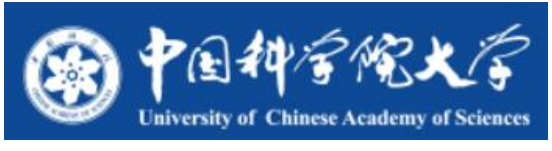
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

轻量级隔离

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



容器

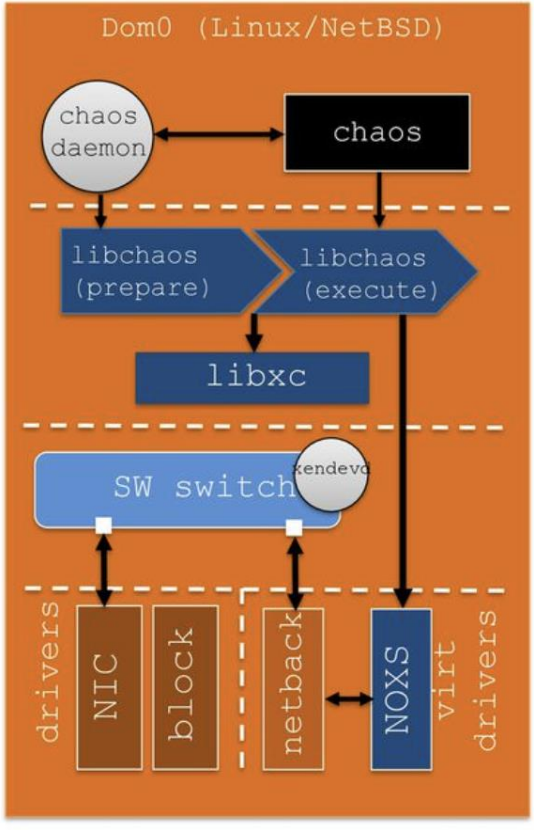
Hypervisor

- **Bare Metal Hypervisors**
  - VMWare ESXi
  - Hyper-V
  - Proxmox
  - Xen
- **Hosted Hypervisors**
  - VirtualBox
  - VMWare Workstation
  - UTM (uses qemu under the hood)
  - qemu (also supports emulation)
  - Hyper-V
  - Parallels

Containers

- **Light(-er) weight**
  - Allows them to be easily distributed
  - Rather than virtualizing the entire OS, it continues to use the host's **kernel**/operating system as a "base" to service whatever is running within the container.
- **Faster than VMs (usually) (?)**
  - Can also use an emulated system if necessary → runs within a VM
- **Designed for ephemerality**
  - Containers are "disposable" – any long-term data should be stored in separate persistent "volumes"

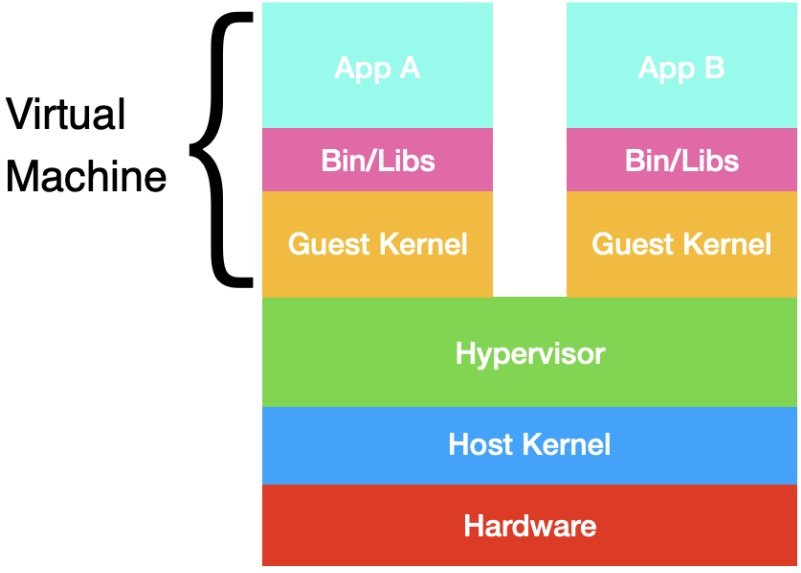
Faster than VMs (usually) (?)



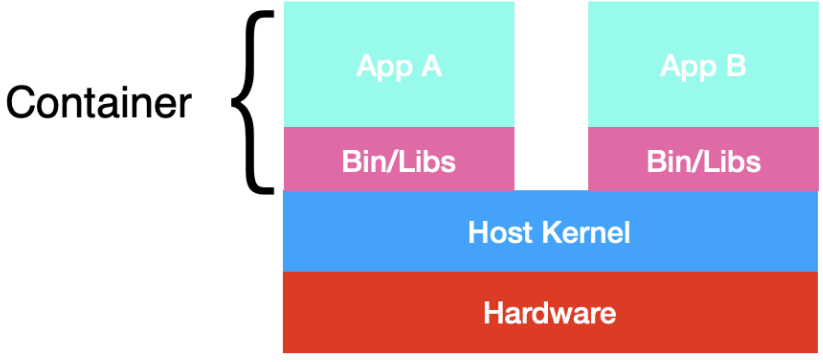
1. **Chaos** – toolstack optimized for paravirtualized guests
2. **Split functionality**
3. **Noxs** - no XenStore

My VM is Lighter (and Safer) than your Container [SOSP'17]

What is a container



- Containers shares the host kernel, but have their own system binaries and libraries



Containerization

- **You can take an application and wrap it in a container to ensure a consistent running environment.**
  - You can define the **operating system it expects** to use (but not the kernel)
  - You can define the **CPU architecture the program expects** (and if the CPU architecture differs from the host, it will have to run within a virtual machine)
  - You can define **dependencies and other programs that the application expects to be installed**
  - You can define **the "hard drive" layout the program expects**
  - All this, and the application gets a level of **isolation** from other applications on the system.

docker

- **docker is one of the most popular tools to create and manage containers. Here's some useful-to-know terms:**
  - **Containers** are an ephemeral object representing a copy of a program, based on an Image
  - Images are built from **Dockerfiles**, and represent a frozen copy of an application and everything needed to run it
  - **Dockerfiles** are special scripts that are used to build images, including:
    - Instructions on adding files (e.g. program files) from your local system
    - Instructions on adding dependencies
  - **Volumes** store persistent data even past the lifetime of a container.

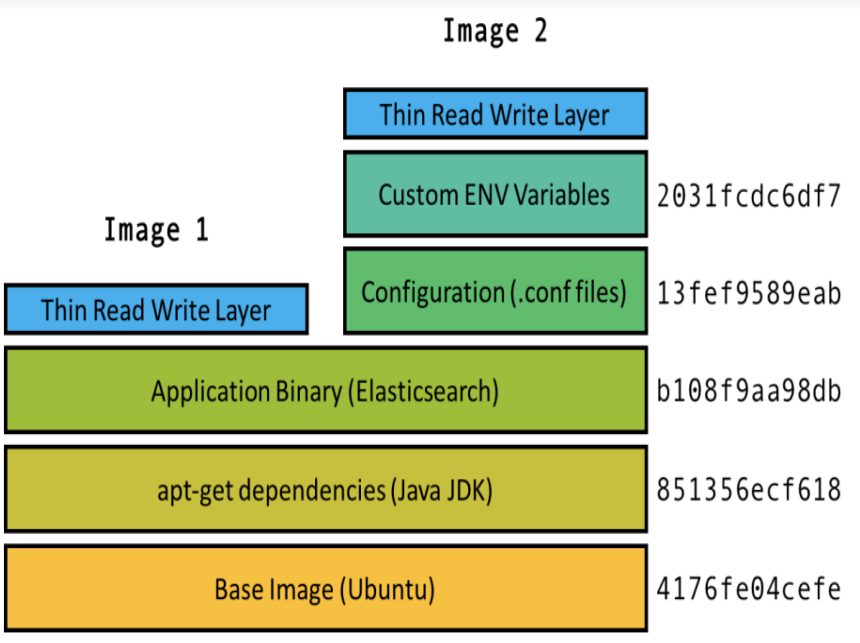
Images

- **A compressed collection of the libraries, binaries, and applications that make up a container**
- **Defined by a file called the Dockerfile**
  - Very similar to a script and sets up the container environment
  - Docker than compresses the environment into a binary
- **Docker offers tools and repositories (container registry) to build, store, and manage these images.**

```
FROM ubuntu:latest

MAINTAINER Edward Elric

RUN apt-get update -y

RUN apt-get install -y python-pip
python-dev build-essential

RUN pip install uwsgi flask

COPY transmute.py ~/transmute.py

WORKDIR ~/

RUN echo “1 soul” > sacrifice.txt

CMD [“python”, “transmute.py”]
```

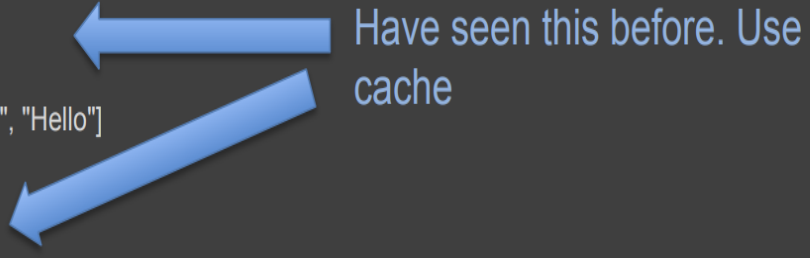
Images

- The image is divided into a sequence of layers. Each command creates a new layer on top of the previous layers.
- Images with common commands will have common layers whose binaries they share. This is the *union file system*.



