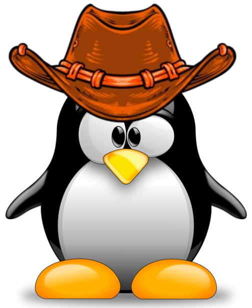


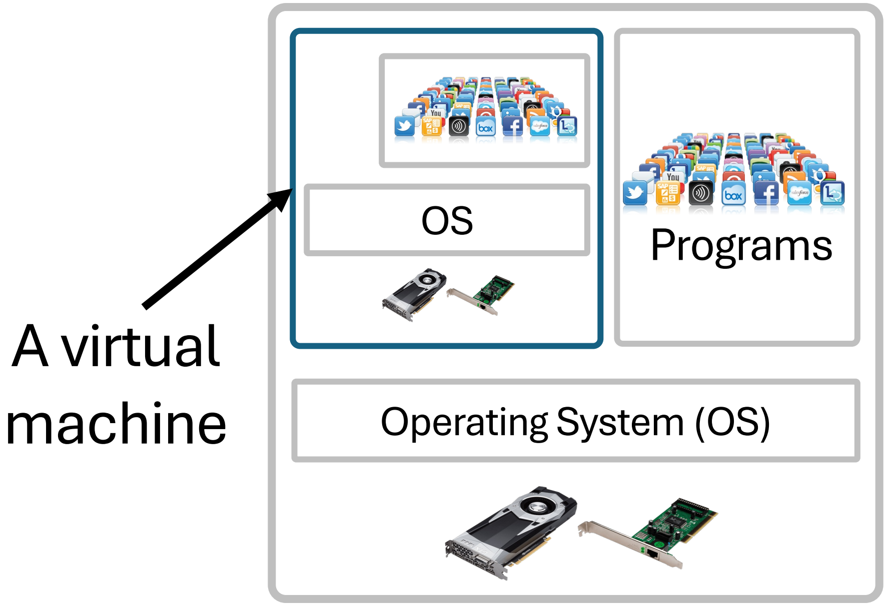
CSE 330: Operating Systems

Adil Ahmad

Lecture #22: HW-Assisted and OS-Level Virtualization

What’s a virtual machine (or a VM)?

A *simulated* instance of a computer inside a computer



Desired VM properties:

1. Accurate simulation
2. Isolation from ‘host’
3. Fast!

Why do we use VMs in today's world?

A. Cloud computing:

- Each user has their own isolated 'machine'
- Can run different OSs and remain isolated from other users
- Better resource management for cloud providers

B. System development:

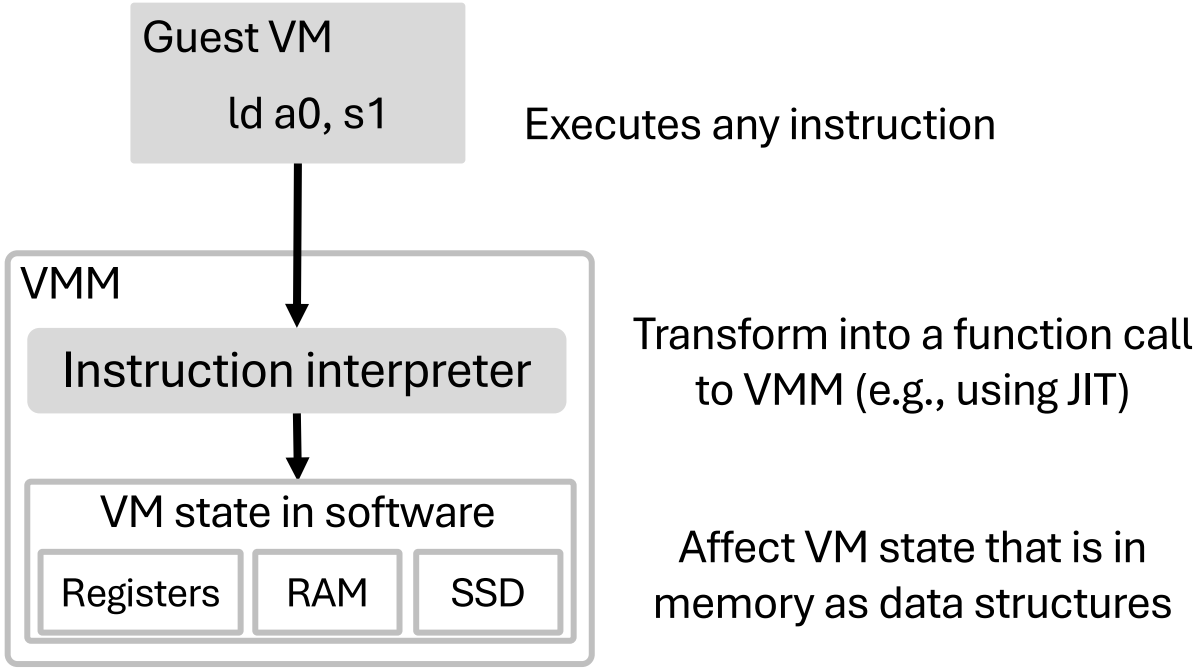
- Testing OSs without breaking your own system
- Testing programs built for certain OS/packages

C. System security:

- Many important OS features can be secured by implementing in a higher privileged layer (remember nested kernel?)

1. Full software emulation-based virtualization

Simulate the entire machine's execution in software-mode



Other state that must be simulated: privileged registers, UART, PMP, etc.

