

# 轻量级隔离

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>

# 虚拟化与隔离

# 云厂商普遍用虚拟化来隔离

- **虚拟化的优势**

- 可以运行完整的软件栈，包括不同的操作系统
- 灵活的整体资源分配（支持动态迁移）
- 方便的添加、删除、备份（只需文件操作）
- 虚拟机之间的强隔离（唯一能抵御 fork bomb 的方法）

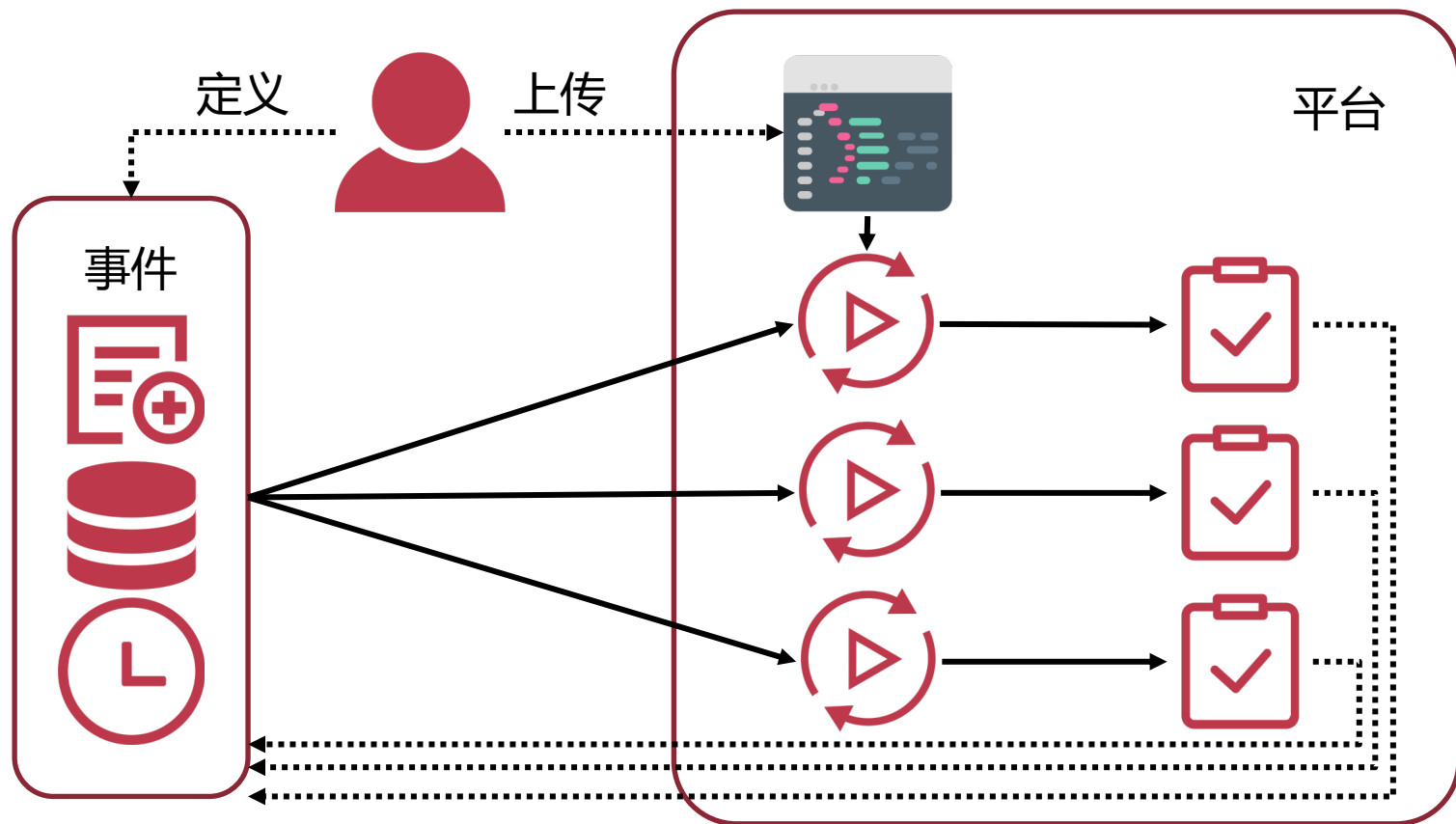
- **虚拟化的问题：太重**

- 云：性能损失，尤其是I/O虚拟化
- 用户：两层操作系统导致资源浪费

# 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

# 如：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



# 是否有更轻量级的隔离？

- **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
- **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