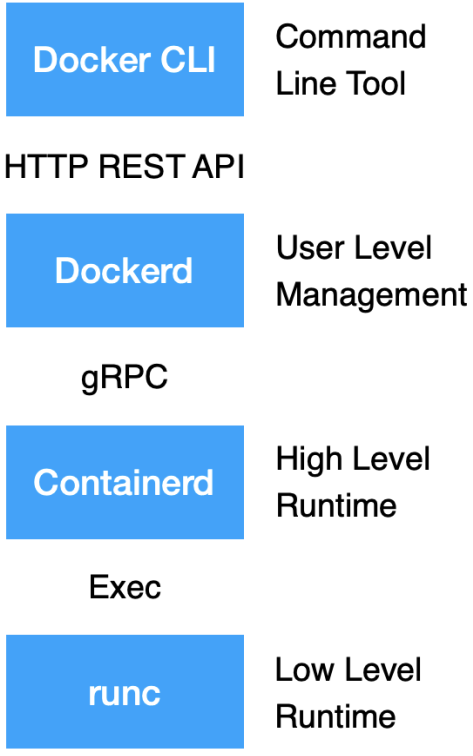
Why layers

```
> docker build -t hello_world_cmd -f Dockerfile_cmd .
Sending build context to Docker daemon   34.3kB
Step 1/4 : FROM ubuntu:latest
 ---> 4e2eef94cd6b
Step 2/4 : RUN apt-get update
 ---> Using cache
 ---> cfc0c414a914
Step 3/4 : ENTRYPOINT ["/bin/echo", "Hello"]
 ---> Using cache
 ---> 7e4f8b0774de
Step 4/4 : CMD ["world"]
 ---> Using cache
 ---> a89172ee2876
Successfully built a89172ee2876
Successfully tagged hello_world_cmd:latest
```



Docker Architecture

- **Dockerd (docker daemon) responds to commands from Docker CLI**
  - It handles management of docker objects (images, containers, volumes, networks, etc)
- **Dockerd uses Containerd to manage the container lifecycle**
  - It handles image push / pull, namespace management, etc
- **Containerd invokes a low level container runtime (runc) to create the container**



Container Runtime

- **A library that is responsible for starting and managing containers.**
  - Takes in a root file system for the container and a configuration of the isolation configurations
  - Creates the cgroup and sets resource limitations
  - Unshare to move to own namespaces
  - Sets up the root file system for the cgroup with chroot
  - Running commands in the cgroup
- **runc by Docker, rkt by CoreOS, gvisor (runsc) by Google, LXC by Google / IBM**

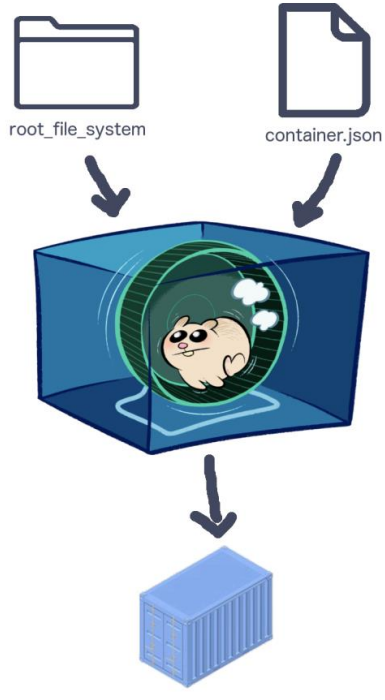


Image source: Cameron Lonsdale, runc

Create a Container

cgcreate -g <controllers>:<path>

1) Create the cgroup with cpu and memory controllers and path = UUID

```
$ UUID = $(uuidgen)
$ cgcreate -g cpu,memory:$UUID
$ cgset -r memory.limit_in_bytes = 100000000 $UUID
$ cgset -r cpu.cfs_period_us = 1000000 $UUID
$ cgset -r cpu.cfs_quota_us = 2000000 $UUID
$ cgexec -g cpu,memory:$UUID \
>   unshare -uinpUrf –mount-proc && \
>   sh -c "/bin/hostname $UUID && chroot $ROOTFS $CMD”
```

Create a Container

2) Set resource limitations for created group

```
$ UUID = $(uuidgen)
$ cgcreate -g cpu,memory:$UUID
$ cgset -r memory.limit_in_bytes = 100000000 $UUID
$ cgset -r cpu.cfs_period_us = 1000000 $UUID
$ cgset -r cpu.cfs_quota_us = 2000000 $UUID
$ cgexec -g cpu,memory:$UUID \
>   unshare -uinpUrf –mount-proc && \
>   sh -c "/bin/hostname $UUID && chroot $ROOTFS $CMD”
```

Create a Container

unshare [options] [*program* [*arguments*]]

3) Unshare the indicated namespaces that were inherited from the parent process, then execute arguments

```
$ cgcreate -g cpu,memory:$UUID
$ cgset -r memory.limit_in_bytes = 100000000 $UUID
$ cgset -r cpu.cfs_period_us = 1000000 $UUID
$ cgset -r cpu.cfs_quota_us = 2000000 $UUID
$ cgexec -g cpu,memory:$UUID \
>   unshare -uinpUrf –mount-proc \
>   sh -c "/bin/hostname $UUID && chroot $ROOTFS $CMD”
```

Create a Container

hostname [-fs] [*name-of-host*]

4) Change the cgroup's hostname to the UUID

```
$ UUID = $(uuidgen)
$ cgcreate -g cpu,memory:$UUID
$ cgset -r memory.limit_in_bytes = 100000000 $UUID
$ cgset -r cpu.cfs_period_us = 1000000 $UUID
$ cgset -r cpu.cfs_quota_us = 2000000 $UUID
$ cgexec -g cpu,memory:$UUID \
>   unshare -uinpUrf –mount-proc && \
>   sh -c "hostname $UUID && chroot $ROOTFS $CMD”
```

Create a Container

chroot new-root [*program* [*arguments*]]

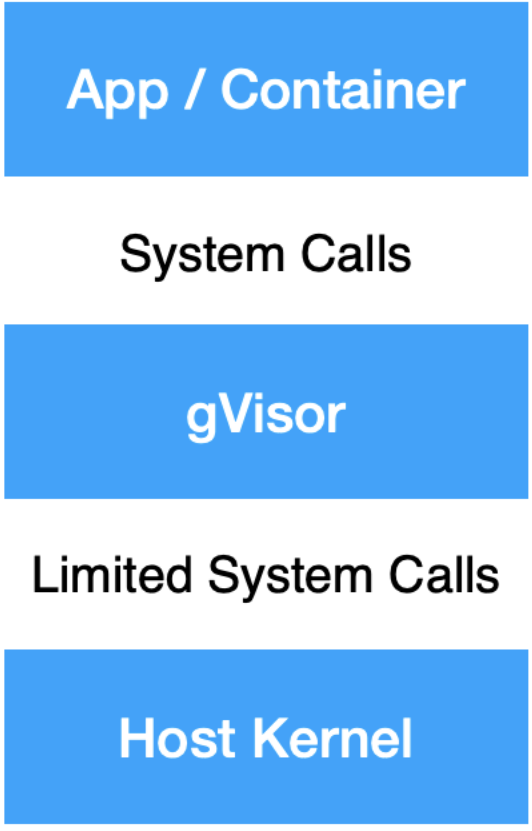
5) Change the cgroup's root to be the subdirectory at new-root, then execute arguments

"chroot /Users/cs162 /bin/ls" will change the root to /Users/cs162 and then execute /bin/ls which is actually /Users/cs162/bin/ls

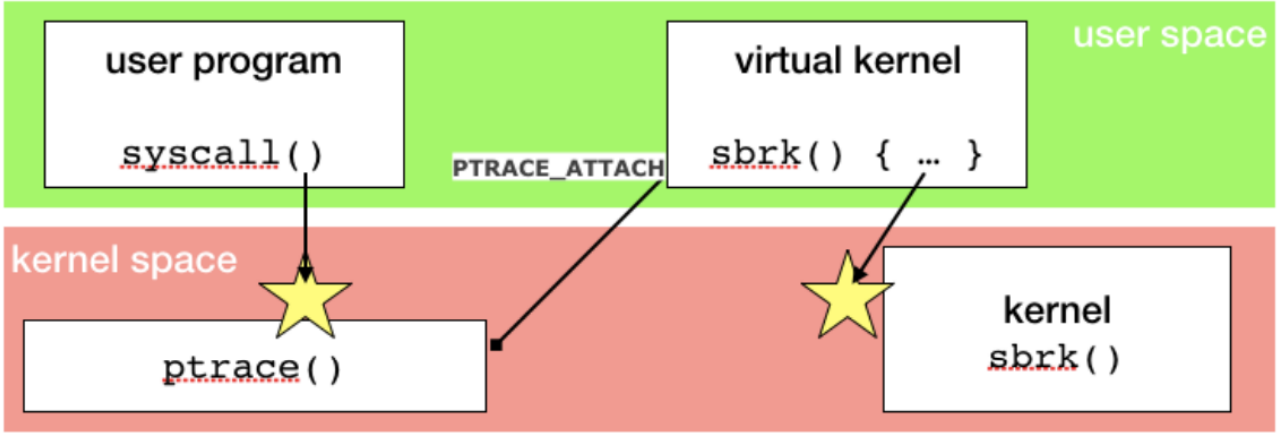
```
$ cgcreate -g cpu,memory:$UUID
$ cgset -r memory.limit_in_bytes = 100000000 $UUID
$ cgset -r cpu.cfs_period_us = 1000000 $UUID
$ cgset -r cpu.cfs_quota_us = 2000000 $UUID
$ cgexec -g cpu,memory:$UUID \
>   unshare -uinpUrf -mount-proc && \
>   sh -c "/bin/hostname $UUID && chroot $ROOTFS $CMD"
```

gvisor

- **A new kind of low level container runtime.**
- **In addition to creating the container, it also sandboxes the container as it runs.**
  - Conceptually similar to a user space microkernel
  - A user space kernel that implements all the Linux syscalls
  - Intercepts all syscalls the container makes and performs them in the secure user space kernel instead of in the actual kernel
  - Defends against privilege escalation attacks and provides stronger container isolation
  - Written in Go for memory and type safety
- **Great for running untrusted applications, i.e. cloud serverless applications**



gvisor



- **Intercepts syscalls through ptrace**
  - Linux syscall that allows one process to “control” another
  - gdb uses it to step through instructors, gVisor to intercept syscalls
- **Can also use kernel based virtual machines (KVM)**
  - Experimental feature that requires hardware support
  - Pushes the idea of what is considered a container and what is a virtual machine

