IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

轻量级虚拟化

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

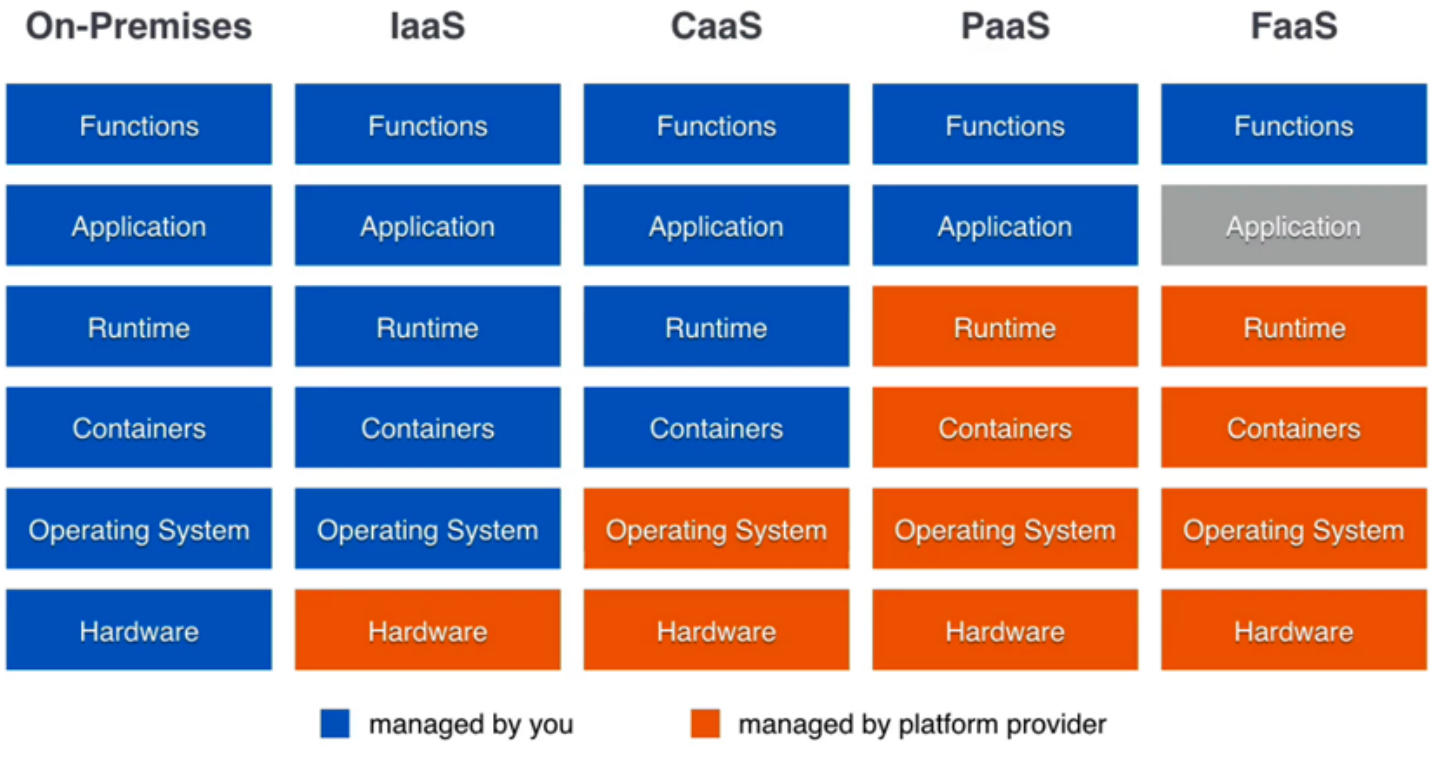
https://ipads.se.sjtu.edu.cn

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

为什么需要轻量级虚拟化

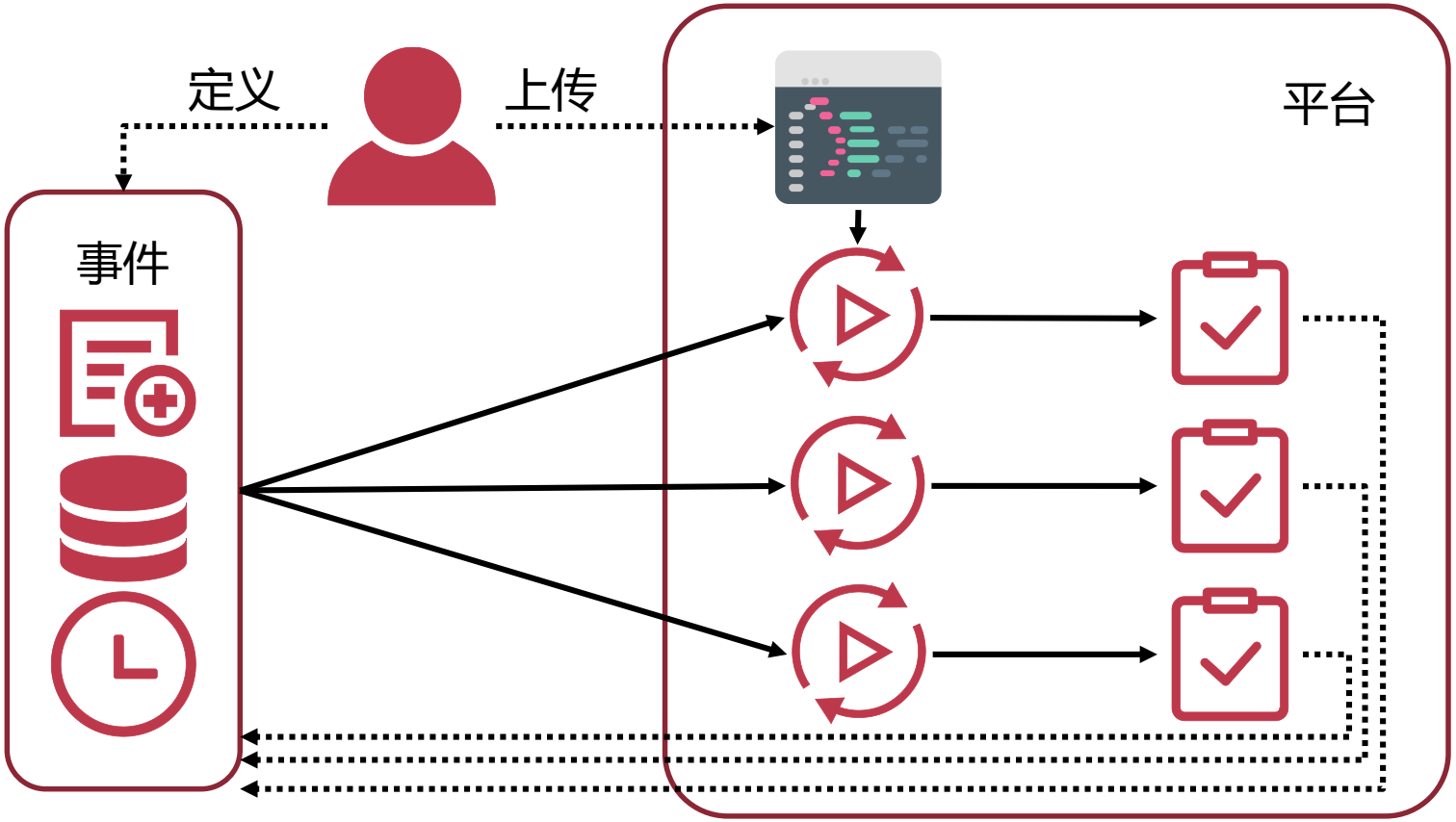
云服务变得越来越轻量


On-Premises
IaaS
CaaS
PaaS
FaaS
Functions
Application
Runtime
Containers
Operating System
Hardware
managed by you
managed by platform provider

