COEN 241
Introduction to Cloud Computing

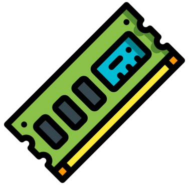
Lecture 5 - OS Virtualization

Lecture 4 Recap

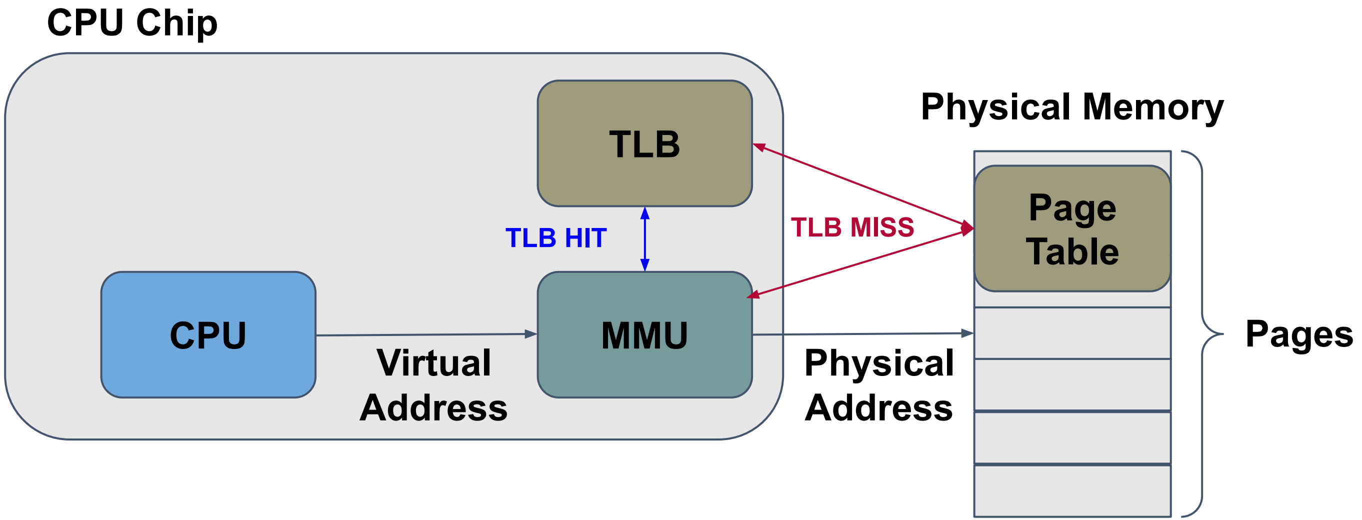
- Advanced Virtualization Concepts II
    - Memory Virtualization
    - Paravirtualization
- Vagrant

Another Virtualization Challenge: RAM

- RAM access patterns are surprisingly complex
- History: CPU’s address pins indicate word to read/write
  - e.g., MOS 6502 has 16 address wires, thus 64kB RAM (216 bytes)
- Early Intel 80x86 chips addressed offsets of ‘segments’
  - Thankfully segmented memory model has died off in x64
- Intel 80386 added **page-based memory mapping**