容器怎样做到更轻量的隔离

- **多用户机制**
  - 如：Windows Server允许多个用户，同时远程使用桌面
  - 多个用户可以共享一个操作系统，同时进行不同的工作
- **缺点：多个用户之间缺少隔离**
  - 例如：所有用户共同操作一个文件系统根目录
- **如何想让每个用户看到的文件系统视图不同?**
  - 对每个用户可访问的文件系统做隔离

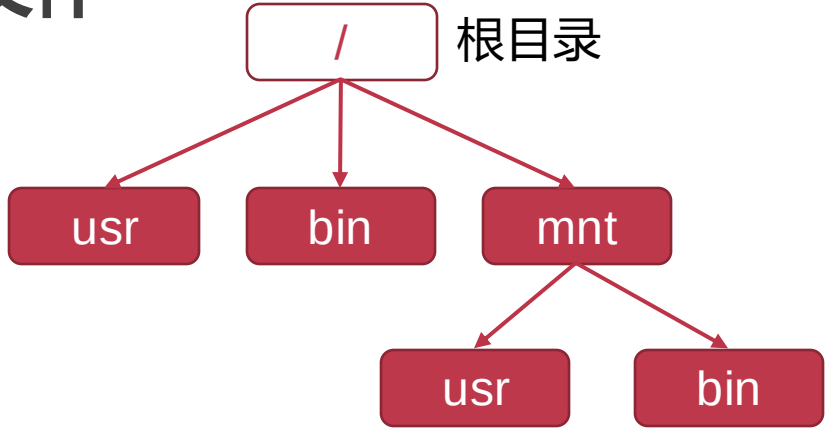
CHROOT

文件系统视图的隔离

- **为每个执行环境提供单独的文件系统视图**
- **原理**
  - Unix系统中的“一切皆文件”设计理念
  - 对于用户态来说，文件系统相当重要
- **方法**
  - 改变文件系统的根目录，即chroot

Chroot效果

- 控制进程能够访问哪些目录子树
- 改变进程所属的根目录
- 进程只能看到根目录下属的文件



Chroot原理

- **进程只能从根目录向下开始查找文件**
  - 操作系统内部修改了根目录的位置
- **一个简单的设计**
  - 内核为每个用户记录一个根目录路径
  - 进程打开文件时内核从该用户的根目录开始查找
- **上述设计有什么问题?**
  - 遇到类似“..”的路径会发生什么?
  - 一个用户想要使不同进程有不同的根目录怎么办?

Chroot在Linux中的实现

- **特殊检查根目录下的“..”**
  - 使得“/..”与“/”等价
  - 无法通过“..”打破隔离
- **每个TCB都指向一个root目录**
  - 一个用户可以对多个进程chroot

```
struct fs_struct{
    .......
    struct path root, pwd;
};

struct task_struct{
    .......
    struct fs_struct  *fs;
    .......
};
```

正确使用Chroot

- **需要root权限才能变更根目录**
  - 也意味着chroot无法限制root用户
- **确保chroot有效**
  - 使用setuid消除目标进程的root权限

```
chdir(“jail”);
chroot(“.”);
setuid(UID); // UID > 0
```

基于Name Space的限制

- **通过文件系统的name space来限制用户**
  - 如果用户直接通过inode访问，则可绕过
  - 不允许用户直接用inode访问文件
- **其它层也可以限制用户**
  - 例如：inode层可以限制用户

Chroot能否实现彻底的隔离?

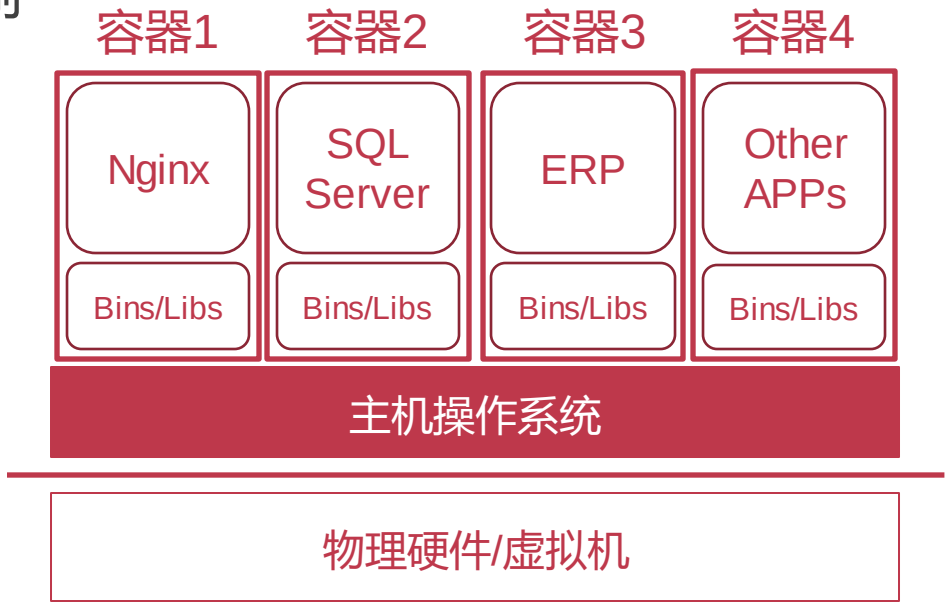
- **不同的执行环境想要共享一些文件怎么办?**
- **涉及到网络服务时会发生什么?**
  - 所有执行环境共用一个IP地址，所以无法区分许多服务
- **执行环境需要root权限该怎么办?**
  - 全局只有一个root用户，所以不同执行环境间可能相互影响

- **不能，因为还有许多资源被共享…**

LINUX CONTAINER

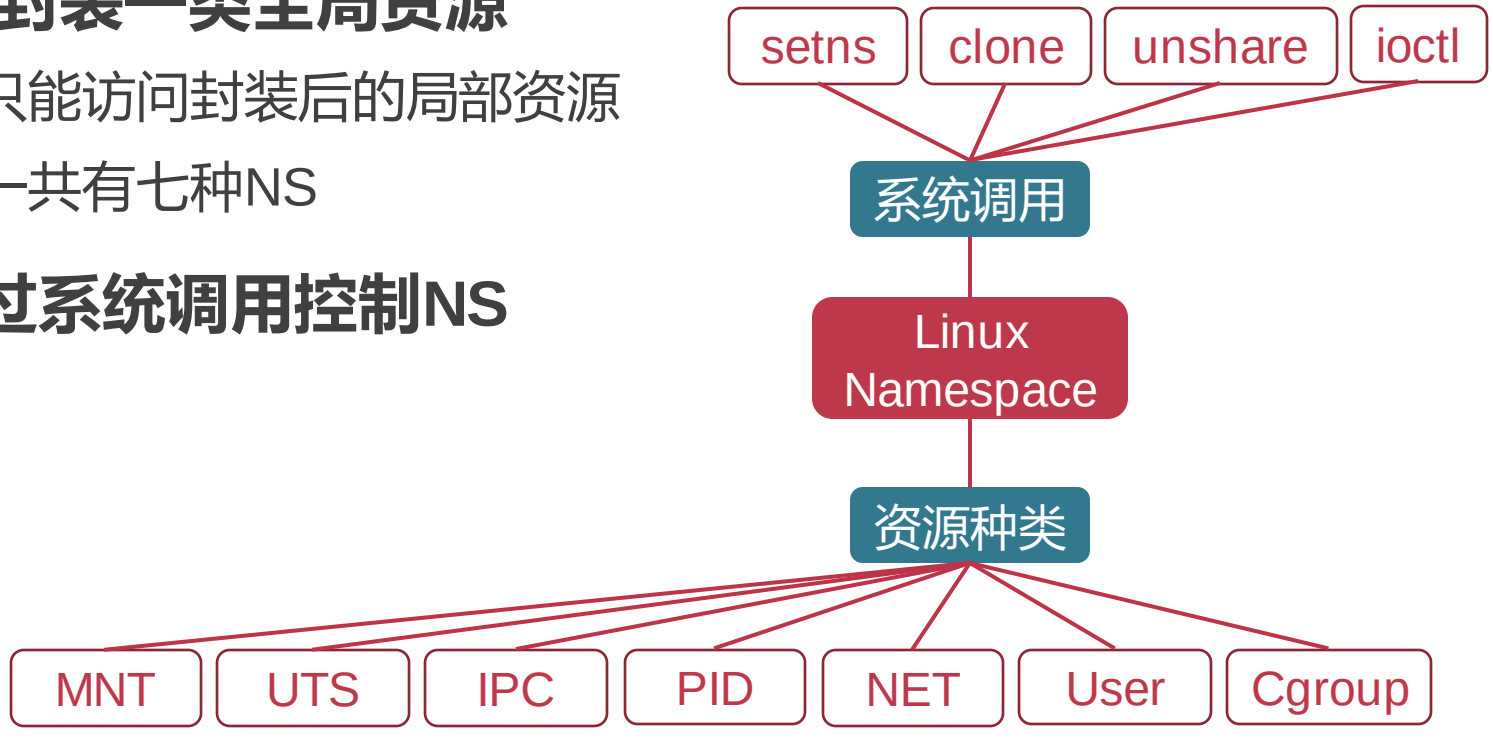
LinuX Container (LXC)

- **基于容器的轻量级虚拟化方案**
  - 由Linux内核提供资源隔离机制
- **安全隔离**
  - 基于*namespace*机制
- **性能隔离**
  - Linux cgroup



Linux Namespace (NS)

