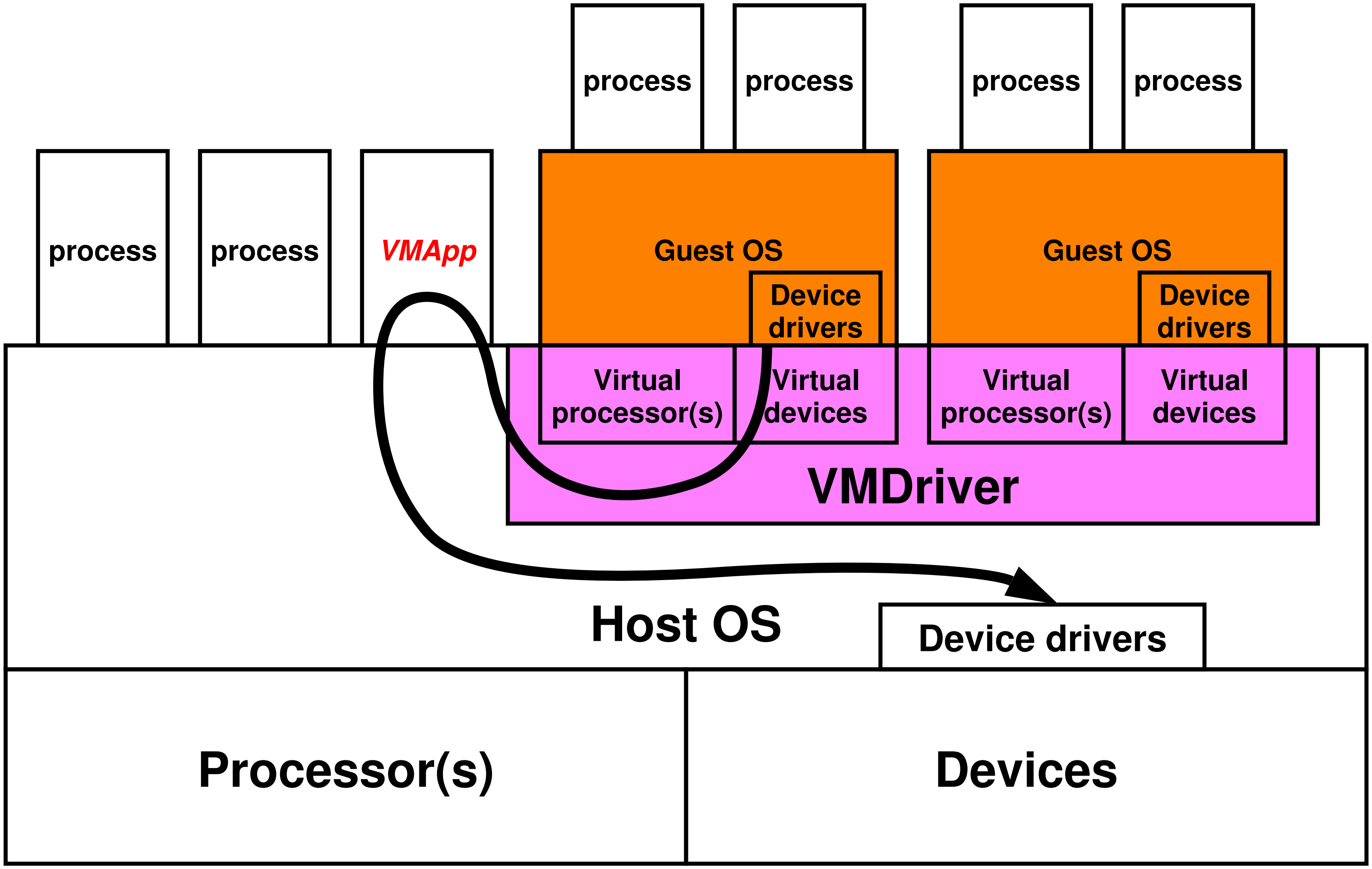
VMware Workstation - Host/Guest Model



- *callback functions* are registered with host OS during VMDriver initialization

- VMware's solution is to use a *guest/host model*
  - VMDriver takes the place of the VMM
  - plenty of device drivers already available on host OS