Page-based Memory Access

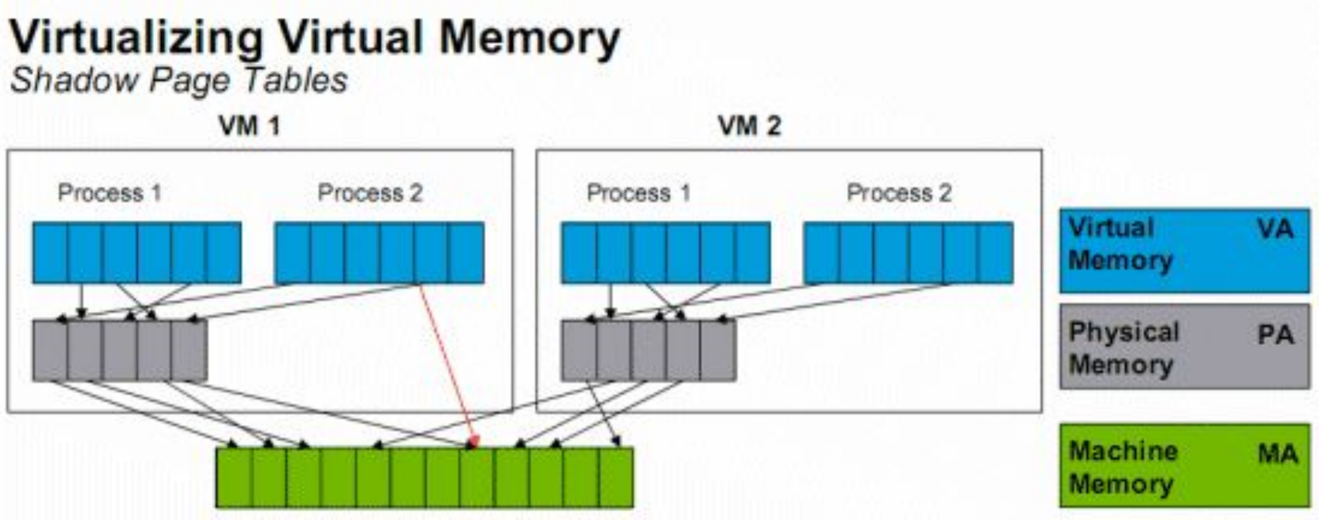


Challenges in Virtualizing Hardware Page Tables

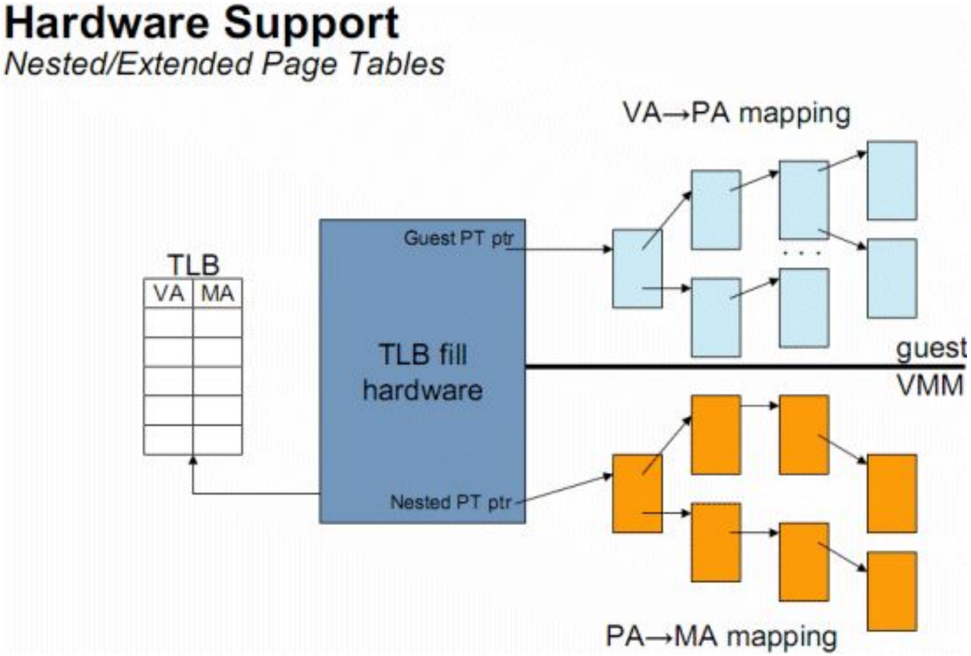
- Hypervisor has no chances to intercept on TLB misses

- Solutions
  - **Software-based**: Shadow Paging
  - **Hardware-based**: Hardware support for virtualizing memory