- **每种NS封装一类全局资源**
  - 进程只能访问封装后的局部资源
  - 目前一共有七种NS
- **进程通过系统调用控制NS**



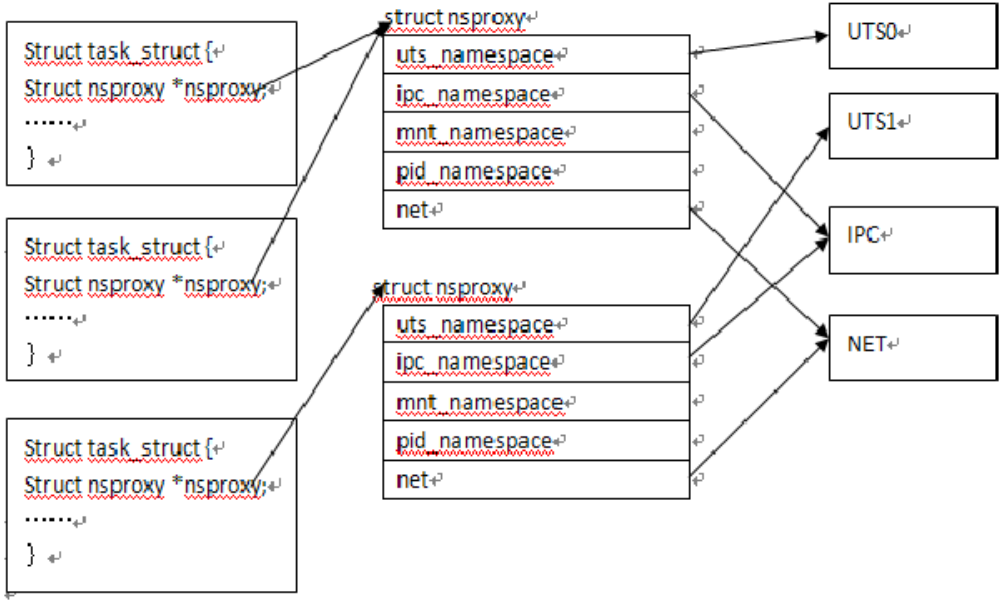
进程的命名空间

- **目的：Namespace（命名空间），可以让每个进程组具有独立的PID、IPC和网络空间。**

- **主机的虚拟化框架能在同一物理机器上实现提供不同的用户操作系统视图**
  - 使用vmware虽然也可以提供一种解决方案，但是资源分配不是很好，而且是在以一种ＯＳ为基础,在其上模拟实现另一种OS，
- **命名空间为系统的虚拟化提供了不同的实现思路，一台主机可以同时运行多个内核，即同时运行多个操作系统（同种操作系统）**
  - 全局系统资源通过命名空间隔离开，各个容器相当于一个独立的ＯＳ，这样的实现更加灵活，命名空间也可以组成层次结构。

进程的命名空间

- **每一个进程其所包含的命名空间都被抽象成一个nsproxy指针，共享同一个命名空间的进程指向同一个指针**
  - 指针的结构通过引用计数（count）来确定使用者数目。
- **当一个进程其所处的用户空间发生变化的时候就发生分裂。通过复制一份老的命名空间数据结构**



进程的命名空间

- 命名空间本身只是一个框架，需要其他实行虚拟化的子系统实现自己的命名空间。
- 子系统的对象不是全局维护的结构，而和进程的用户空间数目一致，每一个命名空间都会有对象的一个具体实例。
- 目前Linux系统实现的命名空间子系统
  - 有UTS、IPC、MNT、PID以及NET网络子模块

Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

调用namespace的API

- **Namespace的API包括**
  - Clone(), setns(), unshare()
  - /proc下的部分文件

- **通过参数指定namespace的种类**
  - CLONE_NEWIPC、CLONE_NEWNS、CLONE_NEWNET、CLONE_NEWPID、CLONE_NEWUSER 和 CLONE_NEWUTS

调用namespace的API

- **Clone()创建新进程的同时创建namespace**

```
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg)
```

  - 参数 child_func 传入子进程运行的程序主函数。
  - 参数 child_stack 传入子进程使用的栈空间
  - 参数 flags 表示使用哪些 CLONE_* 标志位
  - 参数 args 则可用于传入用户参数

调用namespace的API

- /proc/[pid]/ns 文件

```
$ ls -l /proc/$$/ns          <<-- $$ 表示应用的 PID
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user->user:[4026531837]
.
```

```
# touch ~/uts
# mount --bind /proc/27514/ns/uts ~/uts
```

调用namespace的API

- **Setns()——加入一个已存在的namespace**

```
int setns(int fd, int nstype);
```

  - 参数 fd 表示要加入的 namespace 的文件描述符。是一个指向 /proc/[pid]/ns 目录的文件描述符，可以通过直接打开该目录下的链接或者打开一个挂载了该目录下链接的文件得到。

  - 参数 nstype 让调用者可以去检查 fd 指向的 namespace 类型是否符合实际的要求。如果填 0 表示不检查。

调用namespace的API

- Unshare() ——在原进程进行namespace隔离

```
int unshare(int flags);
```

- Unshare运行在原进程，不需要启动一个新进程

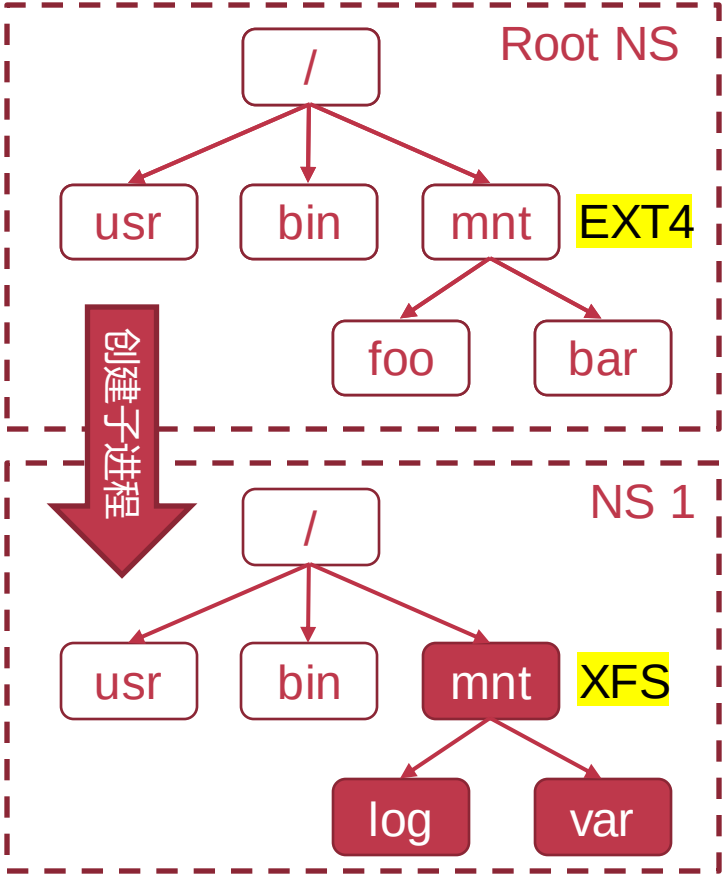
1、Mount Namespace

- **容器内外可部分共享文件系统**
  - 思考：如果容器内修改了一个挂载点会发生什么?

- **假设主机操作系统上运行了一个容器**
  - Step-1：主机OS准备从/mnt目录下的ext4文件系统中读取数据
  - Step-2：容器中进程在/mnt目录下挂载了一个xfs文件系统
  - Step-3：主机操作系统可能读到错误数据

Mount Namespace的实现

- **设计思路**
  - 在内核中分别记录每个NS中对于挂载点的修改
  - 访问挂载点时，内核根据当前NS的记录查找文件
- **每个NS有独立的文件系统树**
  - 新NS会拷贝一份父NS的文件系统树
  - 修改挂载点只会反映到自己NS的文件系统树

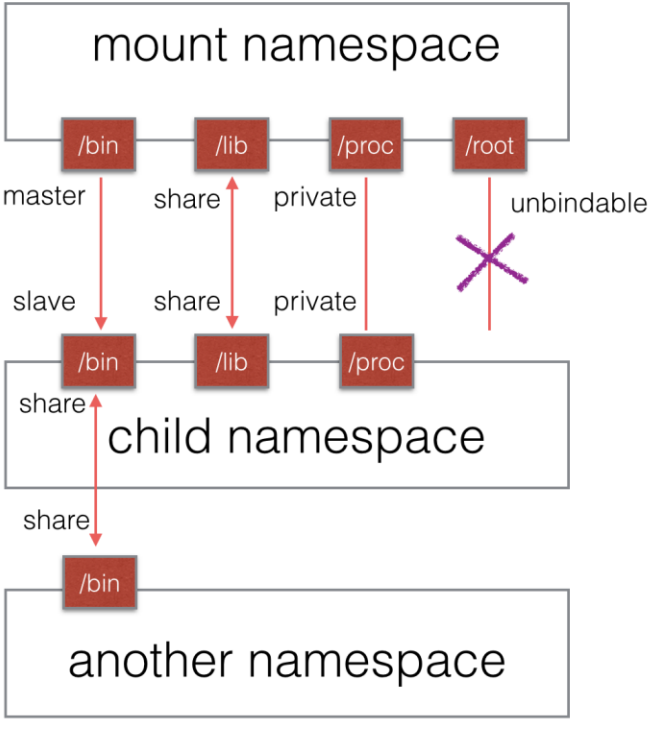


Mount namespaces

- **Mount namespace通过隔离文件系统挂载点对隔离文件系统提供支持**
  - /proc/[pid]/mounts ——查看所有挂在在当前namespace下的文件系统
  - /proc/[pid]/moutstats——查看mount namespace中文件设备的统计信息
  - 进程创建mount namespace
    - 会复制当前的文件结构到新的namespace
    - 新的namespace中的所有mount操作，只影响自身的文件系统
- mount propagation挂载传播：定义了挂载对象之间的关系
  - 共享关系（share relationship）。如果两个挂载对象具有共享关系，那么一个挂载对象中的挂载事件会传播到另一个挂载对象，反之亦然。
  - 从属关系（slave relationship）。如果两个挂载对象形成从属关系，那么一个挂载对象中的挂载事件会传播到另一个挂载对象，但是反过来不行；在这种关系中，从属对象是事件的接收者。

Mount namespace

- 一个挂载状态可能为如下的其中一种：
  - 共享挂载（shared）：传播事件的挂载对象
  - 从属挂载（slave）：接收传播事件的挂载对象
  - 共享 / 从属挂载（shared and slave）
  - 私有挂载（private）：既不传播也不接收传播事件的挂载对象
  - 不可绑定挂载（unbindable）：与私有挂载相似，但是不允许执行绑定挂载，即创建 mount namespace 时这块文件对象不可被复制