2. Trap-and-emulate (T&E) virtualization

- Run the VM as a user-mode process

- User instructions are directly executed by the VM
  - ld a0, s1 → no trap

- Supervisor instructions by the VM OS are trapped to the VMM
  - VMM emulates these just like in full software emulation

- **Advantages of this approach?**
  - Helps avoid the cost of translating *all* instructions
  - Still can build this design in software entirely

We should allow the VM to execute ~~all~~ *most* instructions!

- Does not incur the overhead of *any* traps
  Even privileged instructions can be directly executed on the CPU

- **How is this even possible, let alone secure?**
  - Think about a process – save kernel context in TRAPFRAME and move to process execution at ‘sret’ instruction
  - What if we can save the host context and jump to the guest VM’s execution and resume later?

3. Hardware-assisted virtualization

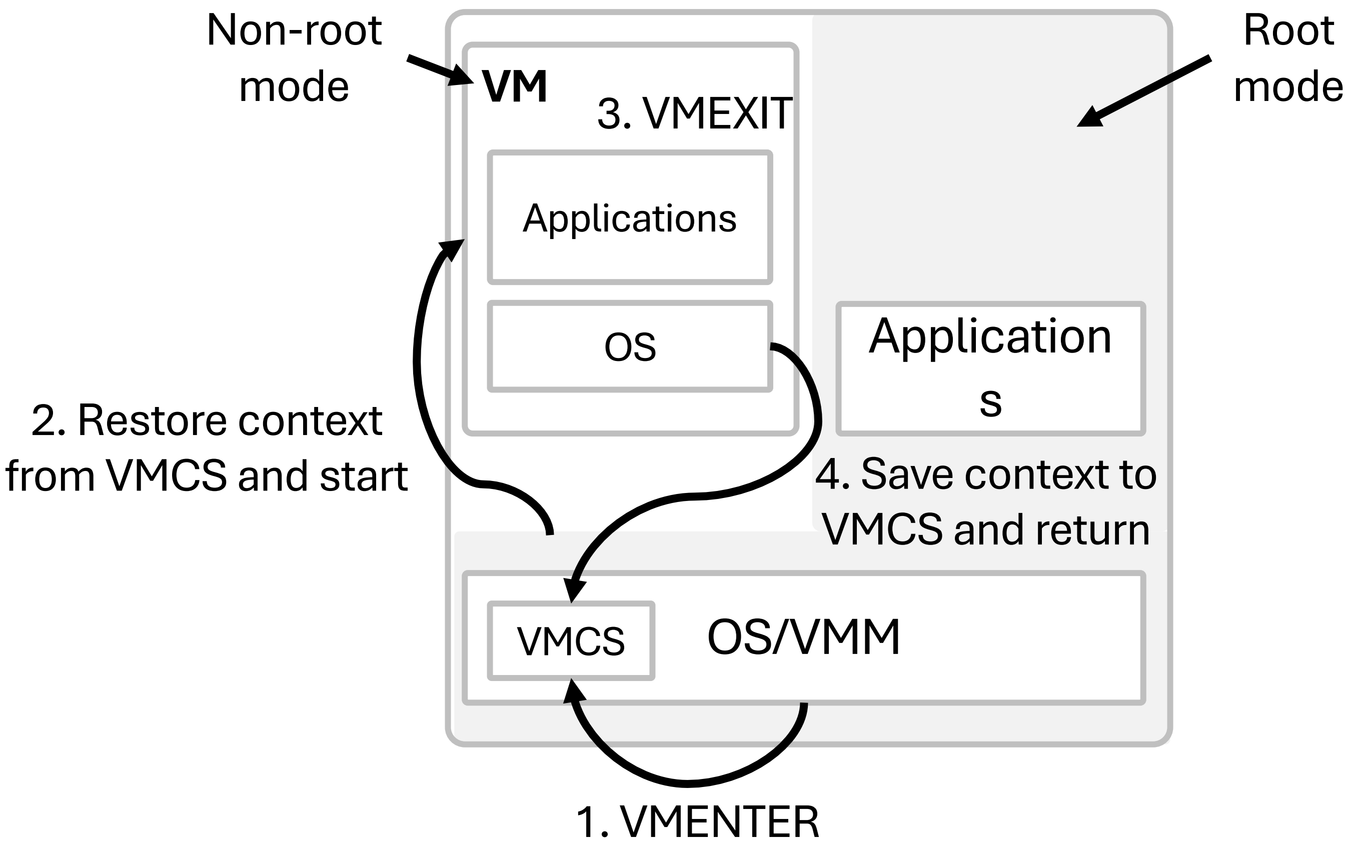
- New mode of execution: root mode and non-root mode.
  - Root mode is for host/VMM, non-root mode for guest/VM
- CPUs provide support for isolating the execution of host and VM
  - VMENTER/VMRESUME – instruction to enter/resume a VM from host
  - VMEXIT – exit the VM back to the host
- On VMENTER, host should save all context that it needs for later
- On VMEXIT, CPU saves guest context in a "TRAPFRAME" called the "Virtual Machine Control Structure" or VMCS (x86)

The context to be saved on exit/entry to VMs

All register contexts including the privileged/unprivileged registers

- Control registers (CRs)
- Segment registers
- Floating point registers, etc.

x86 HW-assisted virtualization illustration



There are three main **security problems**

We need to systematically address the following .

A. Guest could read/write outside its own memory regions

- E.g., using modified page tables

B. Guest could talk to devices that the host wants to isolate.

- Could also refuse to give up control to host at interrupts

C. Guest could adversely impact important system functionality

- Change voltage on a certain CPU core

How would you address these problems? (one-by-one)

We need to systematically address the following .