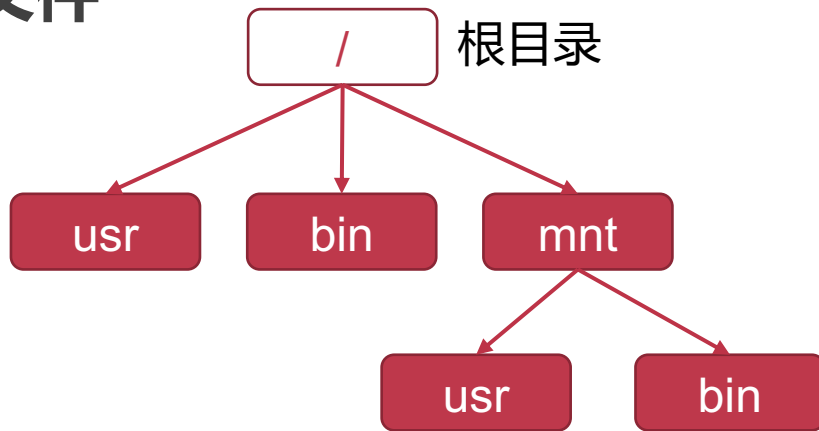
# 第一次尝试：CHROOT

# 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

符号链接层

绝对路径层

路径名层

文件名层

inode number层

文件层

block层



# Chroot能否实现彻底的隔离？

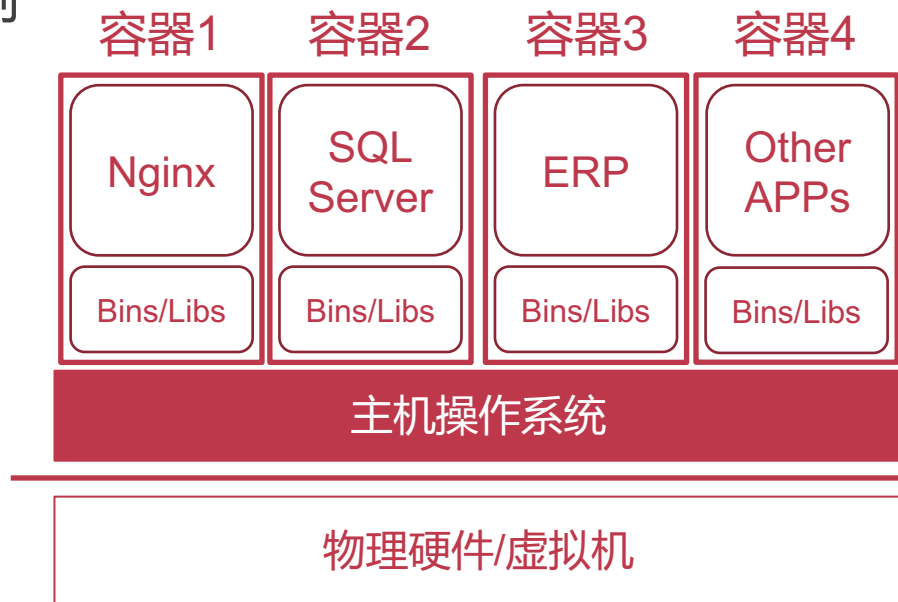
- 不同的执行环境想要共享一些文件怎么办？
- 涉及到网络服务时会发生什么？
  - 所有执行环境共用一个IP地址，所以无法区分许多服务
- 执行环境需要root权限该怎么办？
  - 全局只有一个root用户，所以不同执行环境间可能相互影响
- 不能，因为还有许多资源被共享...