2、IPC Namespace

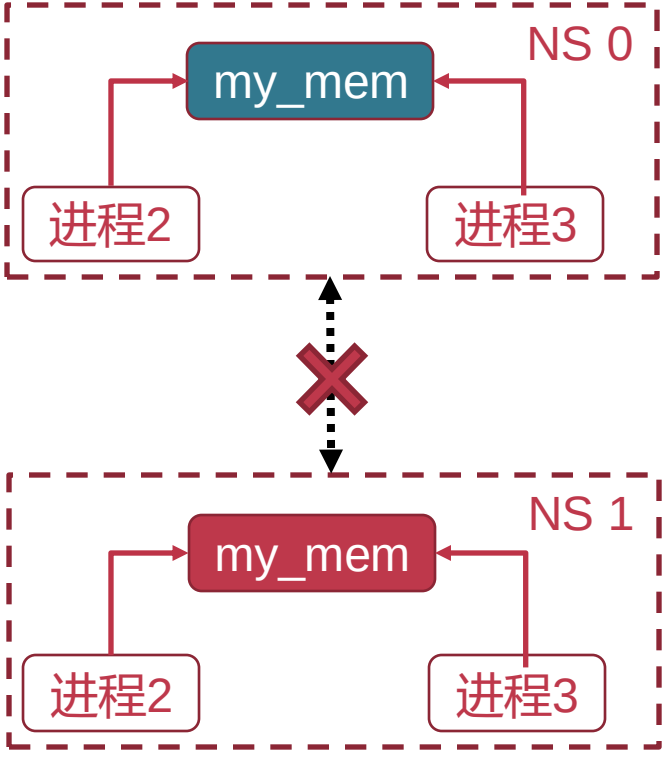
- **不同容器内的进程若共享IPC对象会发生什么?**

- **假设有两个容器A和B**
  - A中进程使用名为“my_mem”共享内存进行数据共享
  - B中进程也使用名为“my_mem”共享内存进行通信
  - B中进程可能收到A中进程的数据，导致出错以及数据泄露

IPC Namespace的实现

- **使每个IPC对象只能属于一个NS**
  - 每个NS单独记录属于自己的IPC对象
  - 进程只能在当前NS中寻找IPC对象

- **图例**
  - 即使不同NS的共享内存ID均为my_mem → 不同的共享内存



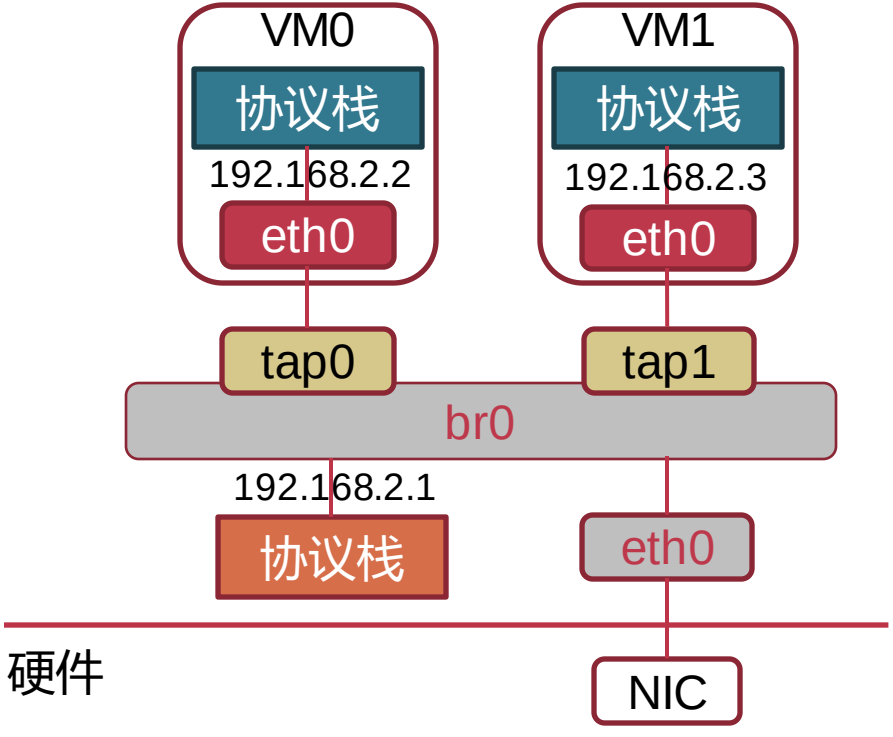
3、Network Namespace

- **不同的容器共用一个IP会发生什么?**

- **假设有两个容器均提供网络服务**
  - 两个容器的外部用户向同一IP发送网络服务请求
  - 主机操作系统不知道该将网络包转发给哪个容器

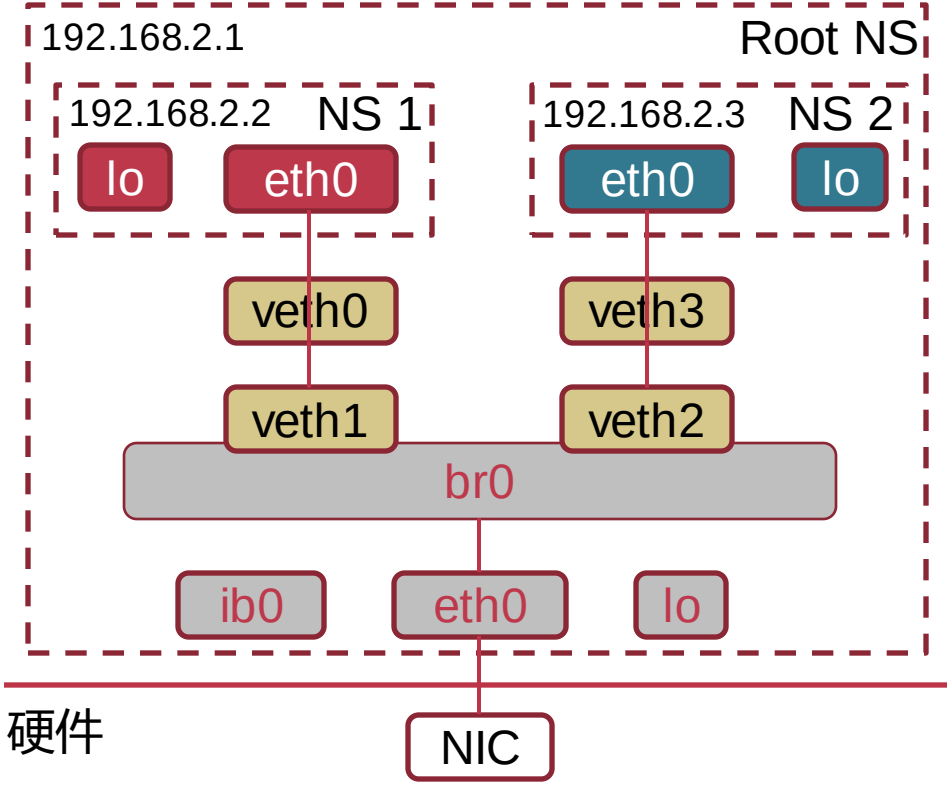
Linux对于多IP的支持

- **在虚拟机场景下很常见**
  - 每个虚拟机分配一个IP
    - IP绑定到各自的网络设备上
  - 内部的二级虚拟网络设备
    - br0: 虚拟网桥
    - tap: 虚拟网络设备
- **如何应用到容器场景?**



Network Namespace的实现

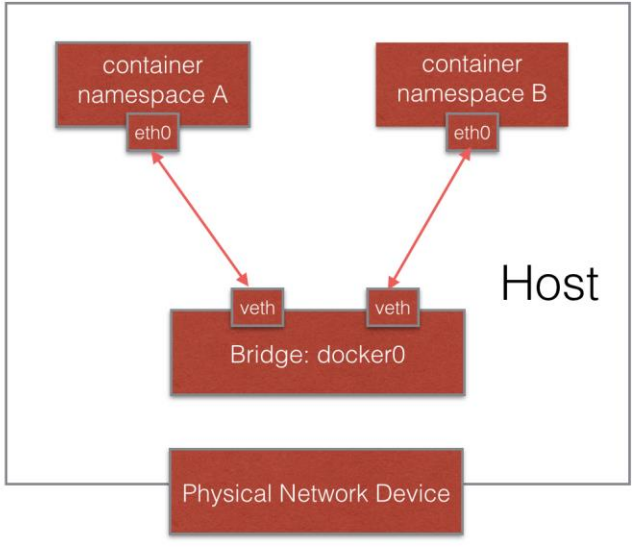
- **每个NS拥有一套独立的网络资源**
  - 包括IP地址、网络设备等
- **新NS默认只有一个loopback设备**
  - 其余设备需后续分配或从外部加入
- **图例**
  - 创建相连的veth虚拟设备对
  - 一端加入NS即可连通网络
  - 分配IP后可分别与外界通信



Network namespace

- Network namespace 主要提供了关于网络资源的隔离
  - 包括网络设备、IPv4 和 IPv6 协议栈、IP 路由表、防火墙、/proc/net 目录、/sys/class/net 目录、端口（socket）等等。
  - 一个物理的网络设备最多存在在一个 network namespace 中
  - network namespace释放时，其物理网络设备怎么办？返回给谁？

- veth pair（虚拟网络设备对）:
  - 在不同的 network namespace 间创建通道，以此达到通信的目的
  - 一端放在新的namespace中，命名为eth0；另一段放在原先namespace中链接物理网络设备，在通过网桥把别的设备连接尽量或进行路由转发



4、PID Namespace

- **容器内进程可以看到容器外进程的PID会发生什么?**

- **假设有容器内存在一个恶意进程**
  - 恶意进程向容器外进程发送SIGKILL信号
  - 主机操作系统或其他容器中的正常进程会被杀死

PID Namespace的设计

- **直接的想法**
  - 将每个NS中的进程放在一起管理，不同NS中的进程相互隔离
- **存在的问题**
  - 进程间关系如何处理（比如父子进程）？
- **更进一步**
  - 允许父NS看到子NS中的进程，保留父子关系