VMware Workstation - Host/Guest Model

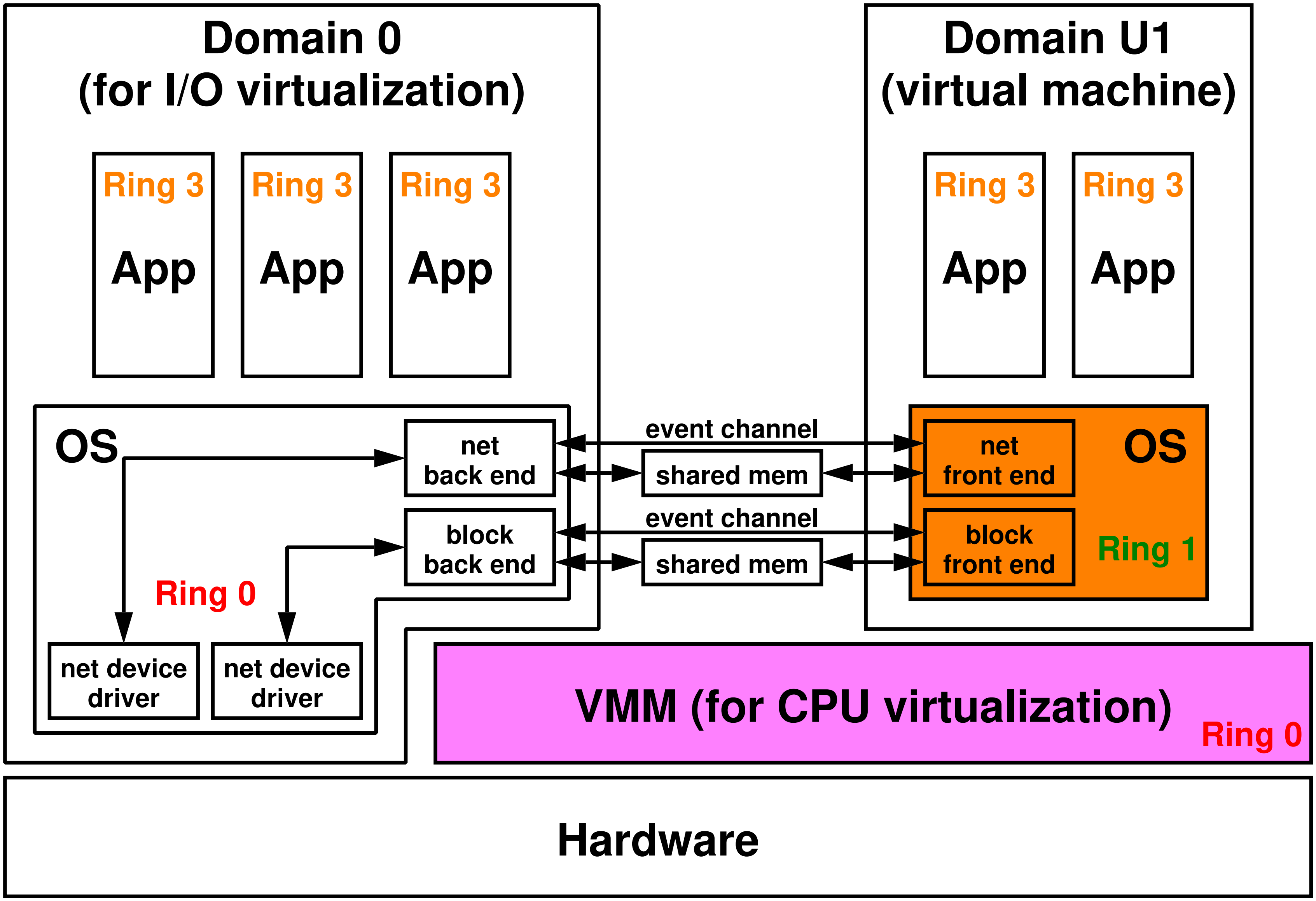


- This is more suitable for "workstations" - more variety of devices
  - *convenience* over *performance*

Paravirtualization

- Sensitive instructions replaced with hypervisor calls
  - traps to hypervisor/VMM
- Virtual machine provides higher-level device interface
  - guest machine has ***no device drivers***
    - OS is changed already, might as well change I/O, if there are sufficient benefits

Xen

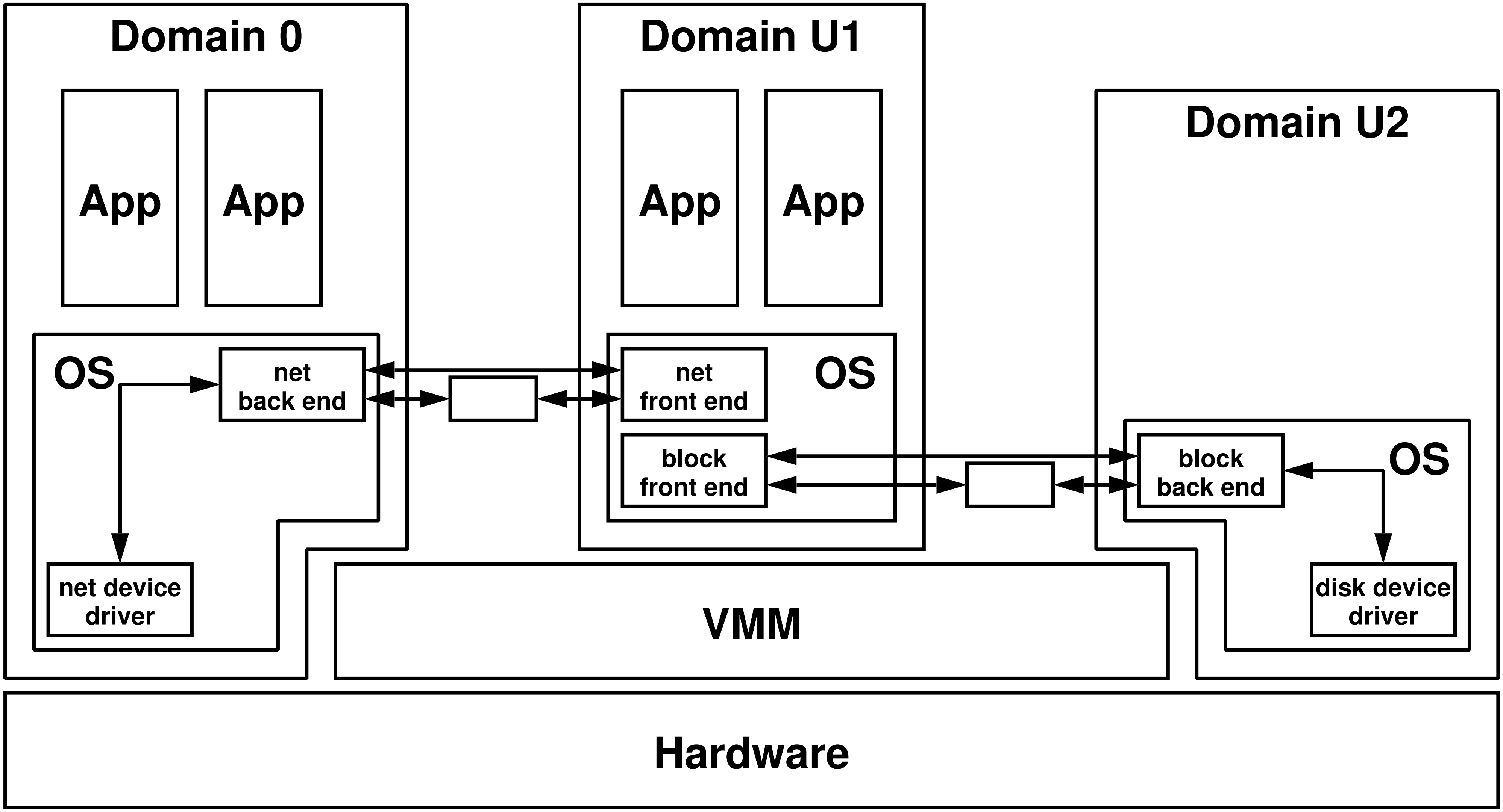


- Domain 0 OS is like the Host OS in VMware but *only for I/O*
  - it directly talks to the hardware