# 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**



# 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

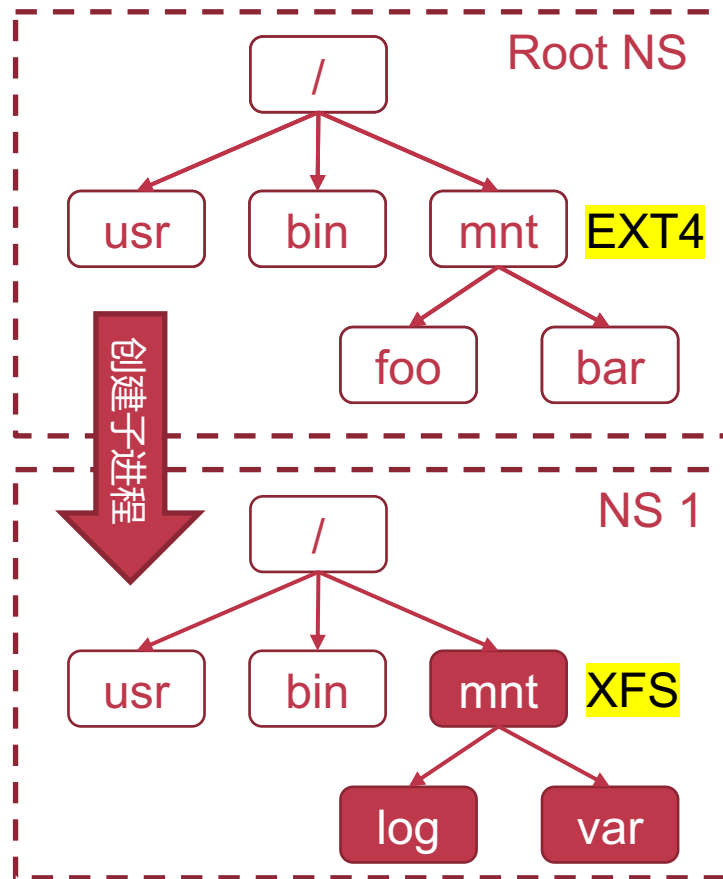
# Mount Namespace的实现

- 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



## 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

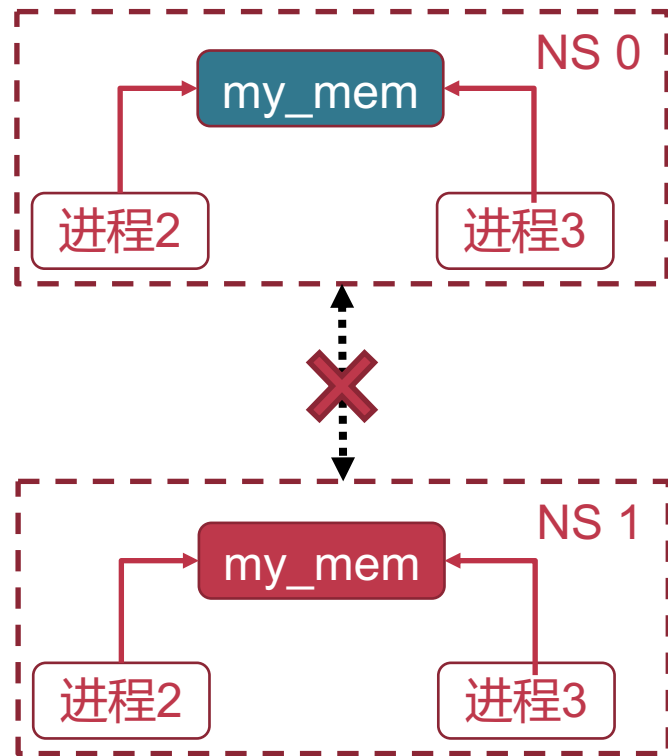


# IPC Namespace的实现

- 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

- 即使不同NS的共享内存ID均为my\_mem → 不同的共享内存

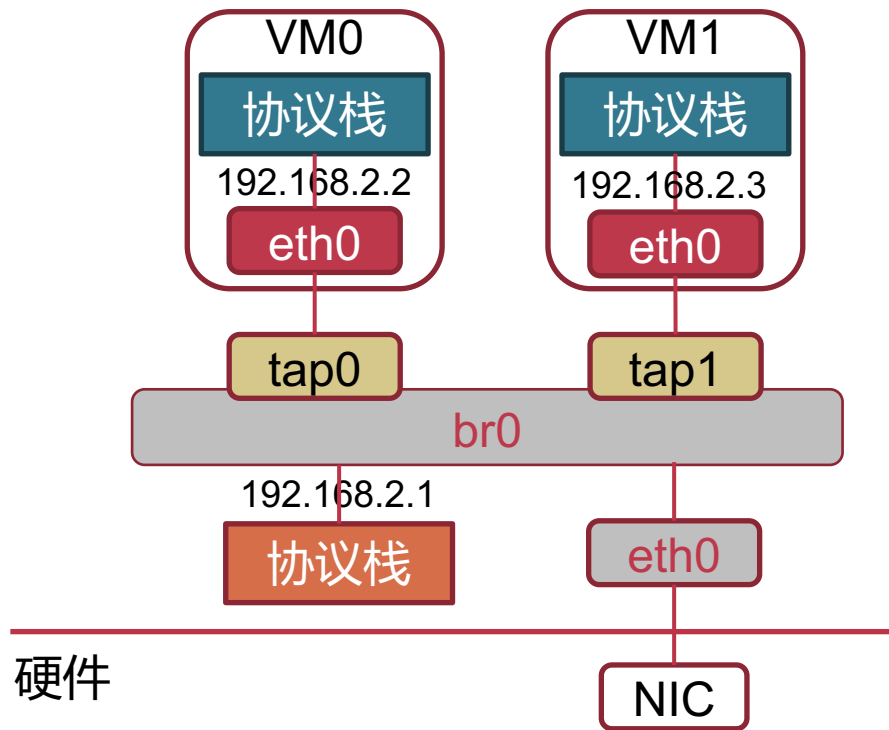


### 3、Network Namespace

- 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

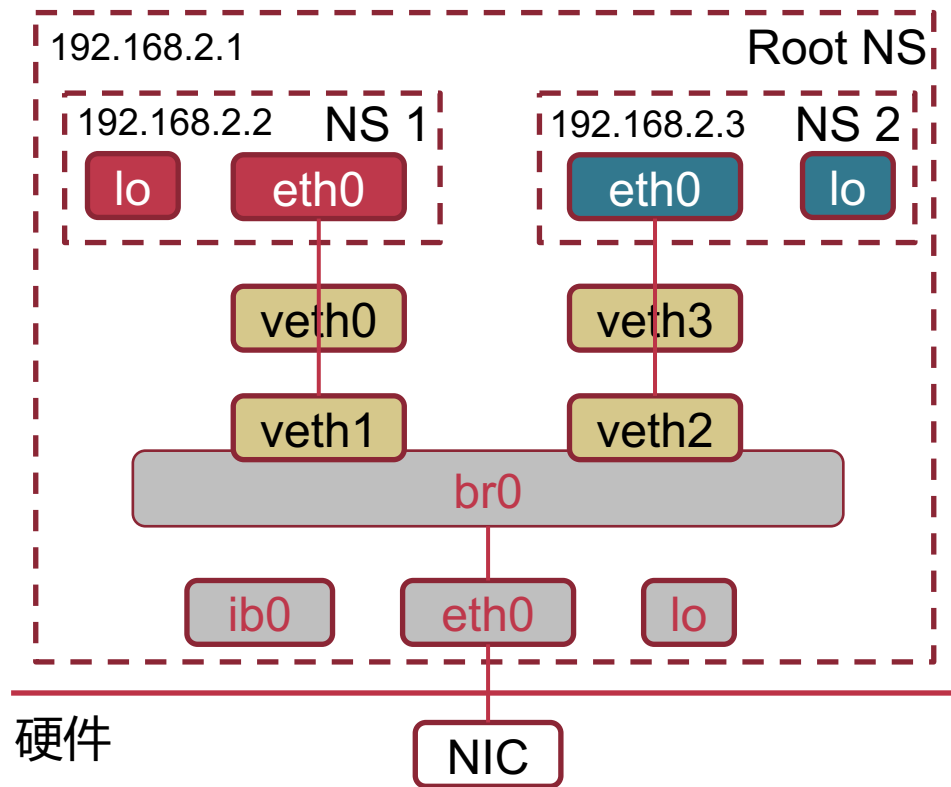
# Linux对于多IP的支持

- 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



## 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**

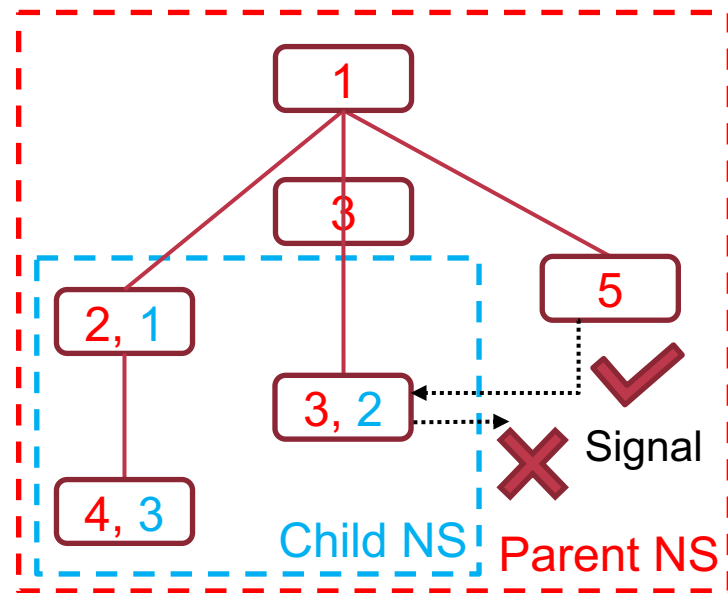
- 允许父NS看到子NS中的进程，保留父子关系

# 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



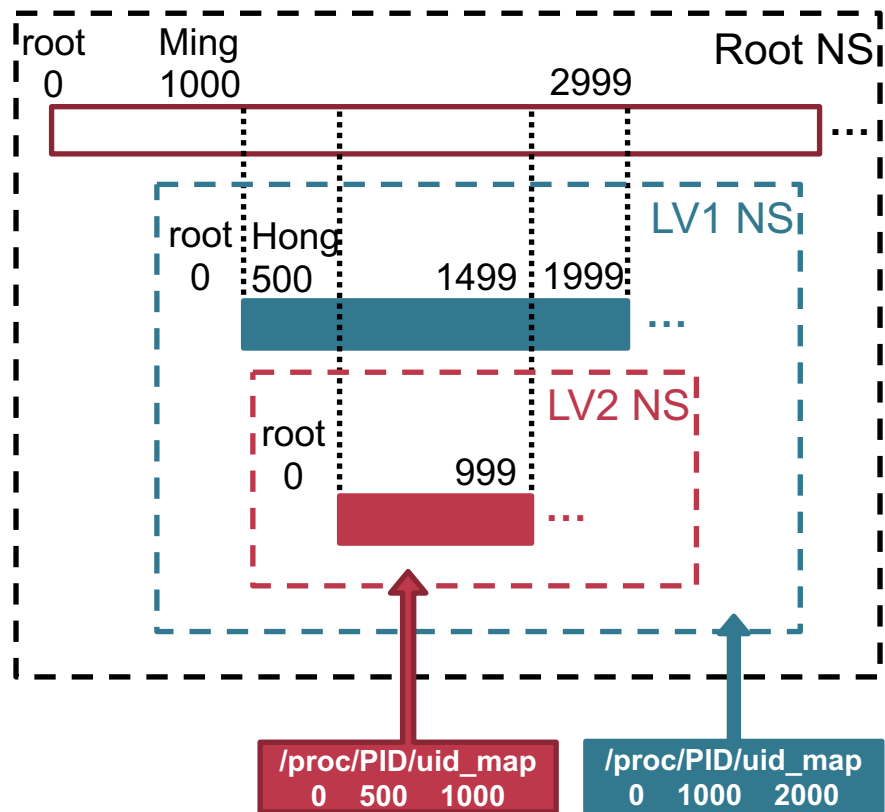
## 5、User Namespace

- 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

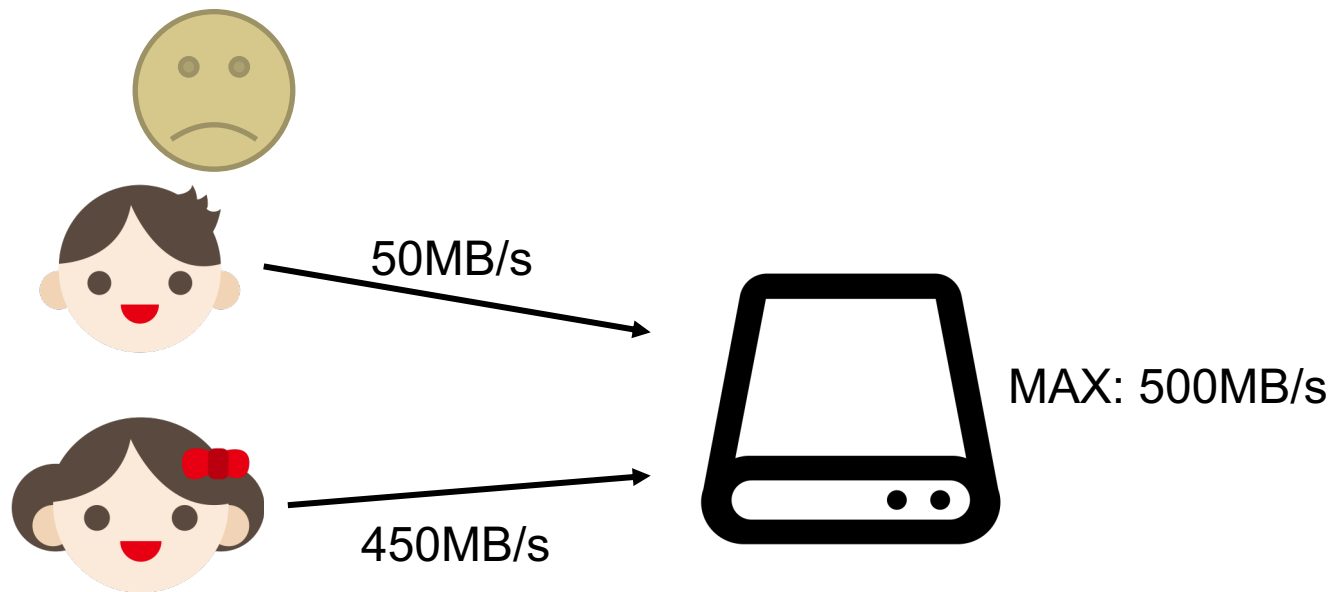
- **7、Cgroup Namespace**

- 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