Virtualizing Paged Memory - Shadow Page Table



Virtualizing Paged Memory - Nested Paging

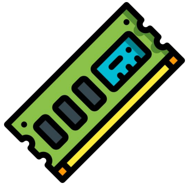


TLB Tagging

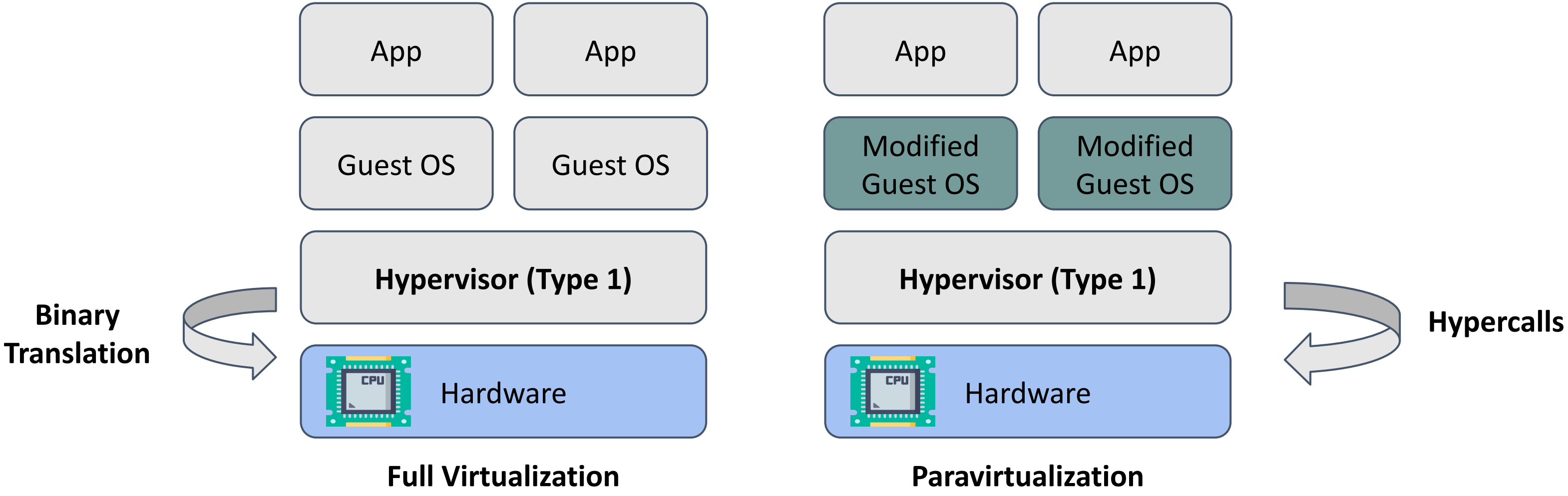
- Traditionally, every time a hypervisor switched between different VMs, the VM and its data structure had to be flushed out of the TLB

- Intel and AMD have facilitated TLB tagging since 2008
  - Intel Virtual Processor IDs (VPIDs) allow VMM to assign VM IDs

- Flush TLB entries of a particular VM only
  - So switching between VMs and VMM may leave TLB entries
  - Significant boost to memory access speed



Paravirtualization



Full Virtualization

Paravirtualization

Vagrant

- A tool for automatically creating and configuring VMs via simple scripts
- Operates on a **Vagrantfile** that contains instructions for how to download and run a particular VM image
- Requires a “provider” on the host machine to start and manage the VM
    - E.g., Virtualbox, QEMU, Hyper-V, ...

Agenda for Today

- OS Virtualization
  - No more hardware details!
  - Features enabling OS Virtualization
    - **Namespaces**
    - **Cgroups**
  - History of OS Virtualization
- Lot of Demos!
- Readings
  - Recommended: Demystifying Containers Part I
  - Optional:
    - https://web.archive.org/web/20140906153815/http://www.cs.bell-labs.com/sys/doc/names.html
    - https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html
    - https://www.redhat.com/sysadmin/pid-namespace