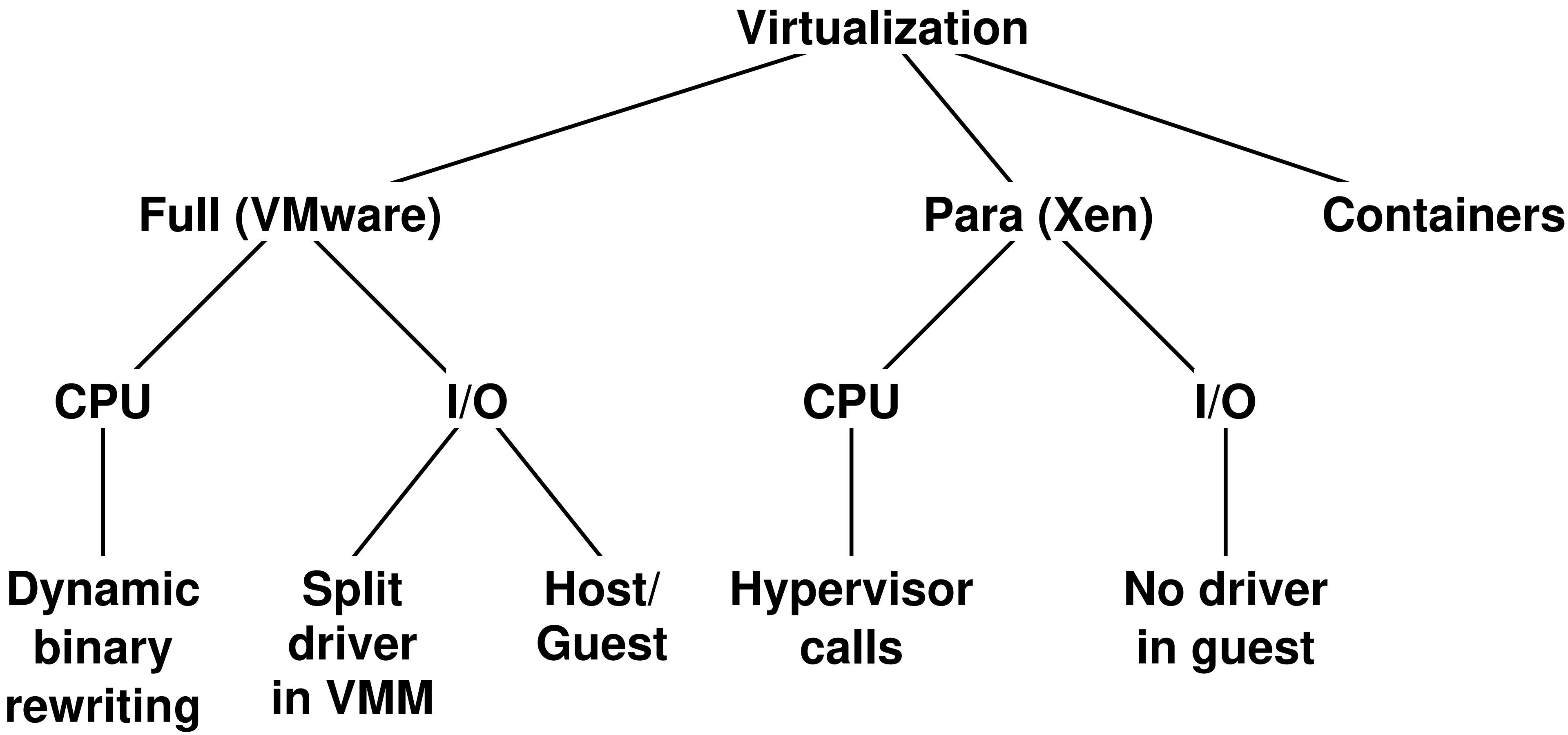
Additional Applications

- Sandboxing
  - isolate web servers
  - isolate device drivers
- Migration
  - VM not tied to particular hardware
  - easy to move from one (real) platform to another