A. Guest could read/write outside its own memory regions
- E.g., using modified page tables

B. Guest could talk to devices that the host wants to isolate.
- Could also refuse to give up control to host at interrupts

C. Guest could adversely impact important system functionality
- Change voltage on a certain CPU core

A. Preventing guest from accessing outside memory

- Introduce a *second layer of translation* inside the memory management unit (MMU), called the Extended Page Tables (EPT)

- *High-level idea:*
  - Without EPT: guest VA → PTs → host PA (insecure)
  - With EPT: guest VA → PTs → guest PA → EPTs → host PA

- Since EPTs are controlled by VMM, a VM can only access regions of memory that the VMM wants it to access

- CPU delivers PFs in 1st PTs to guest, and EPTs to host

B. Isolating devices and interrupts

- HW interrupts are *sent directly to the host* (i.e., using a VMEXIT), which will decide when interrupts should be sent to the guest

- Preventing guest from accessing devices

  - Instructions that access device regions (e.g., IN/OUT) can be **intercepted** directly at the VMM-level

  - MMIO: Use EPT to protect device memory-mapped regions

  - IOMMU: Preventing a guest from using malicious DMA

C. Prevent changes to important system functionality

- CPU provides a variety of controls to force exits from VM when it is trying to make changes to system functions
- CPU voltage example:
  - To change CPU voltage on x86 systems, the guest OS must write to a model-specific register (MSR) using WRMSR instruction
  - X86 allows VMM to intercept the execution of WRMSR and disallow it if needed.
- VMM can also specify what CPU features are available to VM

Quick recap of the three virtualization techniques

- Full system emulation (interpret every instruction)
  - Architecture-agnostic and SW-based
  - Slow due to interpretation of every instruction

- Trap-and-emulate (interpret only privileged instructions)
  - Faster than FSE and SW-based
  - Still slow and not architecture-agnostic

- Hardware-assisted (interpret no instruction)
  - Defacto today in cloud machines since it is very fast!
  - Requires hardware support and not architecture-agnostic

Apart from CPU, how would you virtualize devices to VMs?

Like CPU *virtualization*, there are three (main) techniques

- Full software emulation (of devices)
- *Hybrid* software emulation with hardware acceleration
- Hardware device passthrough

Let's start with **full software emulation of devices**

- *As the name suggests,* a device is fully-emulated in software by VMM
- QEMU sets-up the following for a debug console for VMs
  - ✓ Reserves a part of the physical address space for device registers
  - ✓ Monitors read and write operations to the register space
  - ✓ Emulates the functionality of UART provided by HW vendors
  - ✓ Uses UART functionality to emulate a console/terminal
- **Other example:** Graphical rendering using *LLVM pipes* in QEMU

What are the pros and cons of full software emulation?

- **Does not require any specific hardware**
- **Generally, it is quite slow**
  - Device operations (e.g., graphical rendering) are now emulated by the VMM (using the CPU) and OS
  - CPU may not be well-optimized for device operations

Hybridizing software emulation with HW acceleration

- *High-level idea:* Keep the emulated device interface at the software-level, but speed-up certain computations by directly using devices

- Important example: **accelerated graphics rendering**
(e.g., Virtual GL or VirGL in QEMU)

  - ✓ QEMU creates an emulated GPU like in SW-emulation
  - ✓ The commands received by the emulated GPU are decoded and sent directly to the hardware GPU
  - ✓ The rendered frames are sent back to the guest machine

What are the pros and cons of hybrid HW-SW?

- **Faster than full software emulation**
- **Maintains the *virtual* interface of emulated devices**
- **Generally, it is still much slower than direct device access**
  - For instance, in one of our recent works, we noticed that *VirGL* is still about 16 times than native GPU acceleration!

Direct device passthrough is the *fastest approach!*

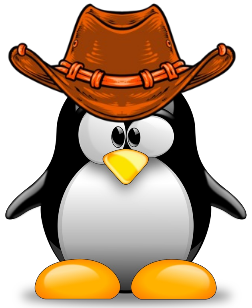
- Each device is isolated and *directly* made accessible to the virtual machine
  - For instance, by providing access to the device’s physical MMIO regions
- The guest “owns” the device and can directly communicate with the device for hardware-accelerated processing
- The host leverages I/O memory management unit (IOMMU) to protect itself from malicious access by the passthrough device

What are the pros and cons of device passthrough?

- (Almost as) fast as native access
- Requires reserving the entire device for a VM and cannot be easily shared amongst N virtual machines (if needed)

Single-Root I/O Virtualization (SR-IOV)

- New technique to mitigate the “single device, single VM” passthrough problem
- The hardware device (e.g., GPU) creates “fixed virtual partitions” within its own core that can each be passthrough to a virtual machine
- Requires the hardware to have advanced SR-IOV functionality
  - Typically cloud-capable devices like Nvidia/AMD graphics card and enterprise SSDs have this capability today
- **Cons/disadvantages?**
  - Fixed partitions avoid full use of resource → slower
  - Not very widely-supported today



Final topic: OS-level virtualization

Retrospective on *system-level* virtualization

- Allows running full **virtual machines** with their own OS kernels, devices, etc.

- Generally, running VMs is quite heavy – even HW-accelerated VMs will consume a significant amount of memory and CPU resources to run
  - Separate OS kernel
  - Background programs/services

- This is acceptable if your goal is to have a fully-isolated and different environment for your programs, but *this is not always required!*

Can anyone give examples where system virtualization is not required?