FaaS与Serverless

- **FaaS：Function as a Service，函数即服务**
  - 用户提供函数，云端提供运行环境与部分运维
  - 特点：用户代码执行的时间非常短
- **传统的以虚拟机服务器为单位的云部署方式**
  - 虚拟机服务器长时间运行，而用户代码执行时间短，不匹配
- **新兴的无服务部署方式：Serverless**
  - 只在请求来时，新建服务器实例执行相关逻辑，完毕后销毁实例
  - 特点：请求驱动，弹性伸缩，按量计费

函数即服务



函数即服务的特点

- **Workload特点**
  - 无状态（stateless）
  - 运行时间非常短（秒级）
- **两个重要的性能指标**
  - 启动时间
  - 运行密度

Serverless该使用何种执行环境?

- **现有方案：虚拟机**
  - Function执行时，动态新建一个虚拟机
  - Function执行完，销毁虚拟机
- **现有方案的缺点**
  - 过于重量级；以一次基于virtio的网络TX为例：
    - 虚拟机内核：VFS → TCP/IP栈 → 网卡驱动
    - 宿主机内核：VFS → OVS → 网卡驱动
  - 启动时延长，运行密度低
  - 根本原因：虚拟机与宿主机的内核功能有重复

是否可以不用虚拟化?

- **Windows Server允许多个用户同时远程桌面**
  - 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
- **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

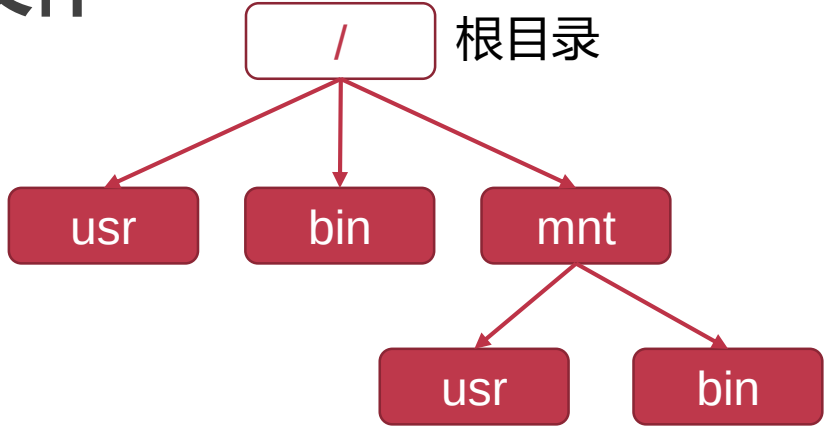
第一次尝试：CHROOT

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?

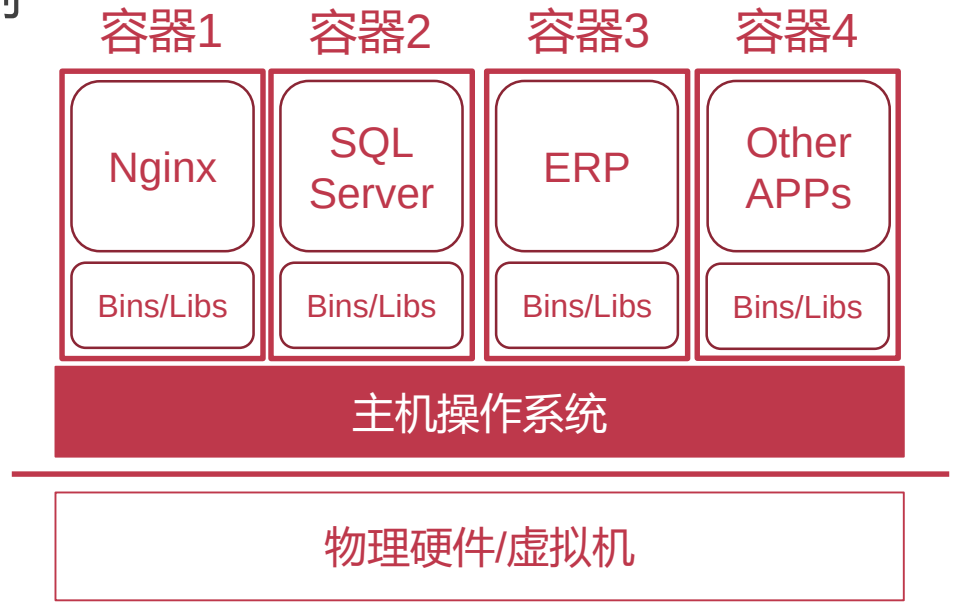
- **不同的执行环境想要共享一些文件怎么办?**
- **涉及到网络服务时会发生什么?**
  - 所有执行环境共用一个IP地址，所以无法区分许多服务
- **执行环境需要root权限该怎么办?**
  - 全局只有一个root用户，所以不同执行环境间可能相互影响

