Goals for Today

- Learning Objective:
  - Understanding how operating systems support containers and modern cloud computing paradigms
- Announcements, etc:
  - MP4 due **May 6th**
    - Get started ASAP!
  - HW1 available! Due **May 8th**
    - Just an "appetizer" for the final exam
    - Multiple attempts allowed, but first attempt is graded

Reminder: Please put away devices at the start of class

CS 423
Operating System Design:
OS Support for Containers

Professor Adam Bates
Spring 2018

Advanced Configuration and Power Interface (ACPI)

Part of UEFI since 2013:

- Exposes different power saving states in a platform-independent manner
- The standard was originally developed by Intel, Microsoft, and Toshiba (in 1996), then later joined by HP, and Phoenix.
- The latest version is "Revision 6.3" published in January 2019!

ACPI Global States

- **G0**: working
- **G1**: Sleeping and hibernation (several degrees available)
- **G2**:, Soft Off: almost the same as G3 Mechanical Off, except that the power supply still supplies power, at a minimum, to the power button to allow wakeup. A full reboot is required.
- **G3**, Mechanical Off: The computer's power has been totally removed via a mechanical switch (as on the rear of a power supply unit).

ACPI Global States

Processor-specific

- **G0**: working
- **G1**: Sleeping and hibernation (several degrees available)
- **G2**:, Soft Off: almost the same as G3 Mechanical Off, except that the power supply still supplies power, at a minimum, to the power button to allow wakeup. A full reboot is required.
- **G3**, Mechanical Off: The computer's power has been totally removed via a mechanical switch (as on the rear of a PSU).

ACPI "Sleep" States

C-States:

Core-specific

- **C0**: is the operating state.
- **C1** (often known as Halt): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.
- **C2** (often known as Stop-Clock): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as Sleep) is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.

