4.2 Rethinking Operating-System Structure

- Virtual Machines
- Microkernel

Monolithic Kernel

- **Major advantage of monolithic kernel**
  - **performance**
- **Major down side of monolithic kernel**
  - **reliability (i.e., buggy kernel)**
- **Proposal to fix the reliability problem**
  - **shrink the code in "privileged mode"**
- **Two major approaches**
  - ***virtual machines***
  - ***microkernel***

Virtual Machines

- A nicely designed and implemented monolithic OS is great
  - but that's not the reality
- Major problem with a monolithic OS *implementation*
  - bugs in one component can adversely affect another component
    - worse if large number of programmers contribute code
      - some coders are not as good as others
      - good coders have bad days
- Modern OSs *isolate* applications from one another
  - code executing in the privileged mode can do things the user mode code cannot
    - e.g., invoking privileged instructions
  - if you invoke a privileged instruction in user mode, you will cause a violation and trap into the kernel
- Can the same kind of *isolation* be provided for *OS components*?
  - if yes, at what cost? (there is no free lunch)

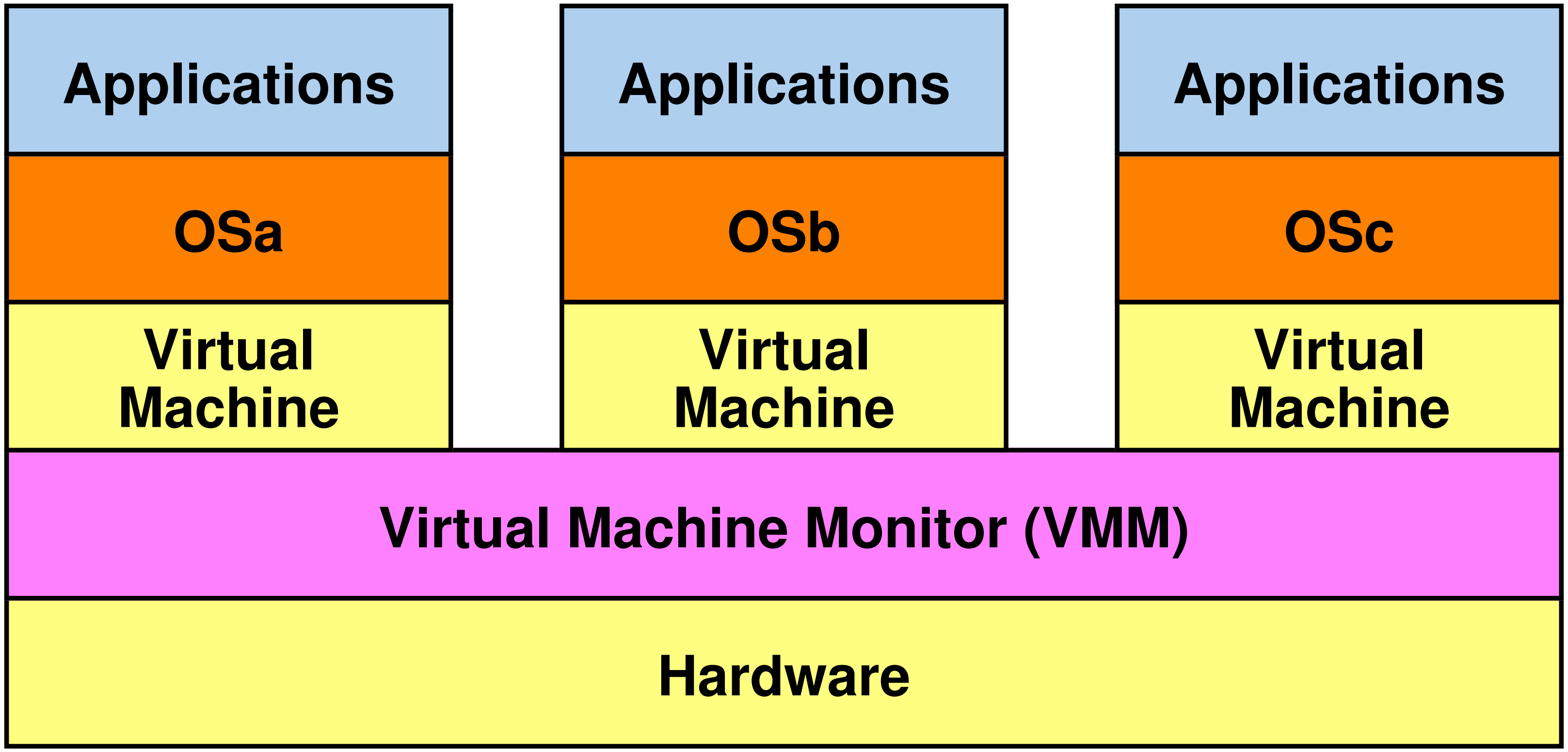
Virtual Machines
Part 1: > 50 Years Ago

- Had a different motivation

It's 1964 ...

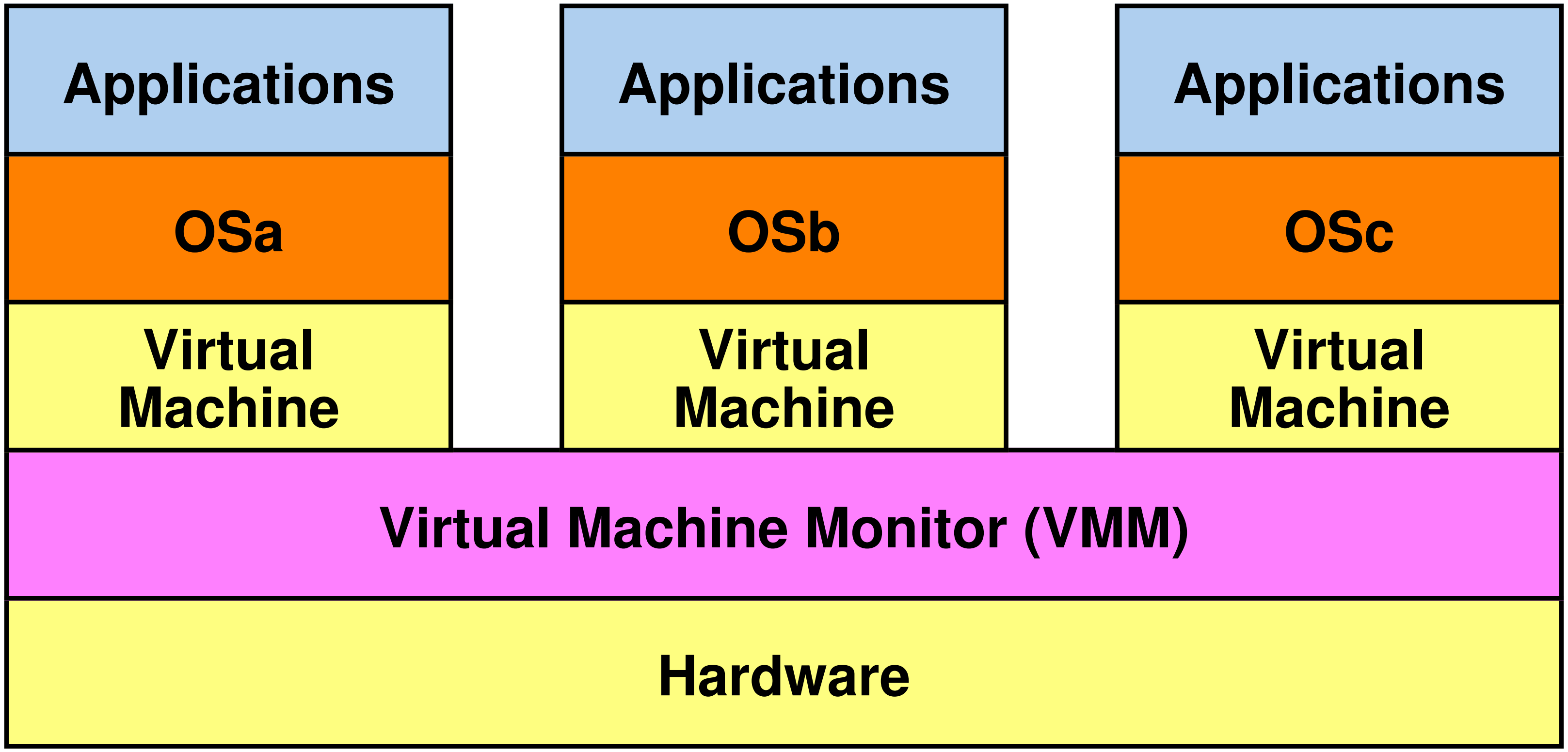
- IBM has a *single-user time-sharing system* called CMS
  - IBM wants to build a *multiuser time-sharing system*
- TSS (Time-Sharing System) project
  - it's a very difficult system to build
  - large, monolithic system
  - lots of people working on it
  - for years
  - total, complete flop
- CP67
  - *virtual machine monitor (VMM)*
  - supports multiple virtual IBM 360s
- Put the two together ...
  - a (working) *multiuser time-sharing system*

Virtual Machines



- A *"monitor"* is a *synchronization construct* that allows executing entities to have both *mutual exclusion* and the ability to *wait* (block) for a certain condition to become true
- What abstraction does a *virtual machine* provide?
  - *hardware*

Virtual Machines



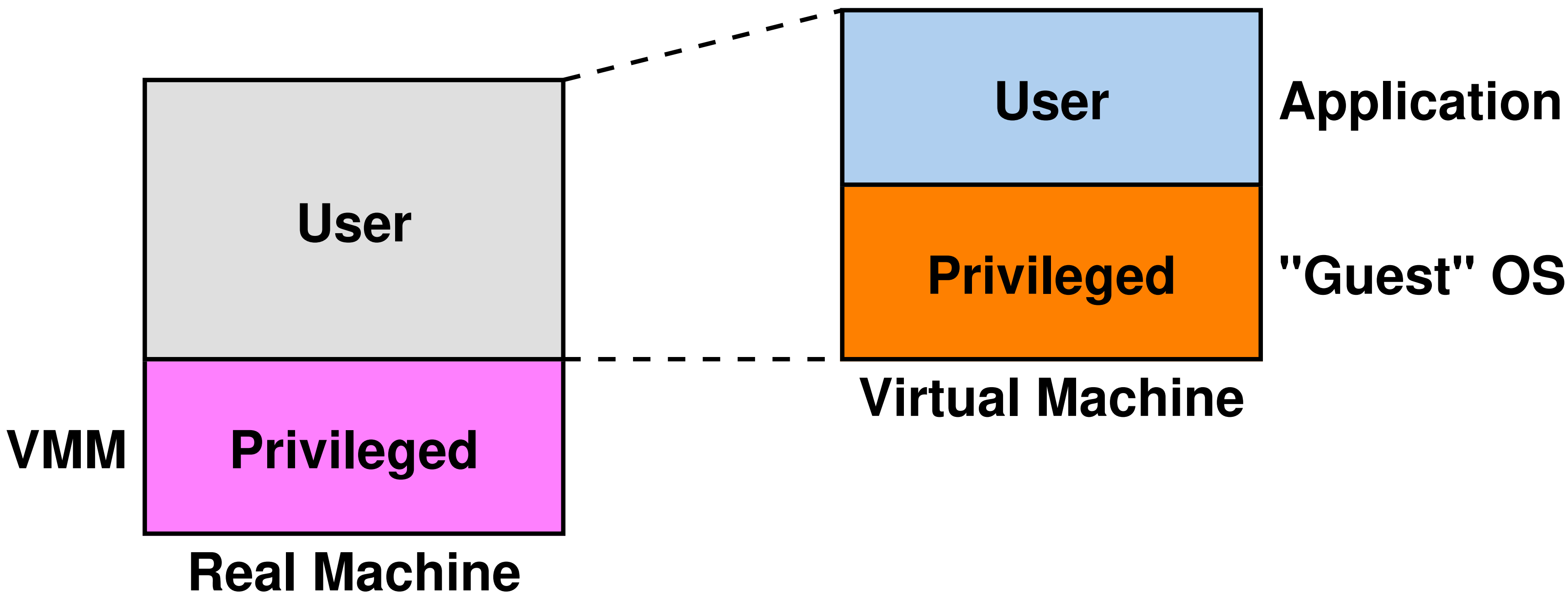
- A single user time-sharing system could be developed independently of the VMM
  - and it can be tested on a real machine (which behaves identical to the VM)
  - no ambiguity about the interface VMM must provide to its applications - *identical* to the *real machine!*

Virtual Machines

- What is considered a virtual machine (for this class)?
  - *run* (not emulate/simulate) OSx "inside" / "on-top-of" OSy
    - we will refer to OSx as the "guest OS" and OSy as the "host OS" (these terms came from VMware)
    - a virtual machine is *not* an OS *emulator*
      - must *execute* "guest OS" on the real CPU *directly*
  - make "guest OS" think that it's running on hardware, but in reality, it is running inside a *virtual machine*
    - therefore, the *code* and *data structures* you put into "host OS" so that you can run "guest OS" in it is called "virtual machine"
  - "host OS" may be a specialized OS

- Different types of virtualization technologies
  - *pure virtualization:* "guest OS" is unmodified
    - "guest OS" thinks it's running directly on hardware
  - *para-virtualization:* "guest OS" is modified
    - modified "guest OS" can only run inside virtual machine
  - something else: we *shouldn't* call it a virtual machine

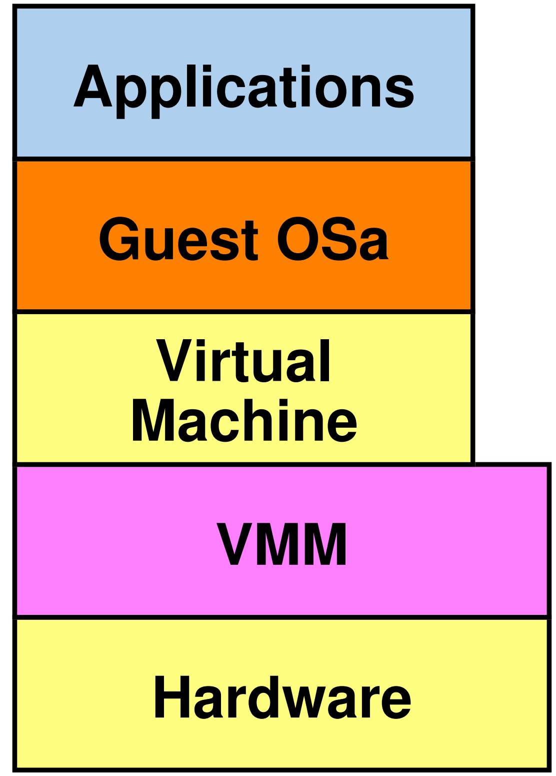
How?



- Run the *entire virtual machine* in *user mode* of the real machine
  - *VMM* runs in the *privileged* mode of the *real* machine
- VMM keeps track of whether each virtual machine is in the *virtual privileged mode* or in the *virtual user mode*
  - *OS* runs in the *(virtual) privileged* mode of the *virtual* machine
  - Applications runs in the *(virtual) user* mode of the *virtual* machine

VMM Operations

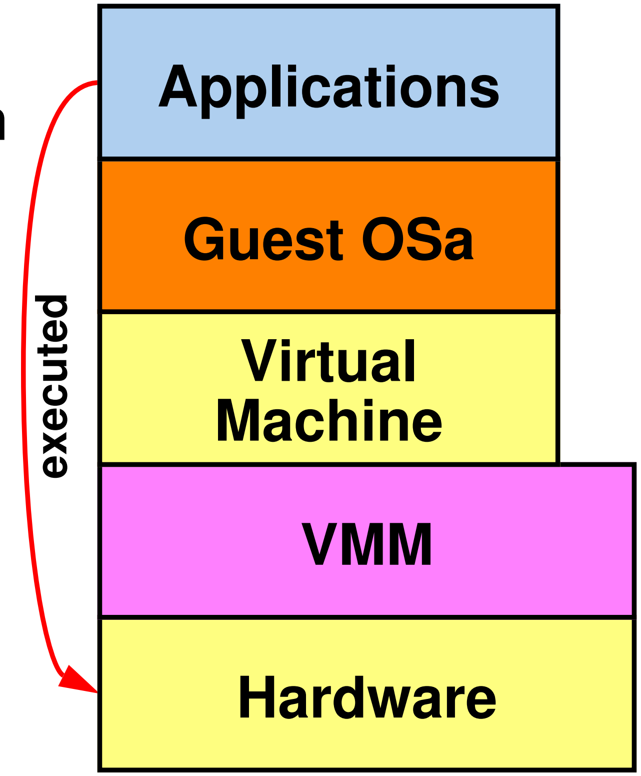
- Execute a *non-privileged* instruction
  - e.g., "add", "mul", pointer manipulation



VMM Operations

- Execute a *non-privileged* instruction
  - e.g., "add", "mul", pointer manipulation

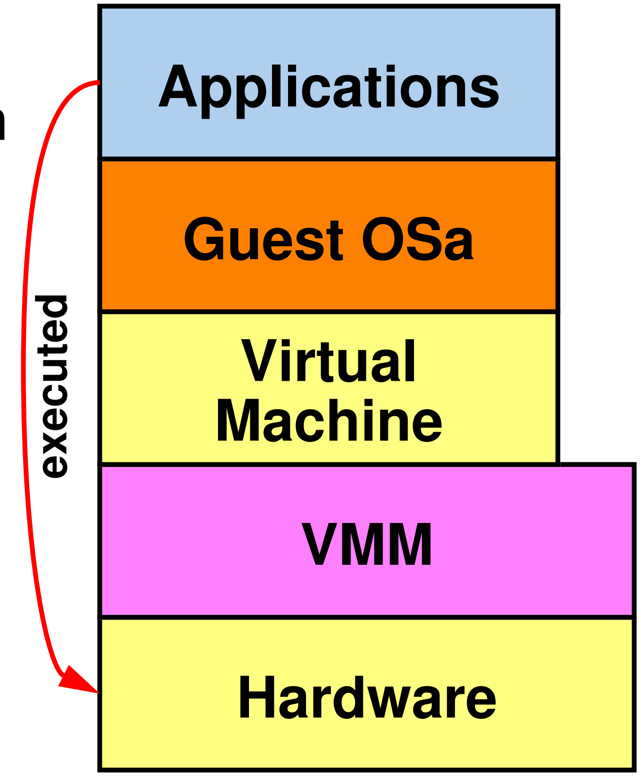
executes directly on hardware
- from application's perspective, no difference running in VM or on hardware



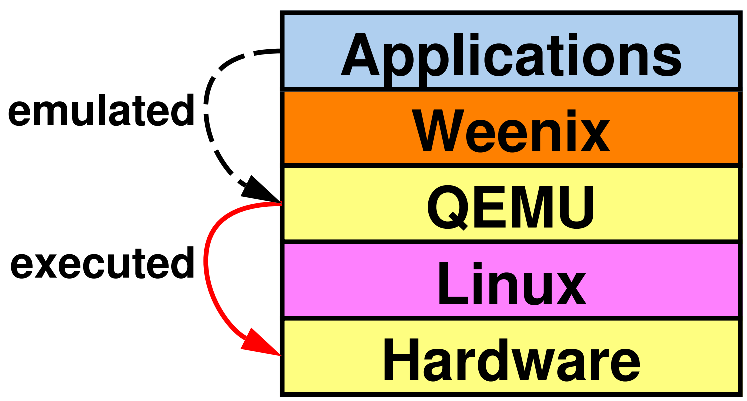
VMM Operations

- Execute a ***non-privileged*** instruction
  - e.g., "add", "mul", pointer manipulation

executes directly on hardware
- from application's perspective, no difference running in VM or on hardware