Xen with Isolated Driver

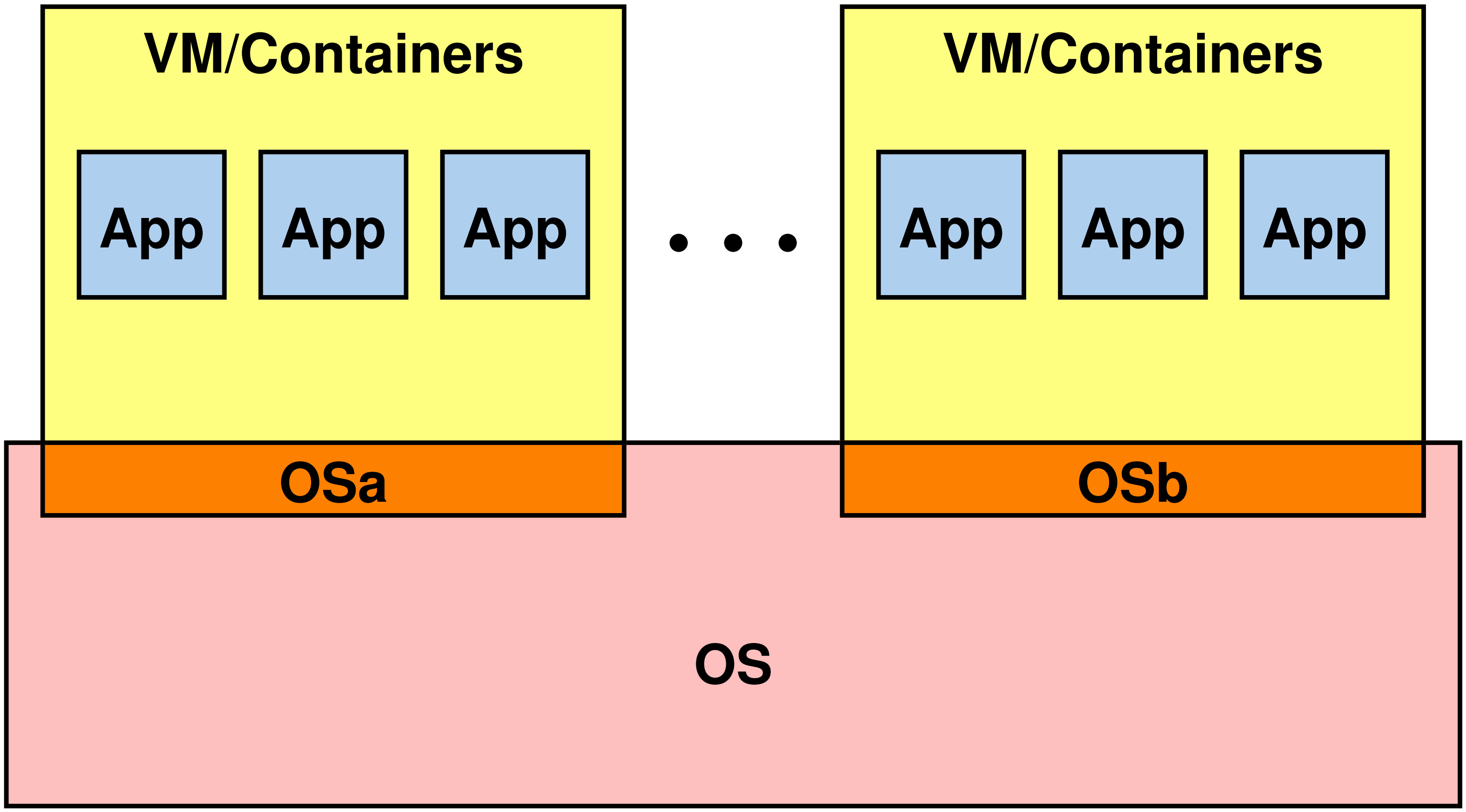


Summary


Virtualization
Full (VMware)
Para (Xen)
Containers
CPU
I/O
CPU
I/O
Dynamic binary rewriting
Split driver in VMM
Host/ Guest
Hypervisor calls
No driver in guest

One More Kind Of Virtualization

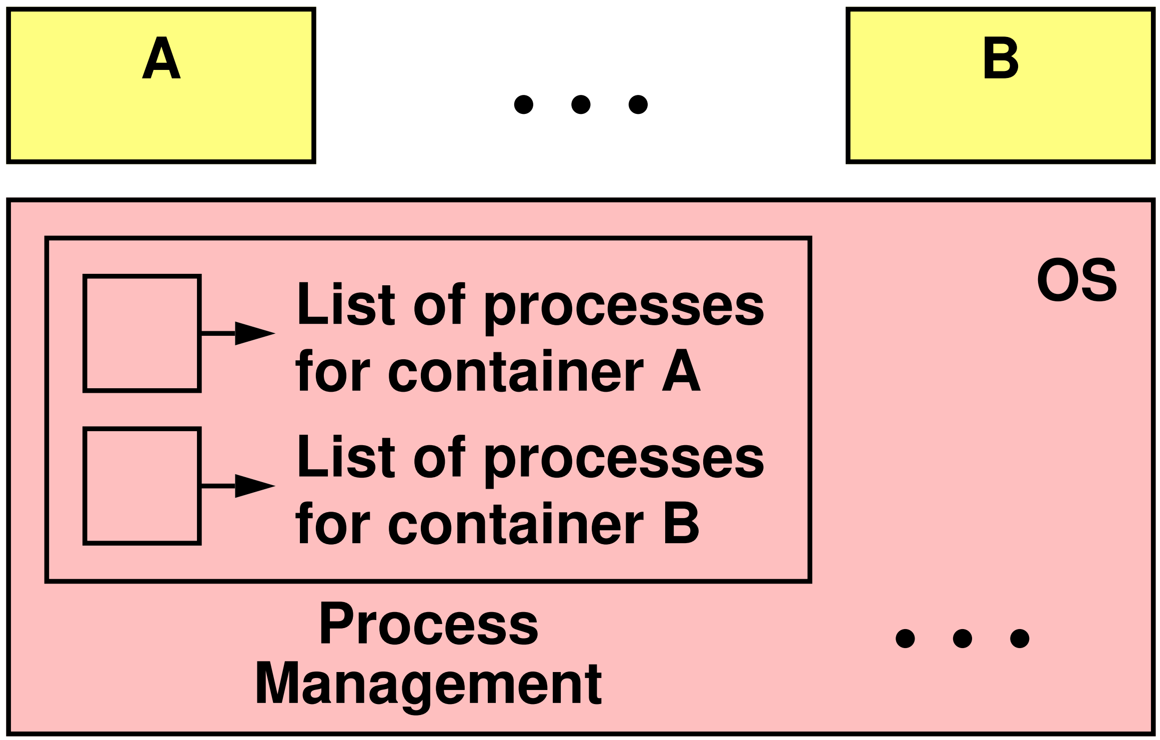
- Containerized OS (or OS Containers)
  - not covered in textbook



  - the OS provides the abstraction that each container runs on top of a separate OS (but there is really only one OS)
  - e.g., OpenVZ, Linux Containers (LXC), Docker

Containerized OS

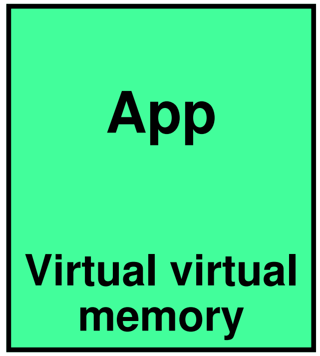
- Within the OS, the management of resources for each container is separated
  - e.g., processes for container A is kept separate from processes for container B



- Others may consider this "virtual machine", but we shouldn't
  - because "guest OS" does not run in user space here and there is really no "guest OS"