Example: Running programs with old set of libraries

- Programs rely on shared libraries (e.g., C library), but Linux has *poor backwards-compatibility* on these libraries!
- A programmer now wants to run an environment:
  - They can install old version of libraries for a certain program
  - Keep new libraries for other newer programs
- *One solution:* run a VM that starts an older version of Linux and run the program
- **Is that a good solution?**
  - No, we don't care/need an OS kernel and all the heavy resource utilization of virtual machines

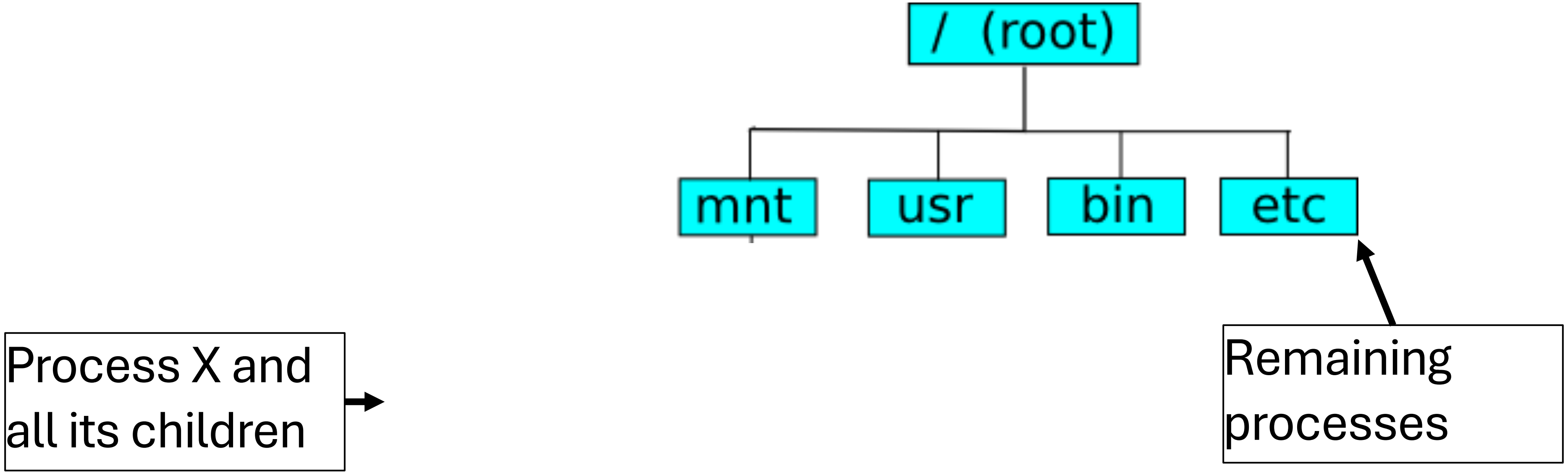
Example: Restricting set of programs to limited resources

- By default, a program in Linux will use all the available memory and CPU resources (if it needs it) and compete with other programs
- You want a certain program (or set) to only use 4 CPU cores and 4GB of memory, instead of all the memory space
- *One solution:* Run a VM that is allocated 4GB of memory and provided with 4 virtual CPUs
- **Why is this not a good solution?**

This realization gives rise to *OS-level virtualization*

- In many important cases (like previous examples), running a full-blown VM is an overkill
- The OS kernel can provide fine-grained mechanisms to control the system view (e.g., file system) and resources for a group of processes
- **What are the two famous OS-level virtualization techniques?**
  - **Change Root (or chroot)**
  - **Containers!**