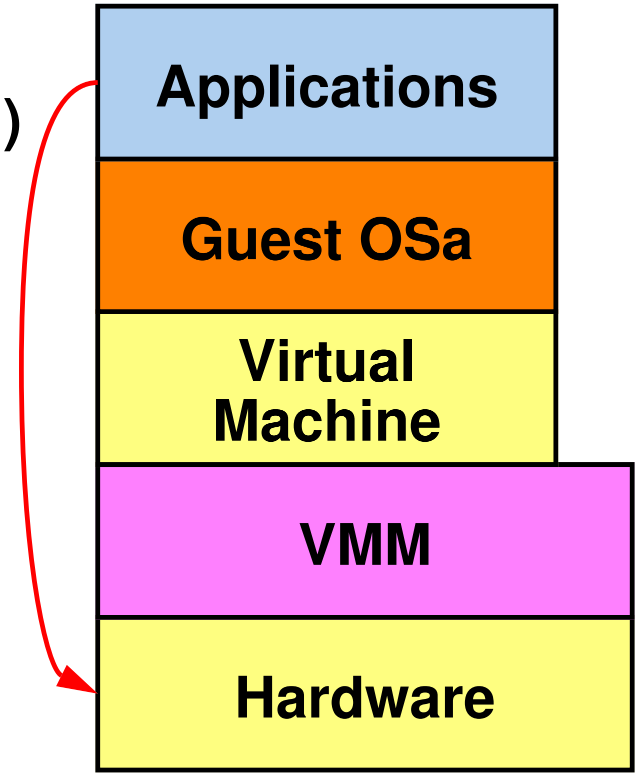
- Note: this looks like our kernel assignments (but quite different)



a ***emulator program*** for the x86 instruction set

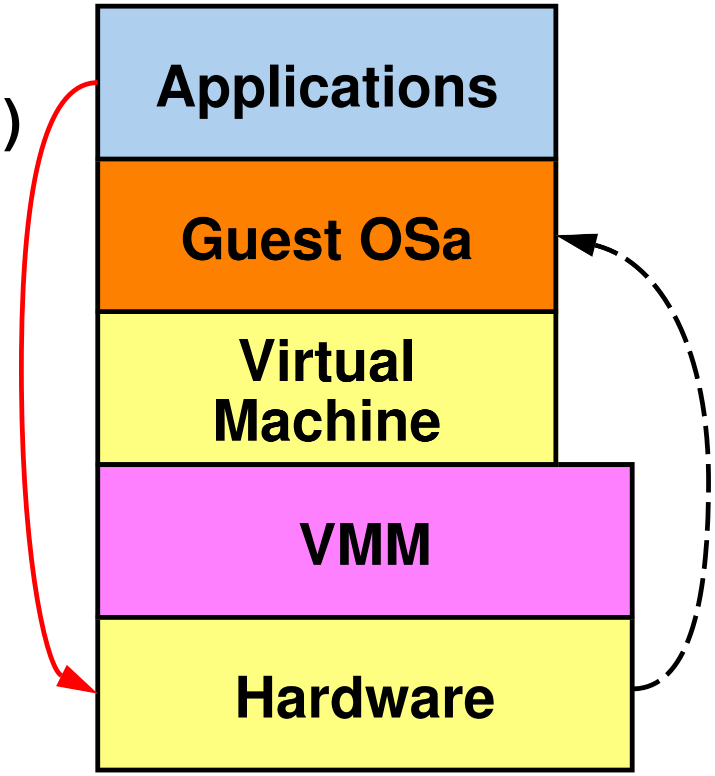
VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)



VMM Operations

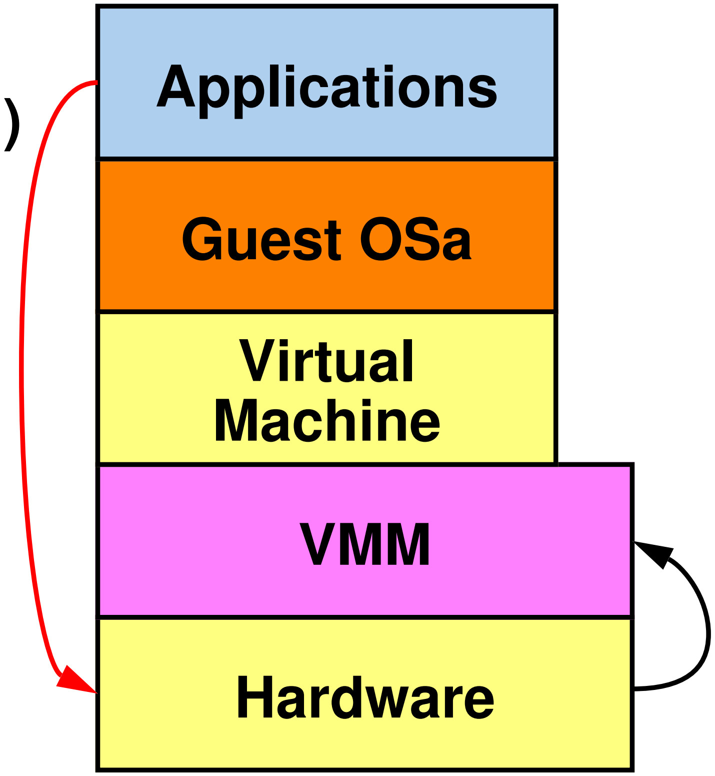
- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)
    - in a real machine, trap handler is indexed by the trap number into a hardware-mandated jump table
    - the VMM needs to find the address of the virtual machine's trap handler in the table and transfer control to it



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

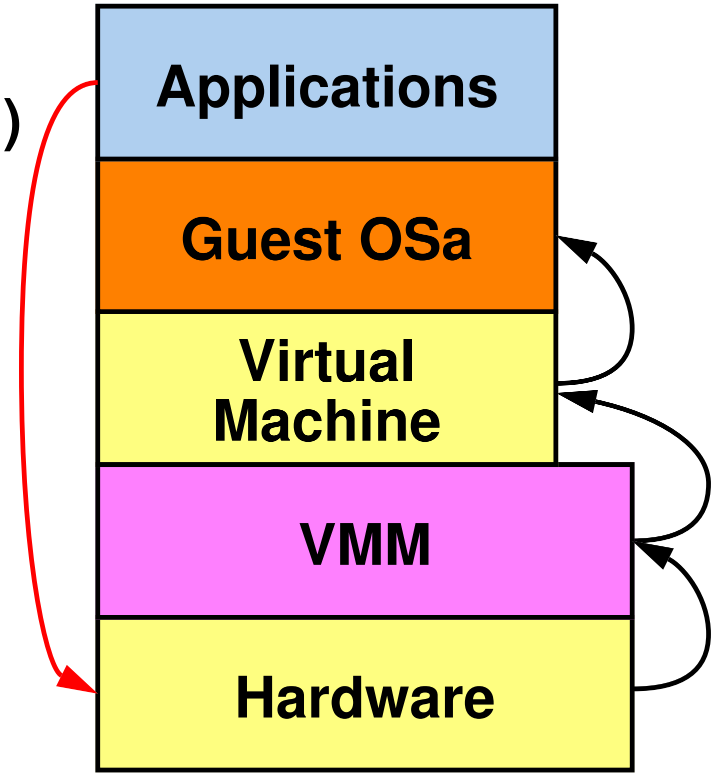
the VMM is invoked
- the VMM figures out which VM is currently executing



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

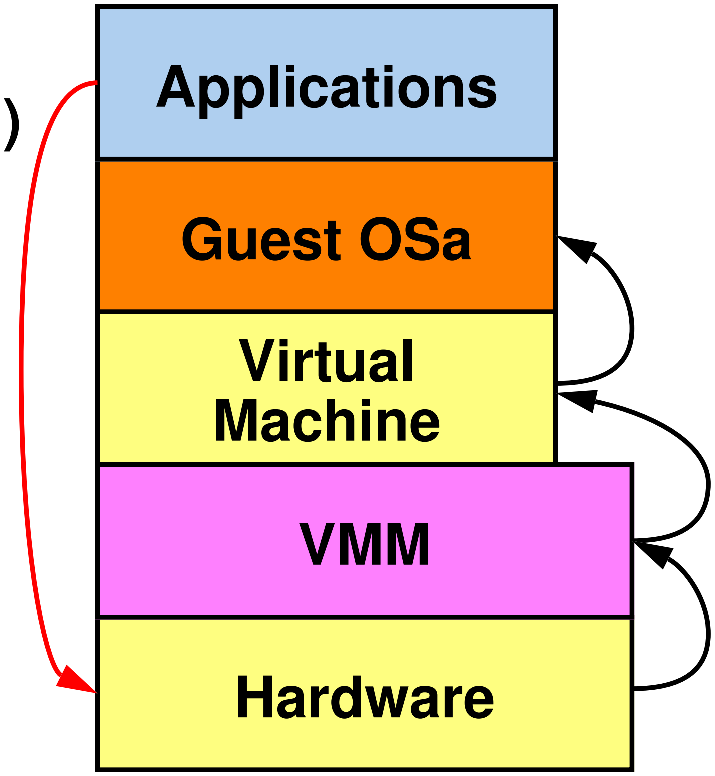
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

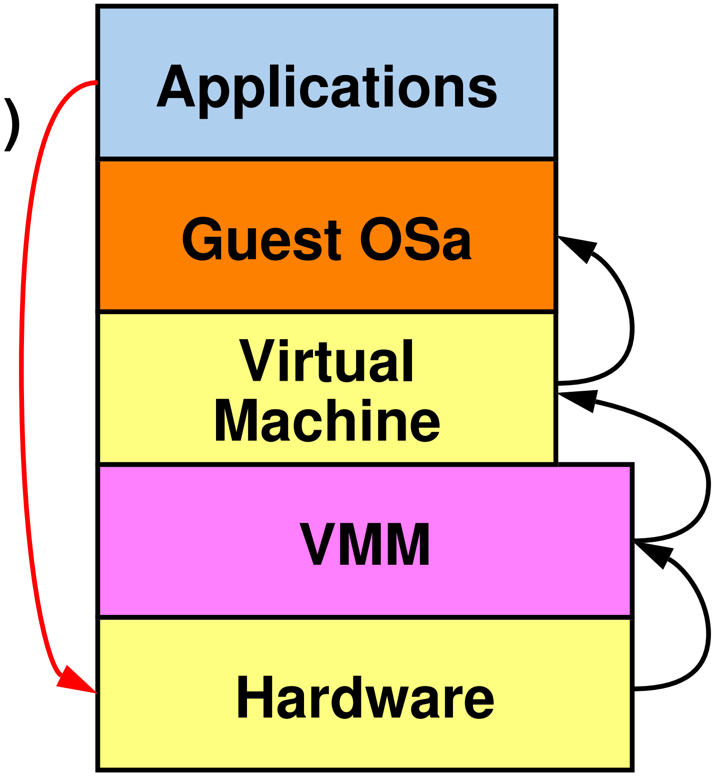
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



- Without VM, the application will simply traps into the OS directly
  - now it's a lot more involved (and slower)
- *Interrupts* pretty much work the same way

VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)



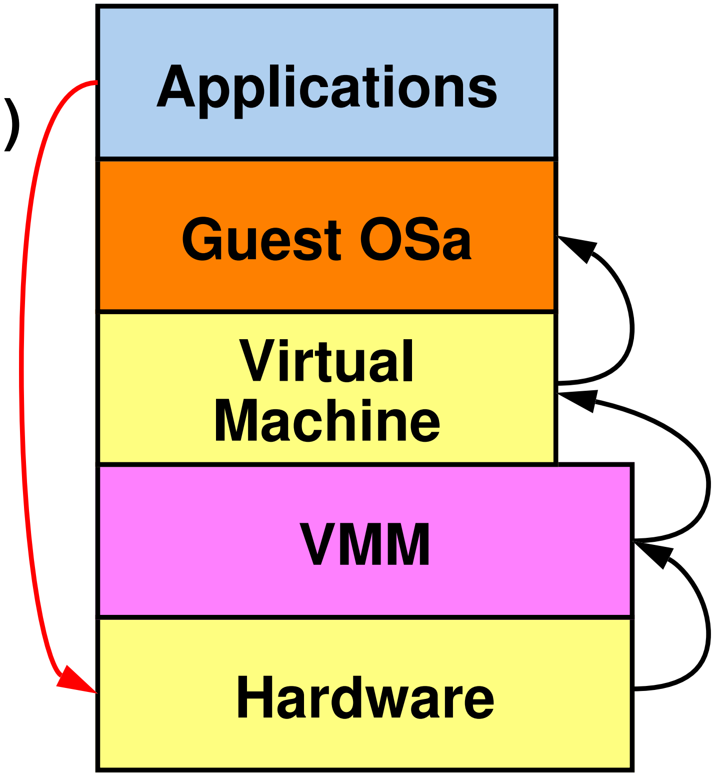
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*

- "Virtual Machine" in the picture contains:
  - virtual CPU, virtual disk, virtual display, virtual keyboard, etc.
    - *data structures* and *code* that *represent hardware components*

VMM Operations

- Execute a *privileged* instruction
  - e.g., "trap" (system call, page fault, etc.)

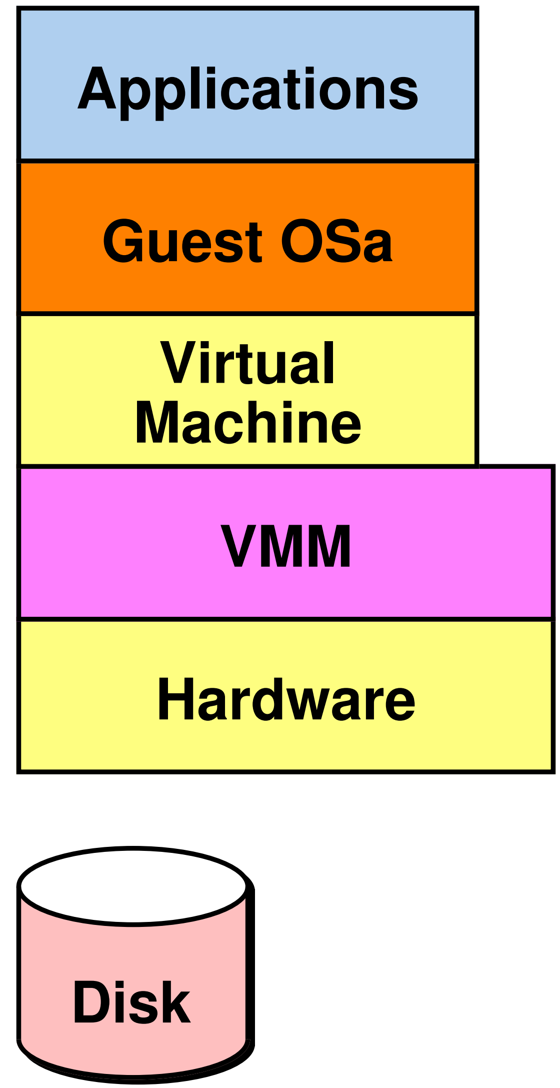
the VMM is invoked
- the VMM figures out which VM is currently executing
- VMM then asks the corresponding VM to *deliver* the trap to its OS
  - VMM should be *virtual machine independent*



- Note that most instructions the trap handler executes are *not* privileged (such as the code to setup PCB, TCB, etc.)
  - clearly, these instructions can run in non-privileged mode
  - what type of code *must* execute in *privileged* mode? (later)
    - what if "return from interrupt" is not privileged?

VMM Operations

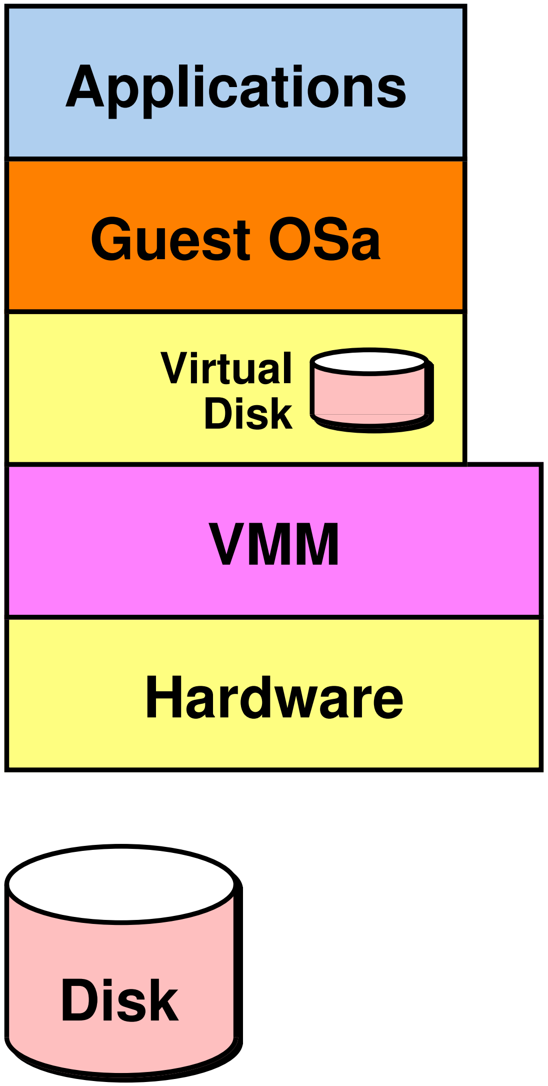
- What about I/O?
  - e.g., `read()`



VMM Operations

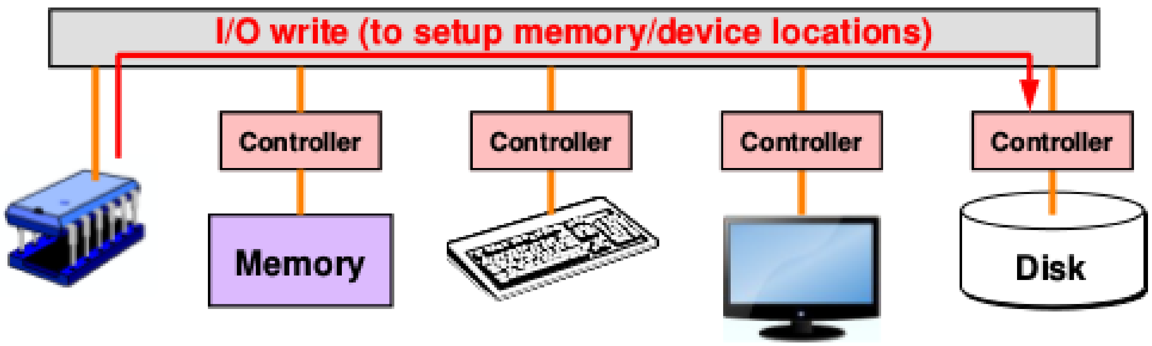
- What about I/O?
  - e.g., `read()`

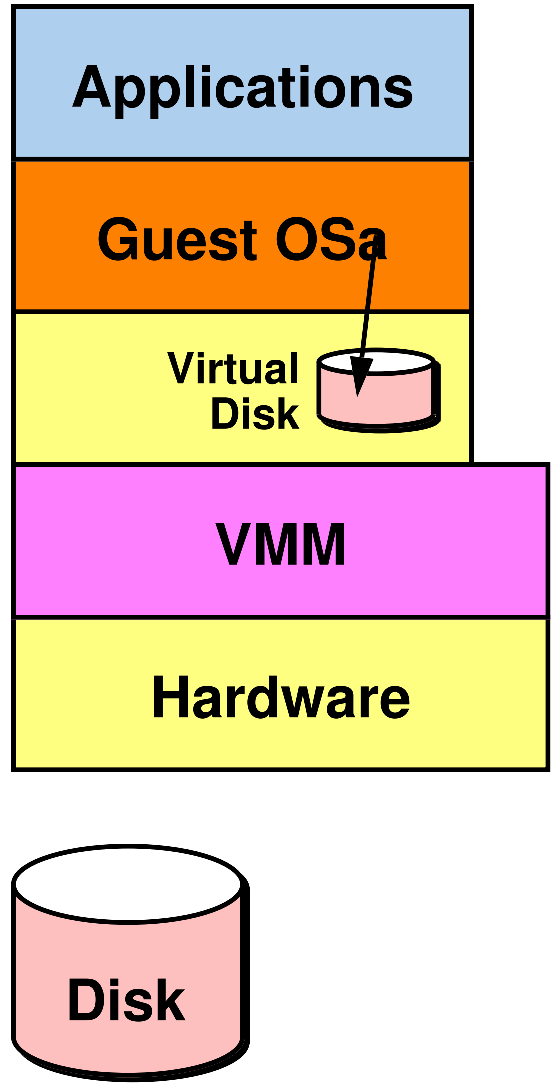
real disk is divvy up among the virtual machines
- each VM has a virtual disk