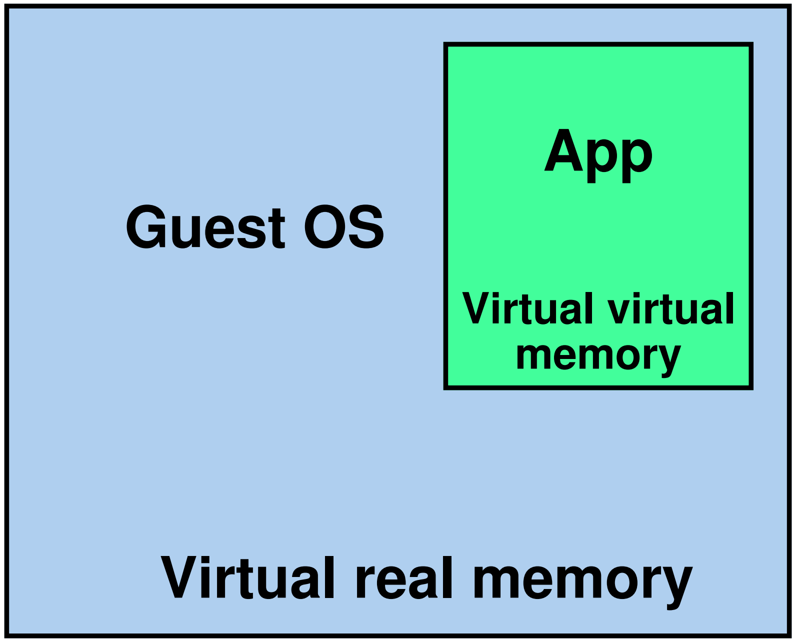
7.2.6 Virtualizing Virtual Memory

Virtual Machines Meet Virtual Memory



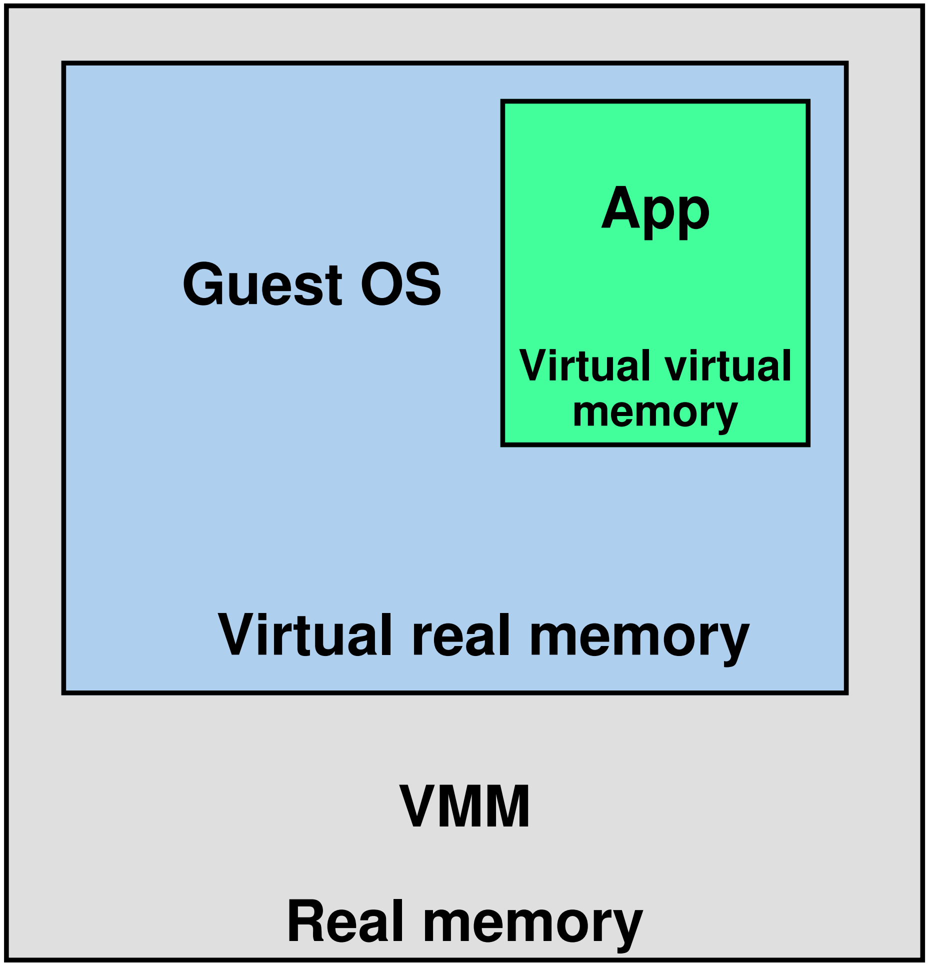
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*