- **不能，因为还有许多资源被共享…**

LINUX CONTAINER

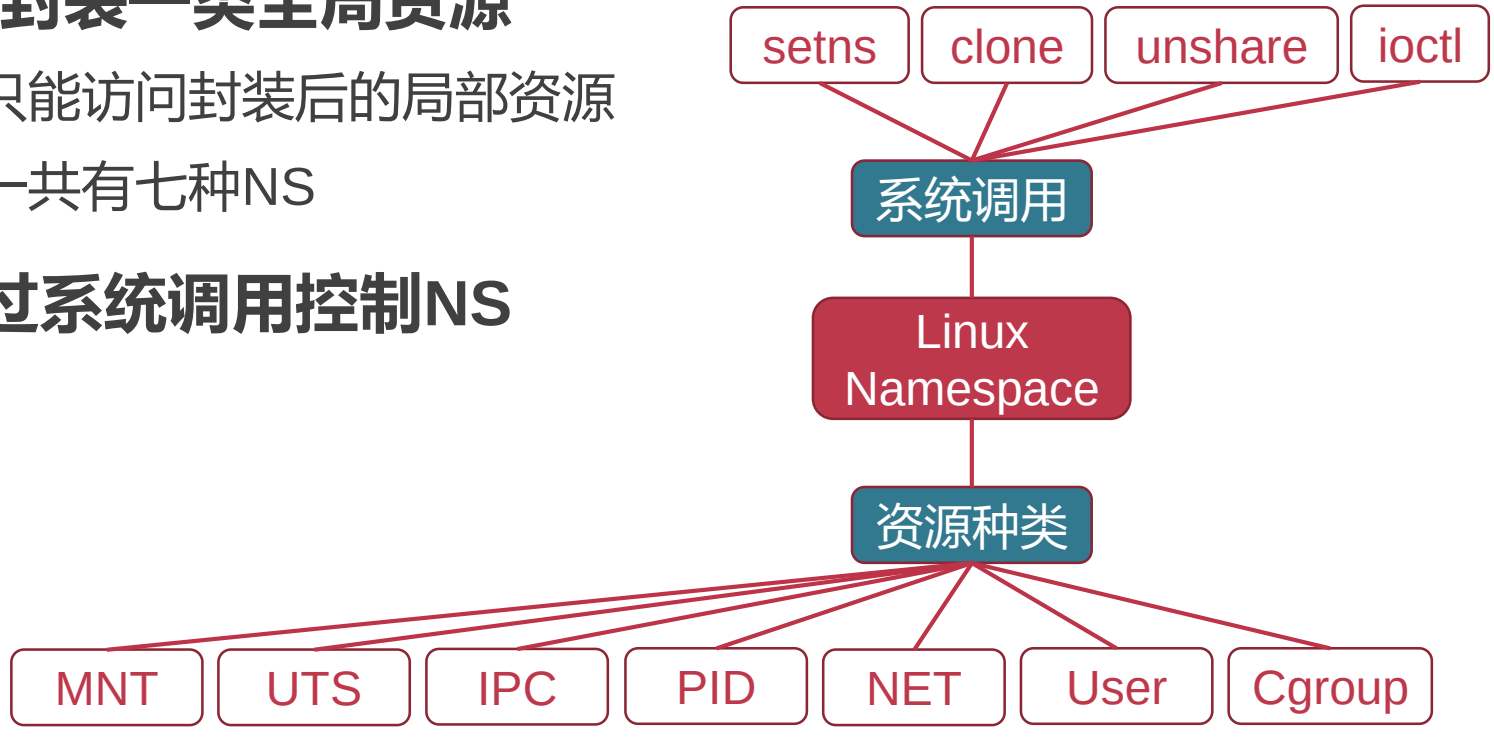
LinuX Container (LXC)

- **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**
  - Linux namespace
- **性能隔离**
  - Linux cgroup



Linux Namespace

- **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**

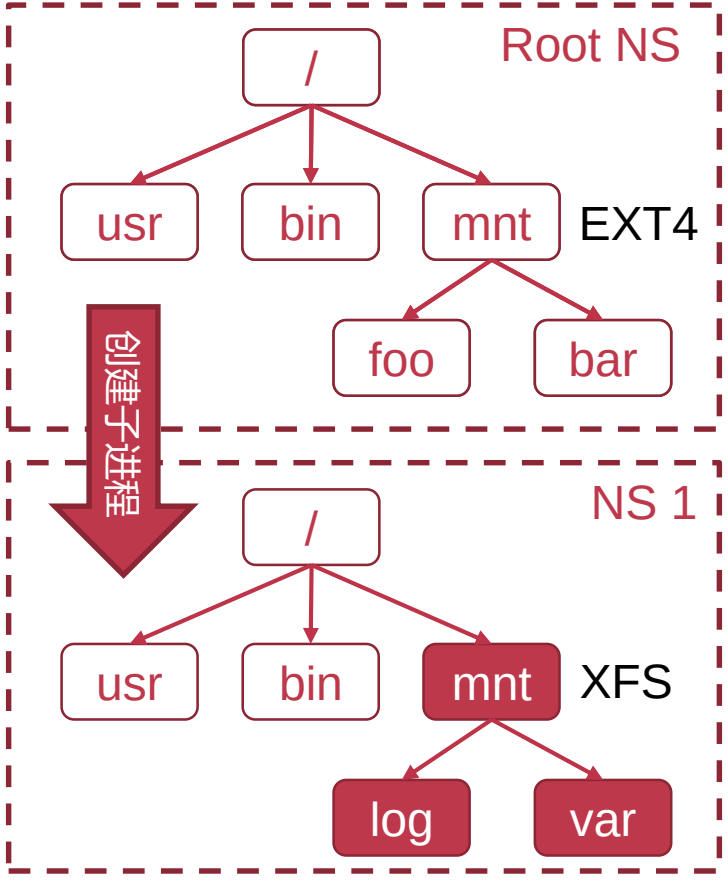


1、Mount Namespace

- **容器内外想要部分共享文件系统**
  - 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**
  - 主机操作系统准备从/mnt目录下的ext4文件系统中读取数据
  - 容器中进程在/mnt目录下挂载了一个xfs文件系统
  - 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



2、IPC Namespace

- **不同容器内的进程若共享IPC对象会发生什么?**

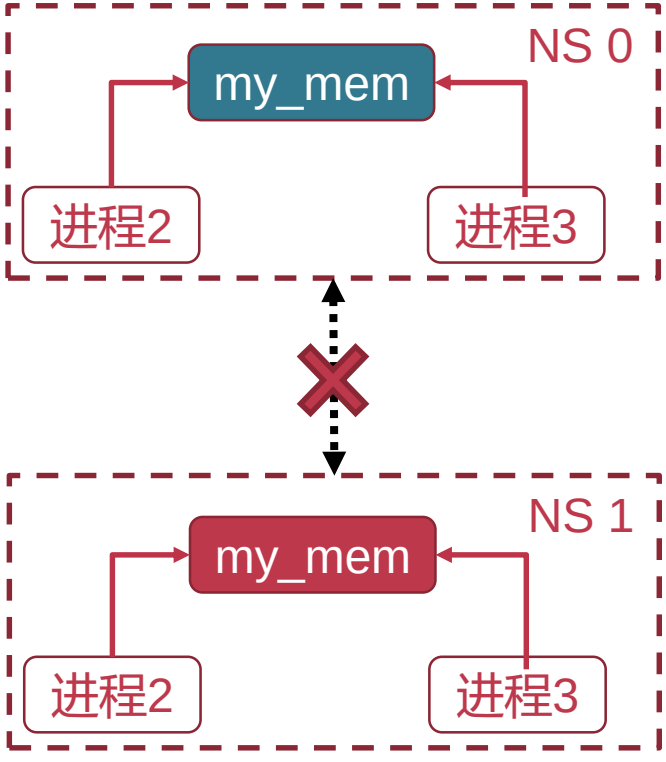
- **假设有两个容器A和B**
  - A中进程使用名为“my_mem”共享内存进行数据共享
  - B中进程也使用名为“my_mem”共享内存进行通信
  - B中进程可能收到A中进程的数据，导致出错以及数据泄露

IPC Namespace的设计

- **直接的想法**
  - 在内核中创建IPC对象时，贴上对应NS的标签
  - 进程访问IPC对象时内核来判断是否允许访问该对象
- **可能的问题**
  - 可能有timing side channel隐患
  - 对于同名的IPC对象不好处理
- **更进一步**
  - 将每个NS创建的IPC对象放在一起管理

