CSE 120
Principles of Operating Systems

Spring 2023

Lecture 17: Virtual Machines

Amy Ousterhout

Administrivia

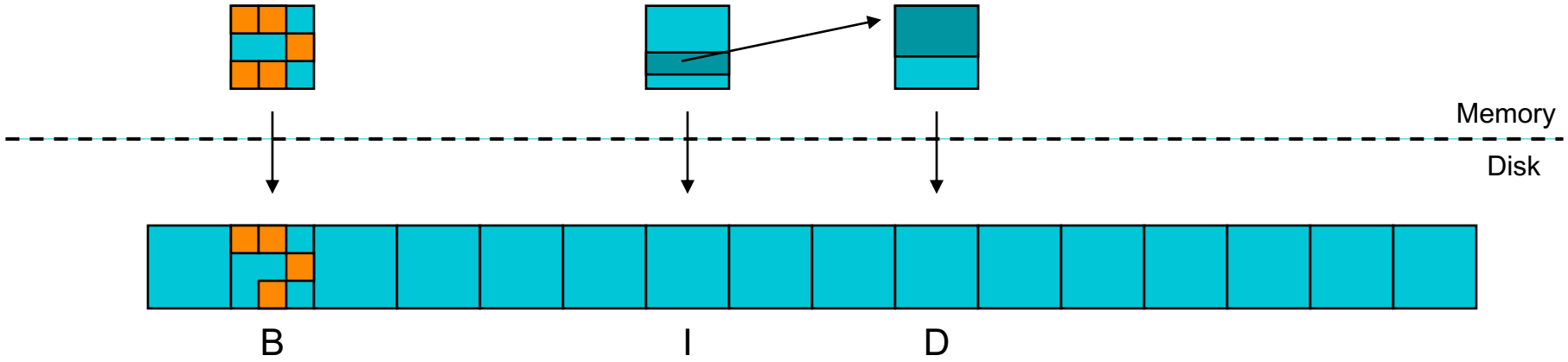
- Homework #4
  - Due Tuesday June 6th
- Project 3
  - Due June 10th – there will be no extensions
  - Get started early!

Administrivia Continued

- Course Evaluation
  - New SET form
  - I really appreciate your feedback
  - Due Saturday June 10th at 8:00 AM
- Final Exam
  - Monday June 12th at 3:00 PM
  - We will review in class on 6/8
  - Extra office hour 1-2 pm on 6/8

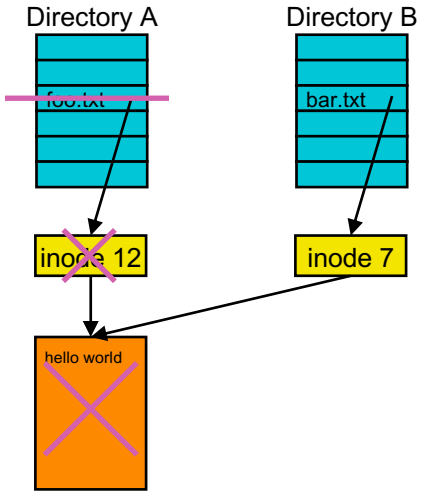
Crash Consistency Problem

- File system operations may involve writing multiple blocks
- Crashes can happen at any time
  - The file system may be left in an inconsistent state
- Goal: ensure that updates to the file system occur atomically