Linux change root (or `chroot`) system call

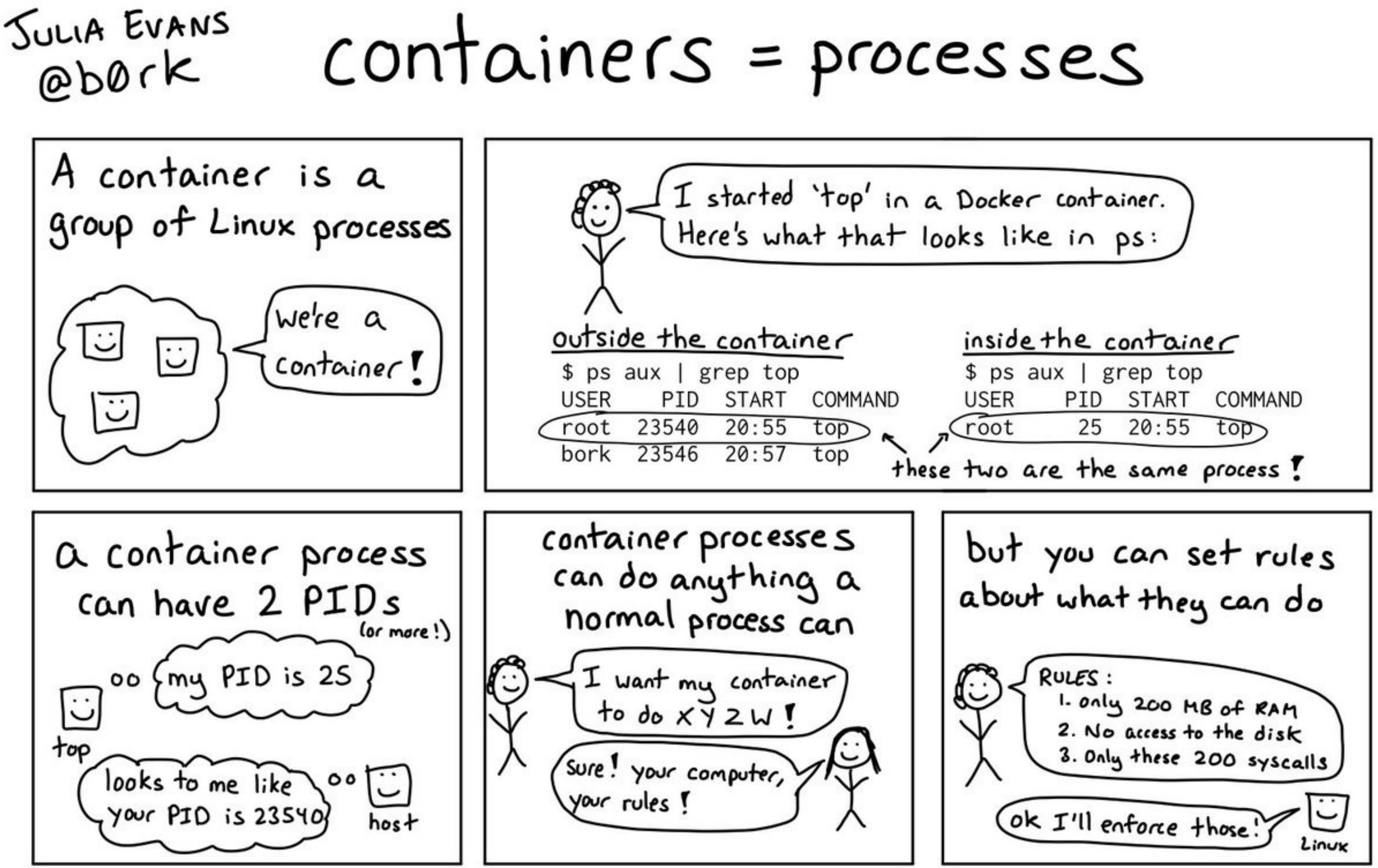


- New “root” filesystem for a certain process (and its children processes) restricting view of the file system and libraries (e.g., /lib)
- Can be used to solve the different/old library problem

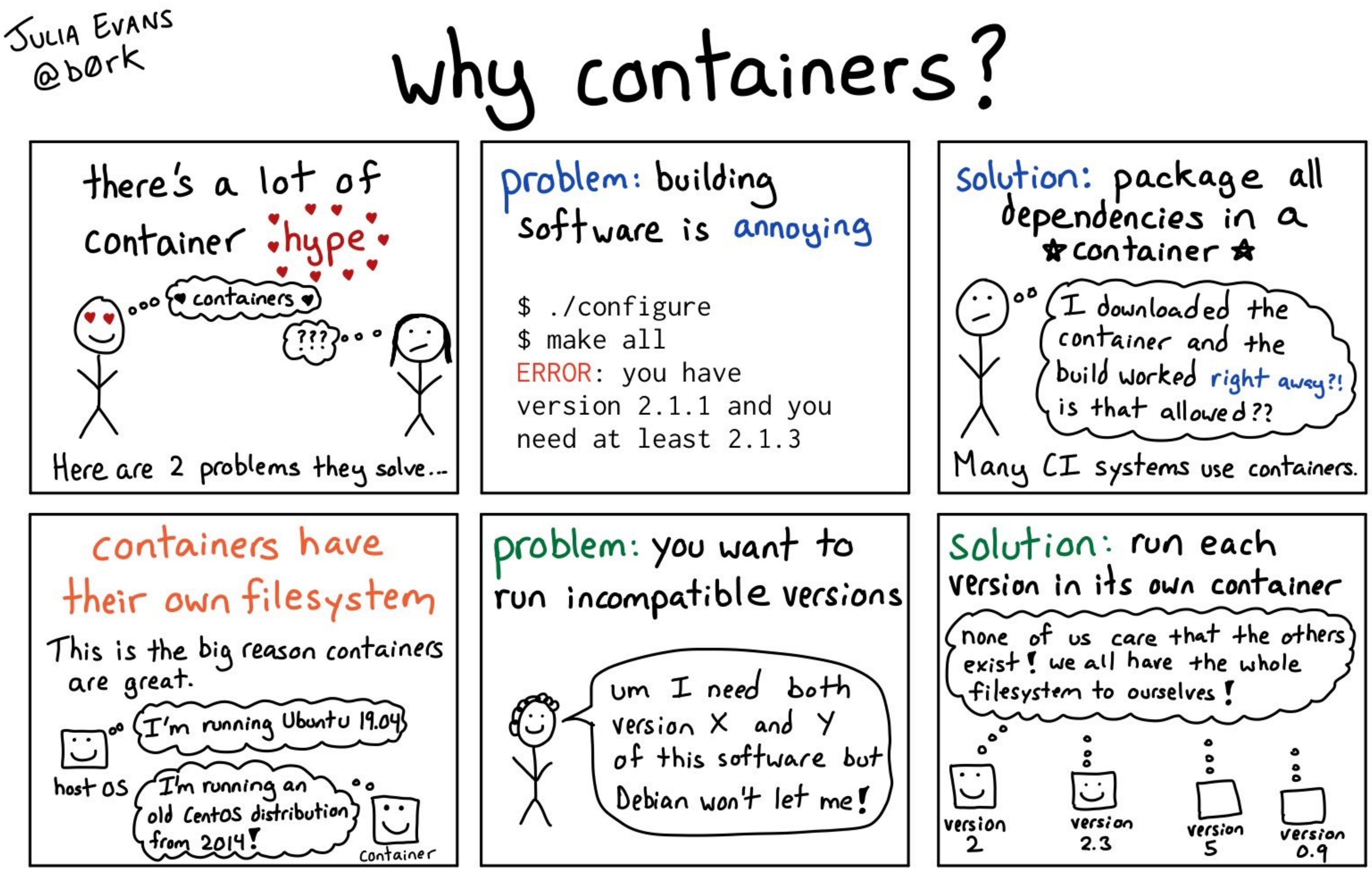
Containers are more advanced than `chroot`