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**
  - ID均为my_mem→不同的共享内存



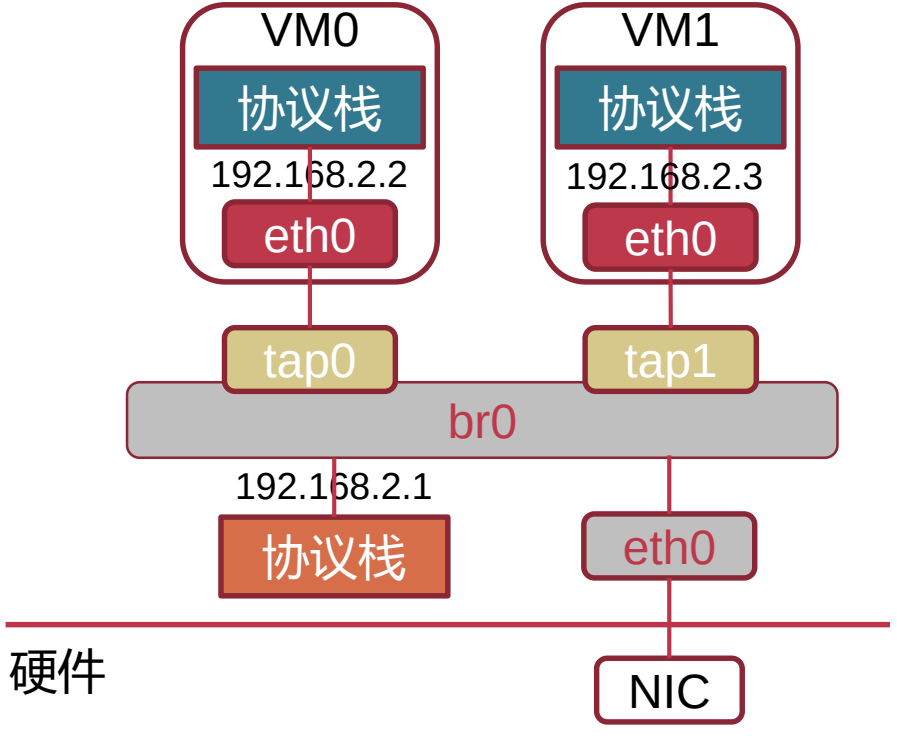
3、Network Namespace

- **不同的容器共用一个IP会发生什么?**

- **假设有两个提供网络服务容器**
  - 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

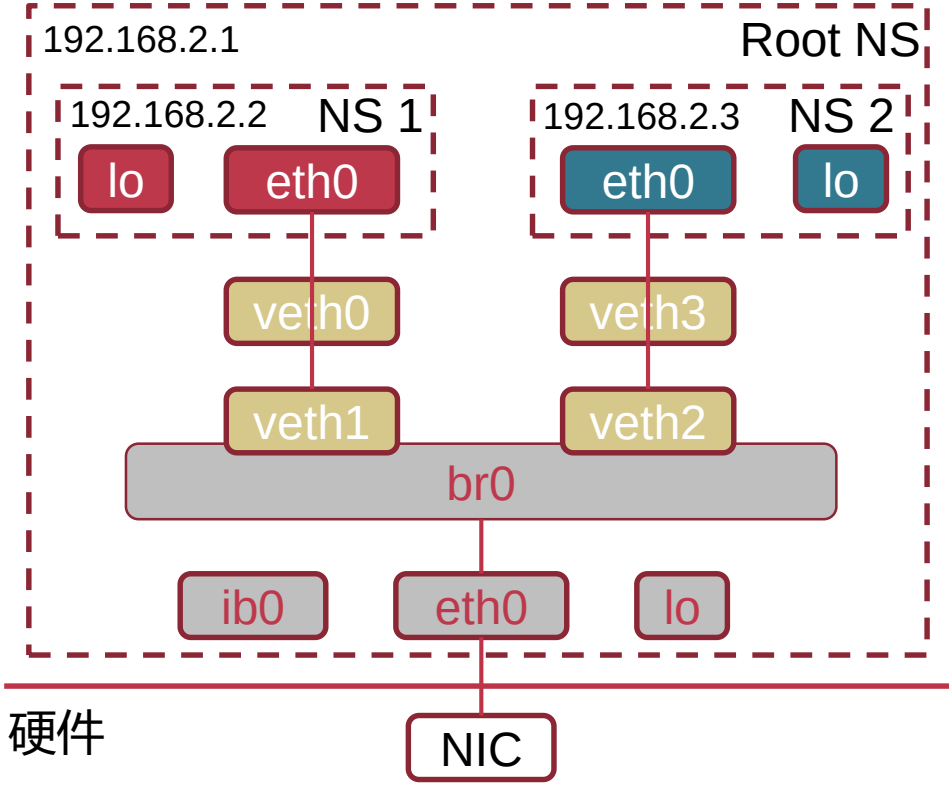
Linux对于多IP的支持

- **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



4、PID Namespace

- **容器内进程可以看到容器外进程的PID会发生什么?**

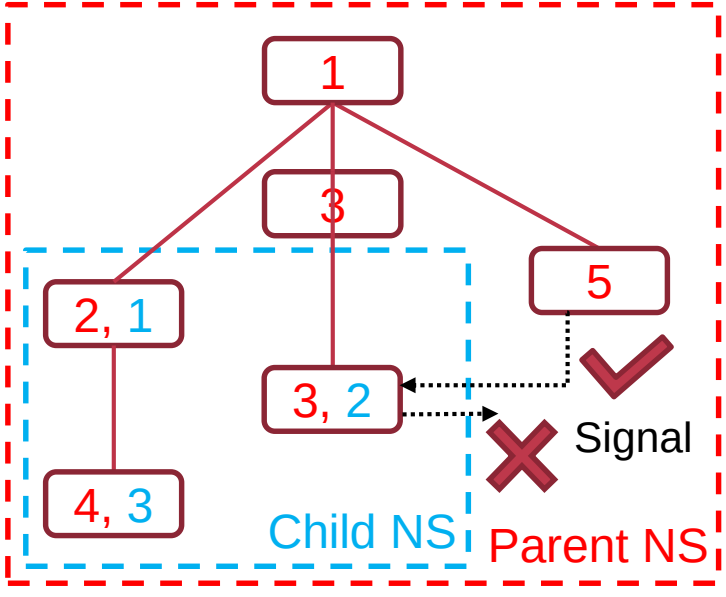
- **假设有容器内存在一个恶意进程**
  - 恶意进程向容器外进程发送SIGKILL信号
  - 主机操作系统或其他容器中的正常进程会被杀死

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**
  - 父子进程等进程间关系如何处理?
- **更进一步**
  - 允许父NS看到子NS中的进程，保留父子关系