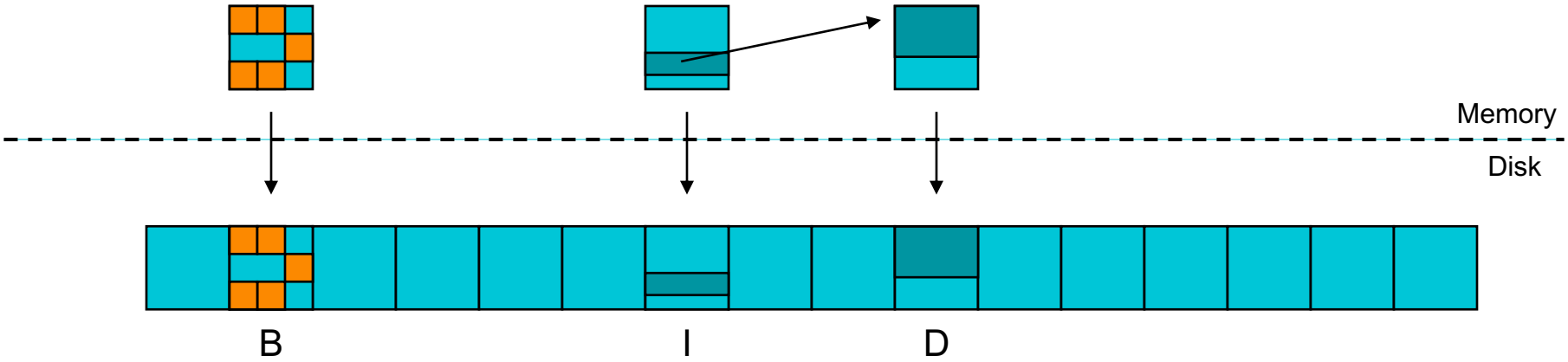
Check and Repair with `fsck`

- `fsck`: the file system checker in Unix
- When system boots:
  - Scan the file system for inconsistencies
  - Repair inconsistencies or notify an admin
  - Example repairs:
    - » Inode points to a data block that is marked as free – update the bitmap
    - » Inode reference count differs from number of links – update reference count
- Cons:
  - Very slow! Can take hours to run on large disk volumes
  - May not restore data to a valid state
  - May pose security issues



Ordered Writes

- Goal: prevent inconsistencies by writing the blocks to disk in a safe order
- Example: appending a new block to an existing file
  - Write the data block
  - Then write the bitmap block
  - Then write the inode

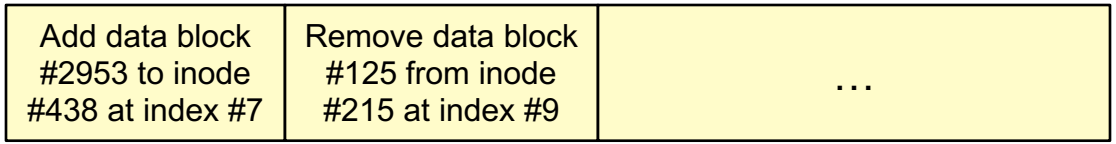


Ordered Writes

- In general:
  - Initialize blocks before writing pointers that point to them
  - Nullify existing pointers to a block before reusing it
  - Set a new pointer to a resource before clearing the last one (e.g., with `mv`)
- Pro:
  - No need to wait for `fsck` on reboot
- Cons:
  - Can leak resources (run `fsck` in the background)
  - Must wait for writes to complete, slows down operations
  - Difficult to find a safe interruptible ordering for all operations

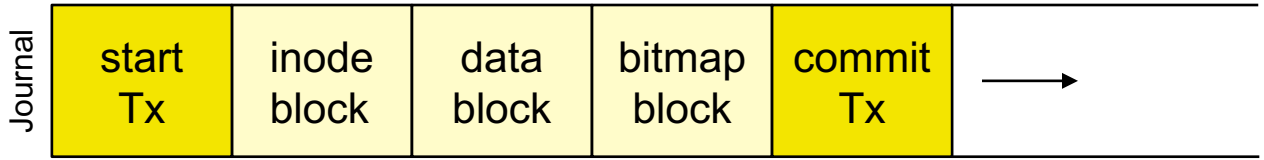
Journaling

- Also referred to as write-ahead logging
- Idea: write down what you are going to do before you do it
  - Record this information in a special append-only log file on disk
  - Flush this log to disk before modifying other blocks
  - When a crash occurs, replay the log to make sure all updates completed



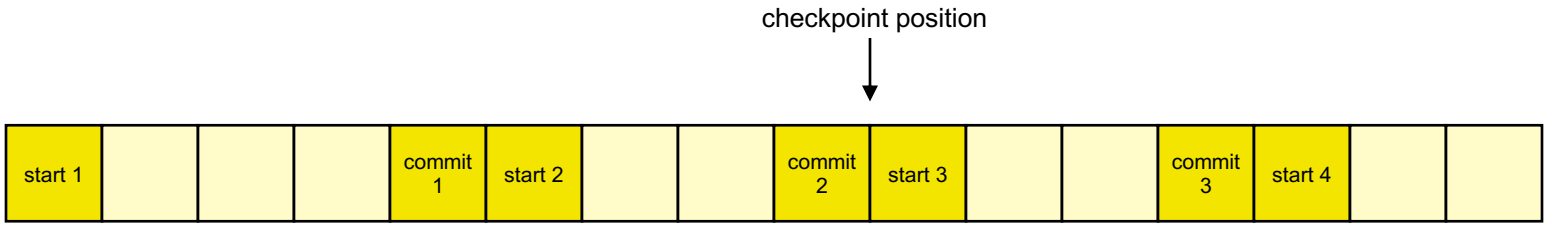
Journal

- Journal: an append-only file containing log records
  - Start transaction
    - » Transaction ID, block addresses, etc.
  - Blocks
  - Commit transaction
    - » Transaction has committed – updates will survive a crash
    - » The transaction is only final after the commit block is written