- chroot alone does not provide required features like:
  - Hardware resource control and accounting
  - Software resource isolation
  - Ease of running different processes
  - Checkpoint and restore

- Containers are a Linux mechanism to add these aspects to a chroot-ed higher-level process and its children



https://x.com/b0rk/status/1224500774450929664/photo/1

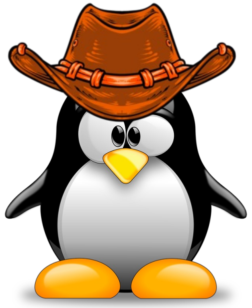


https://x.com/b0rk/status/1224500774450929664/photo/1

Building containers in Linux

- In principle, all the features (e.g., CPU scheduling limits) required to create containers are natively supported by any OS

- It is complex to use these low-level features, and it is useful to have more higher-level abstractions for containers
  - (in addition to chroot)



OS abstractions that make containers tick!

Abstraction #1: *cgroups* (or control groups)

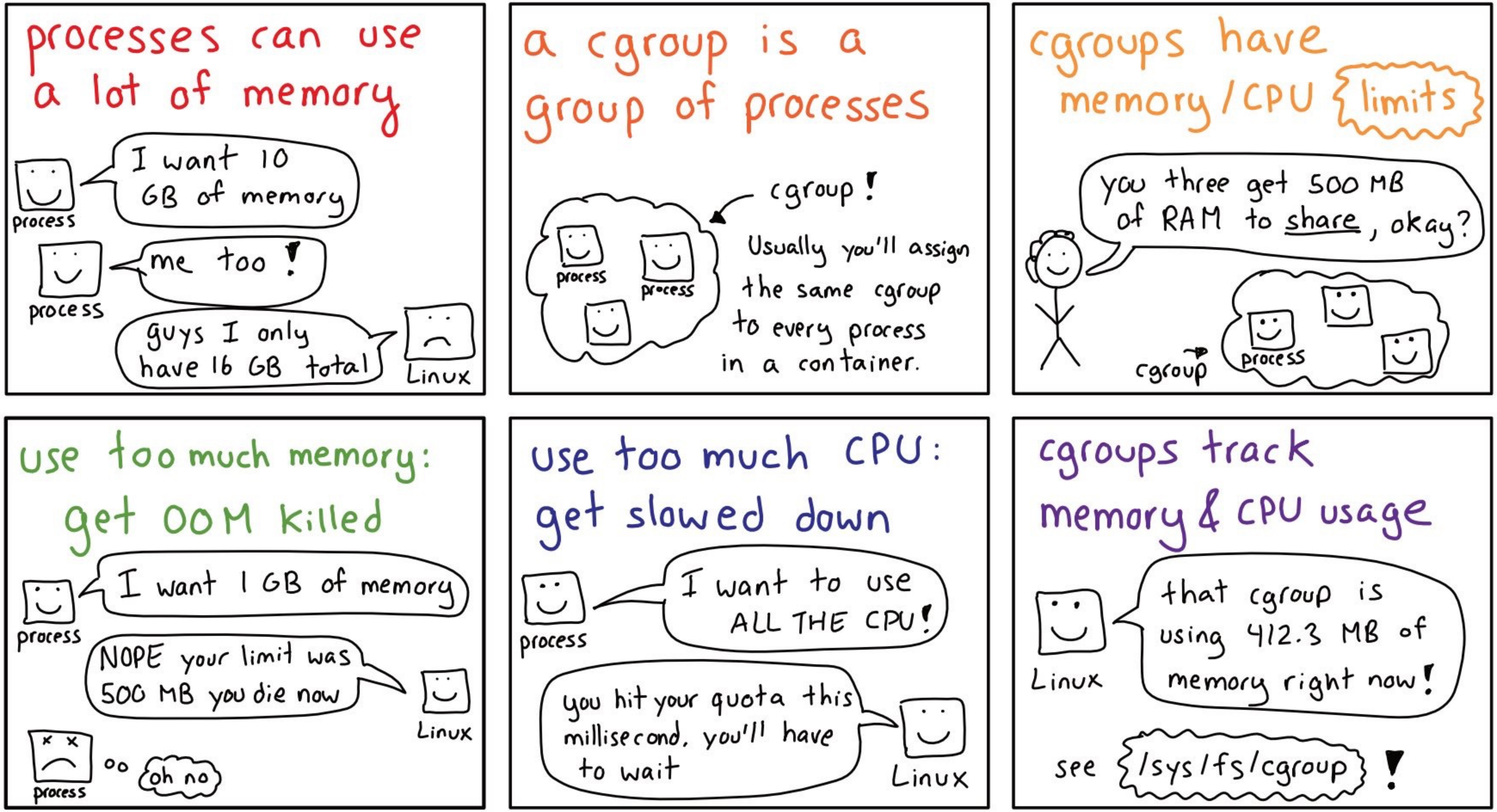
- Linux kernel feature that limits the **hardware resource usage** of a set of processes