VMM Operations

- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O



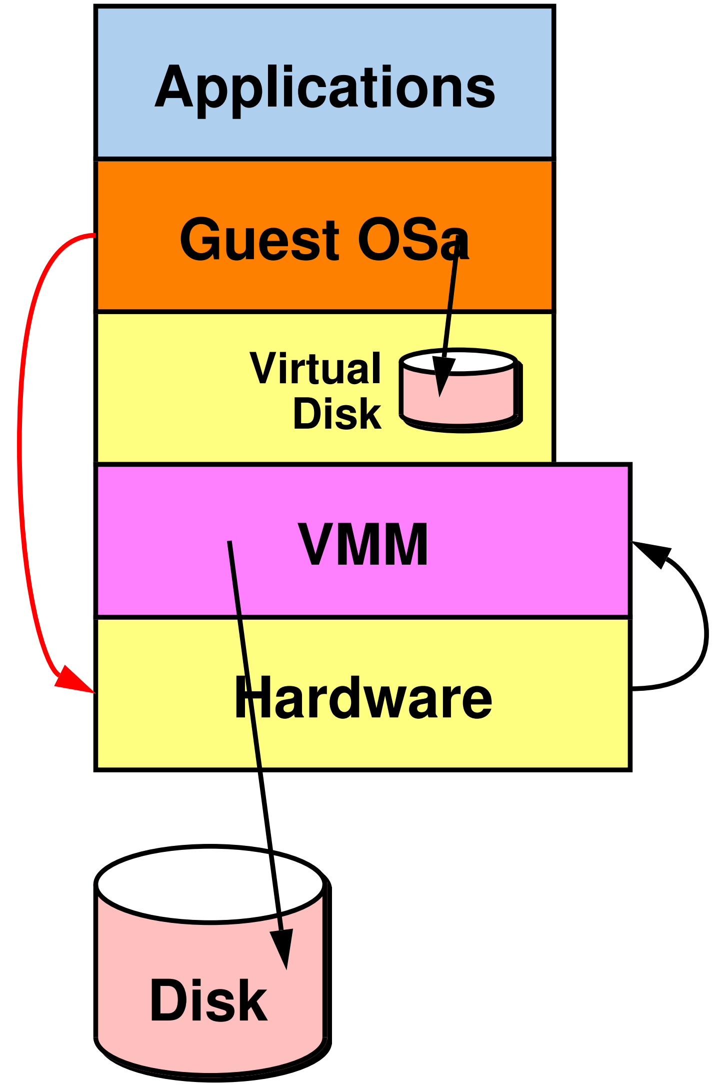


VMM Operations

- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O

memory-mapped I/O causes a trap into VMM

- the VMM *emulates* the instruction (i.e., translates it into a request for the real disk)
  - there is *no "handler"* in the guest OS for I/O instructions
- there's really *no* disk in the VM

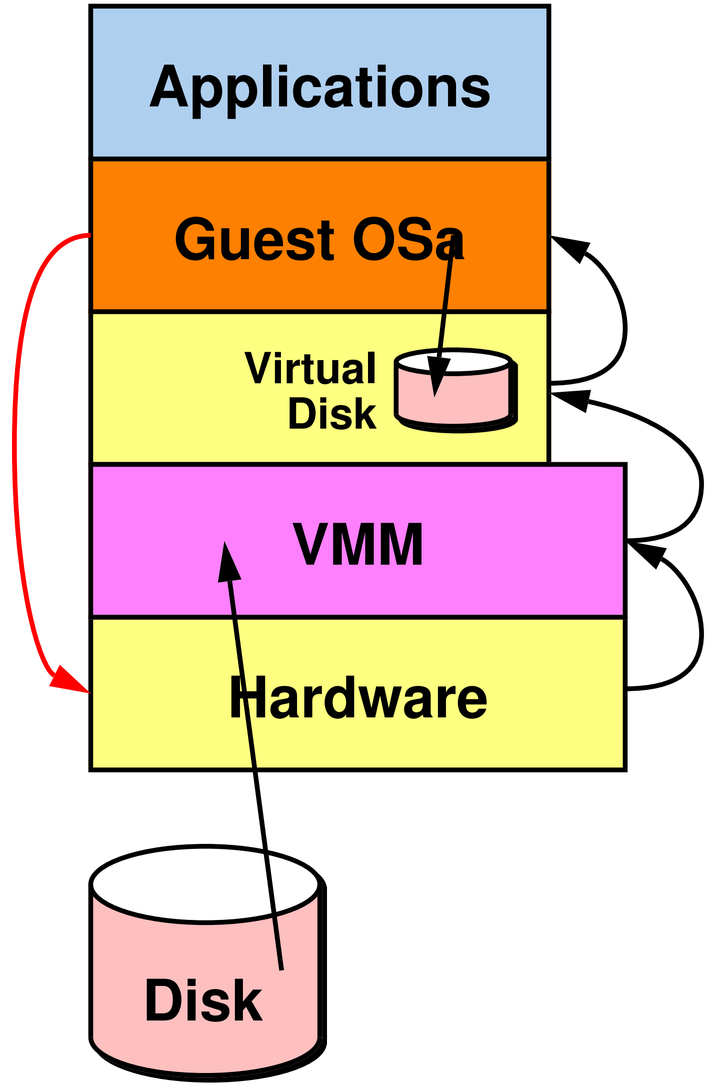


VMM Operations

- What about I/O?
  - e.g., `read()`
  - `read()` eventually reaches the OS
  - the OS asks for a block on the virtual disk
    - in x86: memory-mapped I/O

memory-mapped I/O causes a trap into VMM

- the VMM *emulates* the instruction (i.e., translates it into a request for the real disk)
  - there is *no "handler"* in the guest OS for I/O instructions
- there's really *no* disk in the VM
- the guest OS is expecting an interrupt



Why Virtual Machine?

- It's a good structuring technique for a multi-user system
  - many advantages
- OS debugging and testing
  - run a production OS in a VM, accessible to users
  - test a new OS in a separate VM, accessible to developers
- *Adapt to hardware changes in software*
- *Multiple OSes on one machine*
  - one type of applications run really well in one OS
  - another type of applications run really well in a different OS
  - one physical machine can support both, no user need to suffer
  - today, it's common that a machine in the cloud would run multiple Linux OS instances and multiple Windows OS instances
- *Server consolidation* and *service isolation*
  - web hosting, security concerns
  - cloud computing

Virtualization Requirements

- A virtual machine is an efficient, isolated duplicate of real machine
  - requires faithful virtualization of pretty much all components
    - processor
    - memory
    - interval timers
    - I/O devices
    - etc.
  - this is "*pure*" virtualization
    - costly
- *Paravirtualization:*
  - virtualized entity is a bit different from the real entity
    - so as to enhance scalability, performance, and simplicity
    - it is probably aware that it's not running on a real machine

Pure Processor Virtualization Requirements

- Can all processors be virtualized?
- Virtualizing the processor requires:
  1) multiplexing the real processor among the virtual machines
     - relatively straightforward
  2) making each virtual machine behaves just like a real machine
     - all instructions must work identically
     - generation of and response to traps and interrupts must be identical as well

Pure Processor Virtualization Requirements

- **Processor in the virtual machine is the real processor**
  - **instructions are executed (and not interpreted or emulated)**
  - **traps are generated just as they are on real machines**
    - **in a real machine, trap handler is indexed by the trap number into a hardware-mandated jump table**
    - **the VMM needs to find the address of the virtual machine's trap handler in the table and transfer control to it**
    - **interrupts pretty much work the same way**

- **Pretty much everything can be worked out except for one problem**
  - **if a virtual machine is *executing* in the virtual privileged mode, what's to prevent it from changing things like memory-mapping *(which can affect the execution the virtual machine)?***
    - **or what if "return from interrupt" is *not* privileged?**
      - **this may not be a realistic example because, clearly, "return from interrupt" must be privileged**
    - **such instructions must be identified and make sure that they work properly under virtualization**

Pure Processor Virtualization Requirements

- Under virtualization, we must distinguish between *sensitive instructions* and *privileged instructions*
- *Privileged instructions:*
  - cause privileged-instruction *trap* when executed in *user mode* but *execute fully* when the processor is in *privileged mode*
- *Sensitive Instructions:*
  - ?

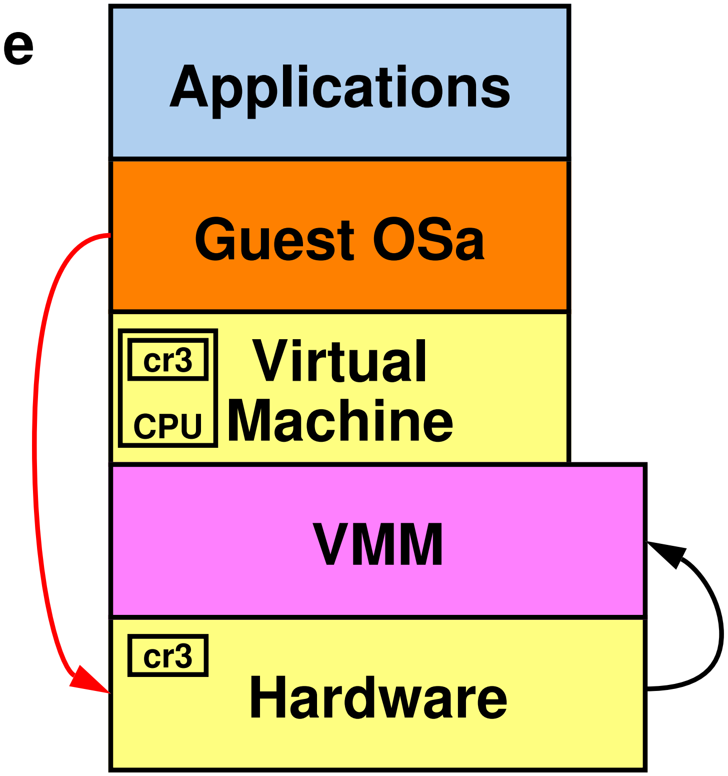
Pure Processor Virtualization Requirements

- ***Sensitive Instructions:***
  - ***Control-sensitive instructions:***
    1) instructions that affect allocation of (real) system resources
       - such as insturctions that change the mapping of virtual to real memory
  - ***Behavior-sensitive instructions:***
    2) instructions whose effect depends on the allocation of (1)
       - such as insturctions that returns the real address of a location in virtual memory
    3) instructions whose effect depends on the current processor mode
       - such as x86's `popf` insturctions that sets a set of processor flags when run in privileged mode, but set a different set of flags otherwise

Sensitive Instruction Example

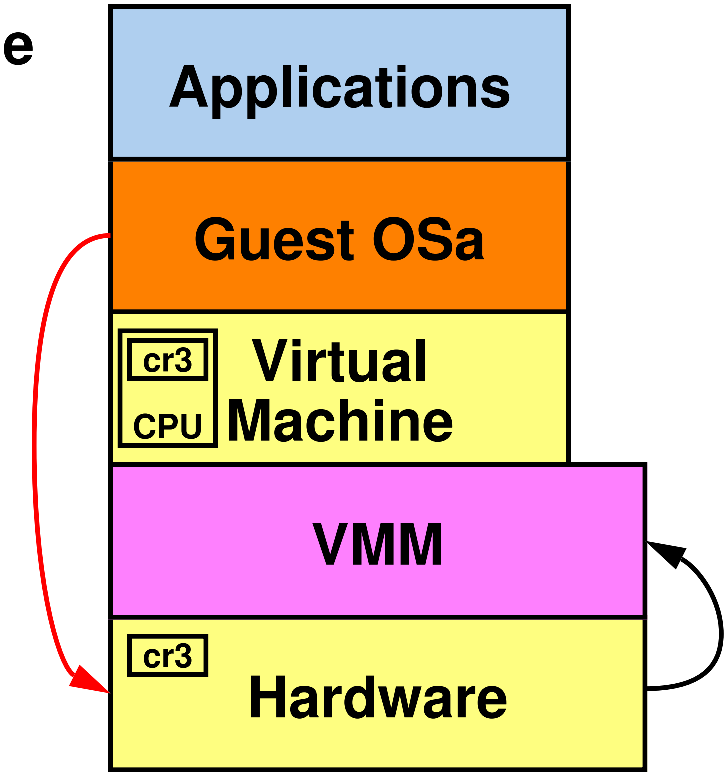
- A *sensitive* instruction must execute in the privilege mode *(in the kernel)*
  - e.g., insturctions that change the mapping of virtual to real memory

Guest OS runs in the *user* mode of the *real* processor
- executing a *sensitive* instruction will cause a trap into the VMM
- must *not* execute such instruction
- the VMM *emulates* the instruction



Sensitive Instruction Example

- A ***sensitive*** instruction must execute in the privilege mode *(in the **kernel**)*
  - e.g., insturctions that change the mapping of virtual to real memory



- All ***sensitive*** instructions must also be ***privileged***
  - but what if it's not?
    - you ***cannot*** build a virtual machine for this processor
  - this gives us another definition of *"**sensitive instruction**"*
    - it's an instruction that if it's not privileged, it will cause the guest OS (inside a virtual machine) to execute ***incorrectly***
      - this ***operational definition*** may be more useful for an introductory class like ours

Pure Processor Virtualization Requirements

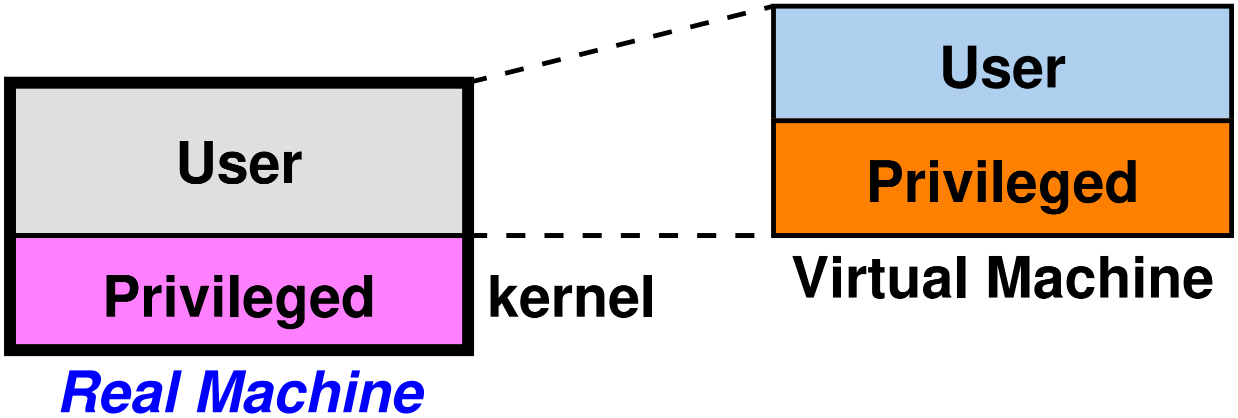
- [Popek and Goldberg, 1974] *proved* that the *sufficient condition* to be able to construct a virtual machine is simply the following:
  - a computer's set of *sensitive instructions* is a *subset* of its *privileged instructions*
  - i.e., if you execute a sensitive insturction in user mode, you will trap into the kernel
    - more importantly, if you execute a sensitive insturction in *virtual user or virtual privileged mode*, you will trap into *VMM*

- The above theorem holds for the IBM 360
  - virtual machines can be constructed for it
- The above theorem does *not* hold for the x86 processors
  - cannot build virtual machines for x86

The (Real) 360 Architecture

- Two execution modes
  - supervisor and problem (user)
  - all sensitive instructions are privileged instructions
- Memory is protectable: 2KB 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

Actions on Real 360

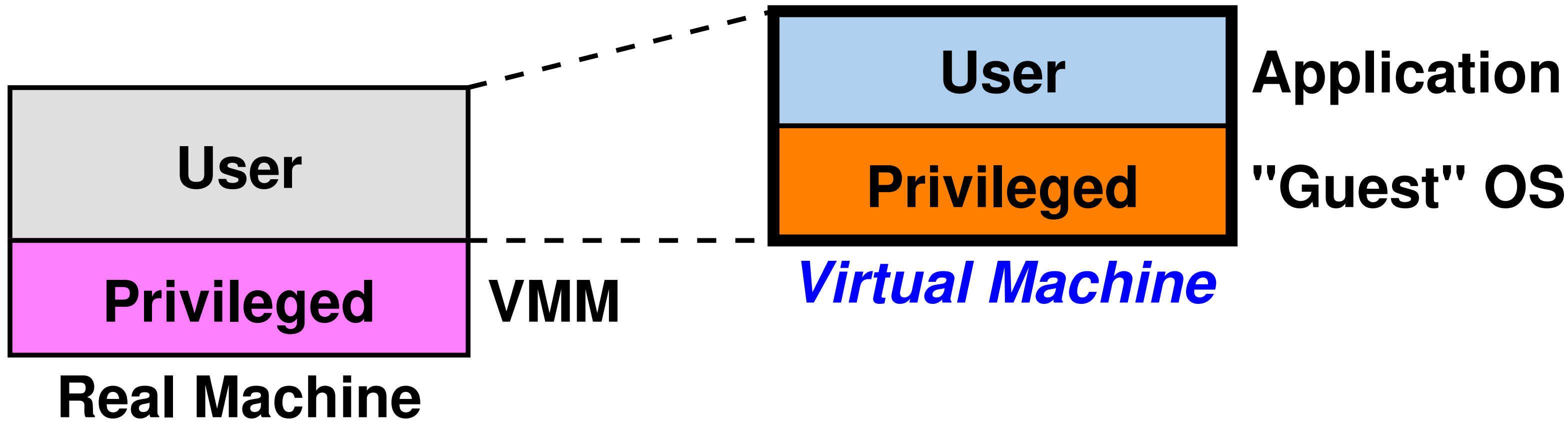


	User Mode	Privileged Mode
non-sensitive instruction	executes fine	executes fine
"errant" instruction	traps to kernel	traps to kernel
sensitive instruction	traps to kernel	executes fine

such as divide by zero, page fault, etc.

since all sensitive instructions are privileged for IBM 360

Actions on Virtual 360



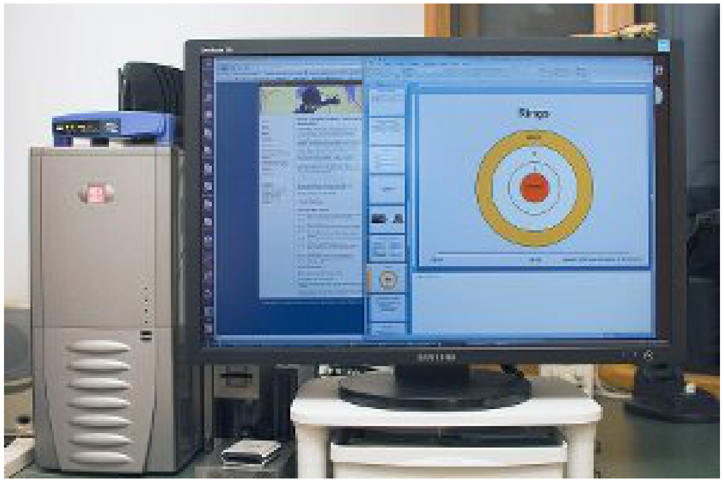
	Virtual User Mode	Virtual Privileged Mode
non-sensitive instruction	executes fine	executes fine
"errant" instruction	traps to VMM; VMM delivers trap to the Guest OS	traps to VMM; VMM delivers trap to the Guest OS
sensitive instruction	traps to VMM; VMM delivers trap to the Guest OS	traps to VMM; VMM verifies and *emulates* instruction

Virtual Devices?