Virtual Machines Meet Virtual Memory



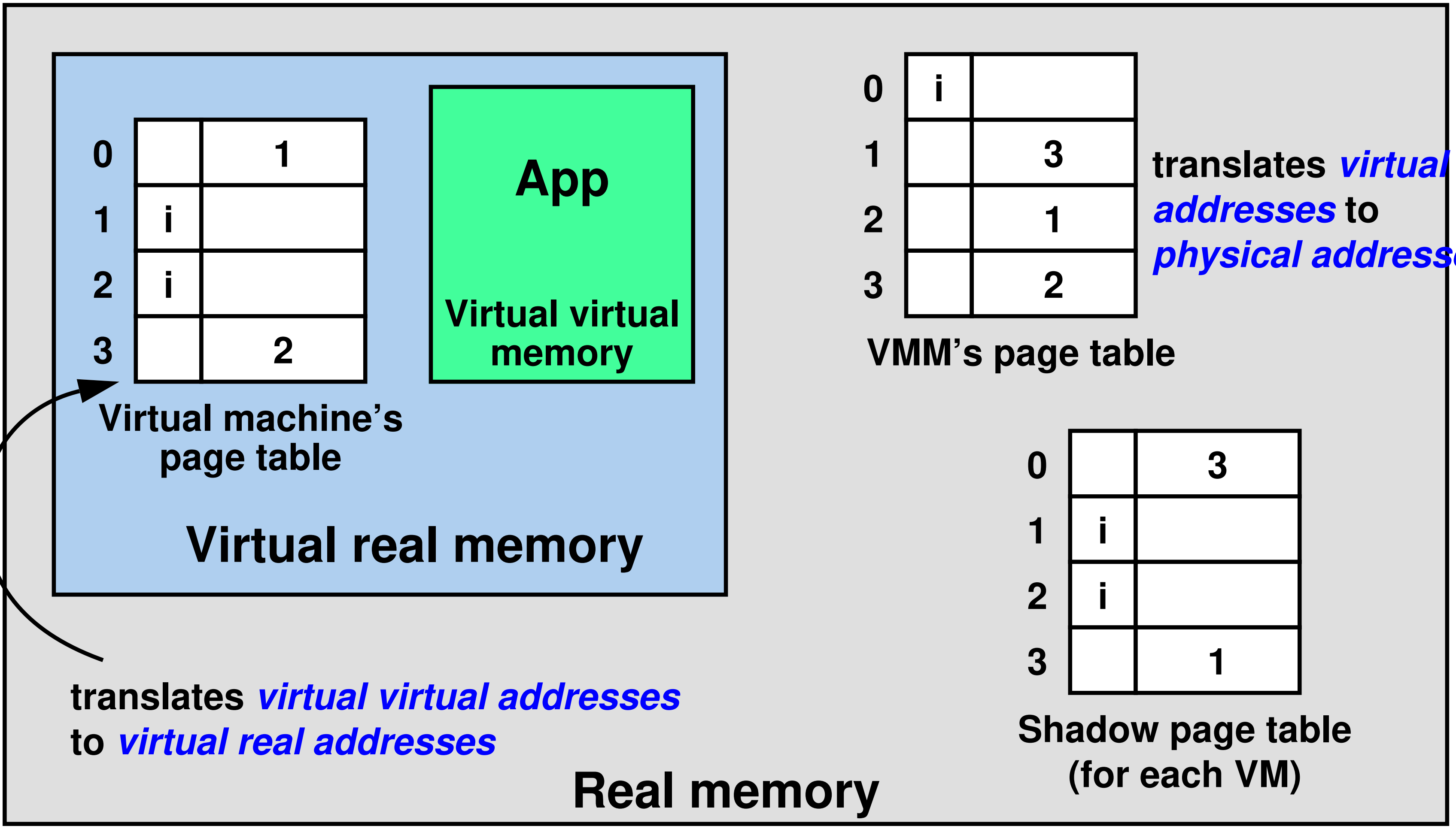
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*

Virtual Machines Meet Virtual Memory



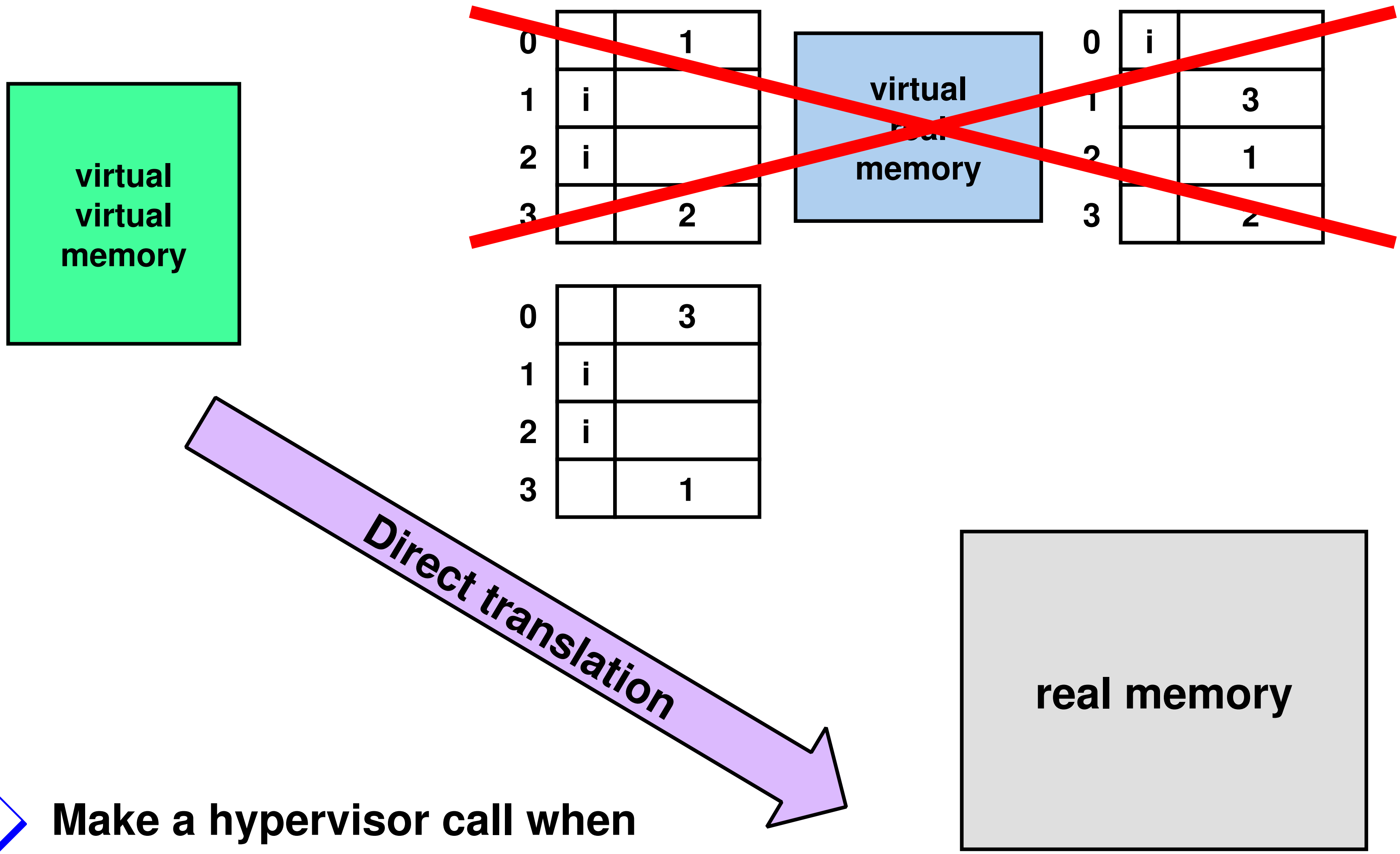
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*
- VMM needs to manage real memory
  - how can we *virtualize virtual memory*?