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



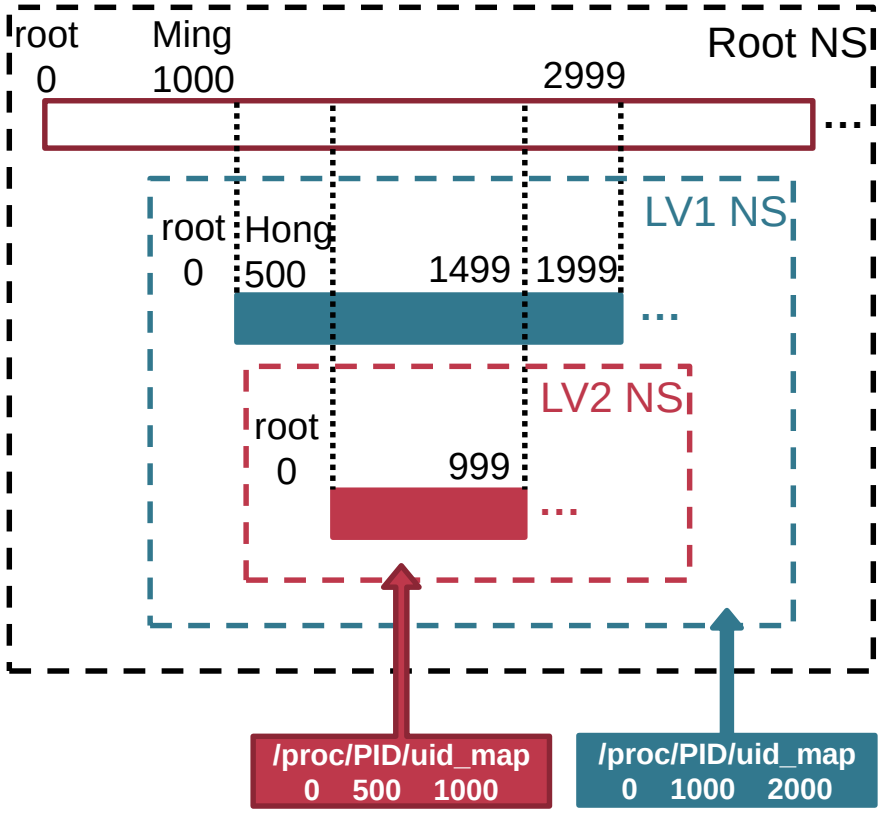
5、User Namespace

- **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拥有更高的权限
    - 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



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

案例：DOCKER

Docker和容器

- **Docker是什么**
  - 基于Go语言实现的开源容器项目
  - 最初基于LXC实现
- **Docker是容器的一种实现**
  - 进一步优化了容器的使用体验
  - Docker在容器生态的发展中起到了重要的作用

Docker发展历史

- **2010年，dotCloud公司成立**
  - 公司创始人Solomon Hykes发起了名为Docker的公司内部项目
- **2013年3月，Docker开源，基于LXC实现**
- **2013年11月，Docker v0.7发布，将LXC替换为libcontainer**
- **2013年底，dotCloud公司更名为Docker**
- **2016年3月，Docker v1.11发布，使用runC运行时**

用Docker编译lab是如何实现的

- **拉取基础Ubuntu镜像**
  - docker pull ubuntu
- **运行基础Ubuntu镜像**
  - docker run -it --name mycontainer ubuntu bash
- **安装必要的编译环境**
  - 与虚拟机内安装软件相同

用Docker编译lab是如何实现的

- **通过容器提交镜像**
  - docker commit mycontainer ipads/chcore_builder:v1.0
  - ipads是我们的用户名
  - chcore_builder是该容器镜像的名字
  - v1.0表示该镜像的版本
- **镜像提交之后，我们便可以直接使用该容器**
  - docker run ipads/chcore_builder:v1.0
  - 一次封装，随处运行，无论使用什么操作系统，都不会遇到因为系统不一致带来的各种问题

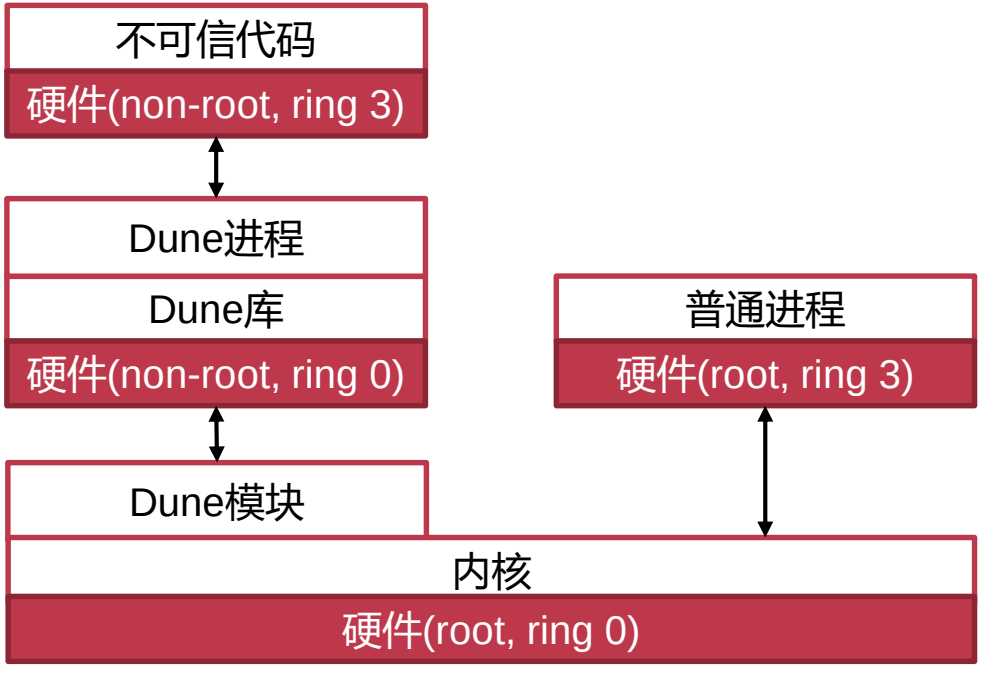
案例：SERVERLESS平台

Google Cloud Functions

支持语言	Node.js, Python, Go
代码大小限制	100MB（已压缩），500MB（解压缩）
并发执行	1000
函数内存	2048MB
函数最大执行时间	9分钟