Journaling

- Checkpointing
  - Record the current position in the log
  - Copy all modified blocks to final destinations (“home locations”) on disk
  - Can clear the log before the recorded position
- Crash recovery
  - Replay all transactions that have committed but are not checkpointed
  - Discard uncommitted transactions



Journaling

- Pros:
  - Recovery is much faster
  - Eliminates inconsistencies
  - Log is written sequentially so writes are fast (no seeks)
  - Can delay writes to improve performance
- Cons:
  - Have to write the data twice

File System Summary

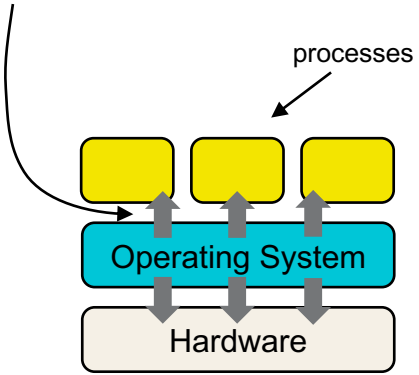
- Physical storage devices
  - Disks, flash, NVM
- File system APIs
  - Files, directories, links
- File system layout
  - Inodes, indirect blocks, super blocks, data blocks
  - Fast File System (FFS)
- File buffer cache
- File system reliability
  - `fsck`, ordered writes, journaling

Other Abstractions for Virtualization

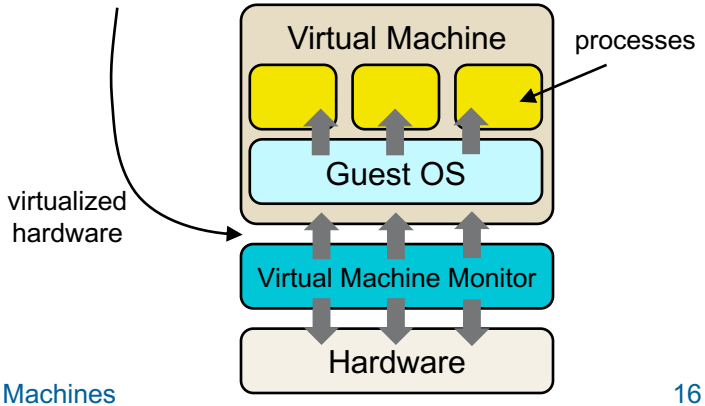
- Virtual machines
  - Types of Virtual Machine Monitors
    - » Type I vs. type II
    - » Full vs. para-virtualization
  - Virtualization components
    - » Processor
    - » I/O
    - » Memory
- Containers

Processes vs. Virtual Machines

- Abstractions for processes:
  - Virtual memory
  - System calls
  - Most instructions in the ISA
  - Most registers



- Abstractions for virtual machines:
  - Physical memory
  - Interrupts
  - All instructions in the ISA
  - All registers
  - I/O devices

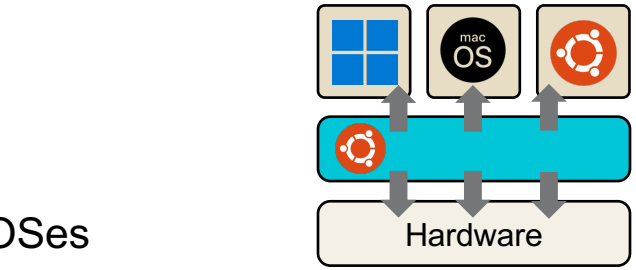


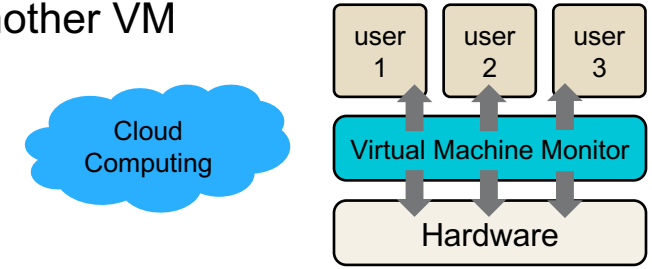
Virtual Machine Monitor (VMM)

- A VMM virtualizes an entire physical machine
- Presents a hardware interface to the OSes above
- Provides the illusion to each OS that it has full control of the hardware
  - Actually, the VMM controls the hardware
- Manages resources among multiple virtual machines (VMs)
- Isolates VMs from each other
- Also called a hypervisor

Why Virtual Machines?