- Terminals
  - connecting (real) people
- Networks
  - didn't exist in the 60s
  - (how did virtual machines communicate?)
- Disk drives
  - CP67 supported "mini disks"
  - extended at Brown into "segment system"
- Interval timer
  - virtual or real?

Virtual Machines
Part 2: Now

≠



How They Are Different

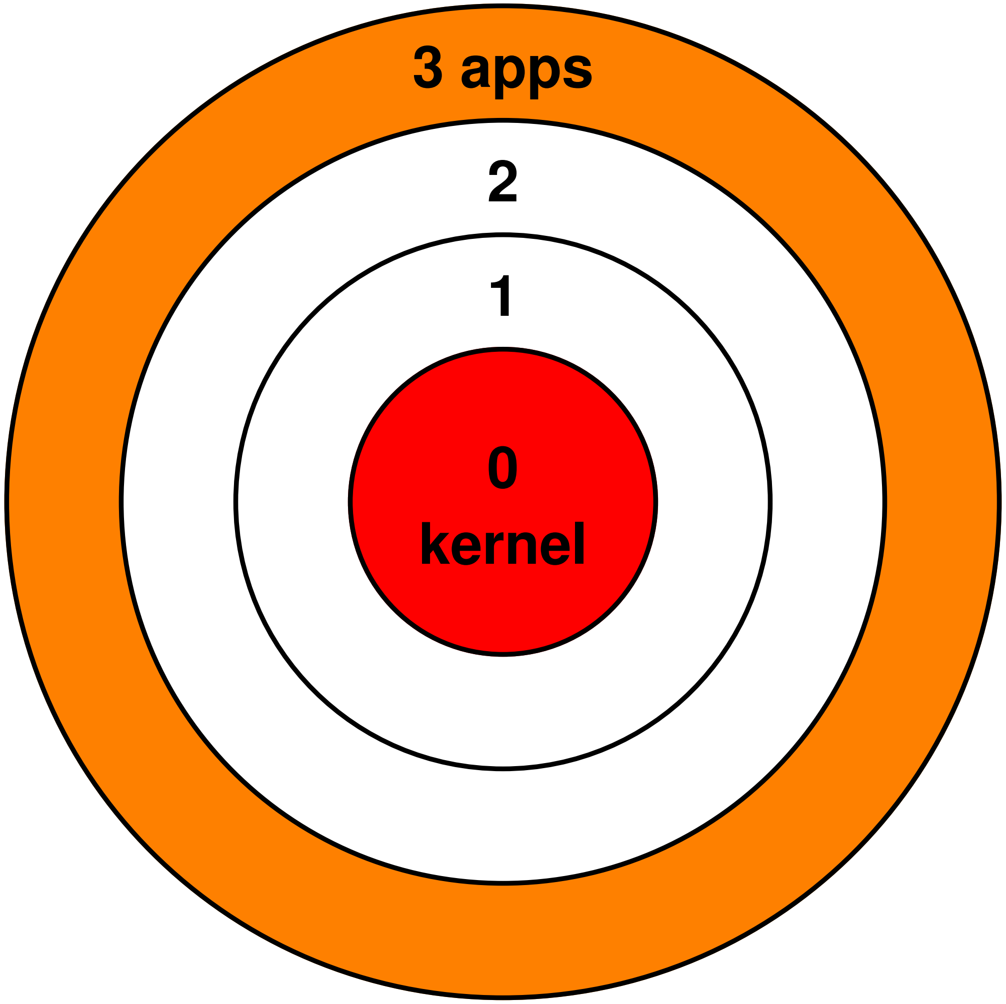
IBM 360

- 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

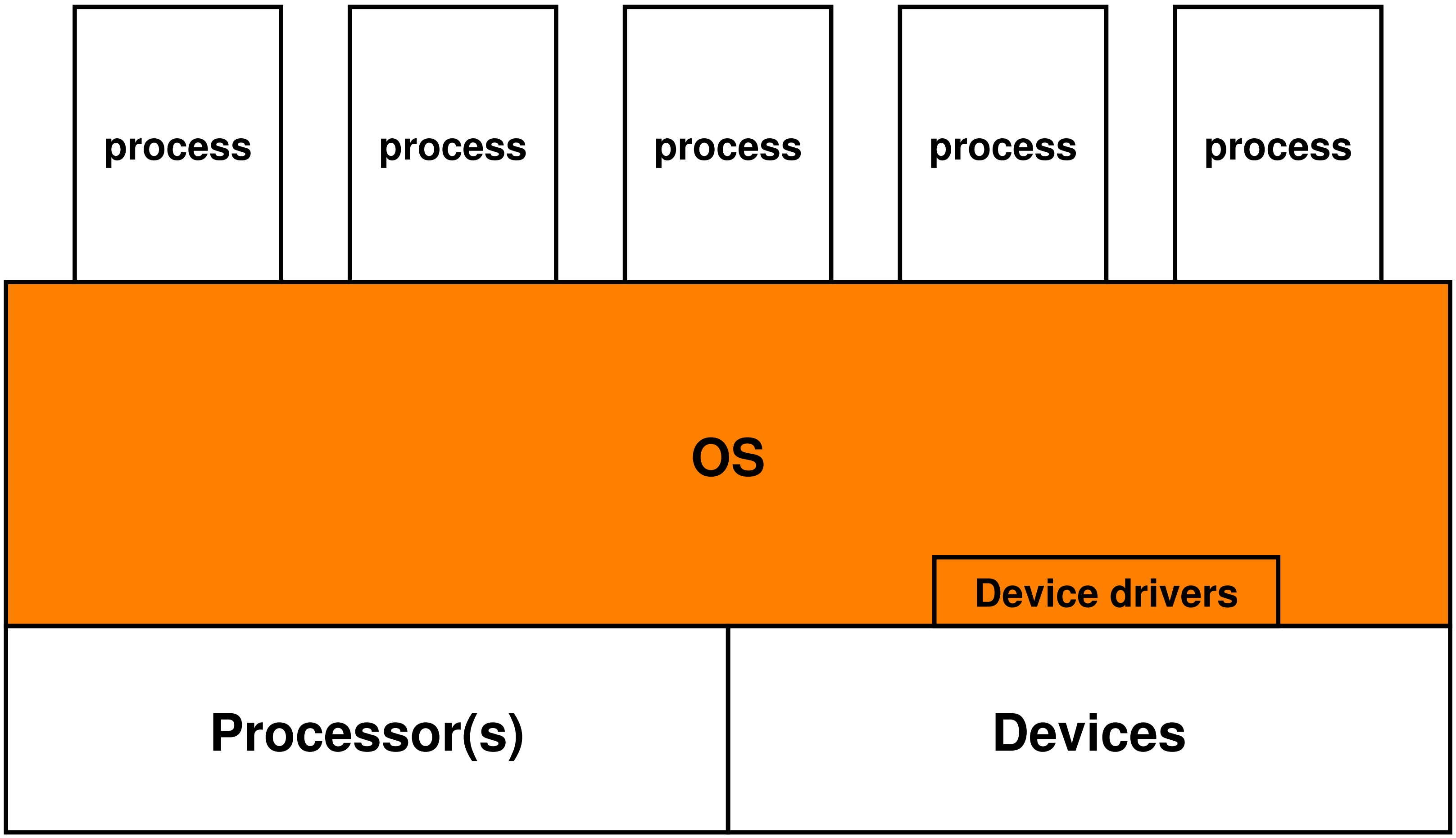
Fixing the Hardware

- 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

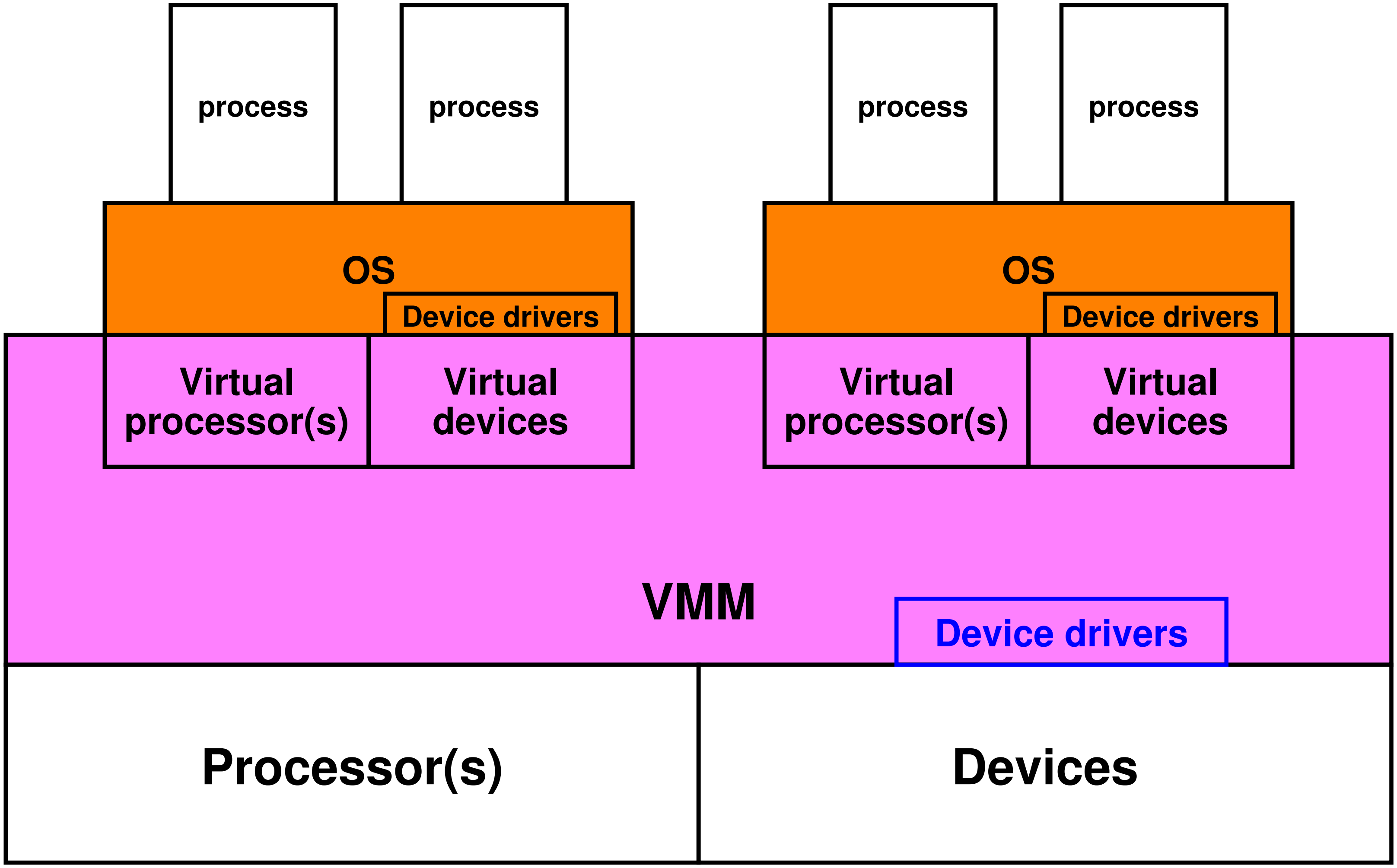
- Channel programs were generic for IBM 360
  - can be emmulated in the VMM
- I/O via memory-mapped registers is not
  - lots and lots and lots of device drivers
  - must VMM handle all of them?
    - problem: *scalability*

Real-Machine OS Structure



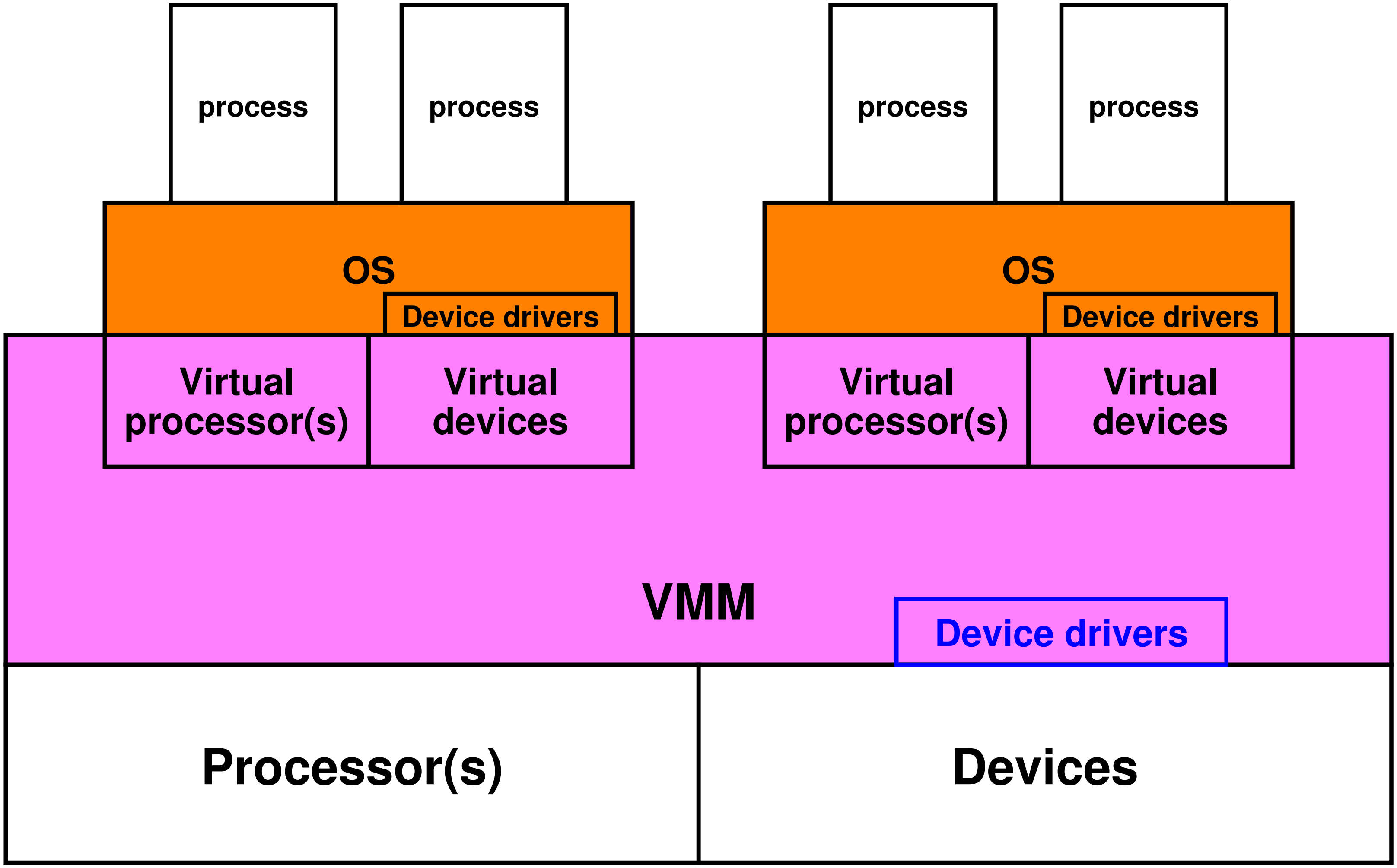
- Lots of devices need to be supported by desktop OSes (such as Windows and Mac OS X)

On a Virtual Machine ...



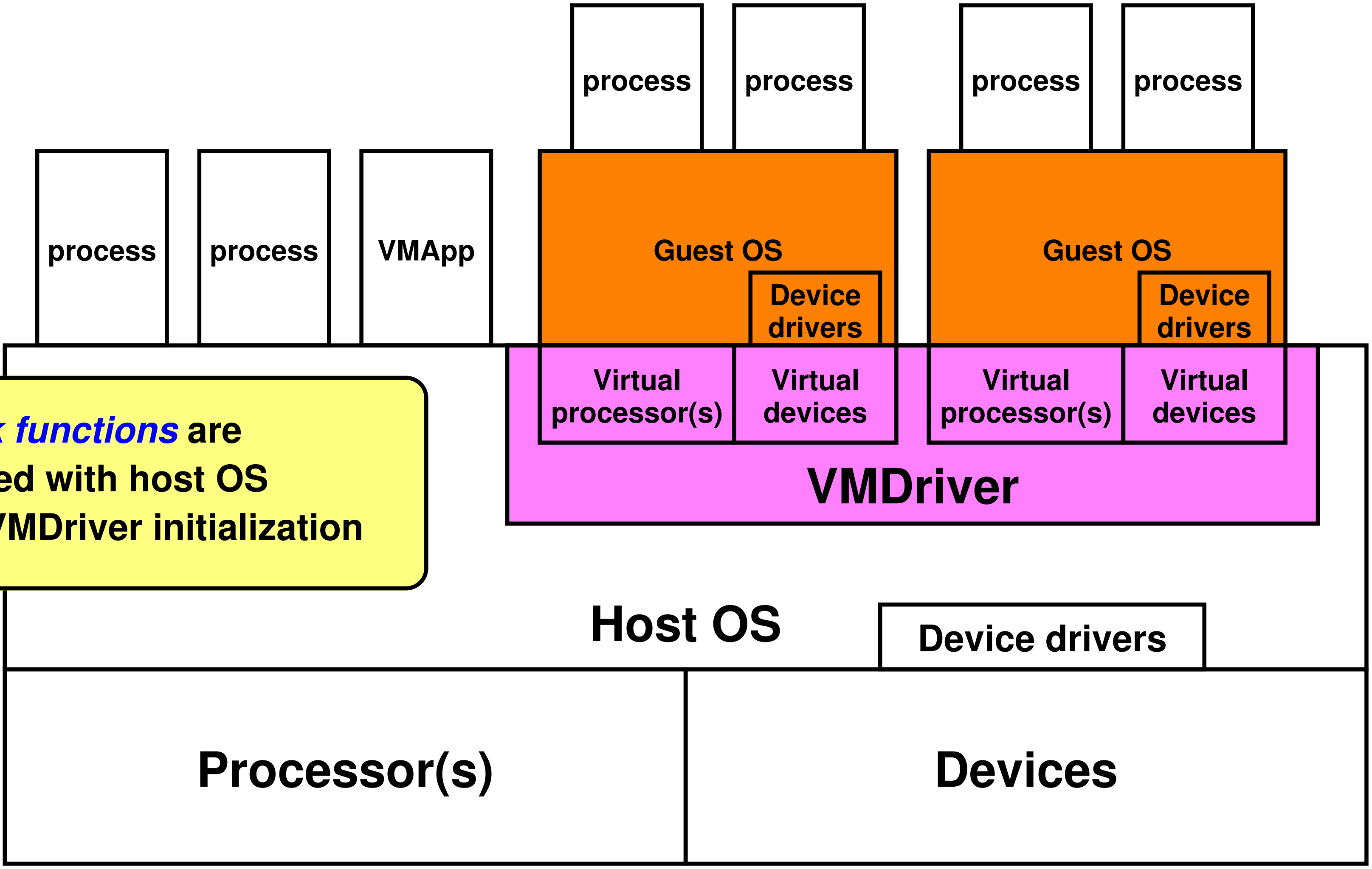
- Who is going to write all the device drives for the new OS?