PID namespace

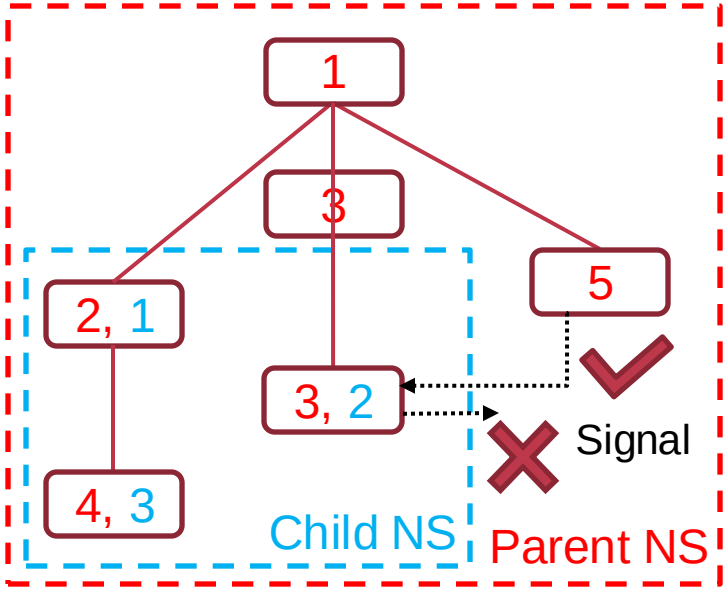
- **Linux为PID namespace维护一个树状结构**
  - 初始化时创建的为root namespace
  - 与进程id类似，pid namespace存在父子关系
  - 父节点可以看到子节点的进程，并通过信号等方式进行影响，反之不可
- **Pid命名空间子模块：层次化**

```
struct pid_namespace {
  //…
  unsigned int level;
  struct pid_namespace *parent;
  //…
};
```

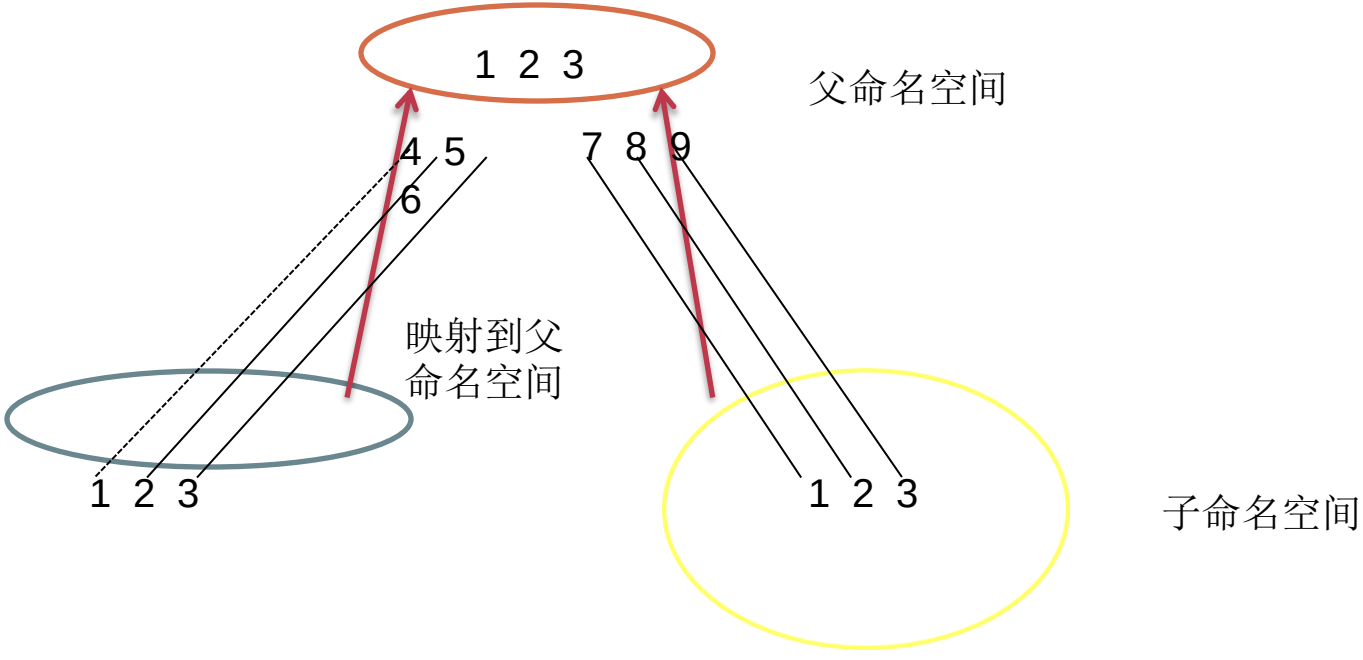
- **怎么监控Docker中运行的程序状态?**

PID Namespace的实现

- **对NS内外的PID进行单向隔离**
  - 外部能看到内部的进程，反之则不能
- **图例**
  - 子NS中的进程在父NS中也有PID
  - 进程只能看到当前NS的PID
  - 子NS中的进程无法向外发送信号



PID namespace



PID namespace

- PID namespace 中的 init 进程
  - 创建的第一个pid namespace
  - Ini进程会负责回收资源和监控、维护后续启动进程的运行状态

- 信号与init进程
  - 信号屏蔽：如果init中没有写处理某个信号的代码逻辑，那么则与init在同一个pid namespace中的进程发送的该信号会被屏蔽
  - 父节点pid namespace中进程发送的信号会被忽略吗?
  - 一旦init进程被销毁，同一pid namespace中的其他进程呢?

- Unshare()和setns()
  - 调用者进程不进入新的pid namespace，而是随后创建的子进程进入
  - 与其他namespace不同的原因?

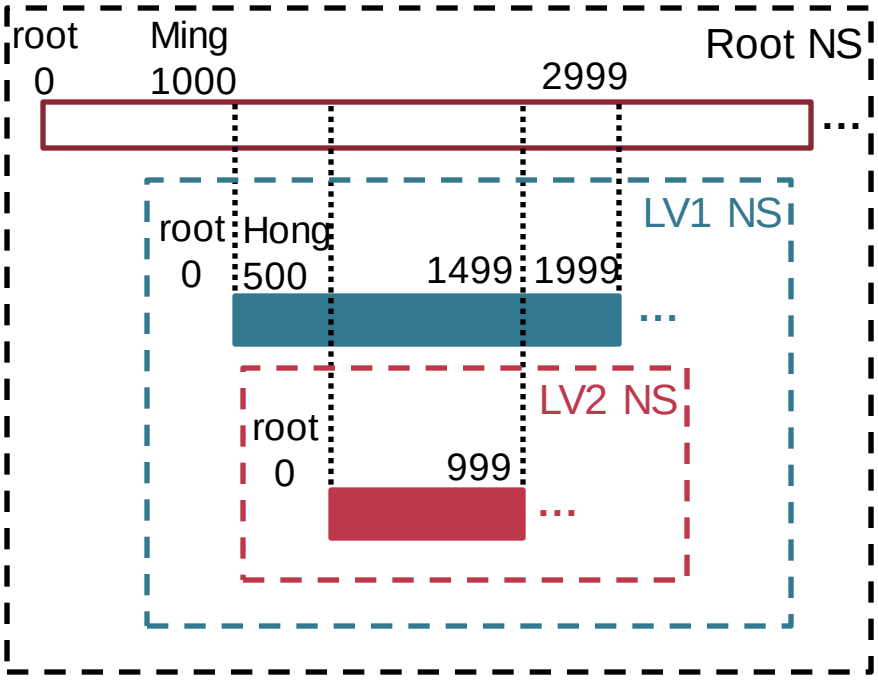
5、User Namespace

- **容器内外共享一个root用户会发生什么?**

- **假设一个恶意用户在容器内获取了root权限**
  - 恶意用户相当于拥有了整个系统的最高权限
  - 可以窃取其他容器甚至主机操作系统的隐私信息
  - 可以控制或破坏系统内的各种服务

User Namespace的实现

- **对NS内外的UID和GID进行映射**
  - 允许普通用户在容器内有更高权限
    - 基于Linux Capability机制
  - 容器内root用户在容器外无特权
    - 只是普通用户
- **图例**
  - 普通用户在子NS中是root用户



进一步限制容器内Root

- **如果容器内root要执行特权操作怎么办?**
  - insmod? 一旦允许在内核中插入驱动，则拥有最高权限
  - 关机/重启? 整个云服务器会受影响
- **1、从内核角度来看，仅仅是普通用户**
- **2、限制系统调用**
  - Seccomp机制

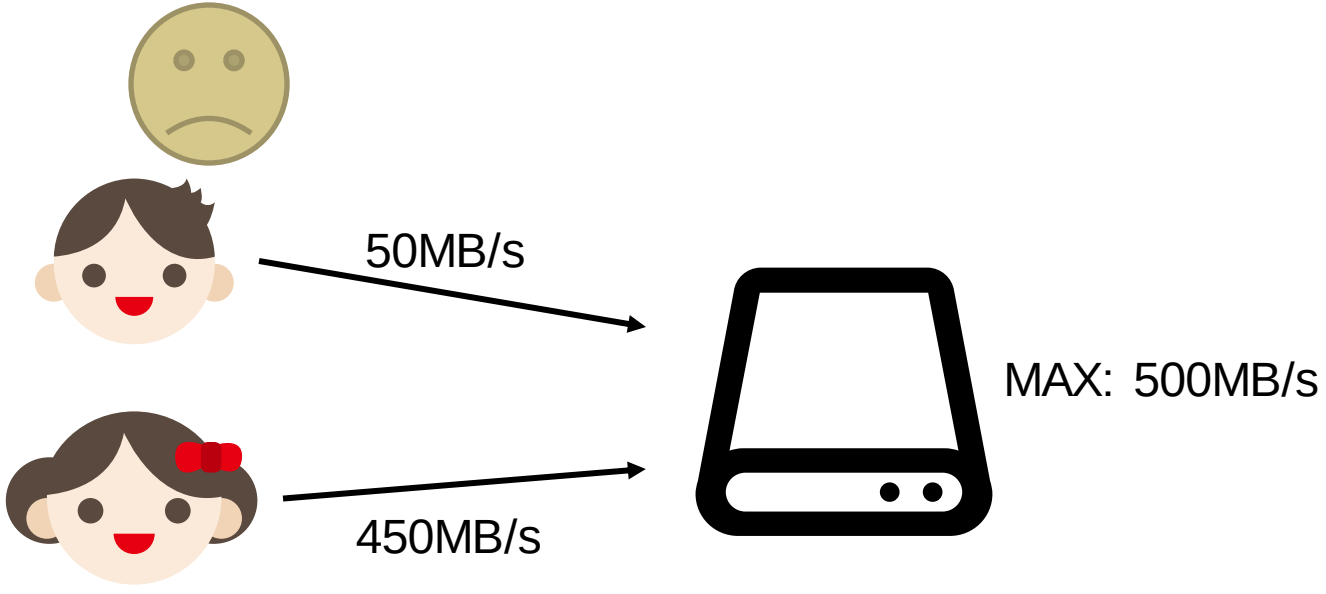
其他Namespace

- **6、UTS Namespace**
  - 每个NS拥有独立的hostname等名称
  - 便于分辨主机操作系统及其上的多个容器
- **7、Cgroup Namespace**
  - cgroupfs的实现向容器内暴露cgroup根目录
  - 增强隔离性：避免向容器内泄露主机操作系统信息
  - 增强可移植性：取消cgroup路径名依赖