# 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

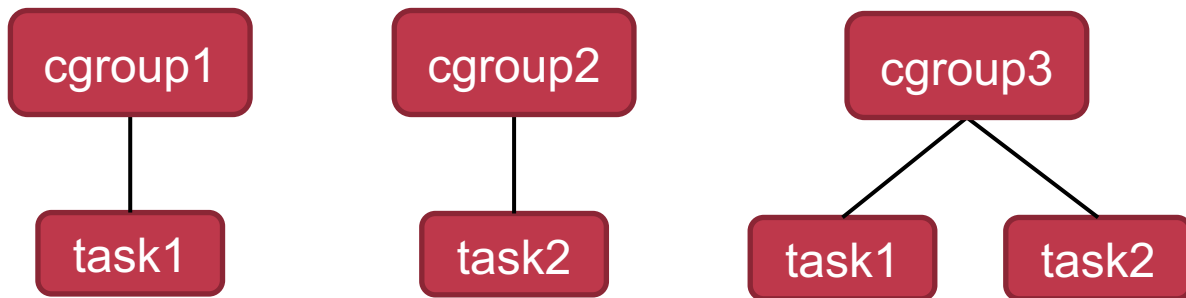
task1

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)
```

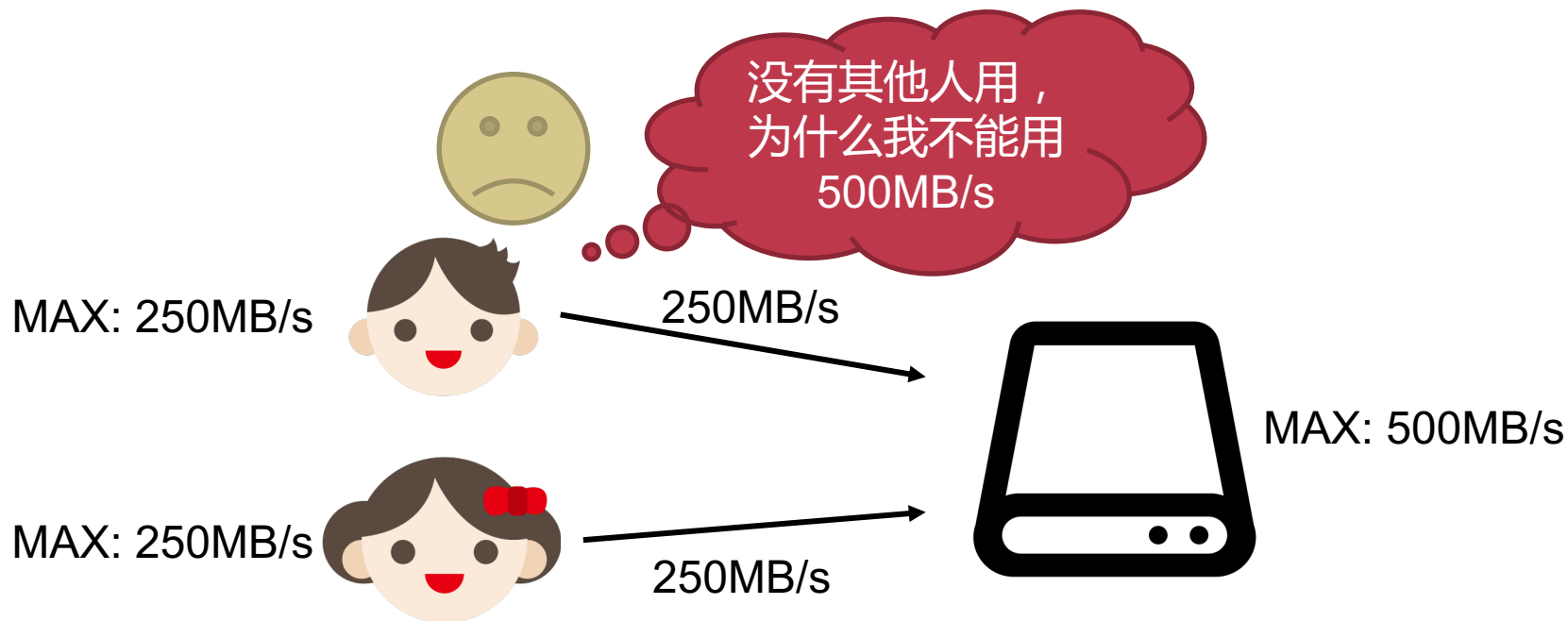
# 资源控制模型

- **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**

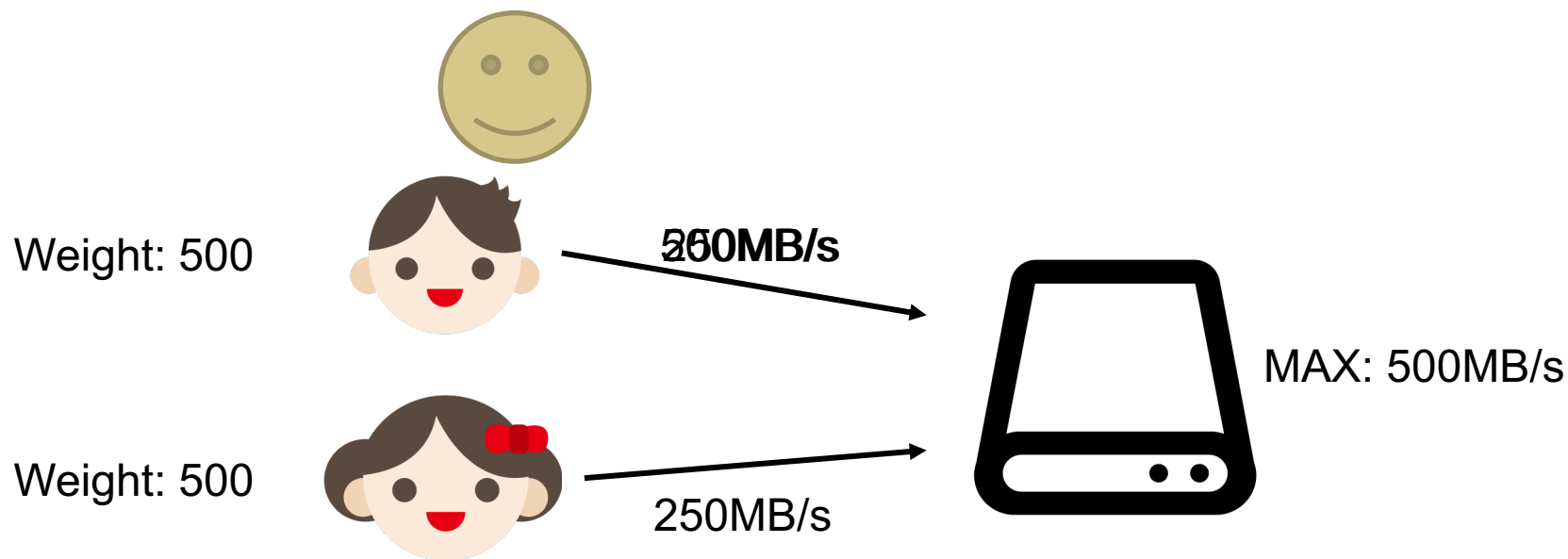
- 直接设置一个控制组所能使用的物理资源的最大值，例如：
  - 内存子系统：最多能使用1GB内存
  - 存储子系统：最大能使用100MB/s的磁盘IO

- **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

# 基于硬件ENCLAVE的隔离

# 硬件隔离环境：Enclave

- **不信任CPU外的硬件**
  - 包括内存（DRAM）、存储设备、网络设备、...
- **仅信任CPU**
  - 包括cache、所有计算逻辑（Anyway，总得信任CPU吧...）
- **Enclave（飞地）**
  - 又称为可信执行环境，TEE（Trusted Execution Environment）

# 硬件提供不同粒度的隔离环境

应用片段抽象



**2015年 Intel SGX**  
由硬件提供高层次接口，但交互漏洞频频

应用抽象



传统的操作系统隔离环境抽象，高度依赖OS语义，硬件不直接参与

容器抽象

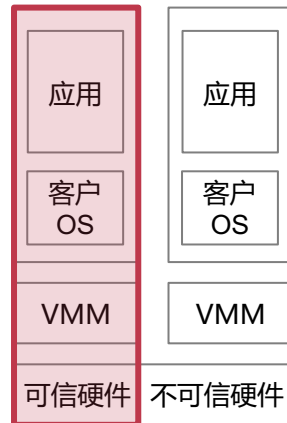


虚拟机抽象



在内部复杂性与交互复杂性间寻找平衡点

物理机抽象



**2003年 ARM TrustZone**  
由于层次结构复杂，导致内部的软件漏洞不断

← 粒度细：内部简单，但交互复杂 ————— 粒度粗：交互简单，但内部复杂 →

# Enclave的隔离方法

- **基于权限控制**
  - 使操作系统没有权限访问用户的数据
- **基于加密**
  - 操作系统即使访问用户数据，也无法解密
- **基于权限控制+加密**
  - 隔离防御软件攻击，加密防御硬件攻击



# 基于权限隔离的隔离方法

- **基于预留的隔离（硬件）**