- Resources can include CPUs, memory, network, and more

https://www.redhat.com/en/blog/world-domination-cgroups-rhel-8-welcome-cgroups-v2 https://x.com/b0rk/status/1214341831049252870

JULIA EVANS
@b0rk

cgroups



Abstraction #2: Kernel-enforced *namespaces*

- Provide a group of processes (i.e., a cgroup) its **own view of operating system resources**
  - Recall this was *hardware resources* for cgroups

- Namespaces can provide different view of aspects like:
  - Process identifiers (pids)
  - User identifiers
  - Network IP addresses

https://x.com/b0rk/status/1248263569629732865?lang=kn

JULIA EVANS
@b0rk

PID namespaces

the same process has different PIDs in different namespaces

PID in host	PID in container
23512	1
23513	4
23518	12

PID 1 is special

PID namespaces are in a tree

host PID namespace
(the root)

child   child   child

Often the tree is just 1 level deep (every child is a container)

you can see processes in child PID namespaces

aw look at all those containers running!

host

if PID 1 exits, everyone gets killed

ok I'm done!

PID 1

I'm kill -9'ing everyone else in this PID namespace IMMEDIATELY

Linux

Killing PID 1 accidentally would be bad

kill 1

container process

do you WANT everyone to die? I'm not gonna let you do that

Linux

rules for signaling PID 1

from same container:

only works if the process has set a signal handler

from the host:

only SIGKILL and SIGSTOP are ok, or if there's a signal handler

Are containers as *secure* or *isolated* as virtual machines?

- Containers implement several features to restrict the hardware and system resource view of a set of programs

- Despite the techniques, containers are still considered much less secure than virtual machines because of two reasons:

  - Lack hardware isolation mechanisms implemented for VMs (e.g., CPU/memory isolation using EPTs and IOMMU)

  - Have a very big communication interface (attack surface) between containers and the host OS kernel

Understanding communication channels + *attack surface*

- A **malicious process** that wants to compromise the kernel will typically execute system calls to trigger vulnerabilities

- Linux kernel has 350+ system calls;

  - Since Linux is monolithic, any of these system calls can be used to trigger vulnerabilities and compromise the (entire) kernel
    - *This is the attack surface*

  - We will look at some kernel vulnerabilities next class!