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?

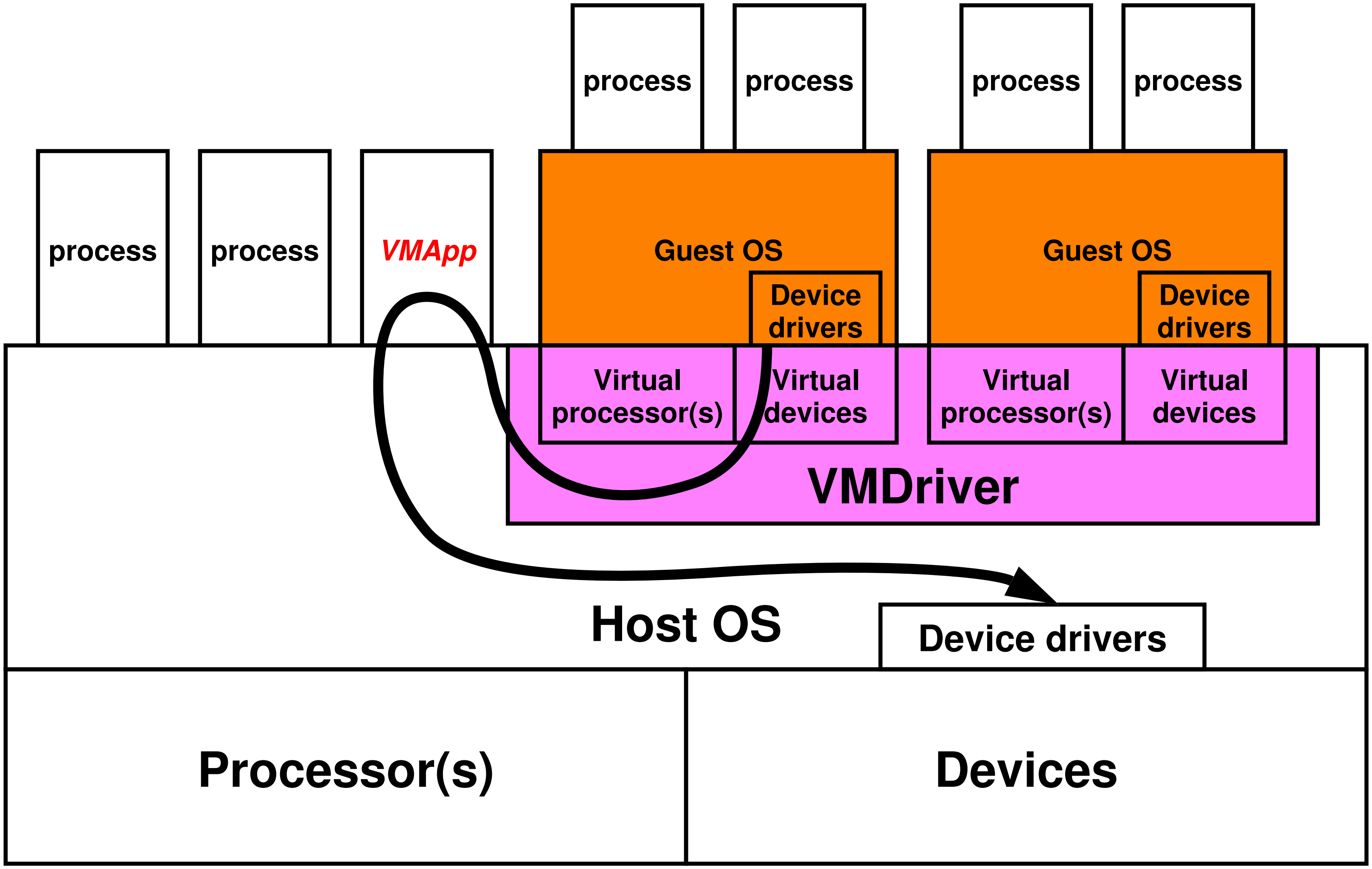
VMware Workstation - Host/Guest Model



- *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

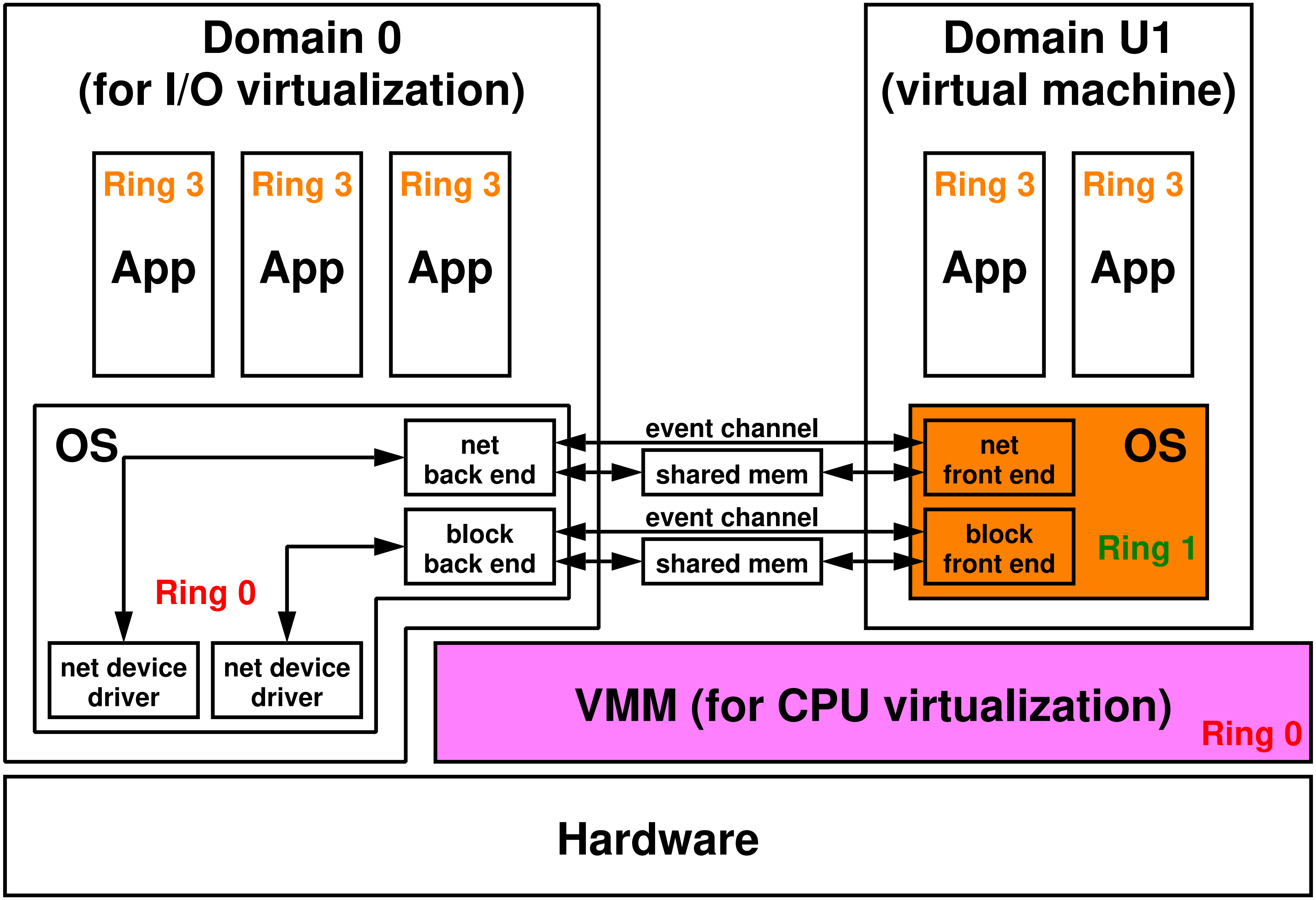


- This is more suitable for "workstations" - more variety of devices
  - *convenience* over *performance*

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

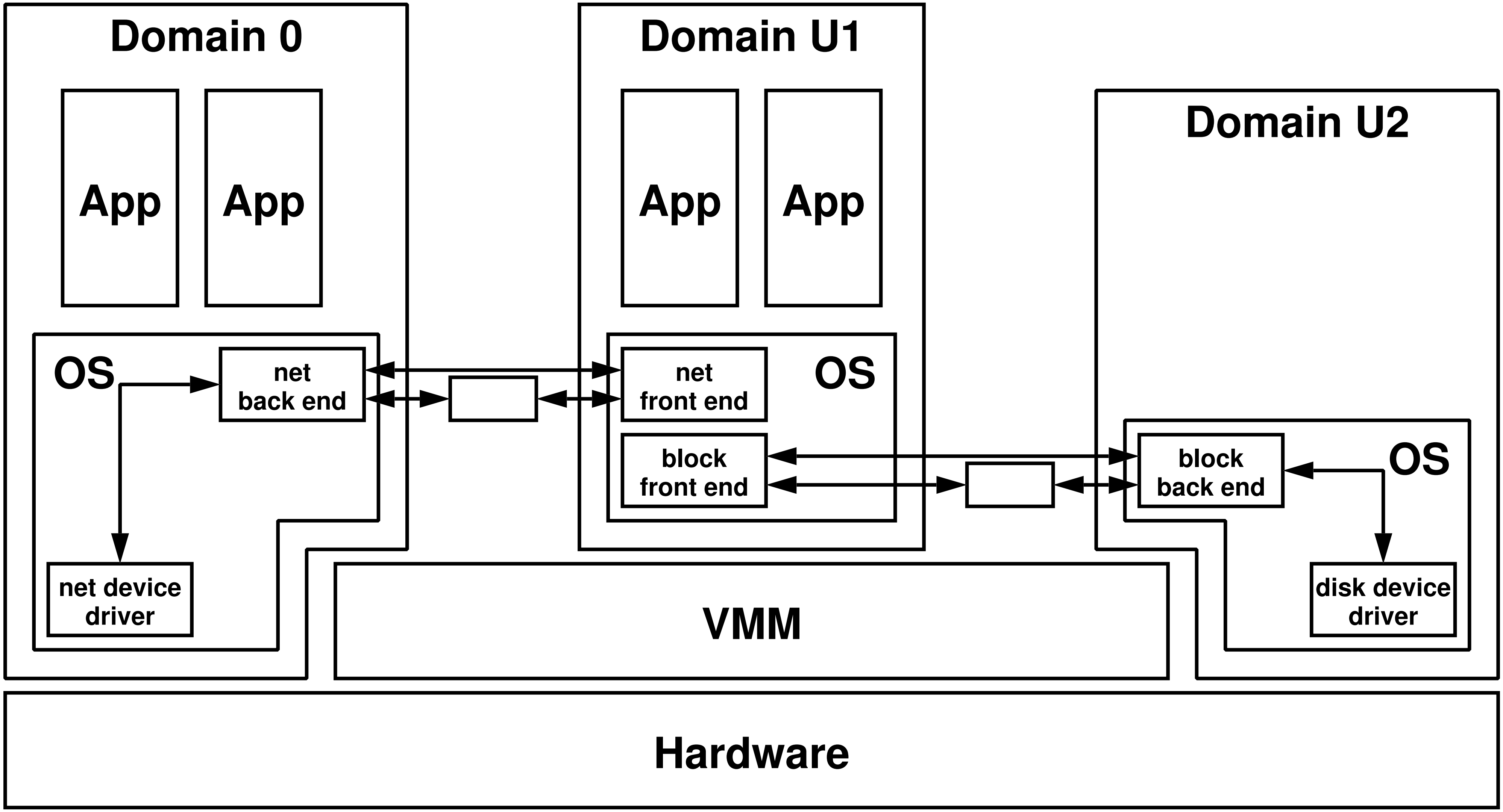


- Domain 0 OS is like the Host OS in VMware but *only for I/O*
  - it directly talks to the hardware