  - 例如：PRM（Processor Reserved Memory）
  - CPU预留一部分物理内存，不提供给操作系统
- **基于页表的隔离（操作系统）**
  - 例如：保证操作系统无法映射应用的物理内存页
    - 问题：页表是由操作系统自己管理的，监守自盗？
- **基于插桩的隔离（编译器）**
  - 例如：SFI（Software Fault Isolation）
    - 在每次访存前插入边界检查，性能损失较大

# Intel SGX

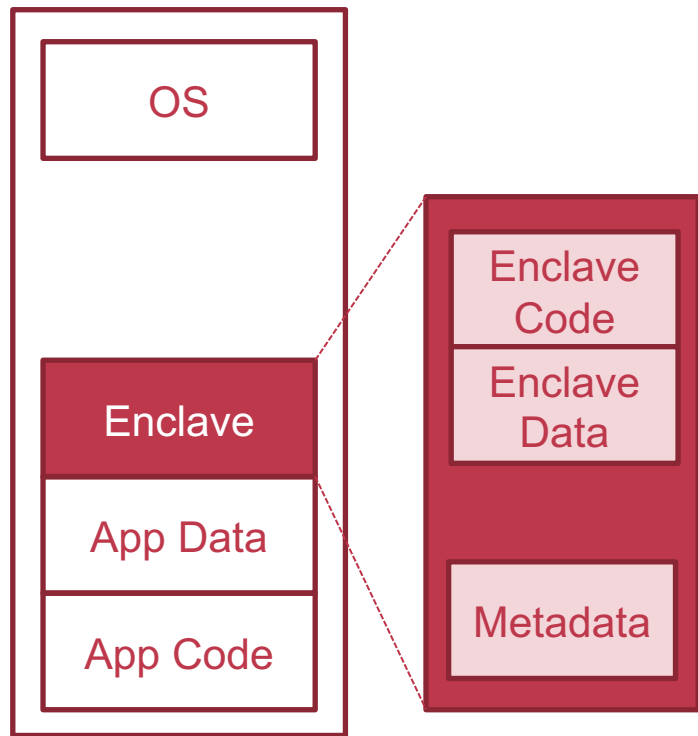
- **SGX: Software Guard eXtension**
  - 2015年首次引入Intel Skylake架构
  - 保护程序和代码在运行时的安全 ( data in-run )
    - 其他安全包括：存储时安全和传输时安全
- **关键技术**
  - Enclave内部与外部的隔离
  - 内存加密与完整性保护
  - 远程验证

# 硬件内存加密与保护机制

- **硬件加密保护隐私性**
  - CPU外皆为密文，包括内存、存储、网络等
  - CPU内部为明文，包括各级Cache与寄存器
  - 数据进出CPU时，由进行加密和解密操作
- **硬件Merkle Tree保护完整性**
  - 对内存中数据计算一级hash，对一级hash计算二级hash，形成树
  - CPU内部仅保存root hash，其它hash保存在不可信的内存中
  - 当内存中的数据被修改时，更新Merkle Tree

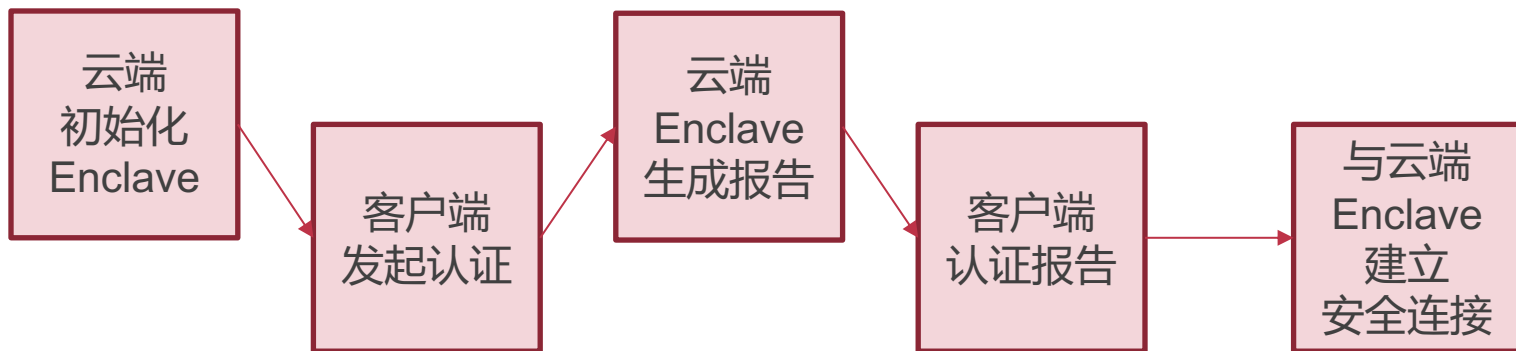
# Enclave与进程的关系

- **Enclave是进程的一部分**
  - Enclave内外共享一个虚拟地址空间
  - Enclave内部可以访问外部的内存
    - 反之则不行
- **创建Enclave的过程**
  1. OS创建进程
  2. OS分配虚拟地址空间
  3. OS将Enclave的code加载到EPC中
    - 并将EPC映射到Enclave的虚拟地址
    - 循环3，完成所有code加载和映射
  4. 完成进程创建



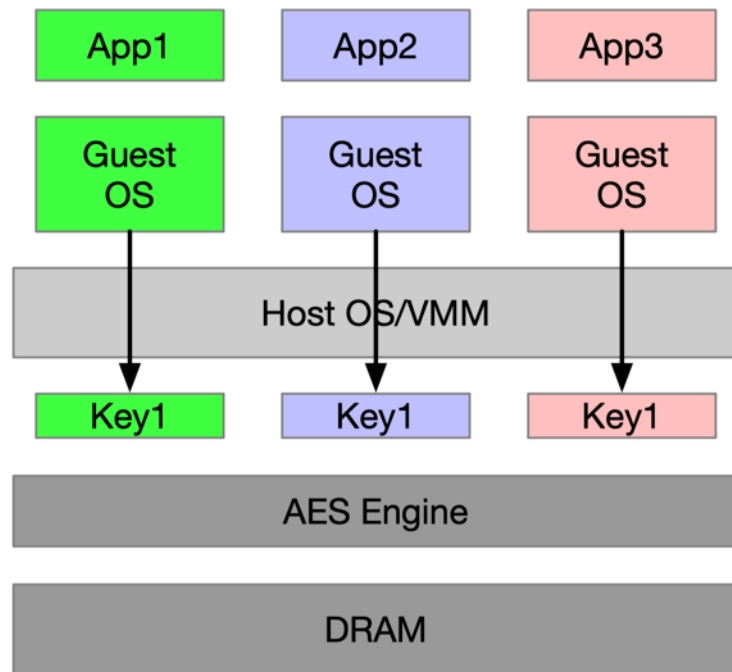
# 远程验证 ( Remote Attestation )

- 要解决的问题：如何远程判断某个主体是Enclave？
  - 例如，如何判断某个在云端的服务运行环境是安全的