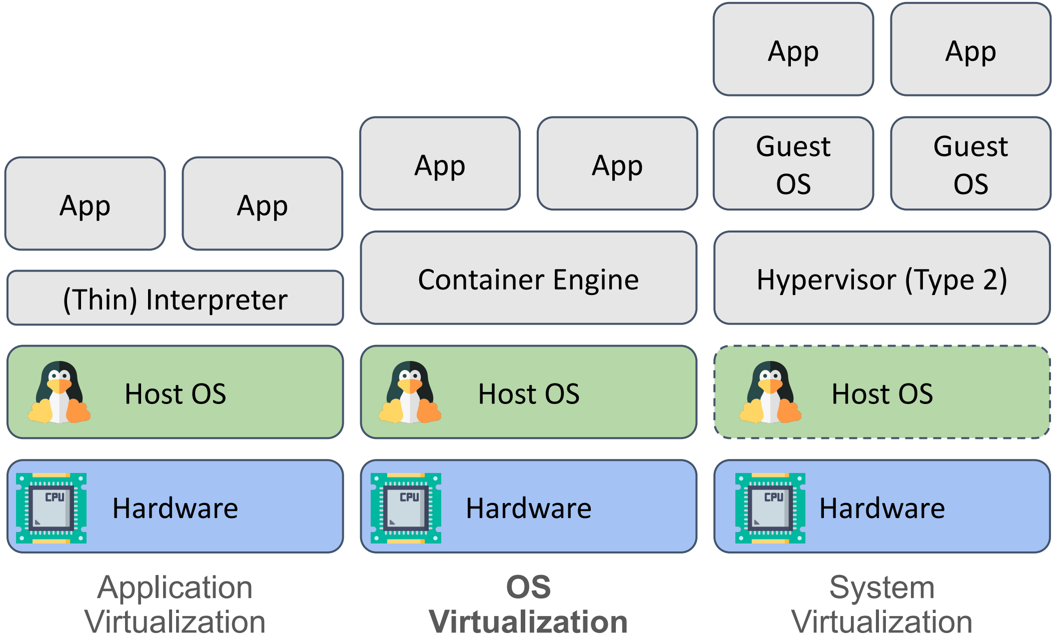
Agenda for Today

- Quiz 1 is out!
    - 10 questions T/F, MC and multiple answers
- Presentation date signup
    - Please submit ASAP! FCFS!
    - https://docs.google.com/spreadsheets/d/10x6tRuOoOK8TEUv3hZjCKSicSoRxcDEewkxTFiuka5w

OS Virtualization

Types of Virtualization

- ~~Application Virtualization~~
  - ~~E.g., JVM, PVM~~
- **OS Virtualization**
  - Containers
- ~~System Virtualization~~
  - ~~Virtual Machines~~
- Network Virtualization
- Storage Virtualization
- Many more...



Different Types of Software Isolation

- Application: In-application isolation
- Thread: Sharing memory within one process
- Process: Each process has its own address space
- **User space: All of the code in an OS that lives outside of the kernel**
- Virtual Machine: What we have covered so far

- **OS virtualization** is to **isolate separate user spaces**
  - Assuming everything below and including the Kernel is shared

Why Isolate User space?

- Less resource intensive if the kernel is secure
  - (i.e., cheaper than needing to run VMs containing OS kernels)
- One example: Android
  - Allocates a unique user ID for every app
  - Each application's processes are (assumed to be) isolated
- **Another example: Containers (Docker)**

Two Features enabling OS Virtualization

- **Namespace**
  - Allows different view of the system for different process
- **cgroup**
  - Controls and isolates the resource usage (CPU, memory, disk I/O, network, etc.) for different processes
- Both are Linux Kernel features that are crucial for OS virtualization
- Warning: Lot of OS details coming up

Namespace

Namespace Overview

- Definition
  - Enable a process (or several processes) to have different views of the system than other processes
- Inspired by Plan 9 Distributed System from Bell Labs
  - Part of optional reading
- Originated in 2002 with “mnt” namespaces
- Now there are 8 kinds of namespaces

Kinds of Namespaces

- **pid: Process IDs**
- **mnt: Mount points, filesystems**
- **net: Virtualizing the network stack**
- ipc: Interprocess communications
- uts: Unix Time Sharing for different hostname
- user: User identification
- **cgroup: Identity of the control group (will cover this in later slides)**
- time: Allow processes to see different times

Namespace Demo

- To query namespaces: **ls -al /proc/<pid>/ns**
- By default, all “native” processes are placed in the same default namespaces
- How to enable separate namespaces?