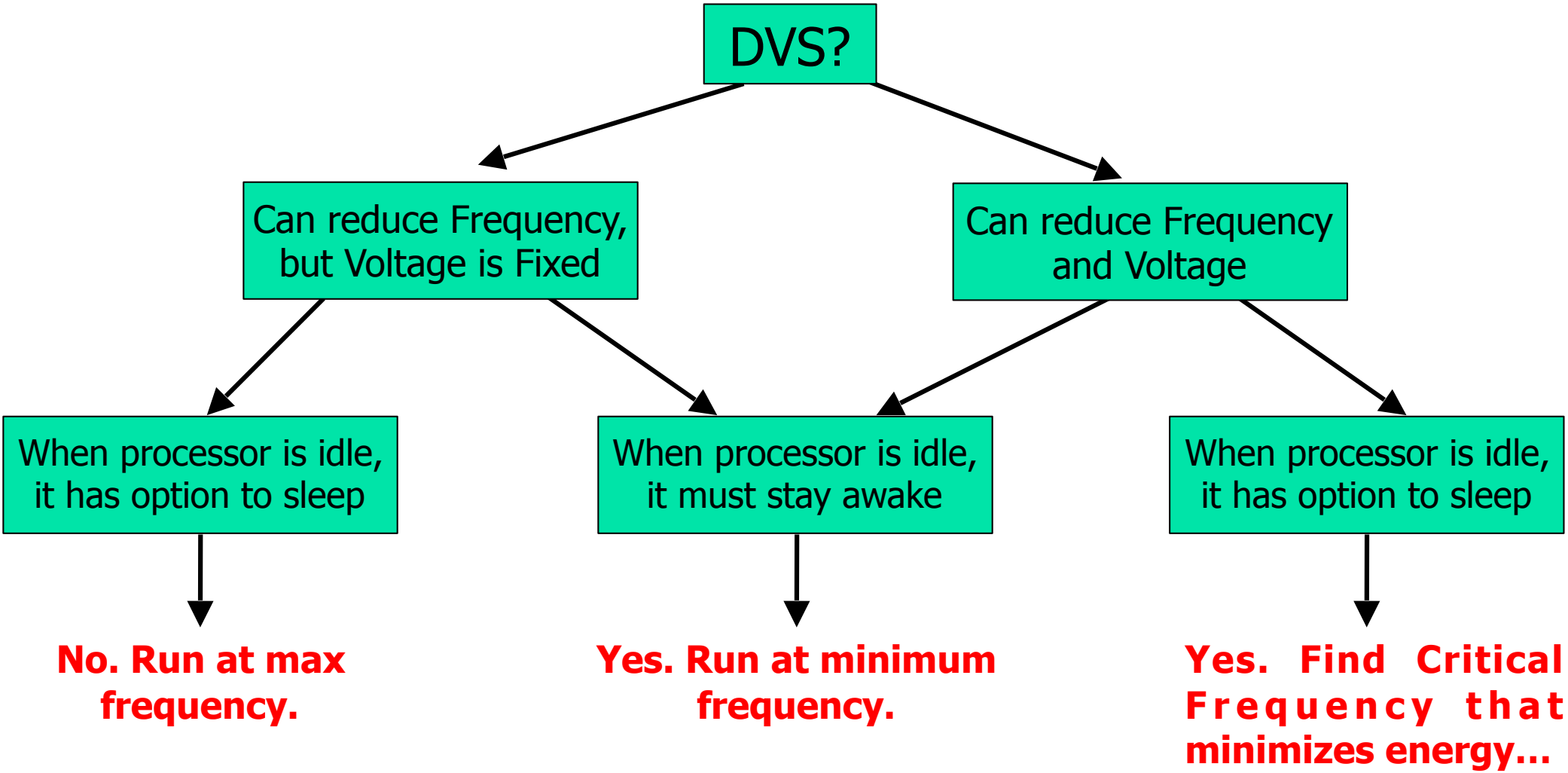
ACPI "Sleep" States

C-States:

- **C0**: is the operating state.
- **C1** (often known as Halt): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.
- **C2** (often known as Stop-Clock): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as Sleep) is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.

Last Class…

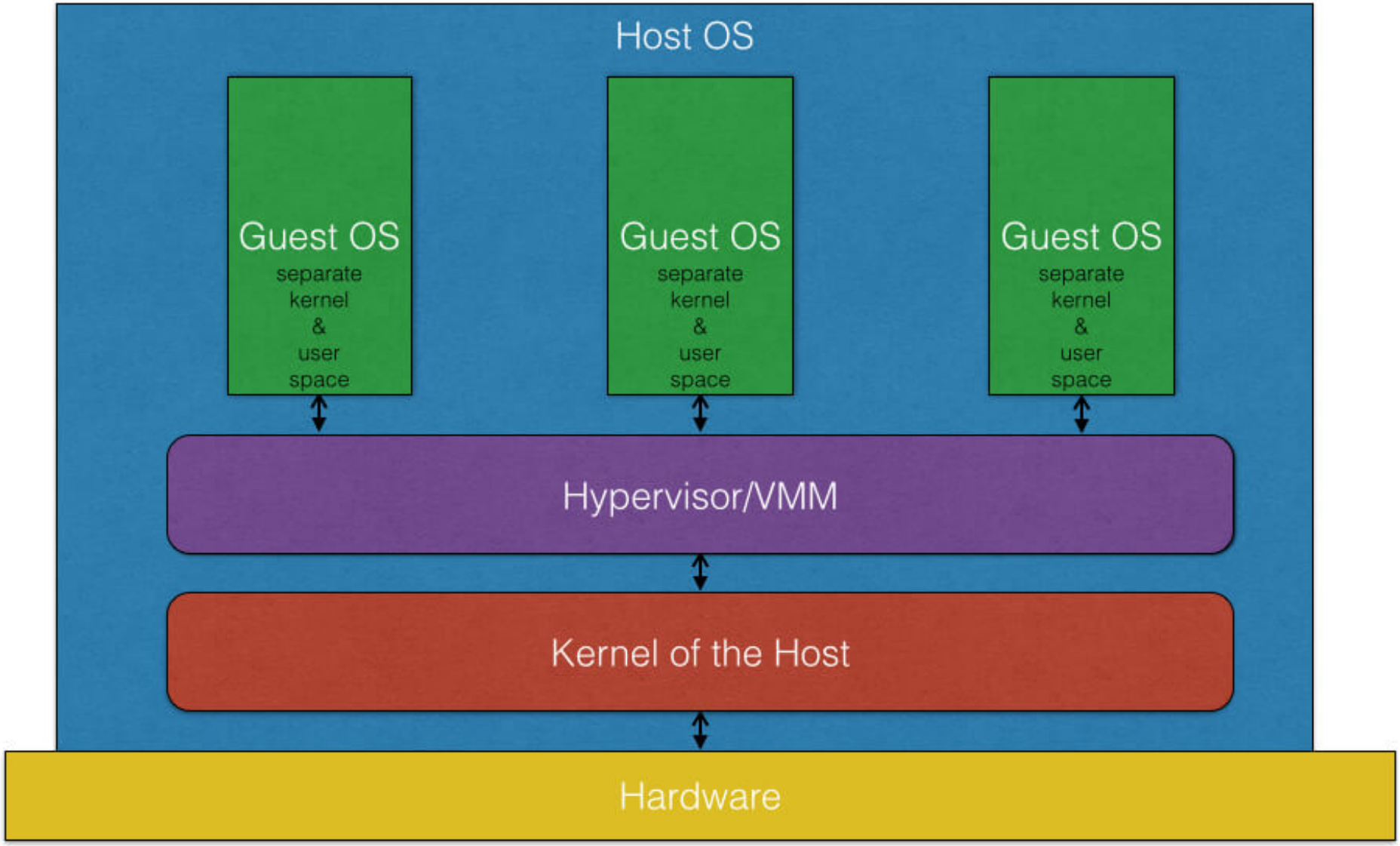
When should we perform dynamic voltage scaling?