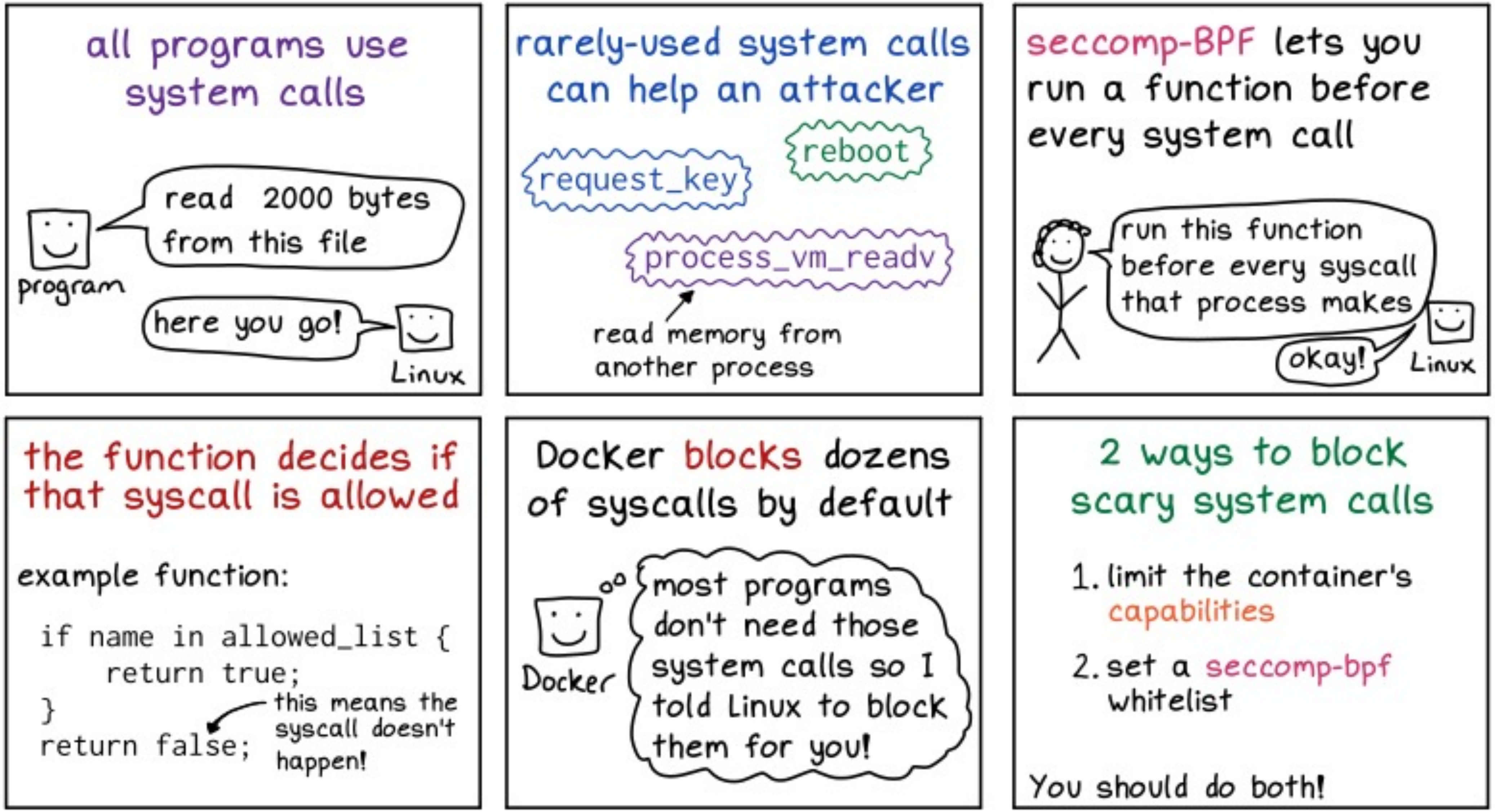
Abstraction #3: *seccomp* filters

- A bid to further restrict the attack surface of containerized programs

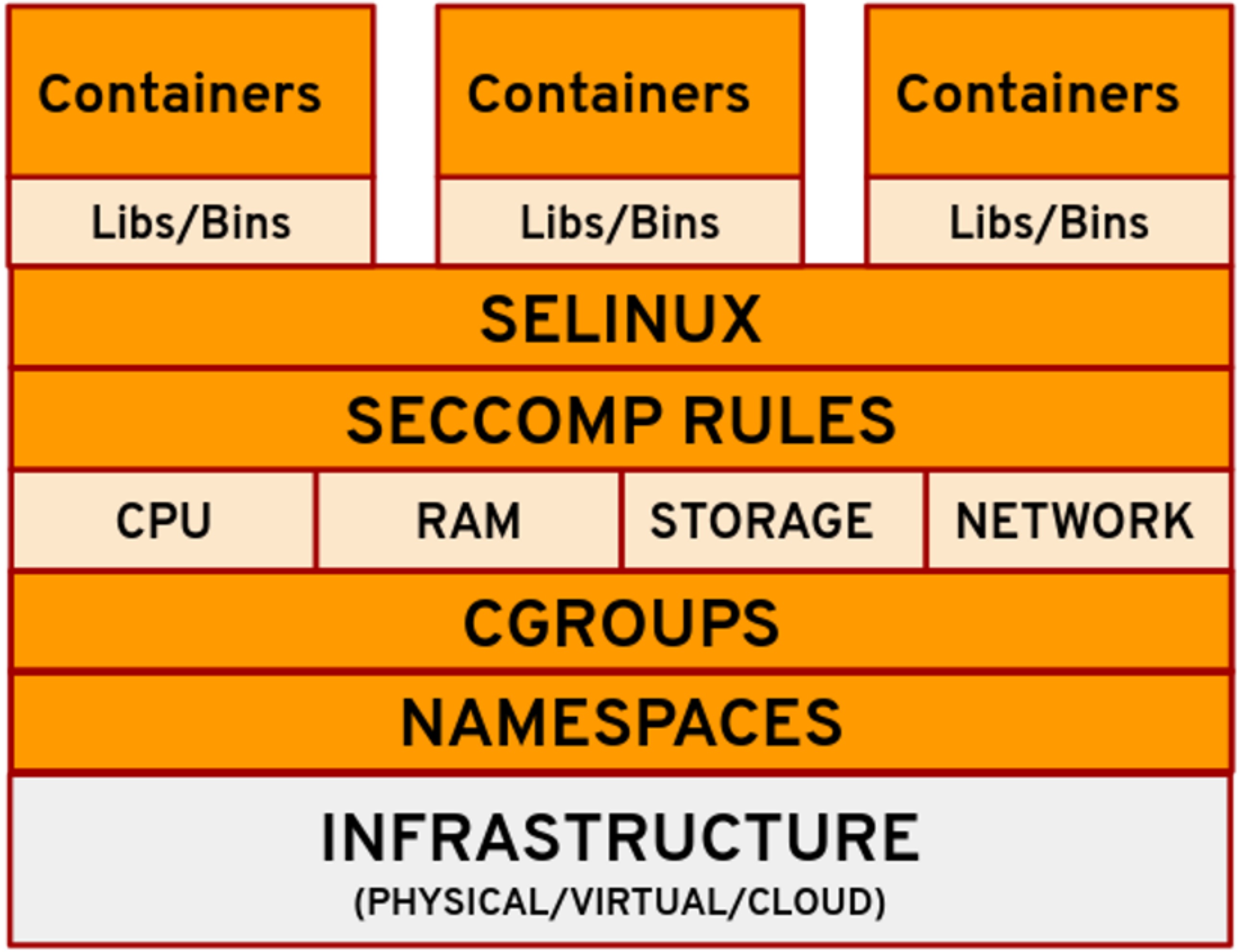
- *Idea:* only allow a specific set of system calls required by the program (e.g., related to files) to execute and restrict all the remaining calls

JULIA EVANS
@b0rk

seccomp-bpf



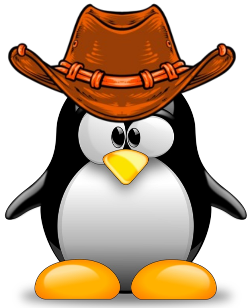
(*relatively!*) Full modern container illustration



https://opensource.com/article/21/8/container-linux-technology

Does seccomp-bpf make container attack surface as small as VMs?

- **Nope!**
- Imagine a **malicious process running inside a virtual machine** wants to compromise the host kernel
  - First compromise the guest kernel using system calls
    - *Can be further restricted using seccomp-bpf inside the VM*
  - Use hypervisor calls (or hypercalls) from the guest kernel to compromise host kernel
    - Linux KVM hypervisor → ~30 hypervisor calls
      - *Still a much smaller attack surface*



Good luck for your final exams!