- Software use
  - Can run software developed for different OSes
- Development and testing
  - Can test apps on different OSes, or test different OSes
- Isolation
  - Bugs or compromises in one VM cannot affect another VM
- Efficiency and cost reduction
  - Can share one machine among multiple users
  - Can migrate VMs from one machine to another

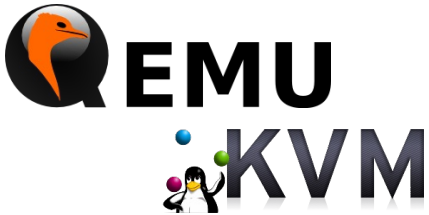




VMM Goals

- Fidelity
  - OSes and applications work the same without modification (although we may modify the OS a bit)
- Manageability
  - Creation, maintenance, administration, provisioning
- Isolation, like separate physical machines
  - VMM protects resources and VMs from each other
- Performance
  - Minimize the overhead of running in a VM
- Scalability
  - Support many VMs at once

Virtual Machine Monitors

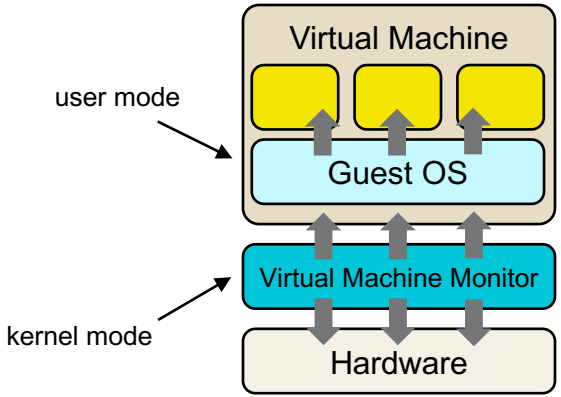


Virtualization via Simulation

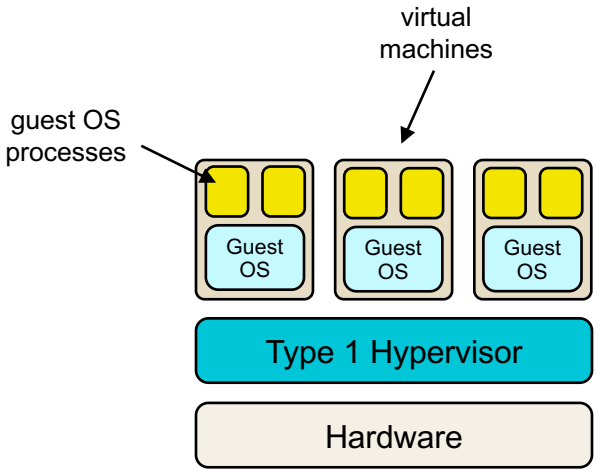
- VMM can simulate instruction execution
  - Simulate memory, I/O, etc.
- Examples:
  - Bochs
  - Nachos – simulates a MIPS processor
- Con: very slow

VMM Approach

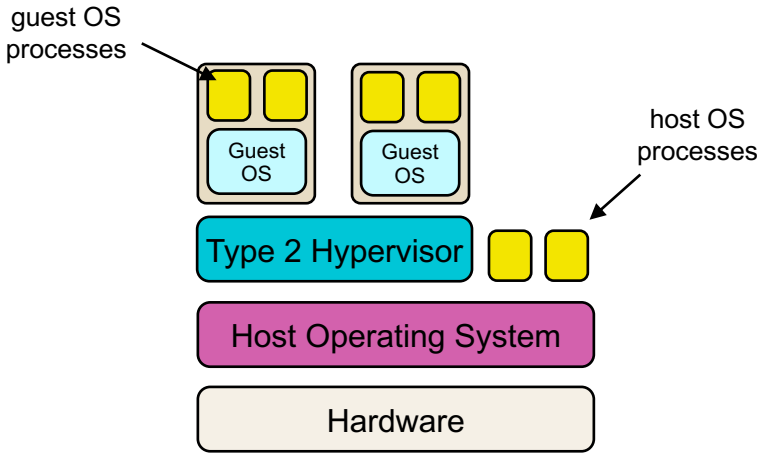
- Run the guest OS in user mode
  - Most instructions execute at regular CPU speed
- Run the VMM in kernel mode
- Anything “unusual” causes a trap to the VMM
  - Privileged instructions (e.g., executed by the guest OS)
  - System calls, exceptions
- VMM simulates the appropriate behavior
  - Known as trap-and-emulate