Cloud Computing (Gen 1)

- Dominated by Infrastructure-as-a-Service clouds (and storage services)
- Big winner was Amazon EC2
- Hypervisors that virtualized the hardware-software interface
- Customers were responsible for provisioning the software stack from the kernel up

Hypervisors

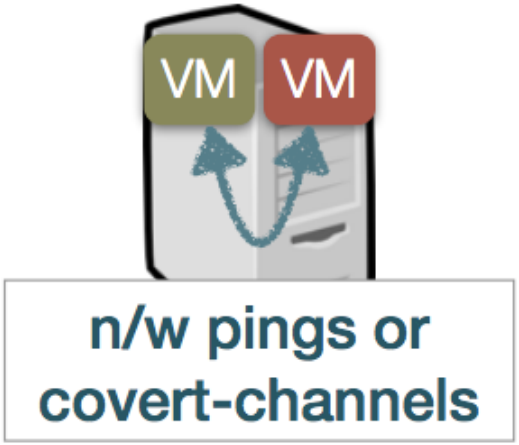


Hypervisors

- Strong isolation between different customer's virtual machines
- VMM is 'small' compared to the kernel… less LoC means less bugs means (~)more security.

Hypervisors

- 'Practical' attacks on IaaS clouds relied on side channels to detect co-location between attacker and victim VM
- E.g., we could correlate the performance of a shared resource
  - network RTT's, cache performance
- After co-resident, make inferences about victim's activities