Namespace Implementation

- Three ways: three system calls are used for namespaces:

- `clone()`: creates a new process and a new namespace; the process is attached to the new namespace.
  - Process creation and process termination methods, fork() and exit() methods, were patched to handle the new namespace CLONE_NEW* flags.
- `unshare()`: does not create a new process; creates a new namespace and attaches the current process to it.
  - unshare() was added in 2005, but not for namespaces only, but also for security. see “new system call, unshare” : http://lwn.net/Articles/135266/
- `setns()`: a new system call for joining an existing namespace

Digression: What are System Calls?

- A way for programs to interact with the operating system
- Provides the services of the operating system to the user programs via APIs
- Services Provided by System Calls :
    - Process creation and management (`fork(), exit(), wait()`)
    - Main memory management (`mmap()`)
    - File Access, Directory and File system management (`open(), read(), write(), close()`)
    - Device handling(I/O) (`ioctl(), read(), write()`)
    - Protection (`chmod(), chown()`)
    - Networking (`socket()`) etc.
- Types of System Calls : There are 5 different categories of system calls
    - Process control: end, abort, create, terminate, allocate and free memory.
    - File management: create, open, close, delete, read file etc.
    - Device management
    - Information maintenance
    - Communication

Namespace Implementation Details

- To support namespace, a member named **`nsproxy`** was added to the process descriptor called **`struct task_struct`**
  - nsproxy includes various namespaces: uts_ns, ipc_ns, mnt_ns, pid_ns, net_ns;
  - https://elixir.bootlin.com/linux/latest/source/include/linux/nsproxy.h
  - https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h
- A method `task_nsproxy(struct task_struct *tsk)` used to access the nsproxy of a particular process
- There is an initial, default namespace for each namespace

Demo: PID Namespace

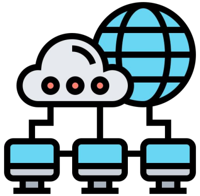
- Run the following commands

```
unshare --fork -p /bin/bash
sleep 8000 &
ps -ef | grep sleep
kill <pid>
lsns |grep bash
sudo cat /proc/<pid>/status |grep NSpid
```

- This create a new PID namespace by unshare() syscall and call execvp() for invoking bash
- Note that you can’t kill other processes in different PIDs
- Building block for Container processes

Network Namespaces

- A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices.
- Network namespace includes all network stack information, like:
  - Loopback device
  - SNMP stats (netns_mib)
  - All network tables: routing, neighbors, etc
  - All sockets – /procfs and /sysfs entries
- Building block for Container networking



Demo: Network Namespaces

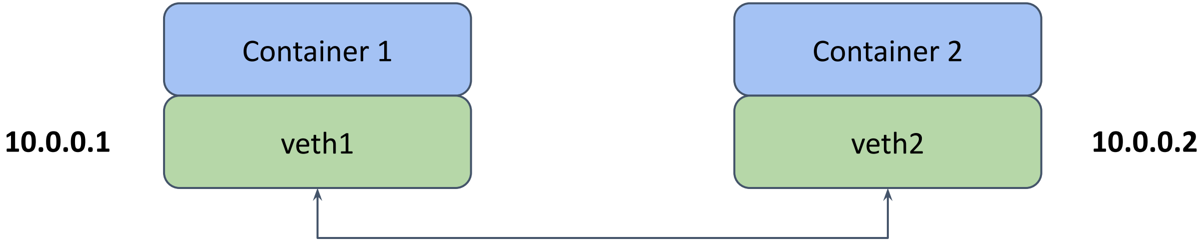
- Create two namespaces, `sfo` & `nyc`:
  - `ip netns add sfo`
  - `ip netns add nyc`
  - `ip netns list`

- Verify that two network namespaces are created:
  - `/var/run/netns/sfo`
  - `/var/run/netns/nyc`

- Which syscall is involved here?
  - `clone(), unshare() or setns()?`

Connecting Two Interfaces

- You can communicate between two network namespaces by:
  - Creating a pair of network devices (veth) and move one to another network namespace.
  - veth (Virtual Ethernet) is like a pipe