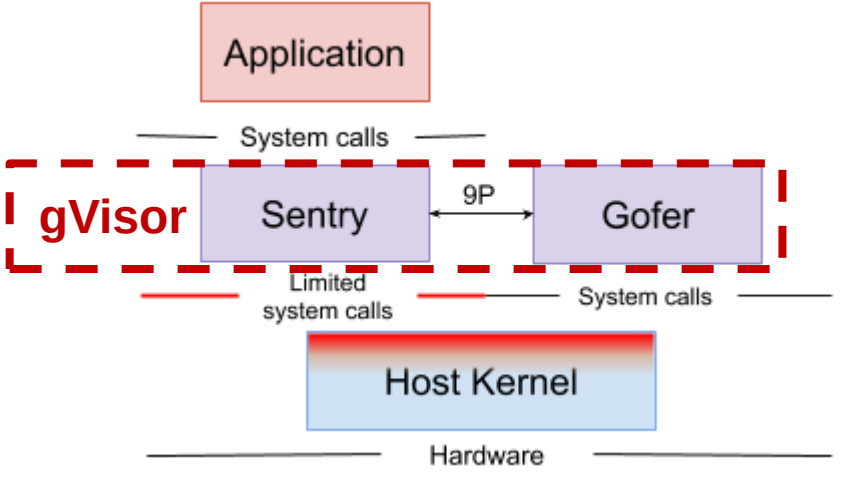
进程级虚拟机：Dune

- 利用硬件虚拟化，允许在非特权级直接执行特权指令



gVisor

- 基于Go语言实现的用户态内核
- 提供与虚拟机相当的强隔离性
- Sentry包含内核，处理大部分系统调用
- Gofer处理IO操作



AWS Lambda

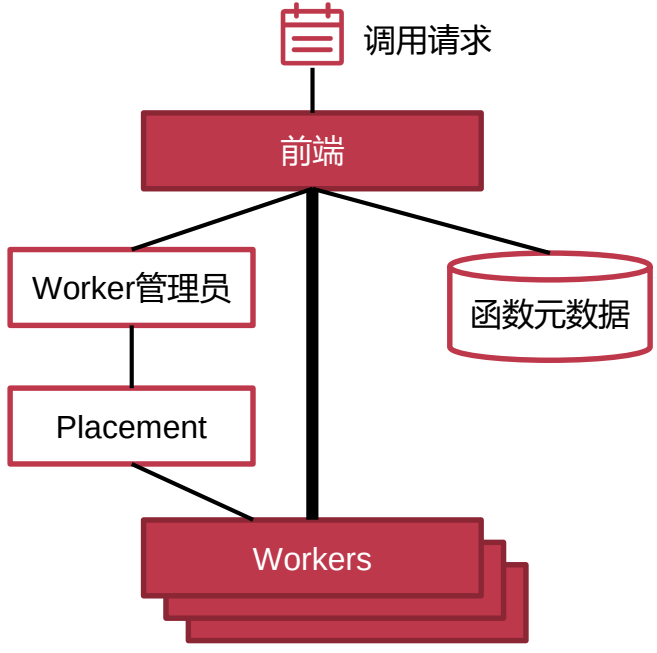
支持语言	Node.js, Python, Ruby, Java, Go, C#, PowerShell
代码大小限制	50MB（已压缩），250MB（解压缩）
并发执行	1000
函数内存	128MB到3008MB，以64MB为增量
函数最大执行时间	15分钟

AWS Lambda


工作原理
运行
下一步：Lambda 响应事件
1 exports.handler = (event, context, callback) => {
2 // 字符串“Hello world!”已成功。
3 callback(null, 'Hello world!');
4 };
只需编写代码
上面是简单的 Node.js Lambda 函数。尝试更改回调值并运行函数，然后再继续下一步。

Lambda的高层架构

- **前端**
  - 接收调用请求并转发给worker
- **Placement**
  - 负责为函数创建新的执行环境
- **Worker**
  - 真正运行函数代码的节点
- **Worker管理员**
  - 负责为调用请求分配worker
    - 若有现成的执行环境，则让前端直接将调用请求发给这个worker
    - 若没有现成的执行环境，则调用Placement服务创建一个新的执行环境



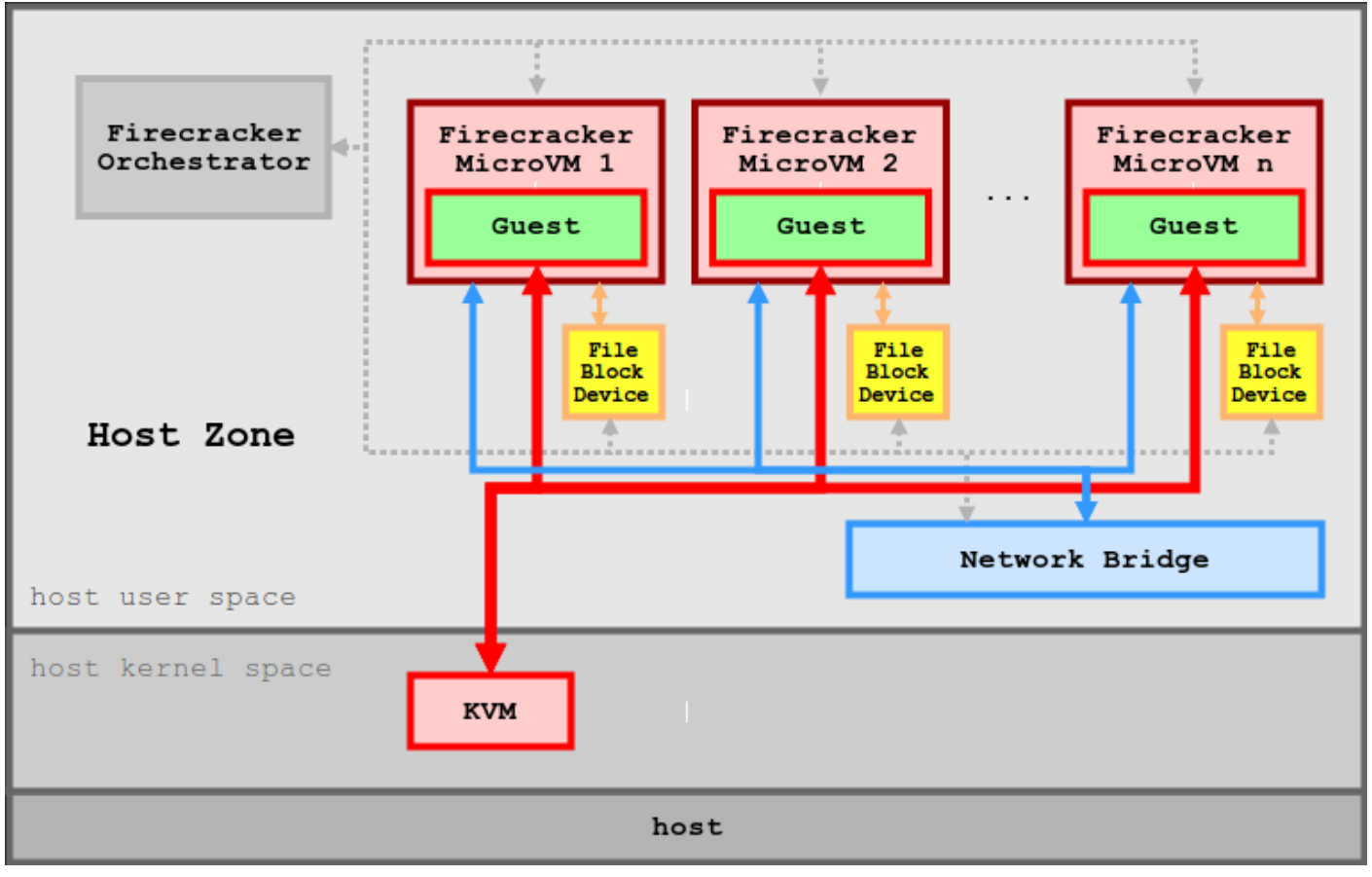
隔离方案

- **最初方案**
  - 使用虚拟机来提供不同用户之间的隔离
  - 使用容器来隔离同一个用户的不同函数
- **AWS Firecracker**
  - Amazon开发的一种轻量级虚拟化运行环境
  - 2018年12月开源，同年被用于Lambda生产环境

AWS Firecracker

- **轻量级虚拟机**
  - 保留了KVM，完全替换了QEMU
- **没有模拟完整的硬件，只实现了运行函数的必要功能**
- **启动一个虚拟机所需内存：5MB**
- **启动时间：小于125ms**