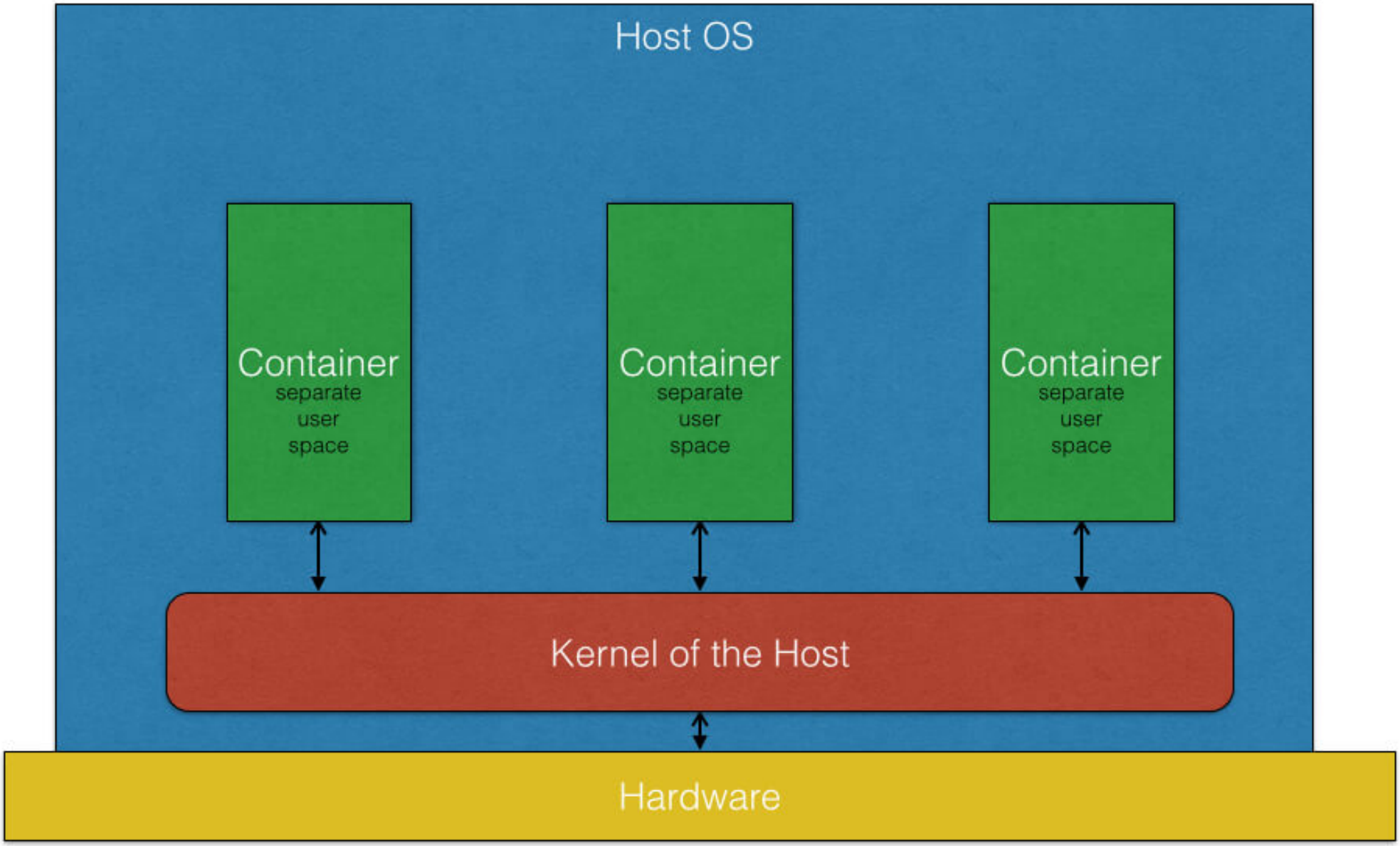
Hypervisors

- Strong isolation between different customer's virtual machines
- VMM is 'small' compared to the kernel… less LoC means less bugs means (~)more security.
- High degree of flexibility… but did most customers really need it?