  - 必须在认证之后，再进行下一步的操作，例如发送数据



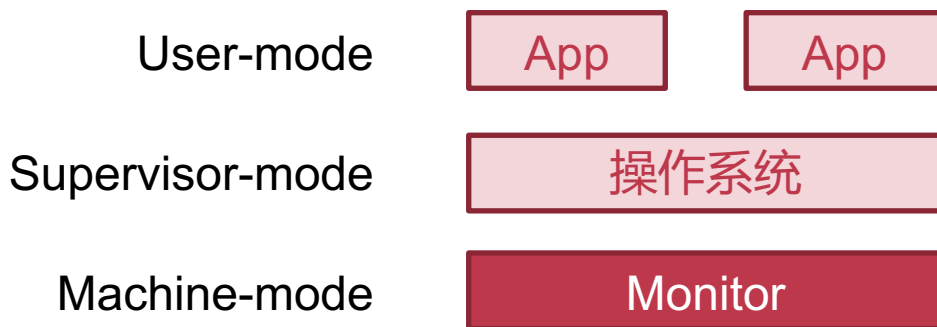
# AMD SEV

- **以虚拟机为粒度的Enclave**
  - 对不同的虚拟机进行加密
  - 每个虚拟机的密钥均不相同
  - Hypervisor有自己的密钥
- **安全模型的缺陷**
  - 依然部分依赖Hypervisor
    - 如：为VM设置正确的密钥



# RISC-V平台的Enclave

- RISC-V具有一个新的模式：Machine-Mode
  - 位于操作系统和Hypervisor之下，直接访问物理地址
  - 具有最高权限，可访问所有的计算资源，并提供新的功能
  - 在M-Mode下实现的软件monitor，可实现Enclave的接口



# 我们自己的Enclave



The screenshot shows the homepage of the Penglai/蓬莱 Enclave project. The header features the project logo and name, followed by the tagline "Open-sourced secure and scalable TEE system for RISC-V." Below this are three buttons: "View on GitHub", "Documentation", and "Team Members". The "Overview" section describes the need for high security-assurance in emerging applications like AI and autonomous cars, contrasting existing systems with the ideal. It lists three reasons: limited scalability, poor performance for high-performance applications, and security limitations. At the bottom, a diagram illustrates the architecture, showing a "User" box containing two "Process" boxes and two "Enclave" boxes, with a dashed line labeled "Penglai isolation" separating the processes from the enclaves.