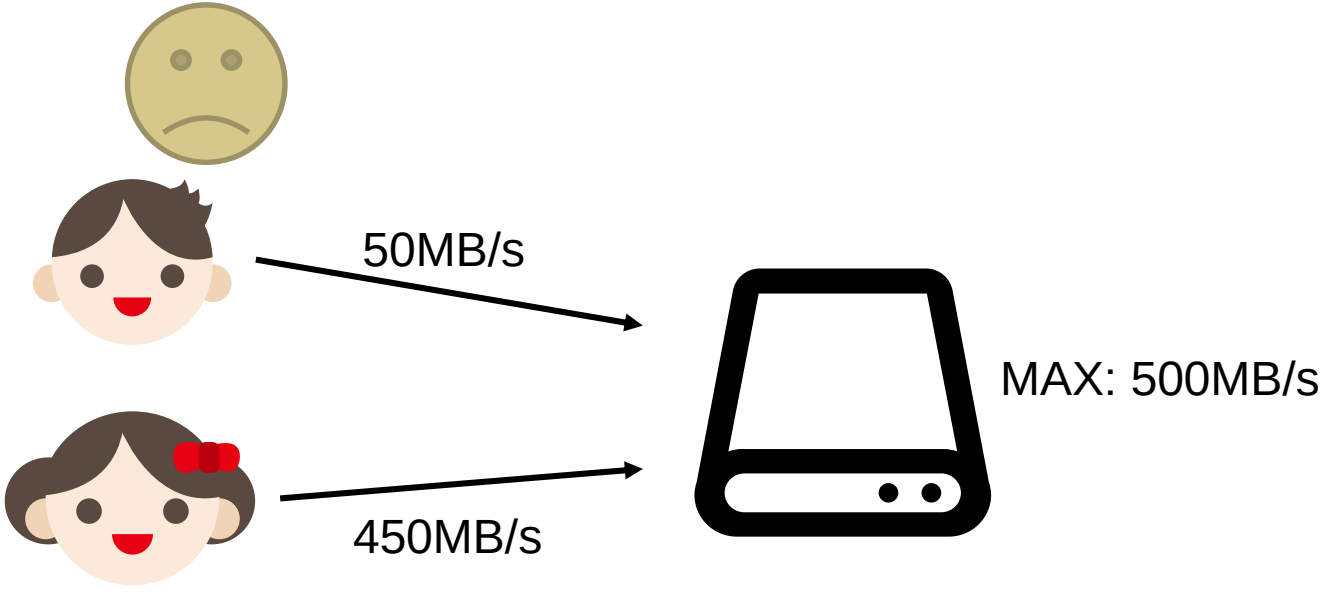
AWS Firecracker



执行环境间的性能隔离

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）

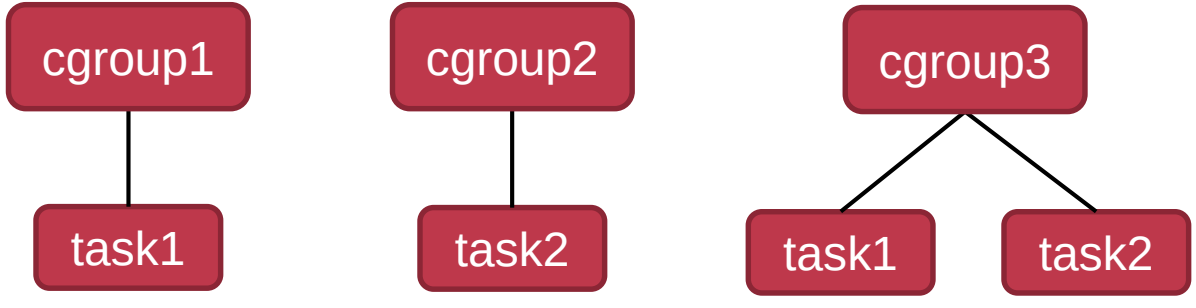
- **系统中的一个线程**
  - 两个任务 task1和task2

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



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

cpu　　cpuacct　　memory

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

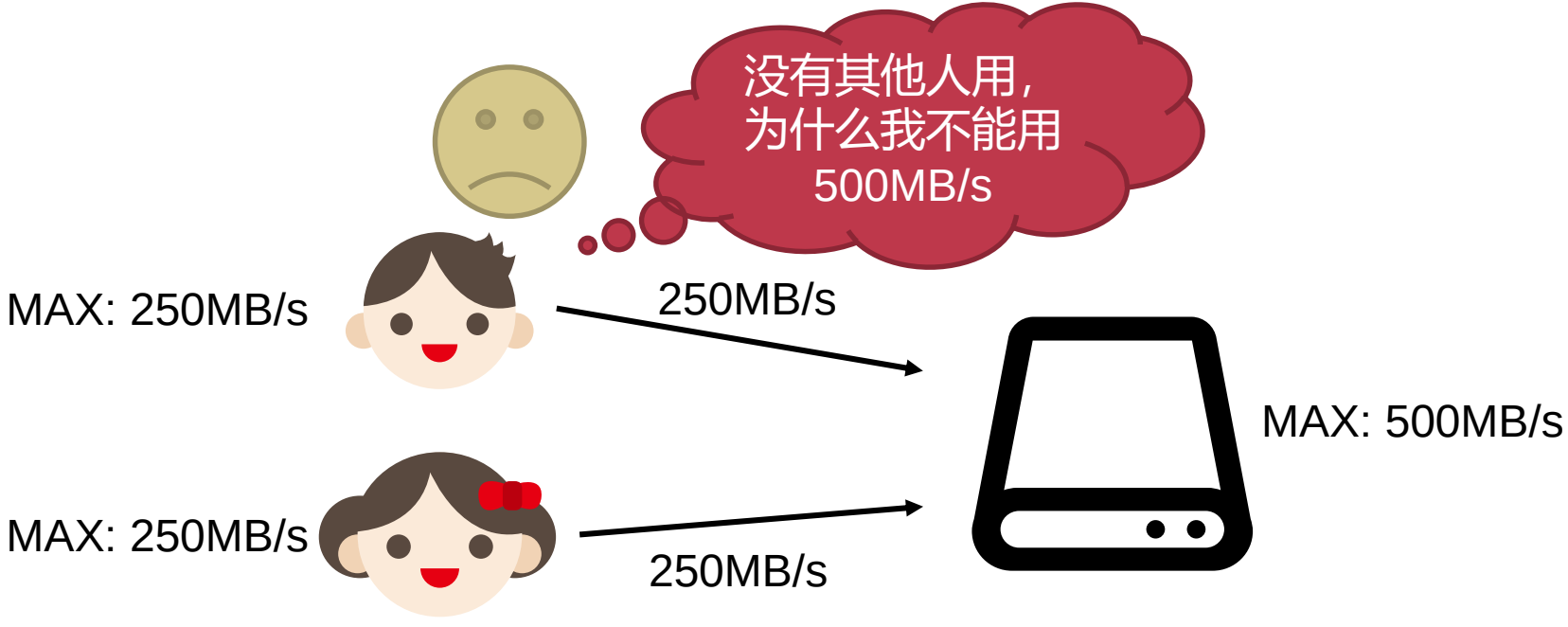
```
$ mount | grep "type cgroup "
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,name=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
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

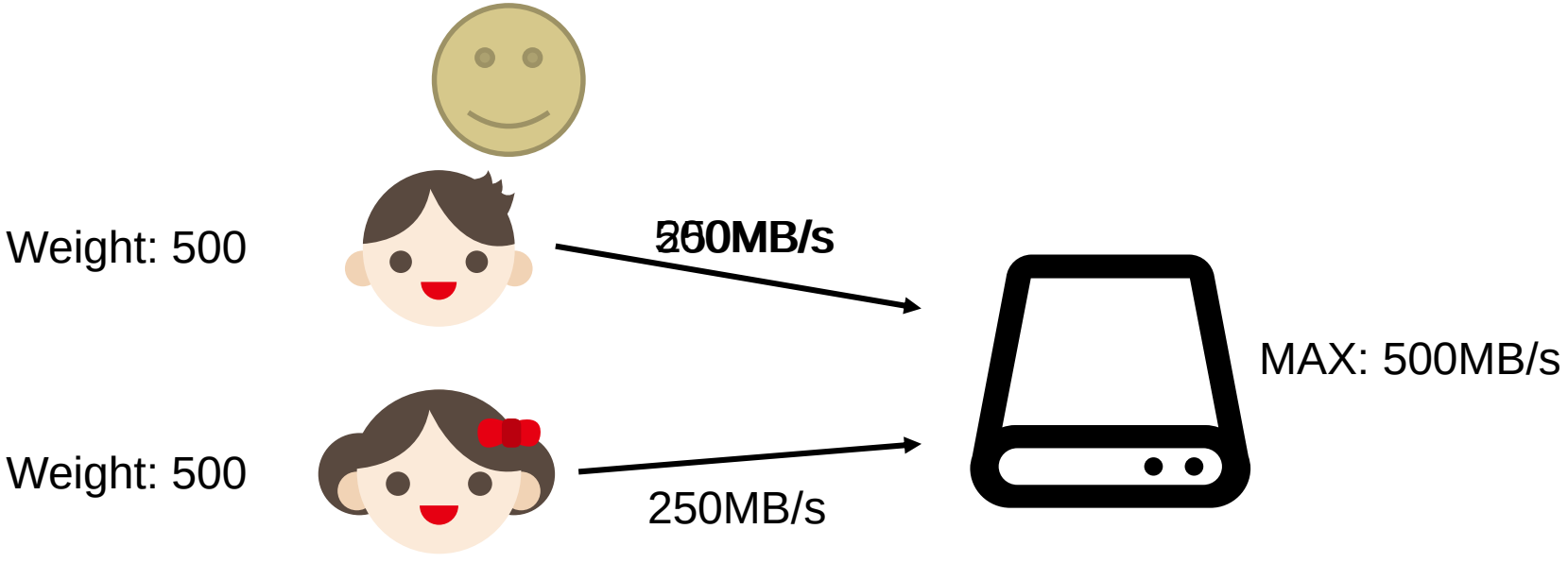


资源控制模型

- **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;
}
```