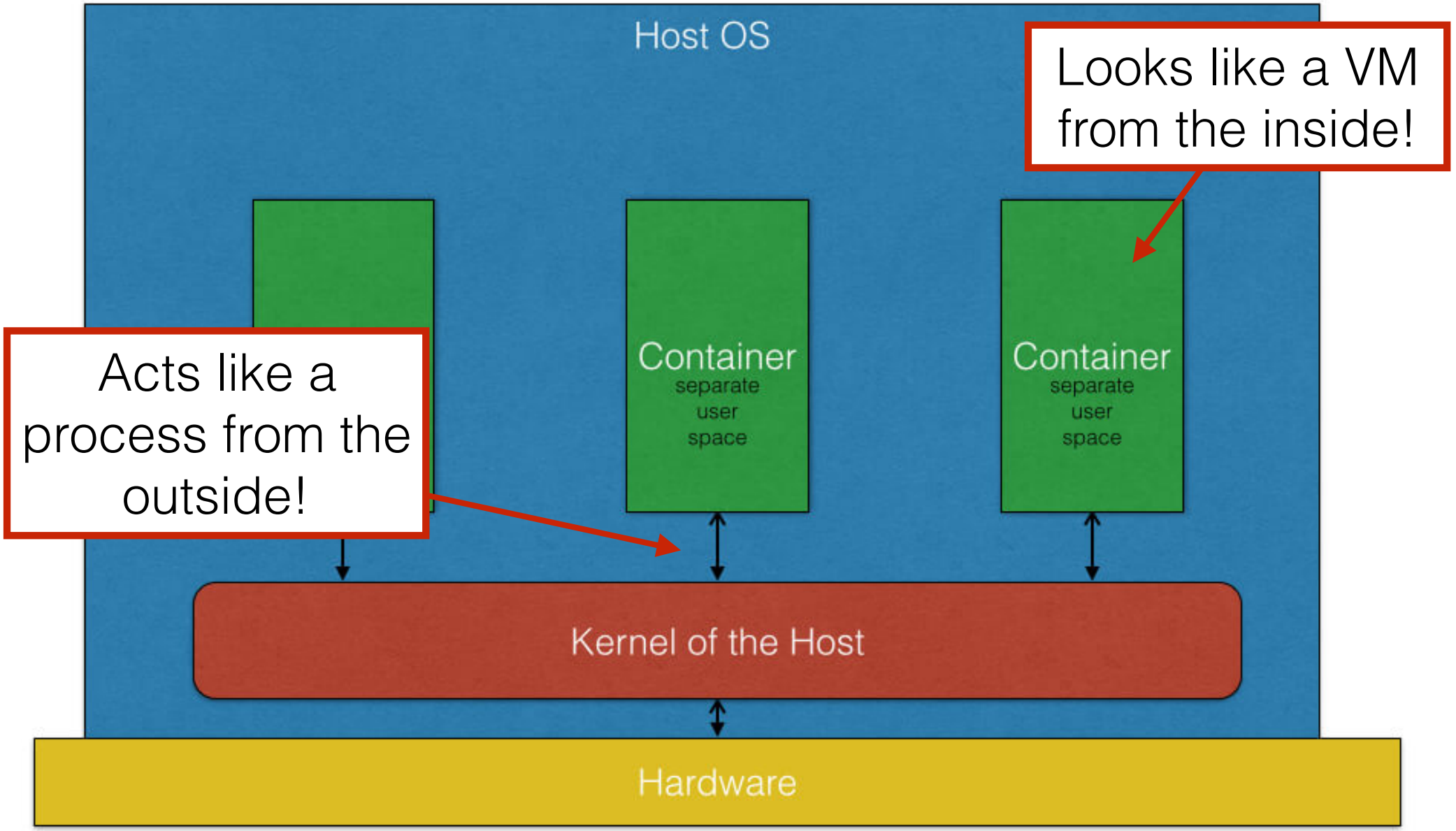
Enter Containers

- Rather than virtualize both user space and kernel space… why not just 'virtualize' user space?
- Meets the needs of most customers, who don't require significant customization of the OS.
- Sometimes called 'operating system virtualization,' which is highly misleading given our existing taxonomy of virtualization techniques
- Running natively on host, containers enjoy bare metal performance without reliance on advanced virtualization support from hardware.

Enter Containers



Enter Containers



OS Support for Containers

- Linux Containers (LXC):
  - chroot
  - Kernel Namespaces
    - PID, Network, User, IPC, uts, mount
  - cgroups for HW isolation
  - Security profiles and policies
    - Apparmor, SELinux, Seccomp

containers = chroot on steroids

- `chroot` changes the apparent root directory for a given process and all of its children
- An old idea! POSIX call dating back to 1979
- Not intended to defend against privileged attackers… they still have root access and can do all sorts of things to break out (like `chroot`'ing again)
- Hiding the true root FS isolates a lot; in *nix, file abstraction used extensively.
- Does not completely hide processes, network, etc., though!

Namespaces

- The key feature enabling containerization!
- Partition practically all OS functionalities so that different process domains see different things
- Mount (mnt): Controls mount points
- Process ID (pid): Exposes a new set of process IDs distinct from other namespaces (i.e., the hosts)
- Network (net): Dedicated network stack per container; each interface present in exactly 1 namespace at a time.
- ….

Namespaces

- The key feature enabling containerization!
- Partition practically all OS functionalities so that different process domains see different things
- Interprocess Comm. (IPC): Isolate processes from various methods of POSIX IPC.
    - e.g., no shared memory between containers!
- UTS: Allows the host to present different host/domain names to different containers.
- There's also a User ID (user) and cgroup namespace

User Namespace

- Like others, can provide a unique UID space to the container.
- More nuanced though — we can map UID 0 inside the container to UID 1000 outside; allows processes inside of container to think they're root.
- Enables containers to perform administration actions, e.g., adding more users, while remaining confined to their namespace.