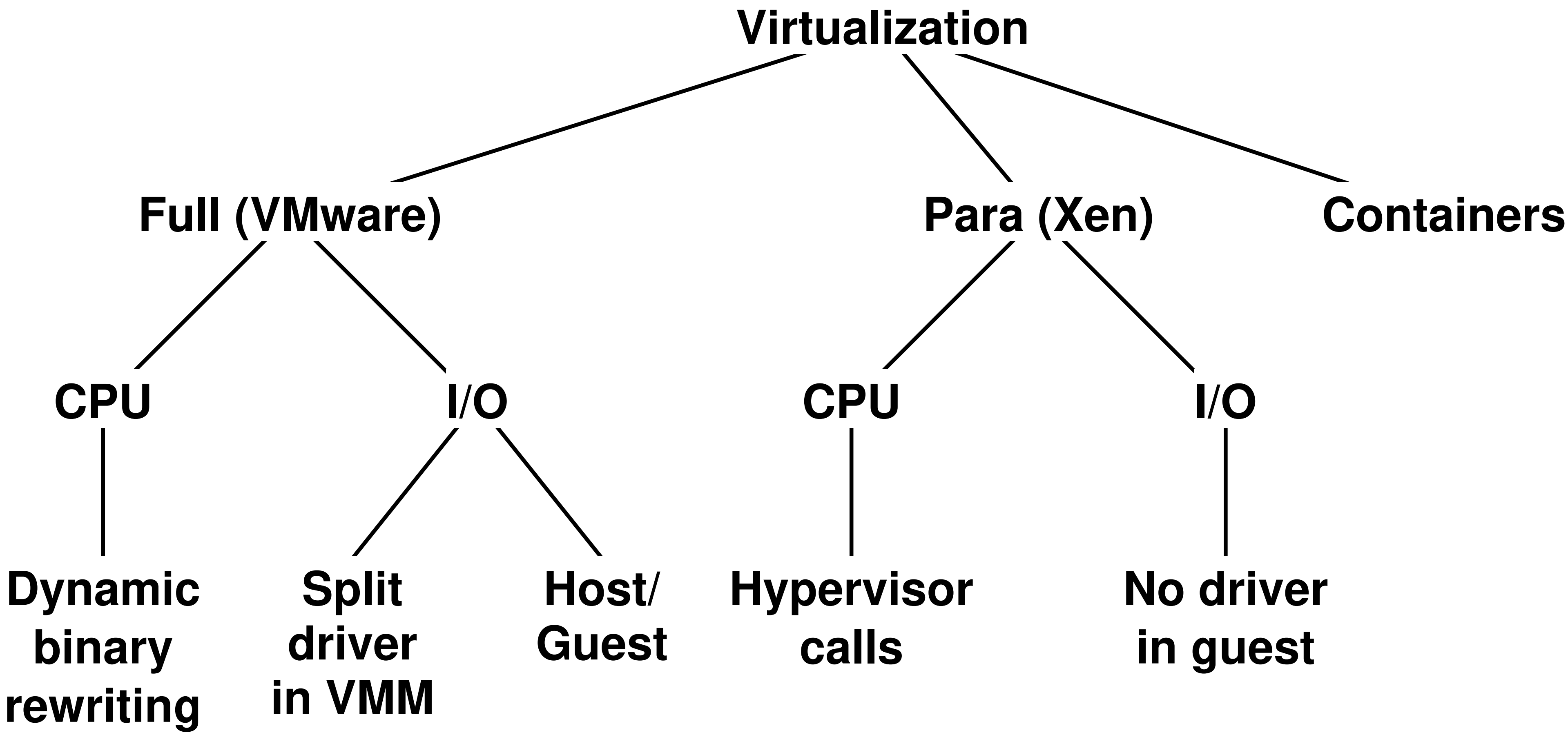
Additional Applications

- 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

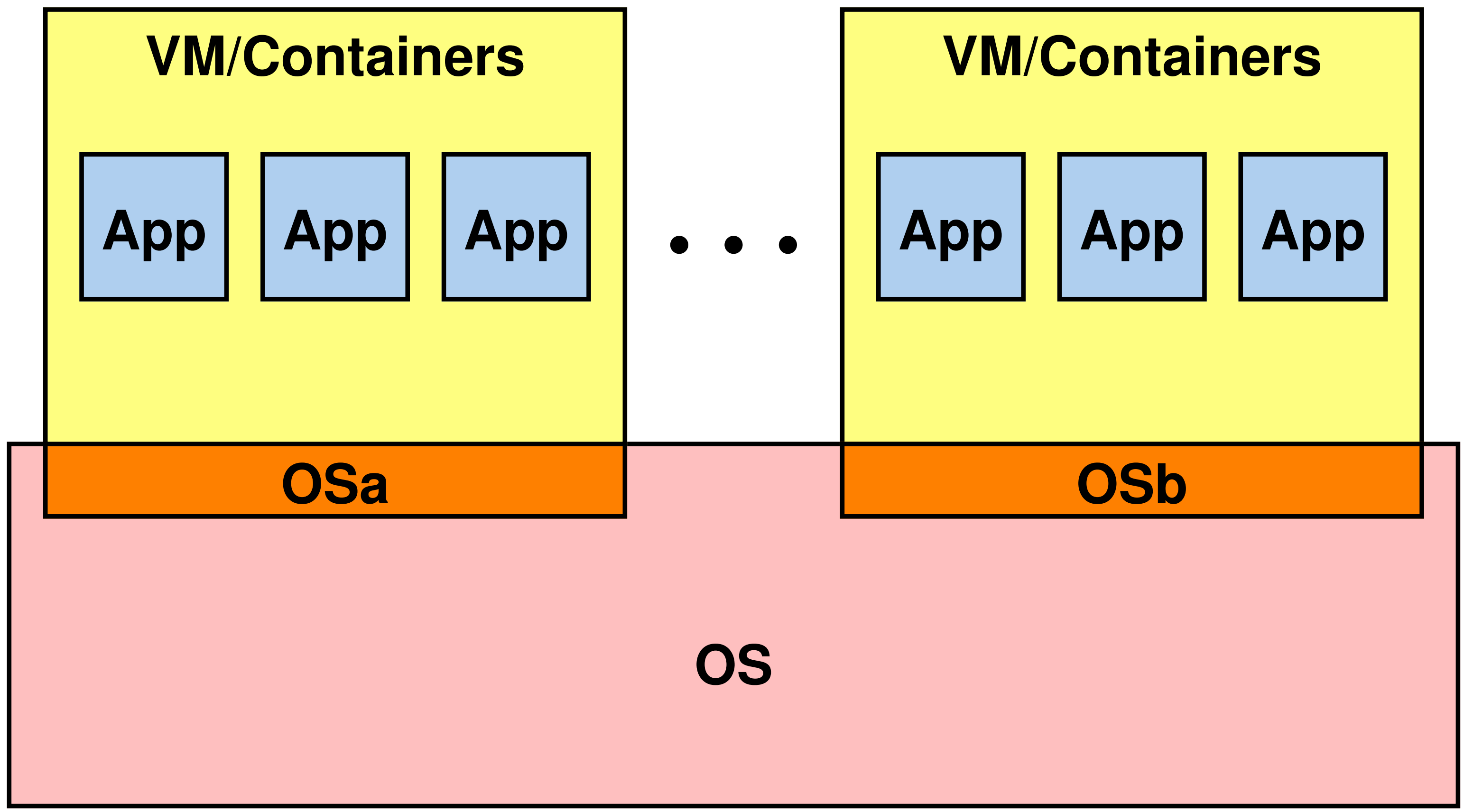


Summary


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

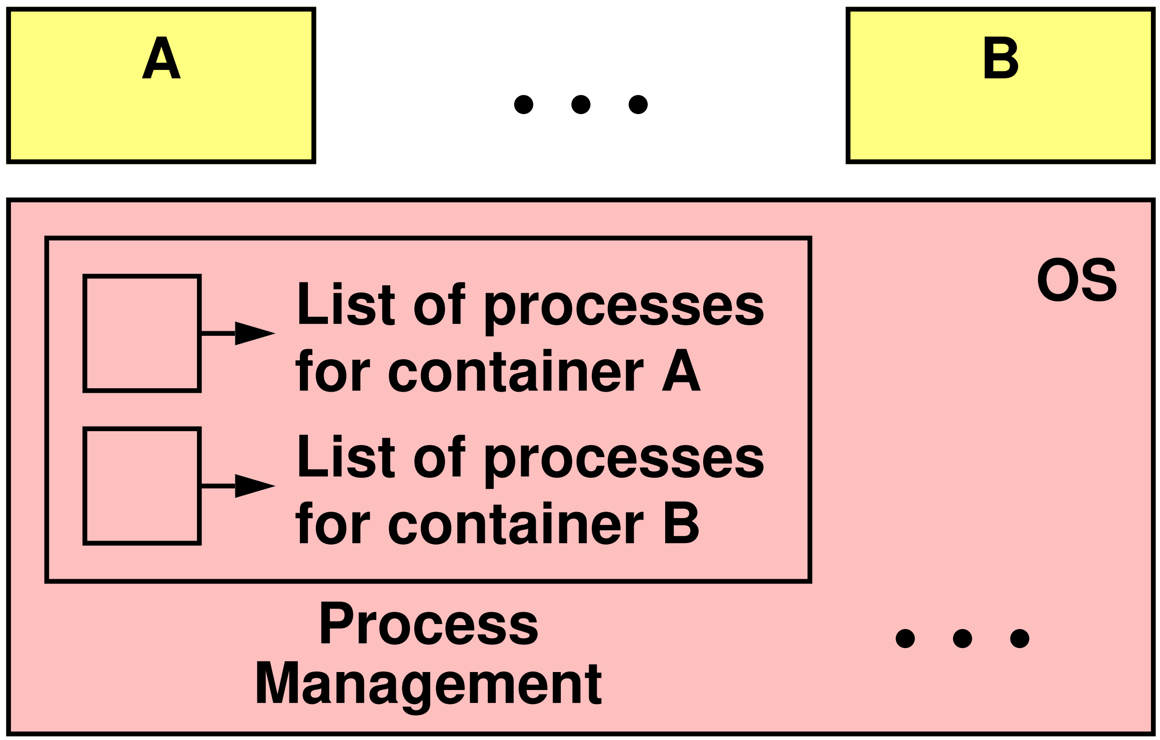
- Containerized OS (or OS Containers)
  - not covered in textbook



  - 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

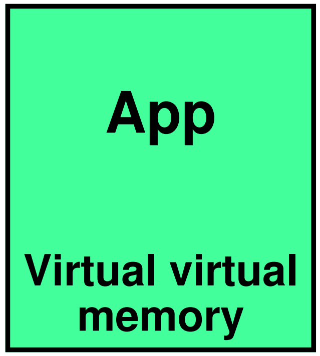
- Within the OS, the management of resources for each container is separated
  - e.g., processes for container A is kept separate from processes for container B



- 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"

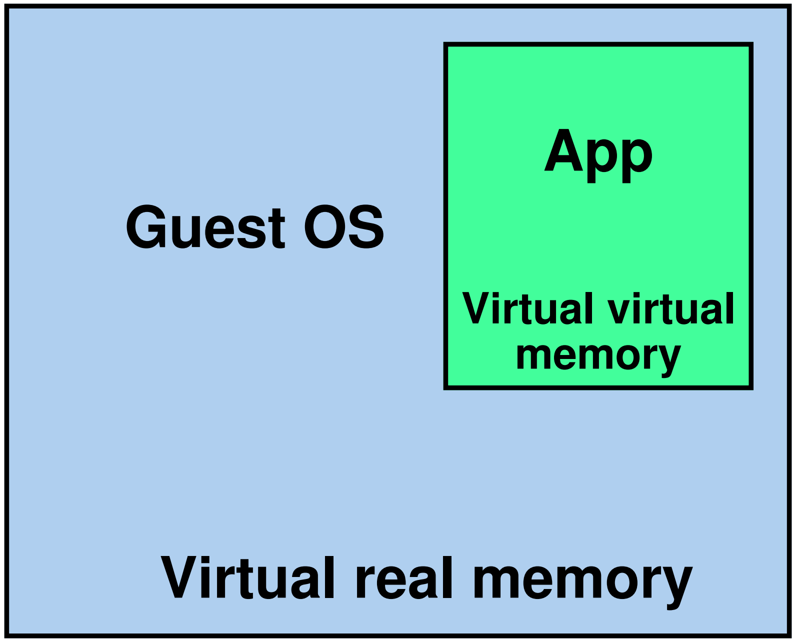
7.2.6 Virtualizing Virtual Memory

Virtual Machines Meet Virtual Memory



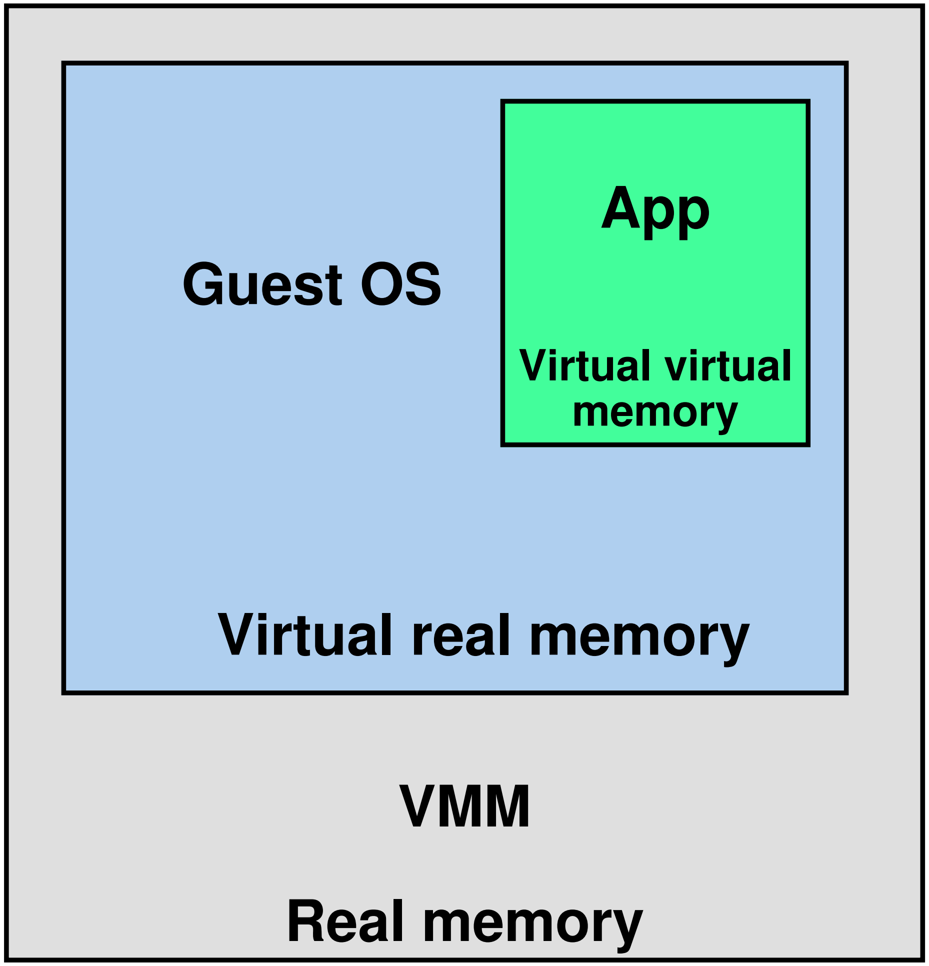
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*

Virtual Machines Meet Virtual Memory



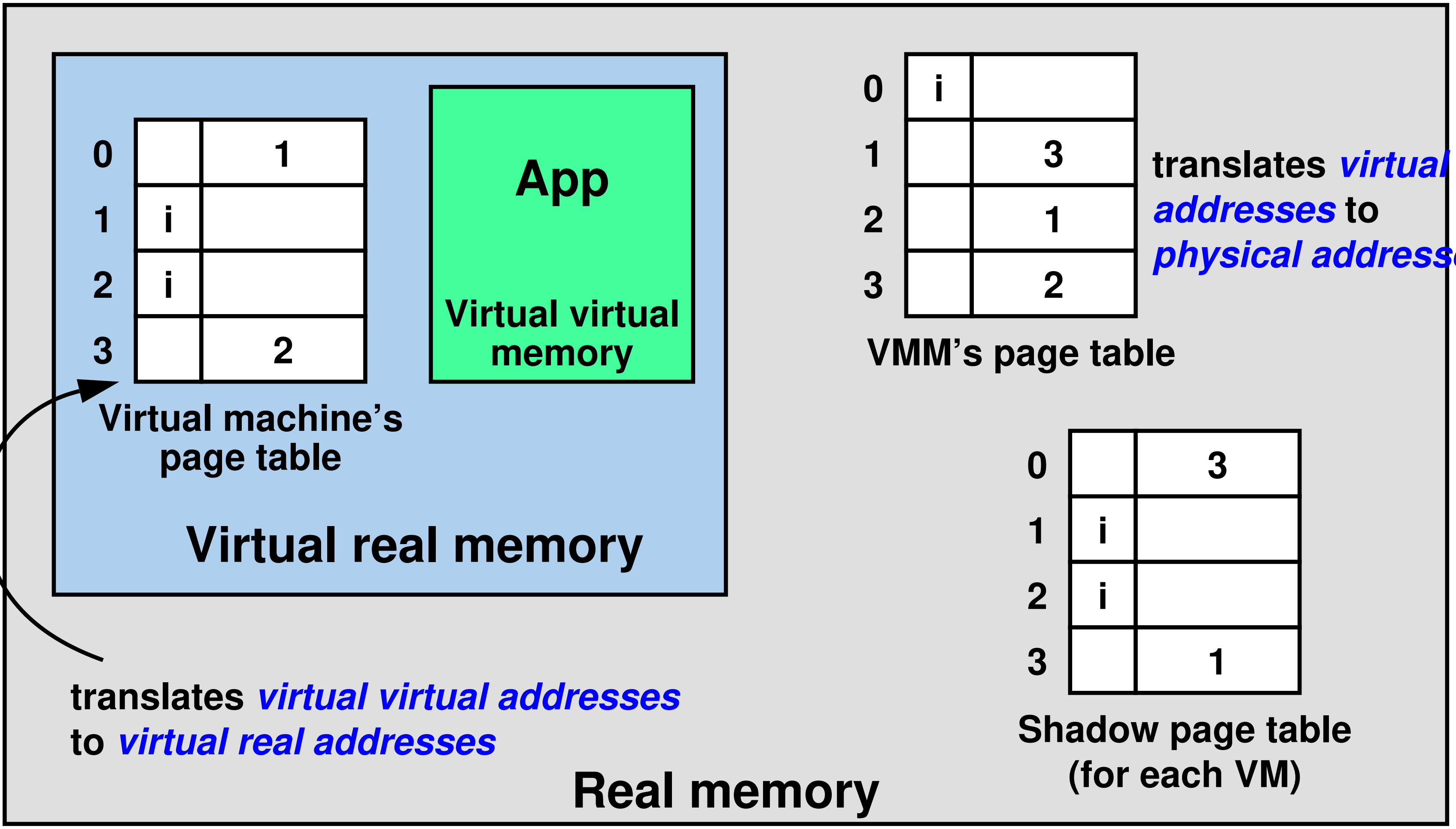
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*

Virtual Machines Meet Virtual Memory



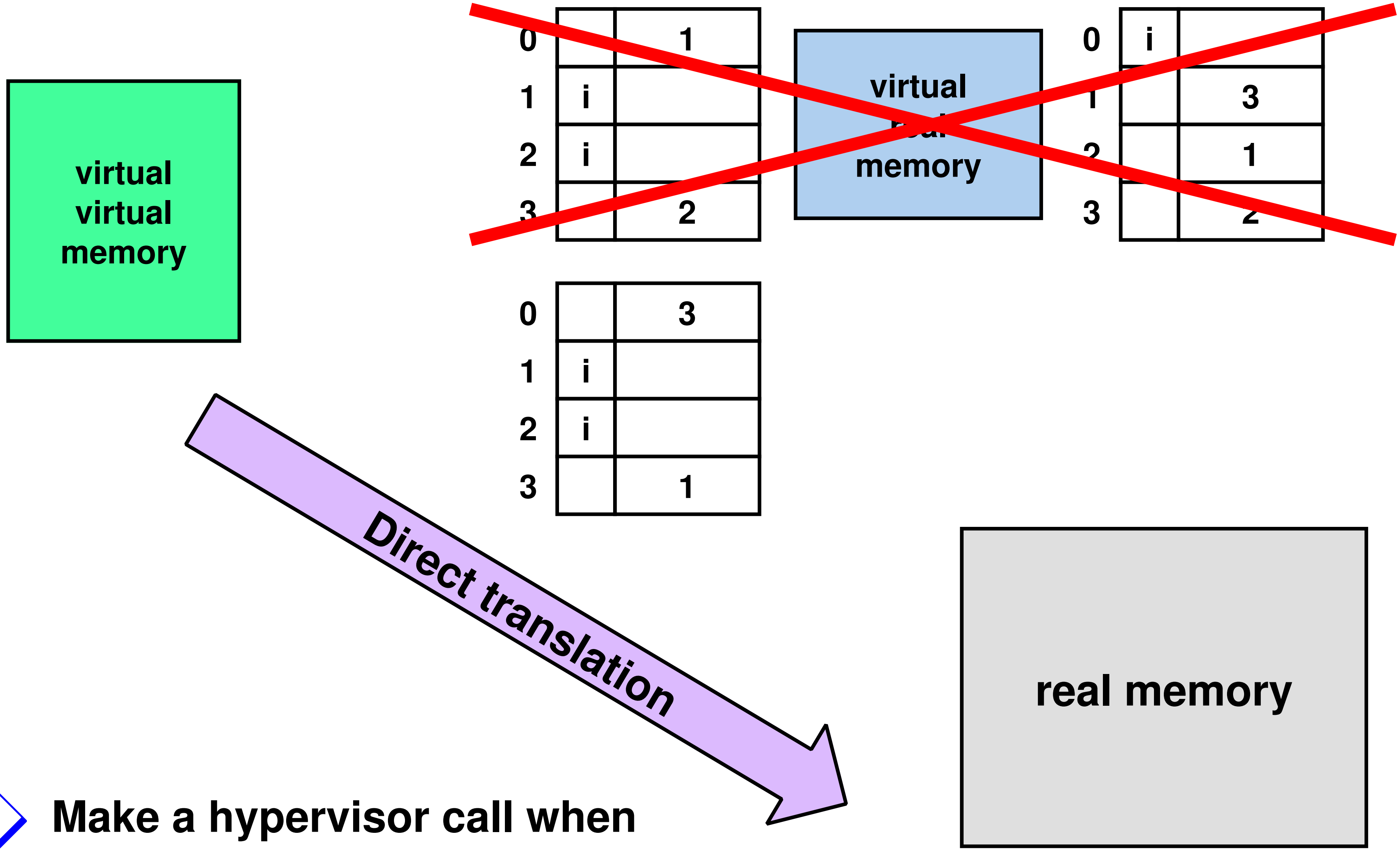
- A user process thinks it's accessing virtual memory
  - but it's really dealing with *virtual virtual memory*
- The OS in a VM thinks it's managing real memory
  - but it's really dealing with *virtual real memory*
- VMM needs to manage real memory
  - how can we *virtualize virtual memory*?

Virtual Machines Meet Virtual Memory



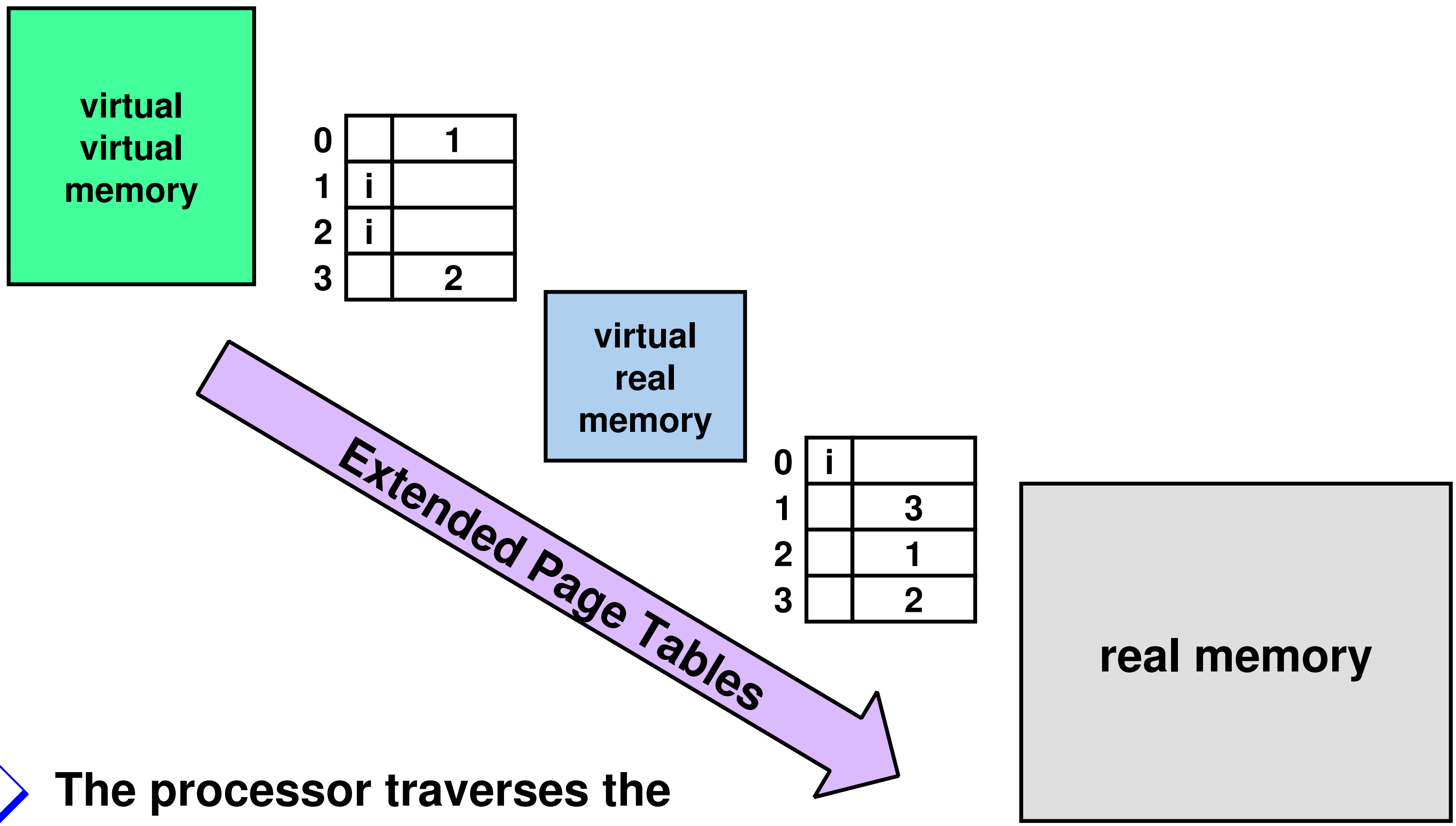
- **When a VM changes its page table, VMM must update the corresponding *Shadow Page Table***
  - **main problem: poor performance**