Type 1 and Type 2 Hypervisors



Type 1

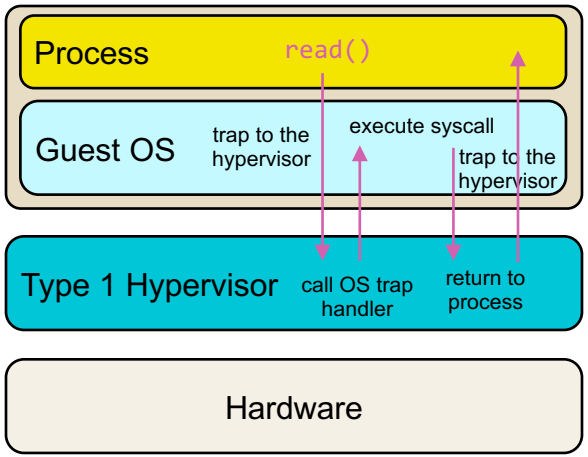


Type 2
(hosted hypervisor)

What Needs to be Virtualized?

- Events (exceptions and interrupts)
- CPU
- I/O devices
- Memory

Example: System Call (Type 1 Hypervisor)



Virtualizing the x86 Architecture

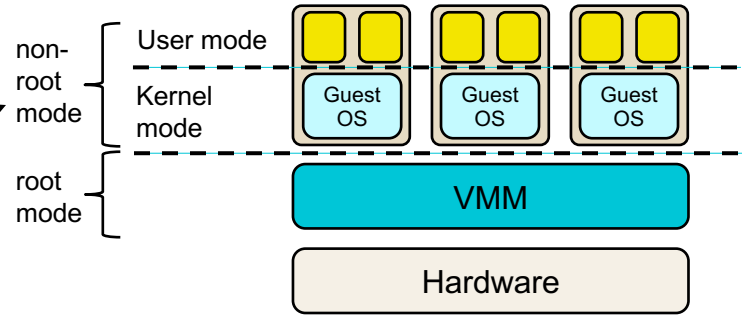
- x86 architecture was not fully virtualizable
- Problems:
  - Some privileged instructions behave differently when run in unprivileged mode
    - `popf` does not trap when it cannot modify system flags
  - Hardware-managed TLB
    - VMM cannot easily interpose on a TLB miss

Virtualizing the x86 Architecture

- Paravirtualization
  - Change the guest OS to better cooperate with the VMM
  - Sacrifices transparency for better performance
  - E.g., VMM can provide a “hypervisor API” so guest can perform certain functions
- Binary translation
  - Run guest OS code under control of a binary translator
  - Rewrites privileged instructions with emulation at runtime (may trap to VMM)
  - Incurs overhead, but can be kept small
- Hardware support
  - Intel and AMD added virtualization support in 2005 (Intel VT-x, AMD-V)

Virtualizing Privileged Instructions

- For instructions that cause a trap:
  - Trap to VMM, handle the instruction, return to guest OS or process
- For instructions that do not:
  - With paravirtualization – modify guest OS to call into the VMM
  - With binary translation – rewrite OS instructions to emulate or call into VMM
  - With hardware support – add a new CPU mode and instructions to support trap-and-emulate

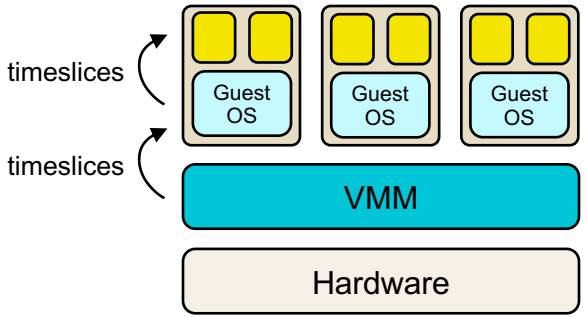


Virtualizing Events

- VMM receives interrupts and exceptions
- Need to vector events to the correct VM
  - With paravirtualization – modify OS to use a virtual interrupt register, event queue
  - With full virtualization – craft an appropriate handler invocation, call into it from VMM
  - With hardware support – hardware delivers events directly to the guest OS