Demo: Connecting Two Interfaces

- Create a veth pair

```
ip link add veth-sfo type veth peer name veth-nyc
ip link list | grep veth
```

- Assign them to different network namespace:

```
ip link set veth-sfo netns sfo
ip link set veth-nyc netns nyc
```

- Run two processes associated with these two namespaces

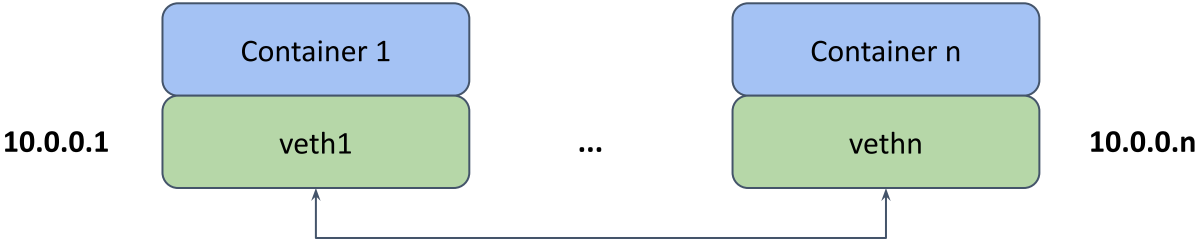
```
ip netns exec sfo ip link
ip netns exec nyc ip link
```

Demo: Connecting Two Interfaces

- Give the interfaces IP addresses
  - `ip netns exec sfo ip address add 10.0.0.1/24 dev veth-sfo`
  - `ip netns exec sfo ip link set veth-sfo up`
  - `ip netns exec nyc ip address add 10.0.0.2/24 dev veth-nyc`
  - `ip netns exec nyc ip link set veth-nyc up`
- Now check the IP addresses
  - `ip netns exec sfo ip addr`
  - `ip netns exec nyc ip addr`
- Now try to ping one another
  - `ip netns exec sfo ping 10.0.0.2`
  - `ip netns exec nyc ping 10.0.0.1`
- Clean up namespaces
  - `ip netns del nyc sfo`

Food for Thought: More Than 2 Containers?

- How to connect multiple containers?
- Our demo was for 1 to 1 network connection
- Hint: Use Linux Bridge



Mount (Filesystem) Namespaces

- Represented by `mnt_ns` (mnt_namespace object) in `nsproxy`
- For Linux, all previous mounts will be visible (inherited) in the new mount namespace, but...
  - mounts/unmounts in the new mount namespace are invisible to the rest of the system
- To create a new mount namespace
  - unshare -m /bin/bash
- How to specify a new root file system to a process
  - **chroot**: relink the root directory of the process to a new root directory (i.e., which includes a complete new file system of a container)

“Old School” Chroot Jail

- Unix servers have to handle users that may be malign
    - Common historical example was running public FTP servers
        - Anonymous users could log into those servers
        - FTP as a protocol allows quite a lot of power over the server
        - Needed to cut down what anonymous users could do
- Solution: change the perceived root directory of the filesystem
    - i.e., a ‘chroot jail’: changes the set of available executables
    - The process and its children can’t access the files outside the new root.
    - Unix accesses binaries from /bin, libraries from /lib, etc.
    - Changing the meaning of / mitigates many vulnerabilities

BSD Jails: OS-level virtualization since Y2K

- BSD Jail takes resource partitioning beyond the filesystem
    - Isolates process IDs, root user, network, device access
    - Also uses a chroot jail to effect filesystem isolation
- Helps avoiding privilege escalation
    - Successful break in to server can’t scan filesystem for vulnerabilities, e.g., reading /etc/shadow and trying to crack weak passwords
- Many operations are blocked within BSD Jails
    - E.g., loading kernel modules, changing network interfaces, mounting and unmounting filesystems, etc.

Cgroup (Control Group)

Linux Cgroup

- This work was started by engineers at Google in 2006 under the name "process containers; in 2007, renamed to “Control Groups”.