## Penglai/蓬莱 Enclave

Open-sourced secure and scalable TEE system for RISC-V.

[View on GitHub](#) [Documentation](#) [Team Members](#)

### Overview

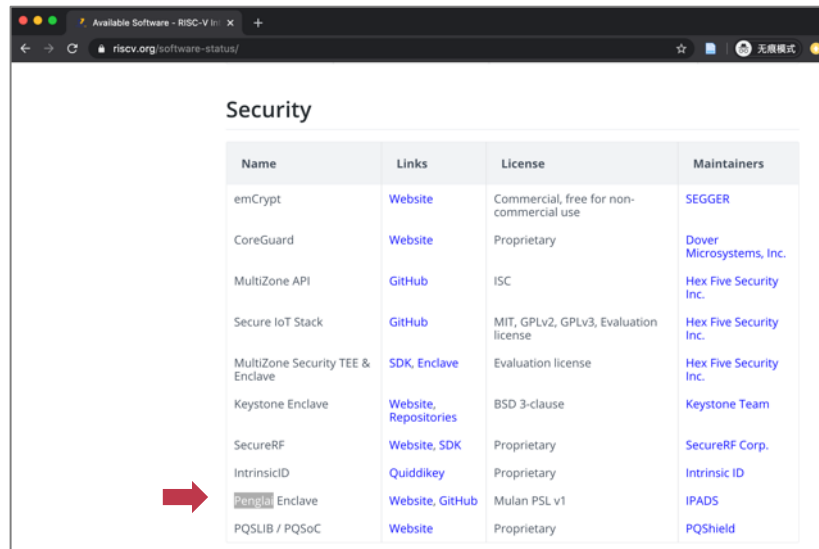
Emerging applications like artificial intelligence and autonomous car require high security-assurance, which stimulates the wide-spread deployment of trusted execution environment (TEE), e.g., Intel SGX, AMD SEV and ARM TrustZone. However, existing enclave systems are far from the ideal for three reasons.

- **Not scalable:** they can only support limited security memory (128MB in SGX) or limited instances (15 instances in SEV);
- Can't support **high-performance applications**, e.g., poor secure communication performance;
- Still have **security limitations**, e.g., PT-based side channels in SGX.

Penglai isolation

User

Process Process Enclave Enclave



The screenshot shows a table titled "Security" listing various software projects. A red arrow points to the "Penglai Enclave" entry in the table.

Name	Links	License	Maintainers