Virtual Machines Meet Virtual Memory



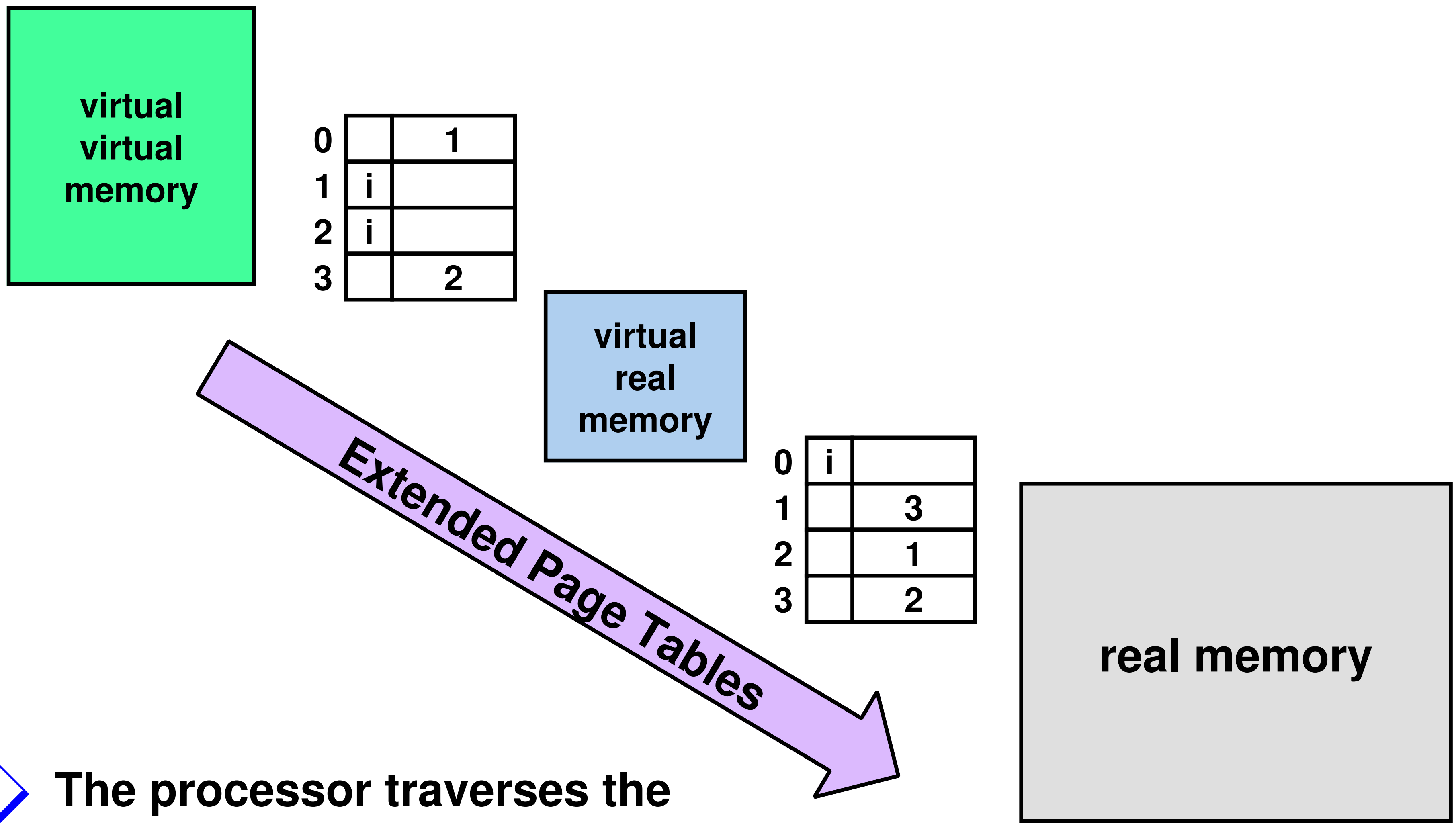
- When a VM changes its page table, VMM must update the corresponding *Shadow Page Table*
  - main problem: poor performance

Solution 1: Paravirtualization to the Rescue



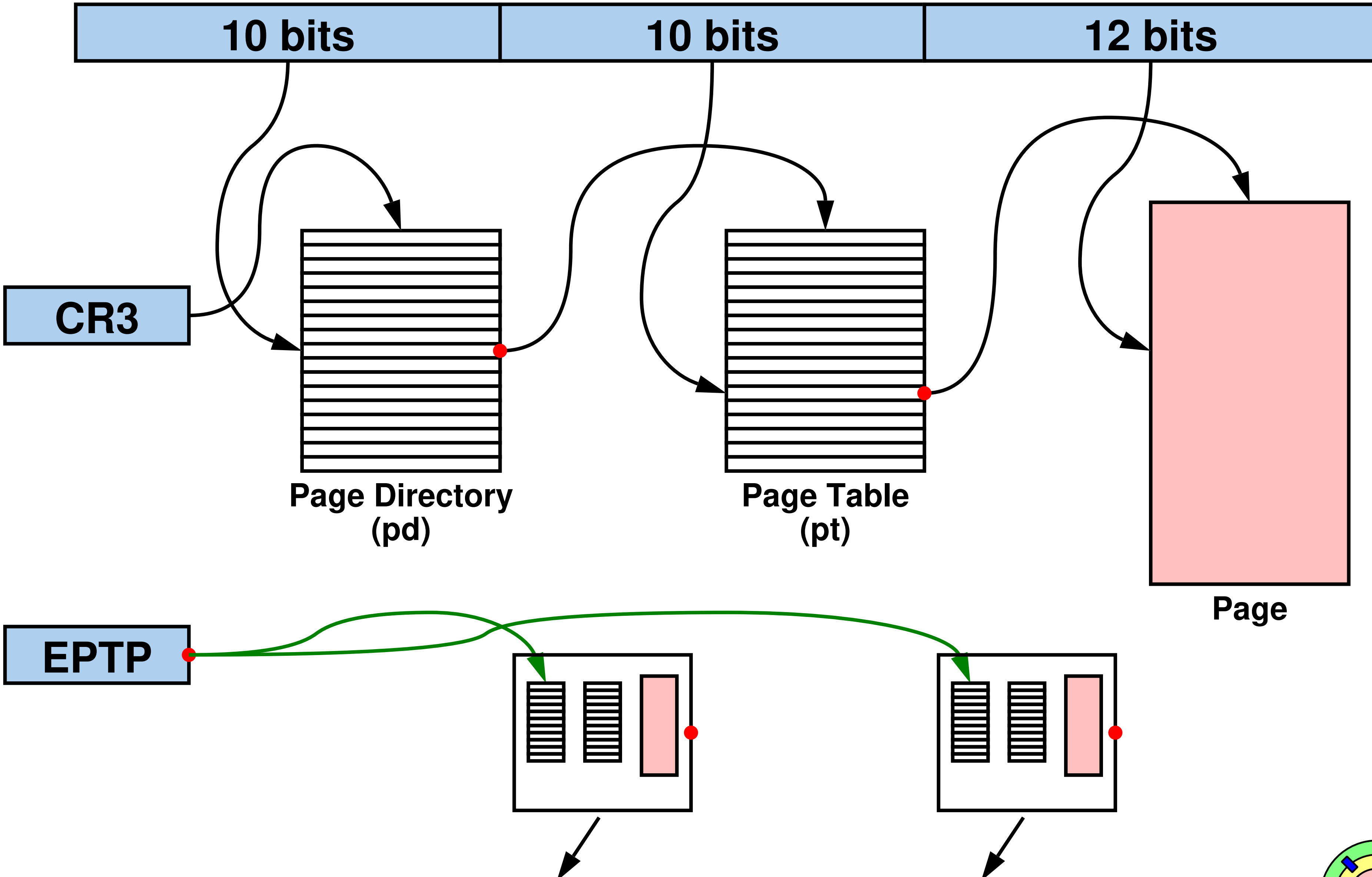
- Make a hypervisor call when page table needs to be modified
  - helps a bit, but not much faster