cgroups

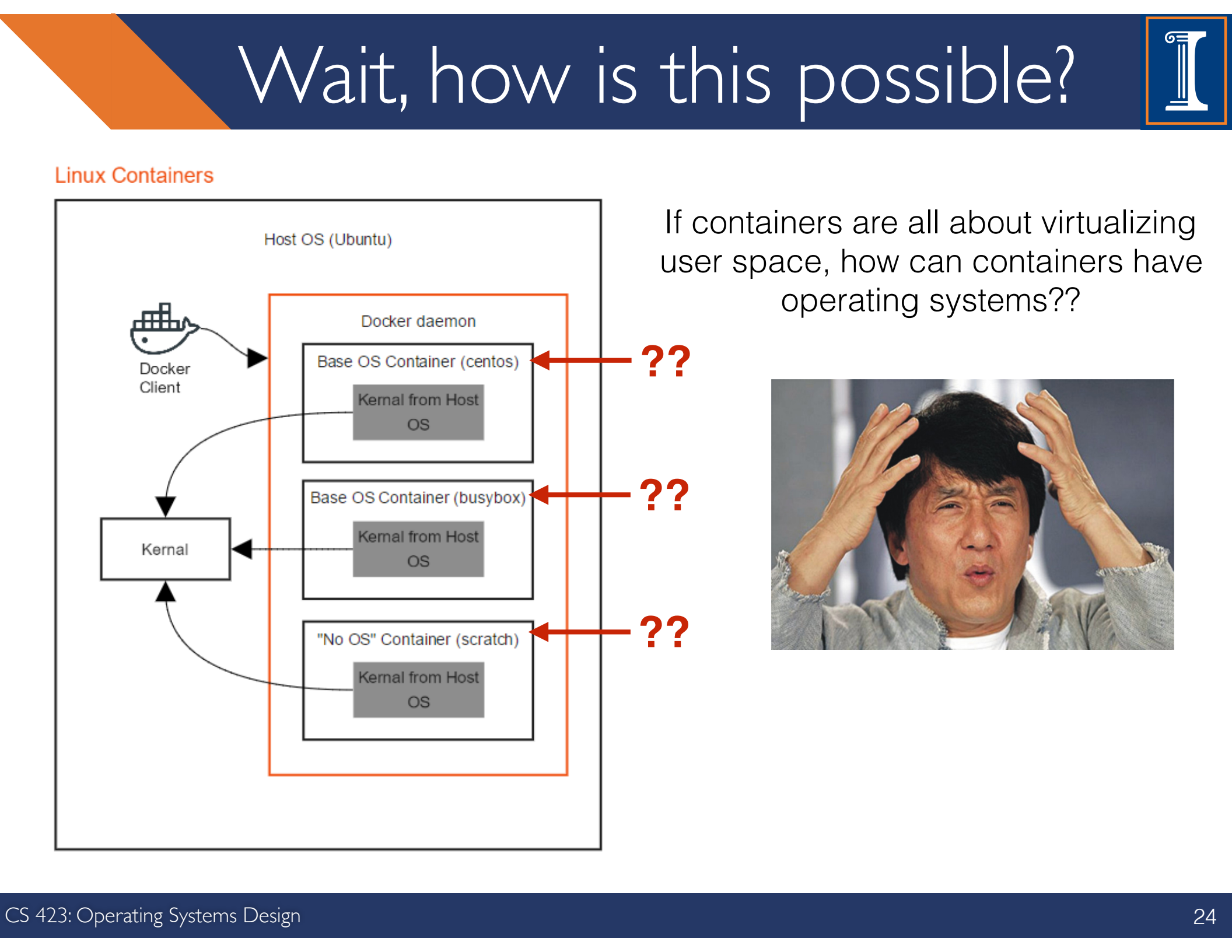
- Limit, track, and isolate utilization of hardware resources including CPU, memory, and disk.
- Important for ensuring QoS between customers! Protects against bad neighbors
- Operate at the namespace granularity, not per-process
- Features:
  - Resource limitation
  - Prioritization
  - Accounting (for billing customers!)
  - Control, e.g., freezing groups
- The cgroup namespace prevents containers from viewing or modifying their own group assignment

Container Security?