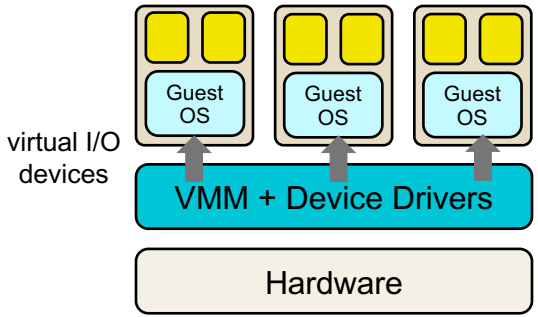
Virtualizing the CPU

- VMM needs to multiplex VMs on the CPU
- Reuse scheduling techniques
  - Timeslice the VMs
  - Each VM will timeslice its OS/applications during its quantum
  - Typically a simple scheduler (e.g., round robin)

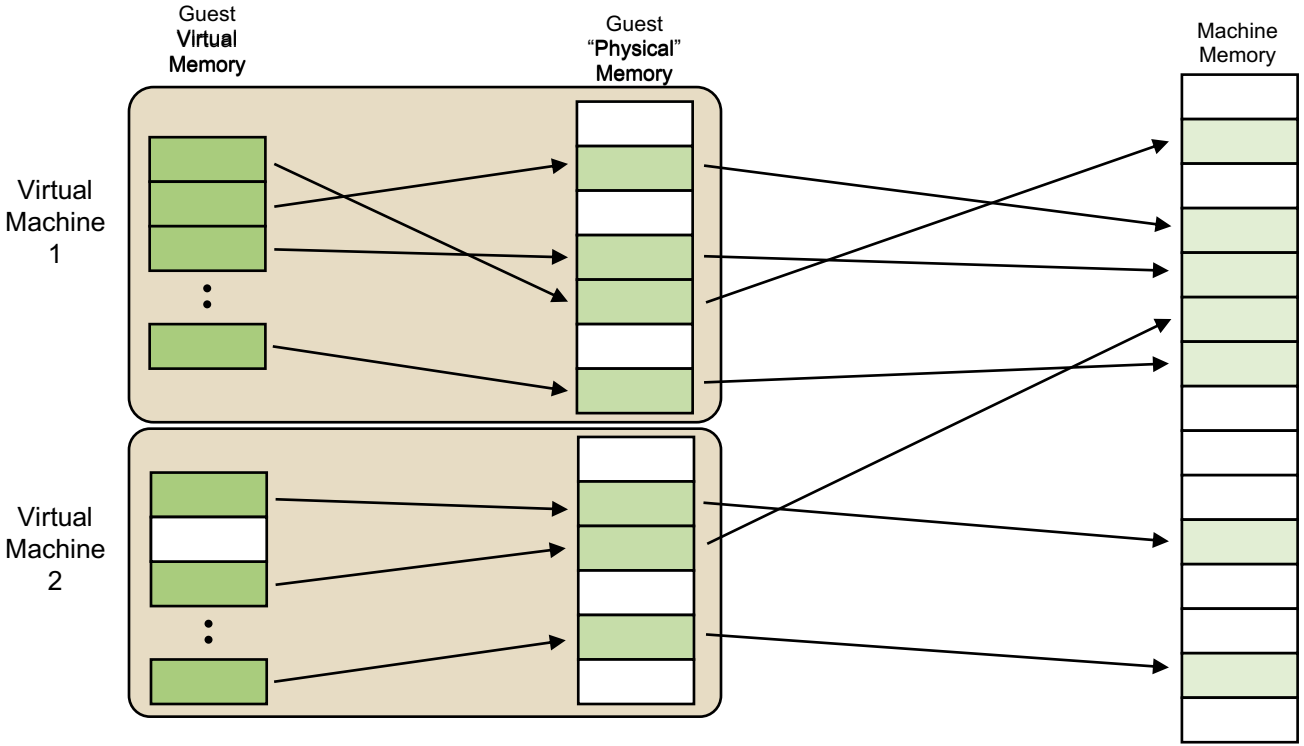


Virtualizing I/O

- Challenge:
  - Lots of I/O devices
  - We don't want to write virtualized device drivers for all possible I/O devices
- One approach:
  - Run device drivers in the VMM
  - VMM presents simple virtual I/O devices to guest VMs
- Can optimize using paravirtualization or specialized hardware



Virtualizing Memory



To be continued next lecture…

For next class…

- Read chapters 53 and 55