Solution 1: Paravirtualization to the Rescue



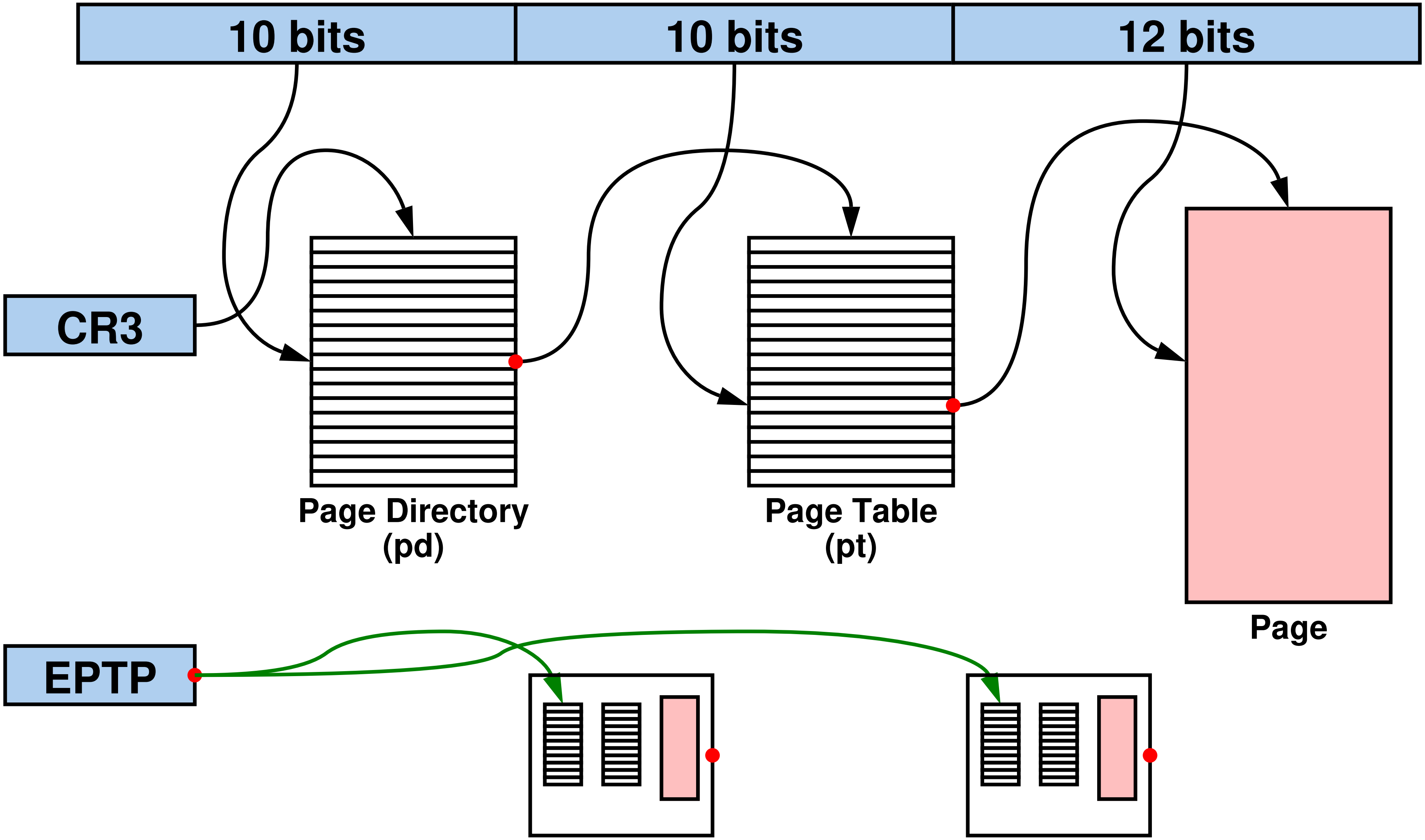
- Make a hypervisor call when page table needs to be modified
  - helps a bit, but not much faster

Solution 2: Hardware to the Rescue

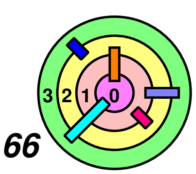


- The processor traverses the two tables in sequence and does the conversion all by itself

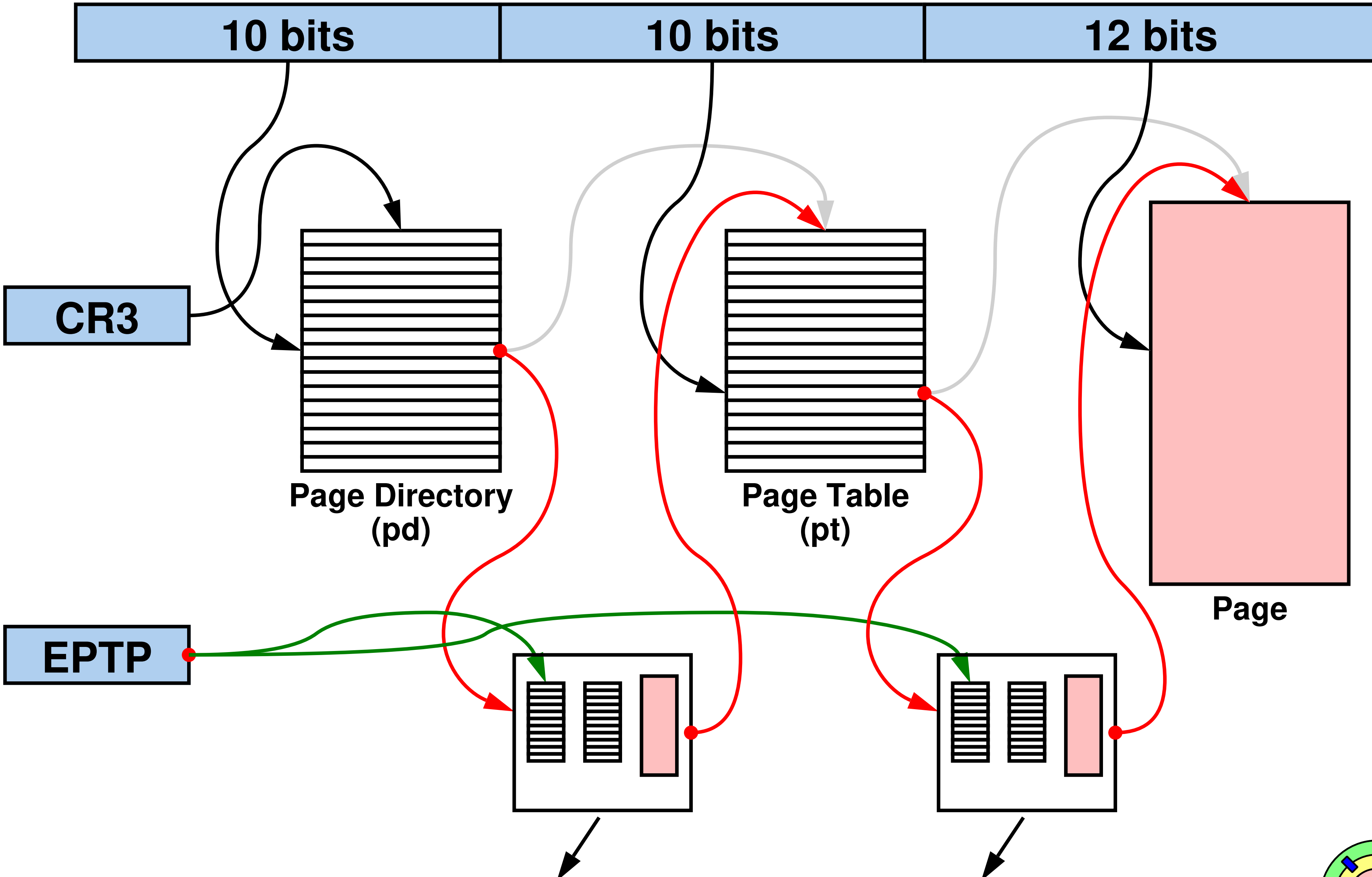
x86 Paging with EPT



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



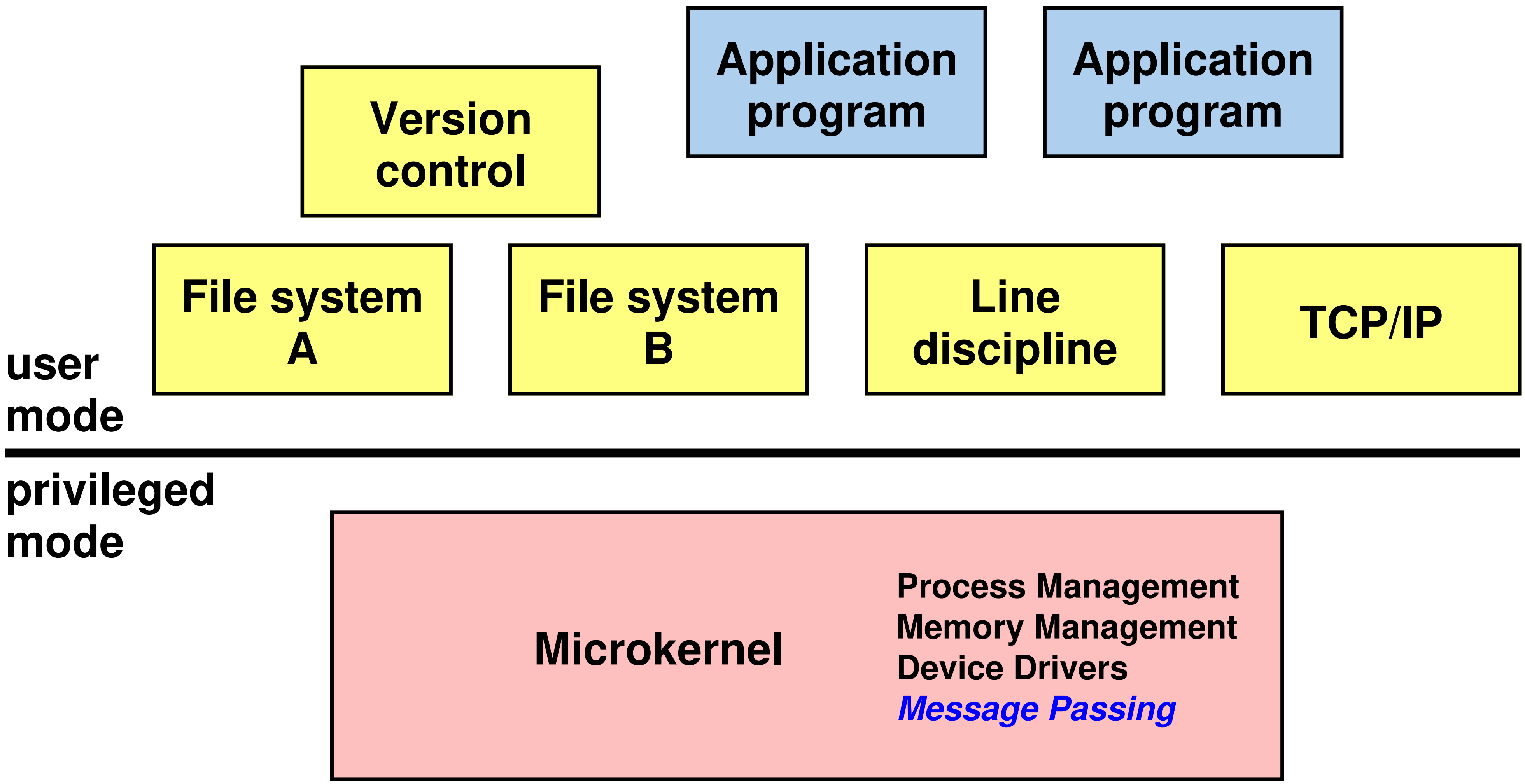
x86 Paging with EPT



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

Microkernels
(Back to Section 4.2)

OS Services as User Apps

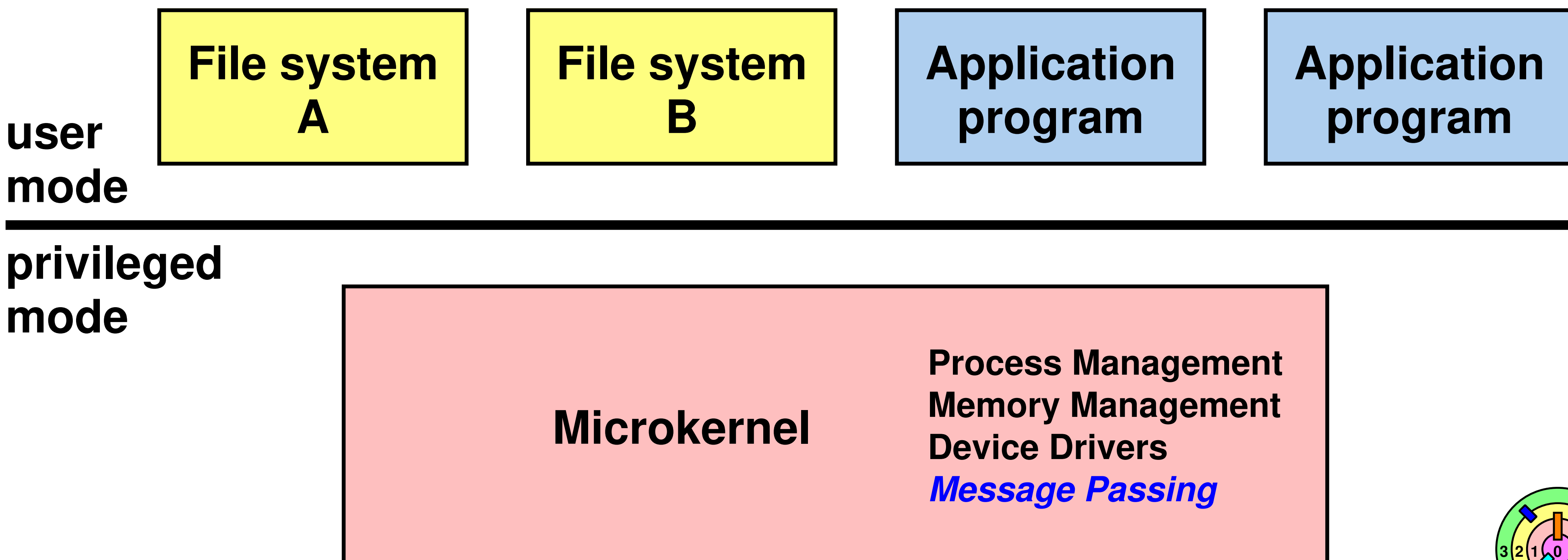


Why?

- It's cool ...
- Assume that OS coders are incompetent, malicious, or both ...
  - OS components run as protected user-level applications
- Extensibility
  - easier to add, modify, and extend user-level components than kernel components

Implementation Issues

- How are modules linked together?
  - e.g., how would you implement `read()/write()`?
    - can't use system calls any more!
  - e.g., which file system supports `read()/write()`?
- How is data moved around efficiently?

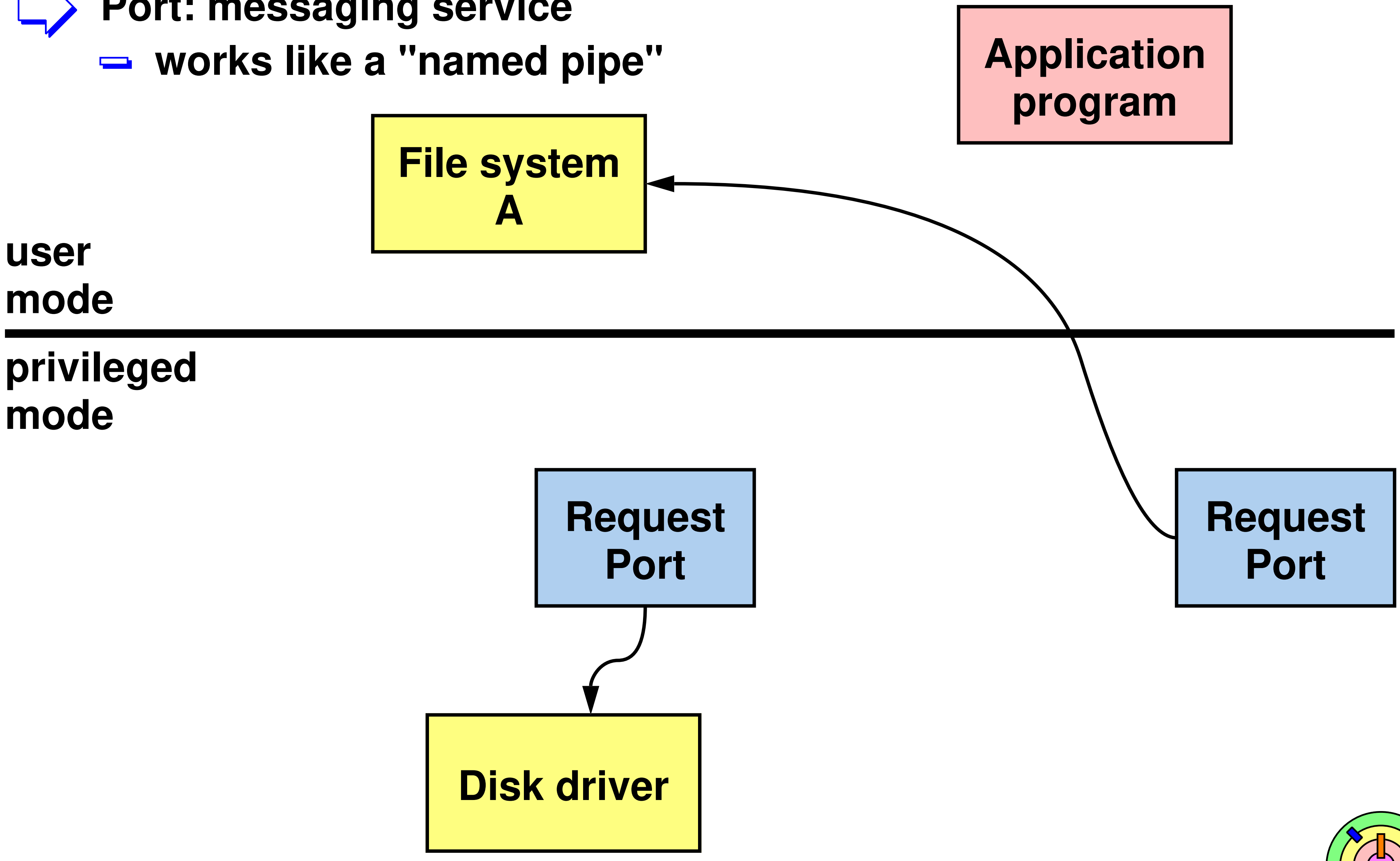


Mach

- Developed at CMU, then Utah
- Early versions shared kernel with Unix
  - basis of NeXT OS
- Later versions still shared kernel with Unix
  - basis of OSF/1
    - basis of Mac OS X
- Even later versions actually functioned as working microkernel
  - basis of GNU/HURD project
    - HURD: HIRD of Unix-replacing daemons
    - HIRD: HURD of interfaces representing depth