性能隔离

资源竞争问题

- 小明和小红同时访问磁盘



Control Cgroups (Cgroups)

- **Cgroups是什么**
  - Linux内核（从Linux2.6.24开始）提供的一种资源隔离的功能
- **Cgroups可以做什么**
  - 将线程分组
  - 对每组线程使用的多种物理资源进行限制和监控
- **怎么用Cgroups**
  - 名为cgroupfs的伪文件系统提供了用户接口

Cgroups的常用术语

- 任务（task）
- 控制组（cgroup）
- 子系统（subsystem）
- 层级 （hierarchy）

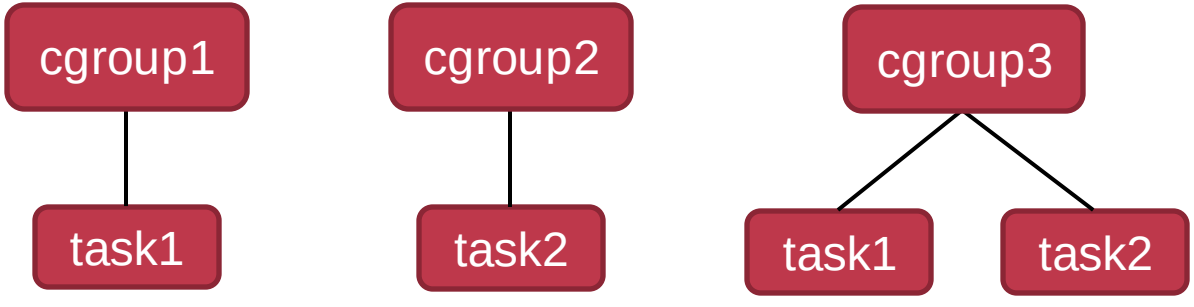
任务（Task）

- 任务就是系统中的一个线程
- 例如：有两个任务 task1和task2

task1　　task2

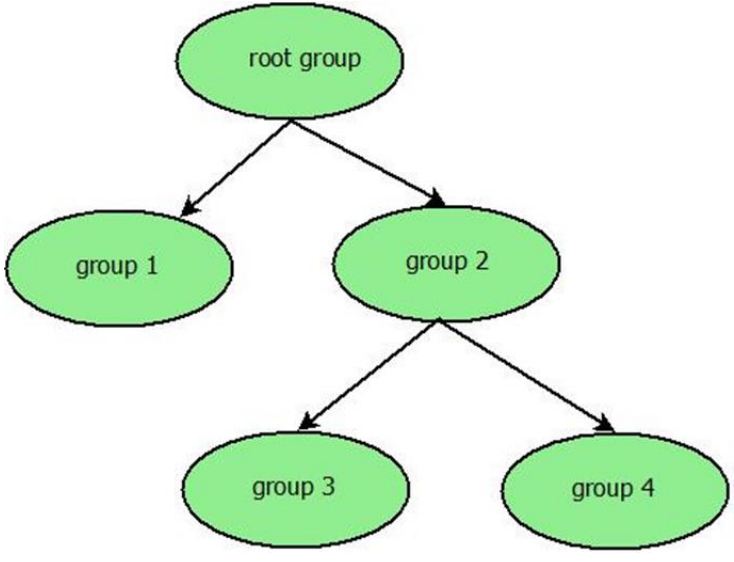
控制组（Control Group）

- **Cgroups进行资源监控和限制的单位**
- **任务的集合**
  - 控制组cgroup1包含task1
  - 控制组cgroup2包含task2
  - 控制组cgroup3由task1和task2组成



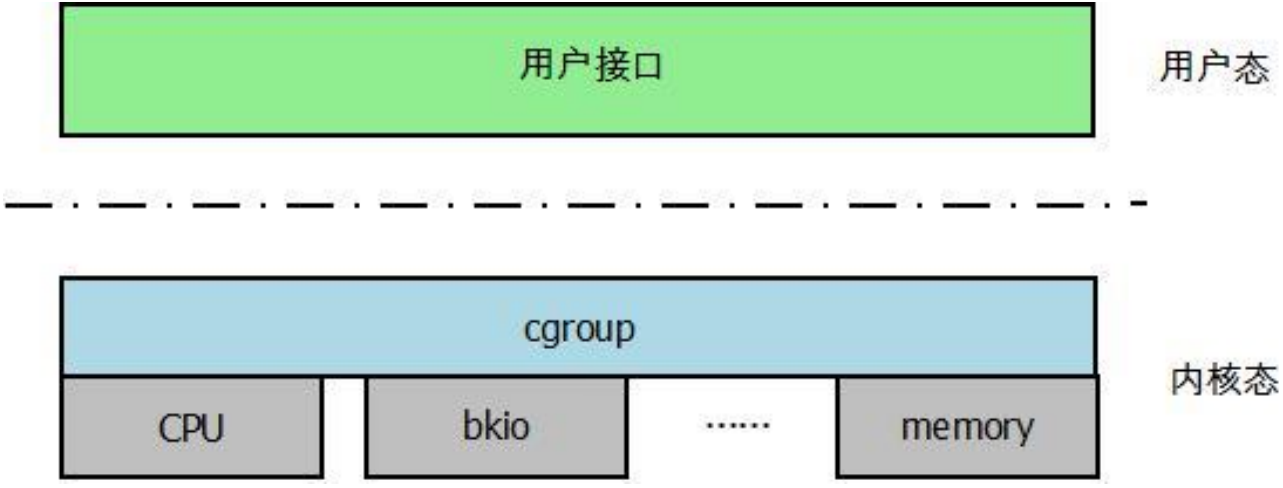
控制组

- **结构**：系统内的所有进程形成一个根进程组，根据系统对资源的需求，这个根进程组将被进一步细分为子进程组，子进程组内的进程是根进程组内进程的子集。经过细分，形成具有层次的进程组树。



子系统（Sub-system）

- 可以跟踪或限制控制组使用该类型物理资源的内核组件
- 也被称为资源控制器



子系统

- blkio -- 这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等等）。
- cpu -- 这个子系统使用调度程序提供对 CPU 的 cgroup 任务访问。
- cpuacct -- 这个子系统自动生成 cgroup 中任务所使用的 CPU 报告
- cpuset -- 这个子系统为 cgroup 中的任务分配独立 CPU和内存节点
- devices -- 这个子系统可允许或者拒绝 cgroup 中的任务访问设备。
- freezer -- 这个子系统挂起或者恢复 cgroup 中的任务。
- memory -- 这个子系统设定 cgroup 中任务使用的内存限制，并自动生成由那些任务使用的内存资源报告。
- net_cls -- 这个子系统使用等级识别符标记网络数据包，可允许 Linux 流量控制程序识别从具体 cgroup 中生成的数据包。

层级（Hierarchy）

- 由控制组组成的树状结构
- 通过被挂载到文件系统中形成

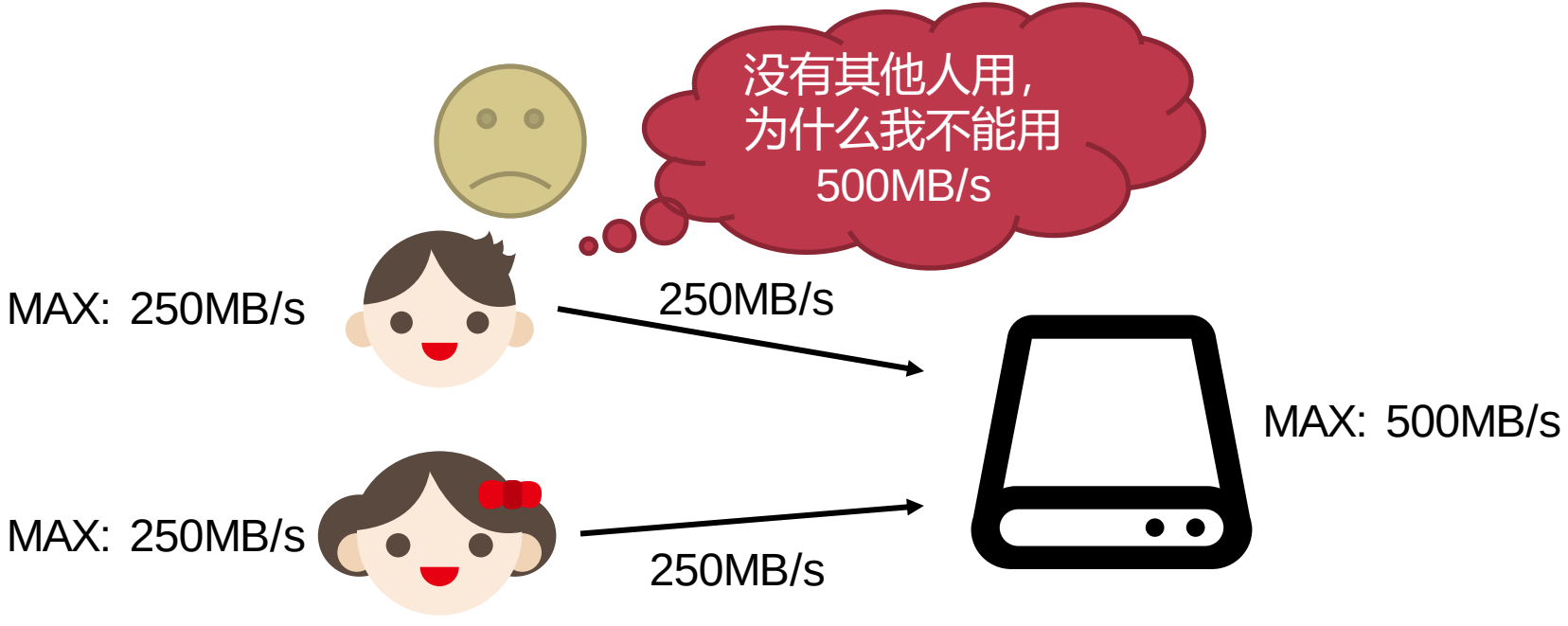
```
$ mount | grep "type cgroup "
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
```