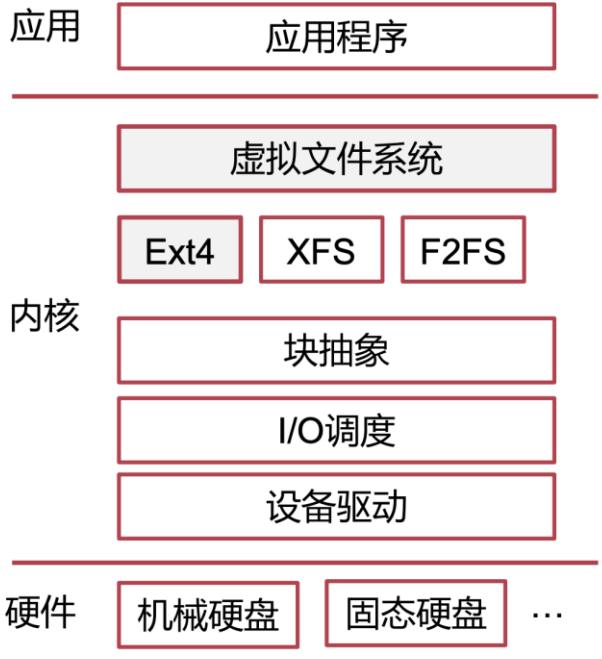
- **释放内存时会执行相反的uncharge操作**

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

回顾Linux中的存储栈

- **你会选择在哪层实现这两种策略?**
- **通用块抽象层**
  - 负责提供和管理块抽象
  - 负责向 I/O调度层提交I/O请求
  - I/O请求的结构体：struct bio
- **I/O调度层**
  - 负责将I/O请求（bio）进行合并和调度
  - 提升I/O性能



最大IOPS/BPS限制（blkio.throttle ）

- **实现于通用块抽象层**
  - 记录当前周期已经发出的IO数量和大小
  - 当新的bio到达时，判断分发了这个bio之后是否超过限制
    - 若未超过限制，则将这个bio分发给I/O调度器
    - 若超过限制，则计算等待时间，到时间再发送这个bio

按比例分配I/O带宽

- **基于完全公平I/O调度器（CFQ）实现**
  - 类似于CFS调度器
  - 每个设备用一棵红黑树维护一些调度队列（CFQ队列）
  - 每个CFQ队列对应唯一的有I/O需求的线程
  - CFQ算法对CFQ队列进行调度
- **CFQ的组调度**
  - 同一控制组的CFQ队列属于同一CFQ_GROUP
  - CFQ算法对CFQ_GROUP进行调度

按比例分配I/O带宽

- **CFQ算法如何选择该该调度的CFQ_GROUP**
  - 每次都选择vdisktime最小的组进行调度
    - 类似于CFS中的vruntime
- **vdisktime如何变化**
  - 在服务完成或时间片耗尽后增加
  - 增加的幅度与权重成反比
    - 权重越大，vdisktime增加越少，被调度到的机会越大

回顾虚拟文件系统中的页缓存

- **为什么需要页缓存**
  - 访问存储设备非常耗时
  - 文件访问具有时间局部性
- **页缓存做了什么**
  - 在一个块被读入内存并访问完成后，并不立即回收内存
  - 在内存中的缓存页被修改后，并不立即将其写回设备
  - 定期或在用户要求时才将数据写会设备

页缓存会给I/O隔离带来什么问题

- **假设以下场景：**
  - 小明要读文件A中的4KB数据，于是将这一页读入页缓存
  - 小红也要访问文件A中的相同页，发现这一页在缓存中，直接从缓存中读到了这页数据
  - 小红将这一页数据读写了10000次
  - 这一页被写回了存储设备

问题：这样公平吗？

- **问：在这个过程中，小明和小红分别发出了多少I/O**
  - 现有实现采取了first-touch policy，谁先访问这一页数据，这一页的所有I/O就算是谁的
  - 所以小明发出了4KB的read和4KB的write，小红没有任何I/O

总结

- 轻量级虚拟化是为了更好的启动性能和运行密度
- 两种思路
  - 更轻量级的虚拟化技术
  - 在内核中增加更多的name space
- 三种隔离技术
  - 虚拟机隔离
  - 容器隔离
  - 虚拟化容器

下次课内容

- **安全**