- Defines parameters about the resource use of a set of processes, e.g.:
  - Limit total memory available to group of processes
  - Indicate non-even share of device input/output priority
  - Affect CPU scheduling to the group
  - cgroups also can assist accounting for resource use
  - cgroups can be applied hierarchically

- cgroups can facilitate starting / stopping processes
  - Important for snapshot functionality

Cgroup Subsystems

- Kernel modules that are used to control the access that cgroups have to various system resources
- Example 10 cgroup subsystems for Redhat
  - `blkio` — sets limits on input/output access to and from block devices such as physical drives
  - `cpu` — uses the scheduler to provide cgroup tasks access to the CPU.
  - `cpuacct` — generates automatic reports on CPU resources used by tasks in a cgroup.
  - `cpuset` — assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
  - `devices` — allows or denies access to devices by tasks in a cgroup.
  - `freezer` — suspends or resumes tasks in a cgroup.
  - `memory` — sets limits on memory use by tasks in a cgroup and generates reports on memory usage.
  - `net_cls` — tags network packets with a class identifier (classid) that allows the Linux traffic controller (`tc`) to identify packets originating from a particular cgroup task.
  - `net_prio` — provides a way to dynamically set the priority of network traffic per network interface.
  - `ns` — the *namespace* subsystem.
  - `perf_event` — identifies cgroup membership of tasks and can be used for performance analysis

Cgroup Virtual File System (VFS)

- Cgroups uses a Virtual File System
- All entries created in it are not persistent and deleted after reboot.
- All cgroups actions are performed via file system actions (create/remove directory, reading/writing to files in it, mounting/mount options).

Mounting Cgroups

- In order to use a filesystem, it must be mounted
- A control group can be mounted anywhere on the filesystem. (e.g., Systemd uses /sys/fs/cgroup.)
- When mounting, we can specify with mount options (-o) which subsystems we want to use
  - mkdir /cgroup/memtest
  - mount -t cgroup -o memory test /cgroup/memtest/

Mounting Cgroups

- Under each new cgroup which is created, some common files are always created
  - tasks: list of pids which are attached to this group.
  - cgroup.procs.: list of thread group IDs (listed by TGID) attached to this group.

- Each subsystem adds specific control files for its own needs
  - memory.max_usage_in_bytes
  - memory.limit_in_bytes
  - memory.kmem.tcp.limit_in_bytes
  - memory.kmem.tcp.max_usage_in_bytes
  - …

Cgroup Demo: cpuset

- `cpusets` provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks

- Creating a cpuset group is done with:

  ```
  mkdir /sys/fs/cgroup/cpuset/group1
  echo 0 > /sys/fs/cgroup/cpuset/group1/cpuset.cpus
  echo 0 > /sys/fs/cgroup/cpuset/group1/cpuset.mems
  echo #pid > /sys/fs/cgroup/cpuset/group1/tasks
  ```

Cgroup Demo: Memory

- `mkdir /sys/fs/cgroup/memory/group1`
- `echo #pid > /sys/fs/cgroup/memory/group1/tasks`
- `echo 10M > /sys/fs/cgroup/memory/group1/memory.limit_in_bytes`

- What would happen if you run a process demanding more than 10 M memory?

- Optional Reading
  - https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory

Summary

- Namespaces and Cgroups are building blocks for containers
- The container runtime is responsible for
  - Creating namespaces
  - Creating cgroups
  - Executing a process within the given namespace and cgroups, which makes it a container!

Agenda for Today

- OS Virtualization
  - No more hardware details!
  - Features enabling OS Virtualization
    - **Namespaces**
    - **Cgroups**
  - History of OS Virtualization
- Lot of Demos!
- Readings
  - Recommended: Demystifying Containers Part I
  - Optional:
    - https://web.archive.org/web/20140906153815/http://www.cs.bell-labs.com/sys/doc/names.html
    - https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html
    - https://www.redhat.com/sysadmin/pid-namespace

TODOs!

- HW 1 Will be released next class
- Project
- Quiz 1

Questions?