资源控制模型

- **最大值**
  - 直接设置一个控制组所能使用的物理资源的最大值，例如：
    - 内存子系统：最多能使用1GB内存
    - 存储子系统：最大能使用100MB/s的磁盘IO

资源控制模型

- 小明和小红同时访问磁盘

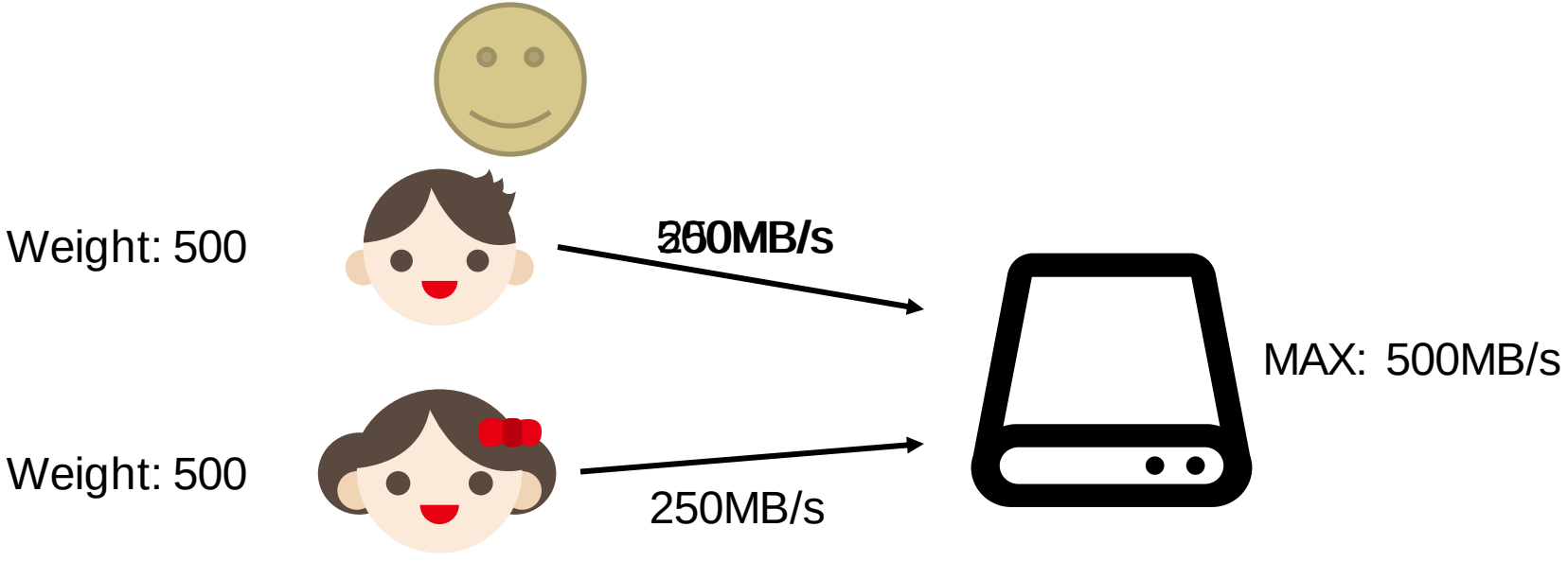


资源控制模型

- **最大值**
  - 直接设置一个控制组所能使用的物理资源的最大值，例如：
    - 内存子系统：最多能使用1GB内存
    - 存储子系统：最大能使用100MB/s的磁盘IO
- **比例**
  - 设置不同控制组使用同一物理资源时的资源分配比例，例如：
    - 存储子系统：两个控制组按照1：1的比例使用磁盘IO资源
    - CPU子系统：两个控制组按照2：1的比例使用CPU时间

资源控制模型

- 小明和小红同时访问磁盘



如何对任务使用资源进行监控和限制

- **Cgroups进行监控和限制的单位是什么?**
  - 控制组
- **如何知道一个控制组使用了多少物理资源?**
  - 计算该控制组所有任务使用的该物理资源的总和
- **如何限制一个控制组**
  - 使该控制组的所有任务使用的物理资源不超过这个限制
  - 在每个任务使用物理资源时，需要保证不违反该控制组的限制

CPU子系统

- **回顾CFS（完全公平调度器）**
  - 可以为每个任务设定一个“权重值”
  - “权重值”确定了不同任务占用资源的比例
- **CPU子系统允许为不同的控制组设定CPU时间的比例**
  - 直接利用CFS来实现按比例分配的资源控制模型
  - 为控制组设定权重：向cpu.shares文件中写入权重值（默认1024）

内存子系统

- **监控控制组使用的内存**
  - 利用page_counter监控每个控制组使用了多少内存
- **限制控制组使用的内存**
  - 通过修改memory.limit_in_bytes文件设定控制组最大内存使用量

内存子系统

- **Linux分配内存首先需要charge同等大小的内存**
  - 只有charge成功，才能分配内存
- **Charge内存的简化代码（大小为nr_pages）：**

```
new = atomic_long_add_return(nr_pages, &page_counter->usage);
if (new > page_counter->max) {
    atomic_long_sub(nr_pages, &page_counter->usage);
    goto failed;
}
```

- **释放内存时会执行相反的uncharge操作**

存储子系统（blkio）

- **限制最大IOPS/BPS**
  - blkio.throttle.read_bps_device 限制对某块设备的读带宽
  - blkio.throttle.read_iops_device 限制对某块设备的每秒读次数
  - blkio.throttle.write_bps_device 限制对某块设备的写带宽
  - blkio.throttle.write_iops_device 限制对某块设备的每秒写次数
- **设定权重（weight）**
  - blkio.weight 该控制组的权重值
  - blkio.weight_device 对某块设备单独的权重值

小结：隔离的不同方式

- **物理机隔离**
- **虚拟机隔离**
- **机密虚拟机隔离**
- **文件系统隔离：**Chroot
- **容器隔离：**Name Space
- **进程隔离：**传统方式
- **性能隔离：**Cgroup

LXC优点

- LXC是所谓的操作系统层次的虚拟化技术，与传统的HAL（硬件抽象层）层次的虚拟化技术相比有以下优势：
- 1、更小的虚拟化开销（LXC的诸多特性基本由内核特供，而内核实现这些特性只有极少的花费，具体分析有时间再说）
- 2、快速部署。利用LXC来隔离特定应用，只需要安装LXC，即可使用LXC相关命令来创建并启动容器来为应用提供虚拟执行环境。传统的虚拟化技术则需要先创建虚拟机，然后安装系统，再部署应用。
- LXC跟其他操作系统层次的虚拟化技术相比，最大的优势在于LXC被整合进内核，不用单独为内核打补丁。

容器/VM 迁移

Hypervisor

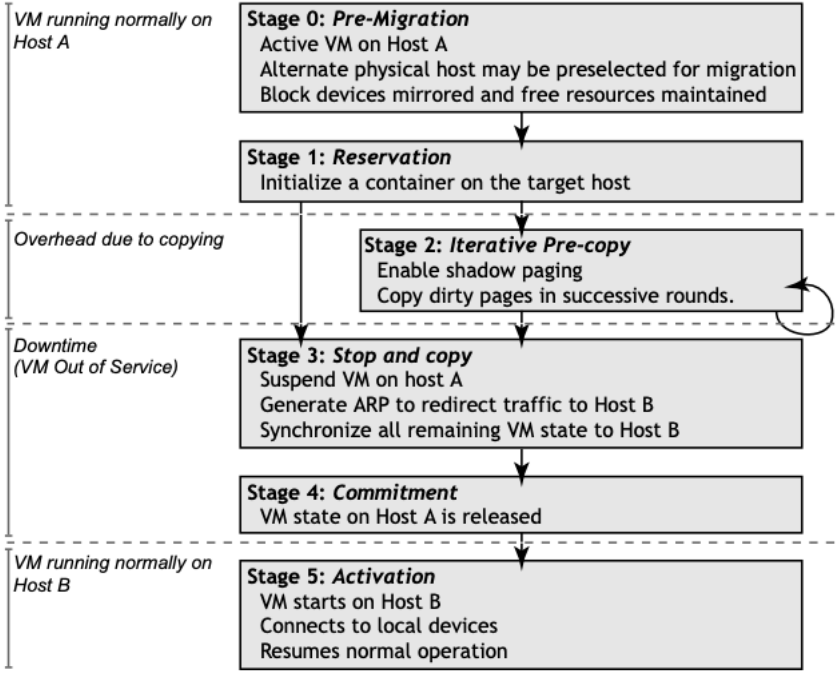


Figure 1: Migration timeline

Live Migration of Virtual Machines [2005, NSDI]

相关参考文献

- **嵌套虚拟化**
  - The Turtles Project: Design and Implementation of Nested Virtualization　OSDI’10