Solution 2: Hardware to the Rescue



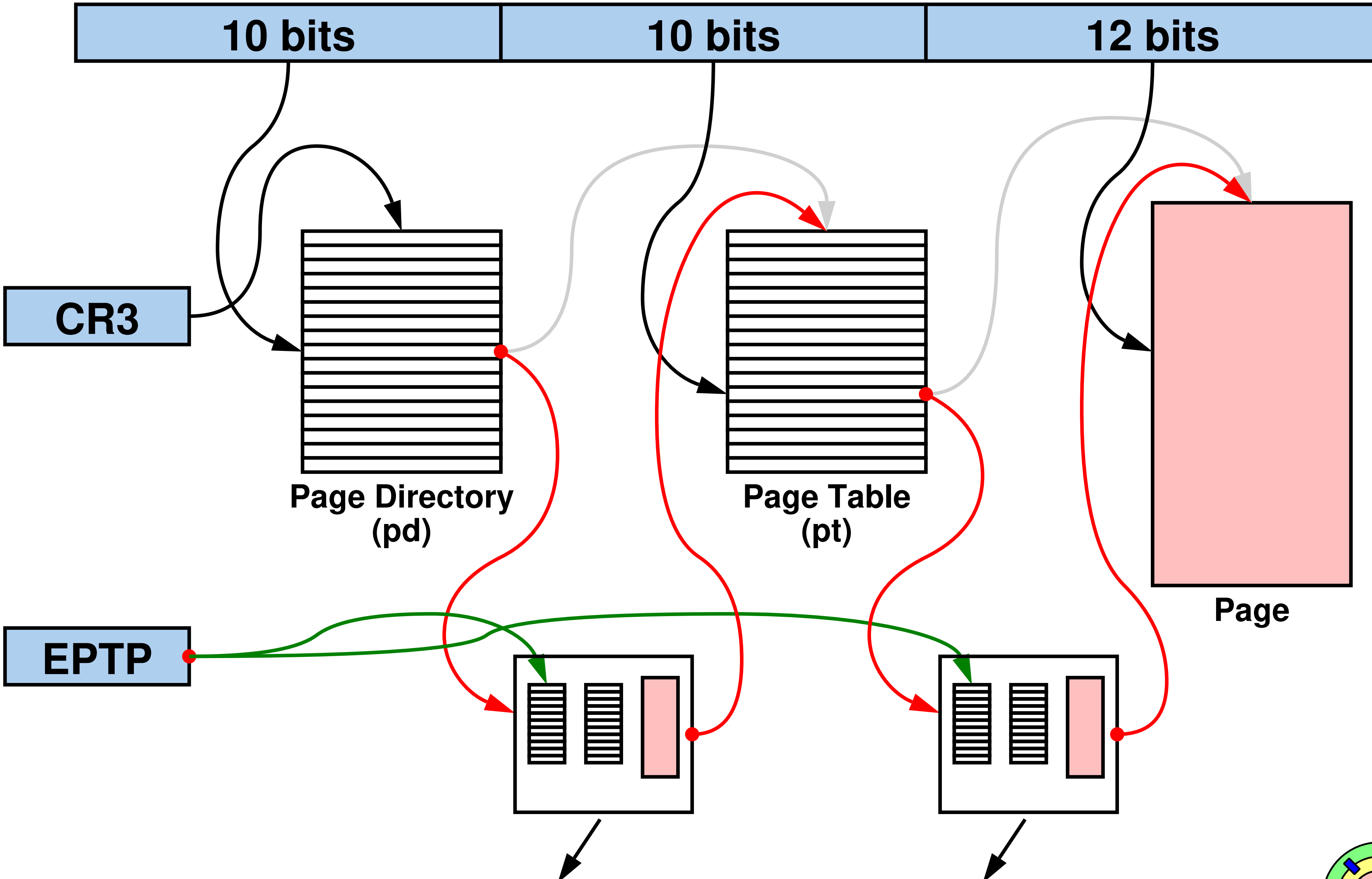
- The processor traverses the two tables in sequence and does the conversion all by itself

x86 Paging with EPT



these translates *virtual real addresses*
to *physical addresses*

x86 Paging with EPT



these translates *virtual real addresses*
to *physical addresses*