"Containers do not contain." - Dan Walsh (SELinux contributor)

- In a nutshell, it's *real hard* to prove that every feature of the operating system is namespaced.
  - /sys? /proc? /dev? LKMs? kernel keyrings?
- Root access to any of these enables pwning the host
- Solution? Just don't forget about MAC; at this point SELinux pretty good support for namespace labeling.
- SELinux and Namespaces actually synergize nicely; *much* easier to express a correct isolation policy over a coarse-grained namespace than, say, individual processes

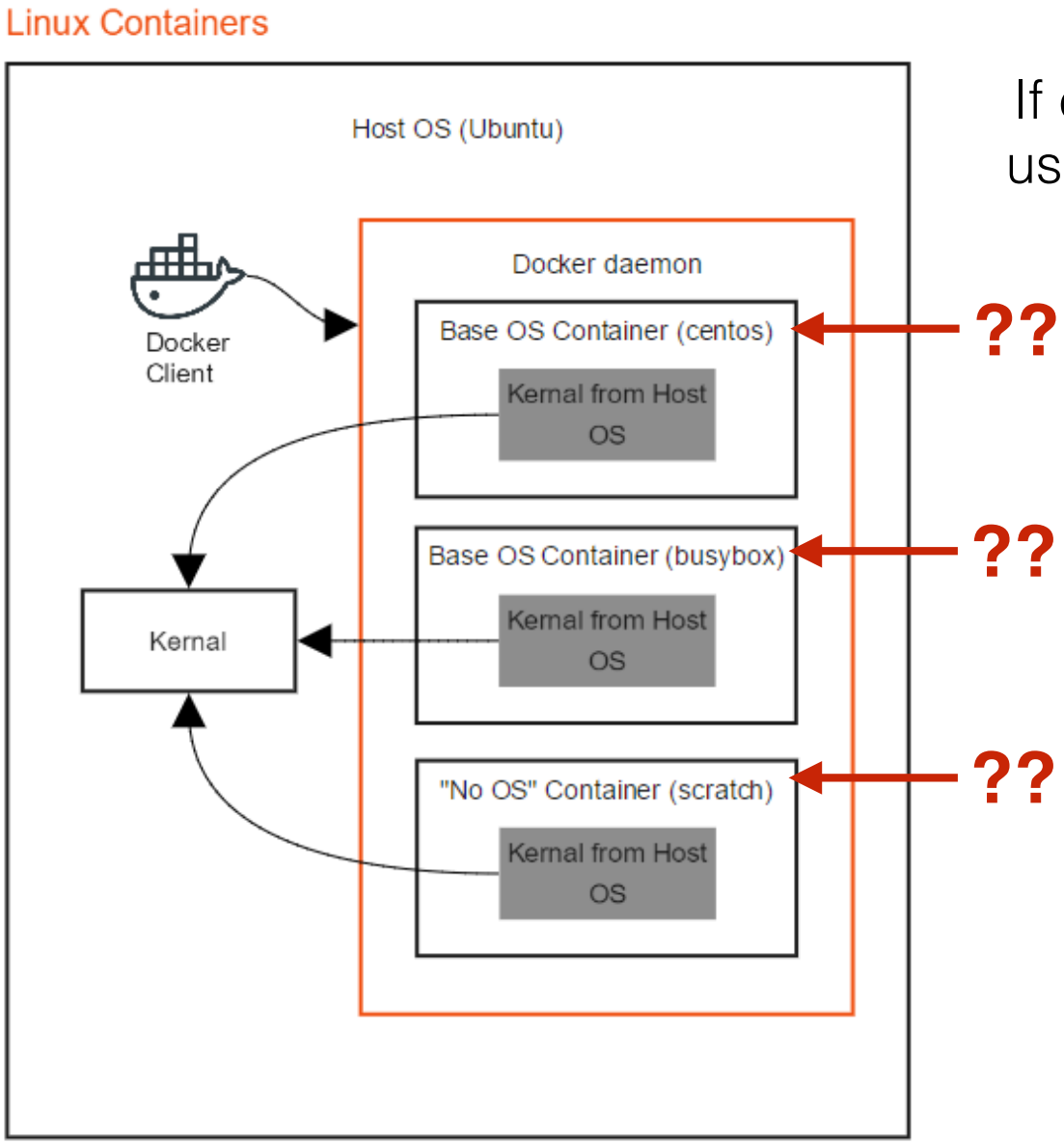
Wait, how is this possible?