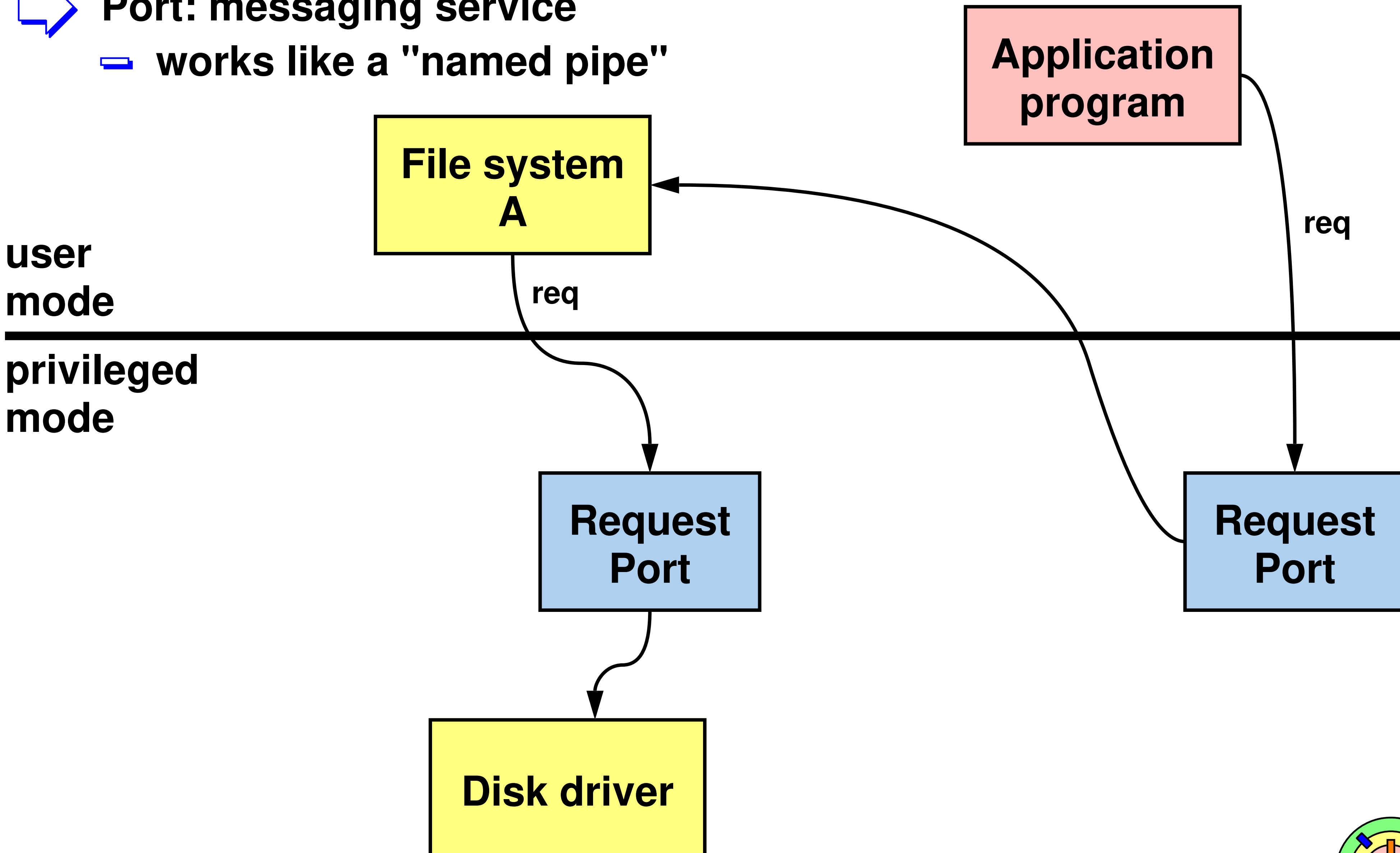
Example

- Port: messaging service
  - works like a "named pipe"



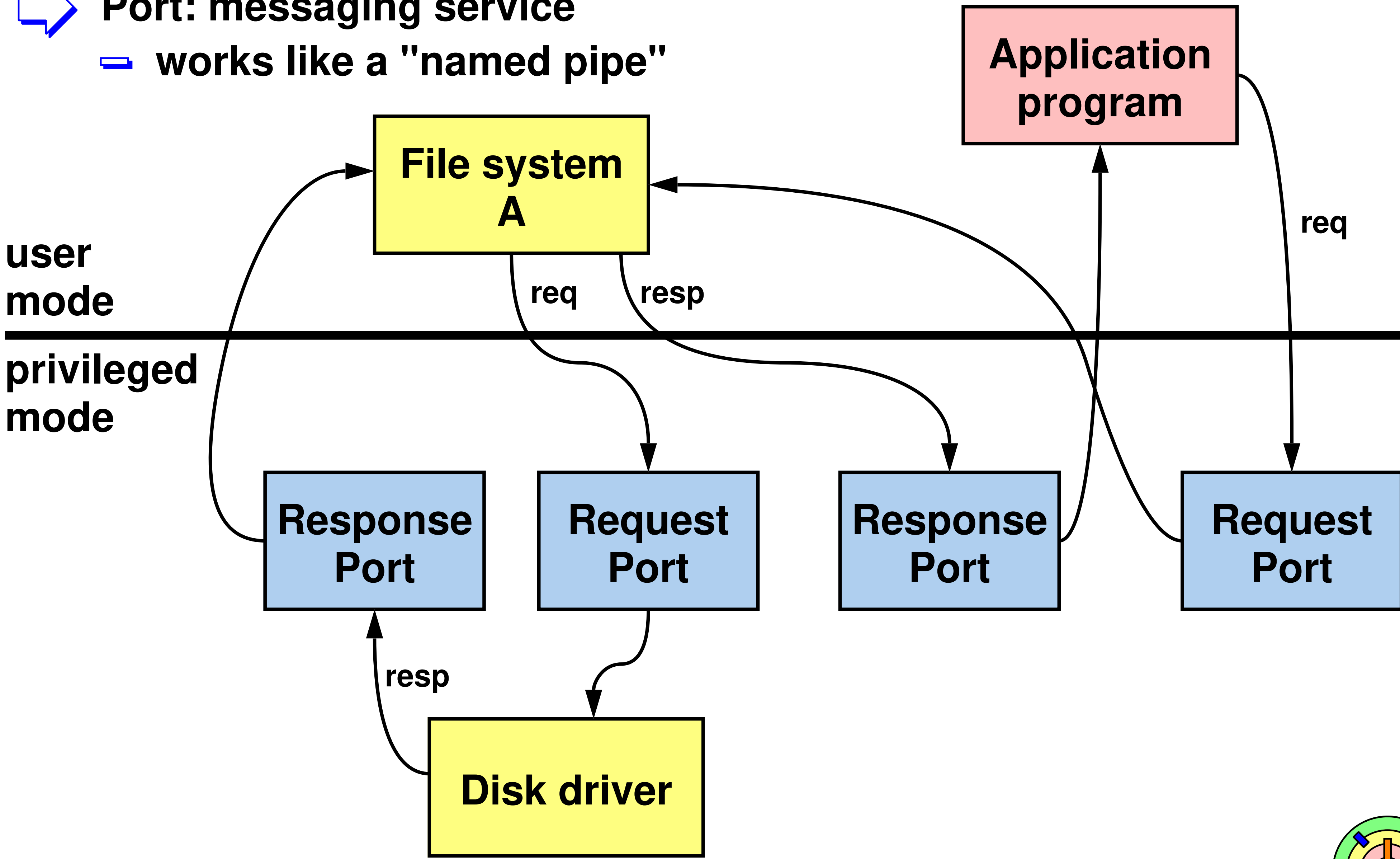
Example

- Port: messaging service
  - works like a "named pipe"



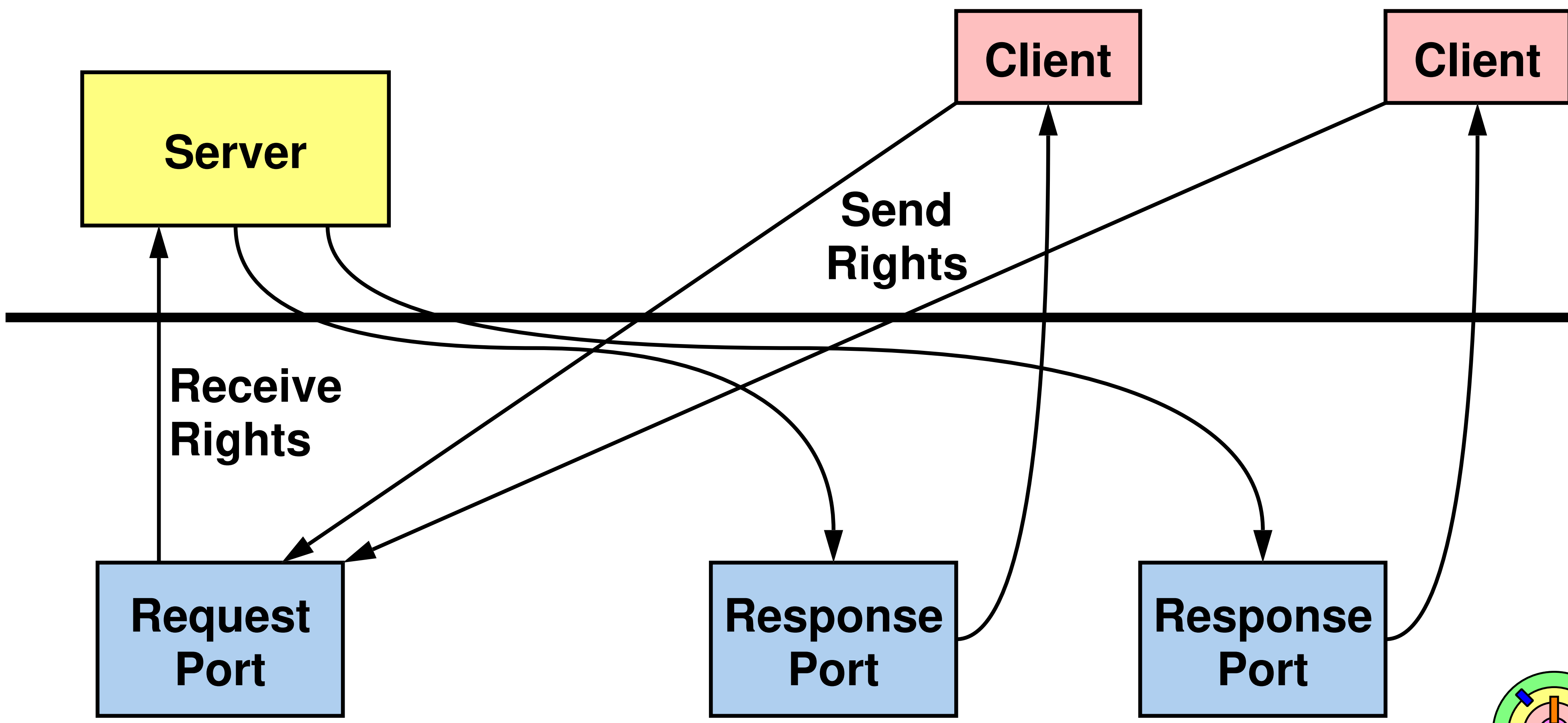
Example

- Port: messaging service
  - works like a "named pipe"



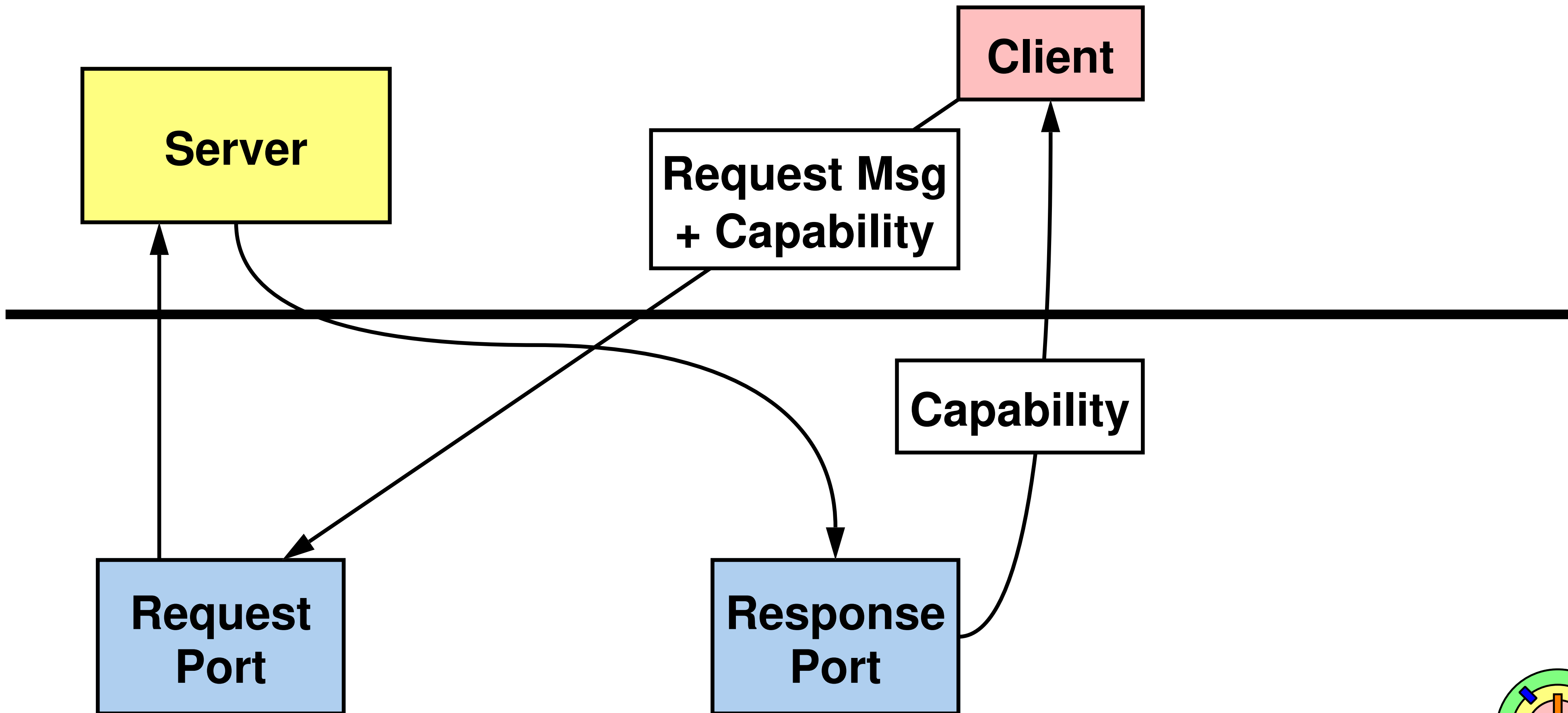
Mach Ports Permissions

- Linkage construct



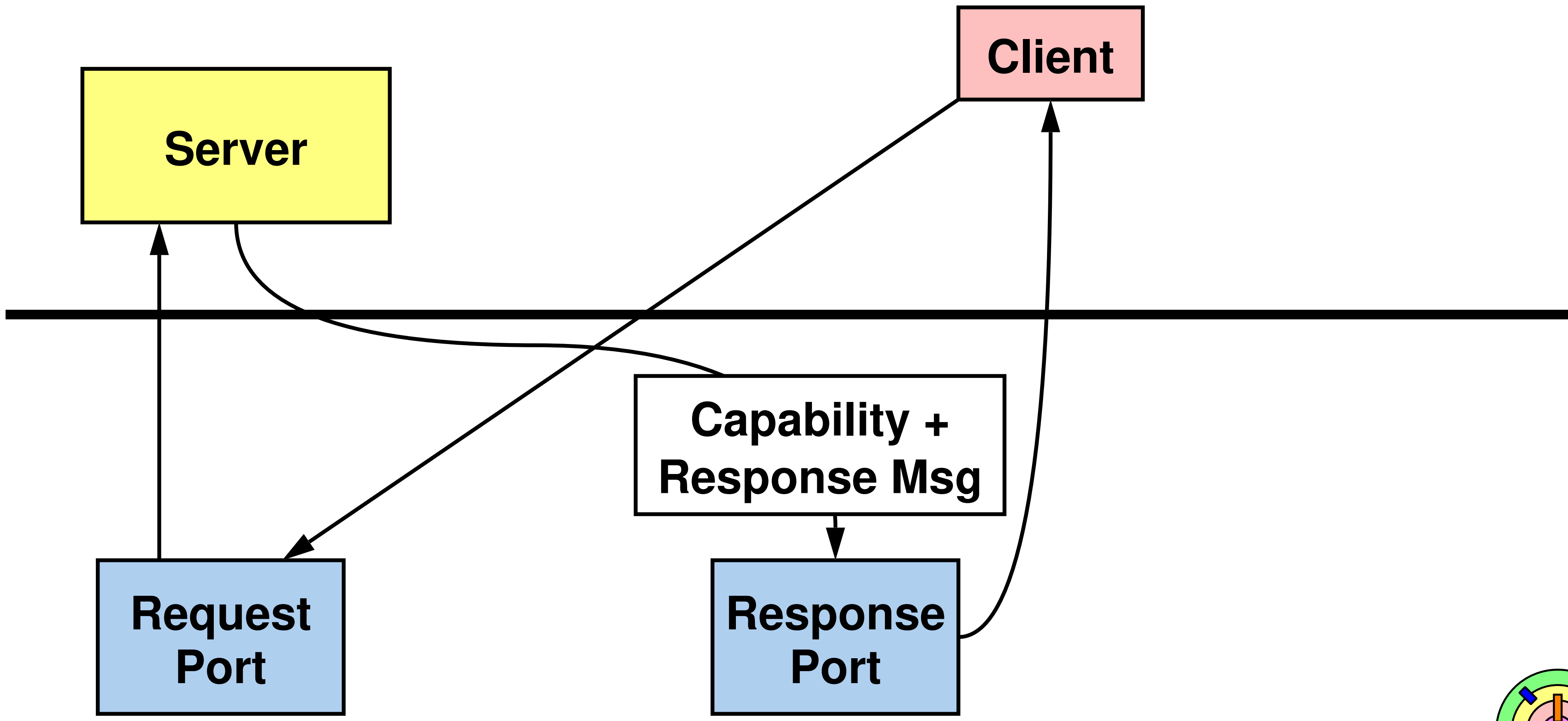
Mach Ports Permissions

- Communication construct
  - client create *response port* and *capability* (like a key) to send data through it
    - include capability in the request message to server



Mach Ports Permissions

- Communication construct
  - client create *response port* and *capability* (like a key) to send data through it
    - include capability in the request message to server



RPC

- Ports used to implement *remote procedure calls*
  - communication across process boundaries
  - if procedures are on same machine ...
    - local RPC

Successful Microkernel Systems

- ?
- ?
- ...

Attempts

- Windows NT 3.1
  - graphics subsystem ran as user-level process
  - moved to kernel in 4.0 for performance reasons
- Mac OS X
  - based on Mach
  - all services in kernel for performance reasons
- HURD
  - based on Mach
  - services implemented as user processes
  - no one uses it, for performance reasons ...