emCrypt	<a href="#">Website</a>	Commercial, free for non-commercial use	SEGGER
CoreGuard	<a href="#">Website</a>	Proprietary	Dover Microsystems, Inc.
MultiZone API	<a href="#">GitHub</a>	ISC	Hex Five Security Inc.
Secure IoT Stack	<a href="#">GitHub</a>	MIT, GPLv2, GPLv3, Evaluation license	Hex Five Security Inc.
MultiZone Security TEE & Enclave	<a href="#">SDK</a> , <a href="#">Enclave</a>	Evaluation license	Hex Five Security Inc.
Keystone Enclave	<a href="#">Website</a> , <a href="#">Repositories</a>	BSD 3-clause	Keystone Team
SecureRF	<a href="#">Website</a> , <a href="#">SDK</a>	Proprietary	SecureRF Corp.
IntrinsicID	<a href="#">Quiddikey</a>	Proprietary	Intrinsic ID
<b>Penglai Enclave</b>	<a href="#">Website</a> , <a href="#">GitHub</a>	Mulan PSL v1	IPADS
PQSLIB / PQSoC	<a href="#">Website</a>	Proprietary	PQShield



<http://penglai-enclave.systems/>



# Enclave隔离的不足

- **仅靠隔离是不够的，还需要考虑交互安全**
  - Enclave依然需要OS提供服务：调度、系统调用、资源分配...
  - 即使隔离，OS依然可能发起的攻击包括
    - 接口攻击：合法的系统调用返回错误的值
      - 例：malloc返回指向栈的地址，导致内部自己破坏掉栈
    - DoS攻击：拒绝分配计算资源（恶意调度）
- **依然受到侧信道等攻击的威胁**
  - Spectre、L1TF