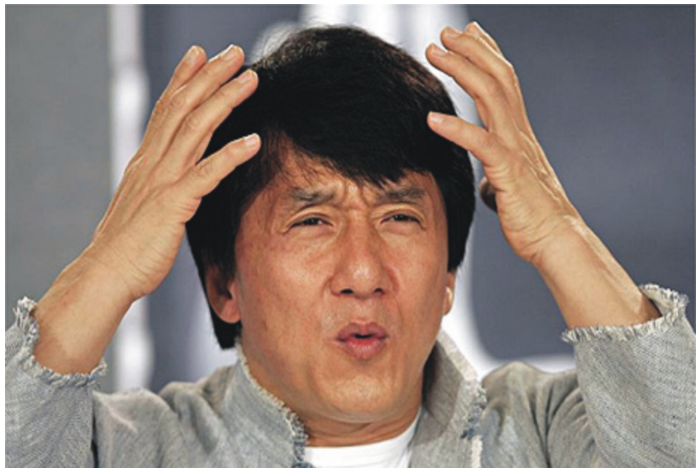
Linux Containers

If containers are all about virtualizing user space, how can containers have operating systems??

Wait, how is this possible?



If containers are all about virtualizing user space, how can containers have operating systems??



Answer: These aren't kernels; they're the system utilities of the distro that have been tricked by process namespaces!

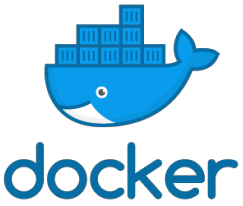
Namespaces in the Linux kernel

- Start by looking in `include/linux/*/*_namespace.h`

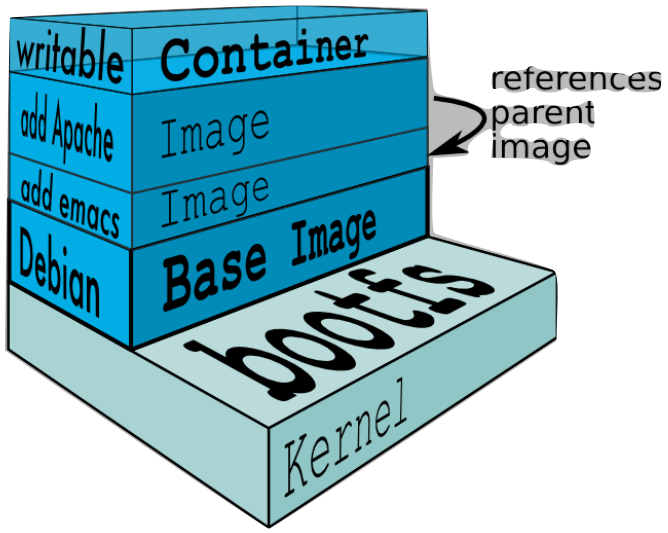
```
struct user_namespace {
        struct uid_gid_map      uid_map;
        struct uid_gid_map      gid_map;
        struct uid_gid_map      projid_map;
        atomic_t                count;
        struct user_namespace   *parent;
        int                     level;
        kuid_t                  owner;
        kgid_t                  group;
        struct ns_common        ns;
        unsigned long           flags;

        /* Register of per-UID persistent keyrings for this namespace */
#ifdef CONFIG_PERSISTENT_KEYRINGS
        struct key              *persistent_keyring_register;
        struct rw_semaphore     persistent_keyring_register_sem;
#endif
};
```

How Docker fits in



- Not an OS thing. ;)
- Utilities that allow you to leverage (e.g.) LXC to build a portable, self-sufficient application using containers.
- Assures that all libraries and dependencies are packaged inside of a container image



Above the clouds…

- Container (~PaaS clouds) are strictly easier to manage than traditional IaaS VMs.
- The era of Container hype has somewhat come and gone… containers still expose more flexibility than most users need!!
- Th hype now is about Function-as-a-Service cloud; individual programs/functions executed by invocation, great for event-driven stuff.
- Enabled by containers
- Instruction-as-a-Service next? ;)

Takeaways

- Container support has existing in Linux for many years
- Foundations of containerization has been around for decades!
- Automating LXC for portability (i.e., Docker) has revolutionized cloud computing
- Lasting legacy of containers may be enabling the Function-as-a-Service revolution… cloud customers can now pay by the method invocation without any idle costs.