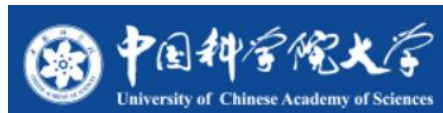




中国科学院软件研究所
Institute of Software, Chinese Academy
of Sciences



机密虚拟化

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。

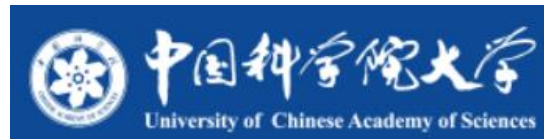


中国科学院软件研究所

Institute of Software, Chinese Academy of Sciences



上海交通大学
SHANGHAI JIAO TONG UNIVERSITY



回顾：实现I/O虚拟化的方式

- 1、设备模拟 (Emulation)
- 2、半虚拟化方式 (Para-virtualization)
- 3、设备直通 (Pass-through)

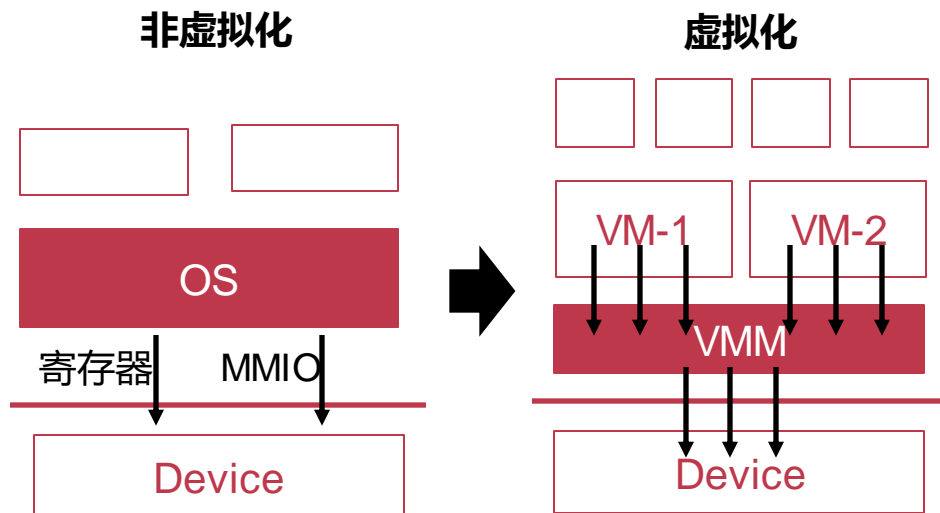
方法1：设备模拟

- OS与设备交互的硬件接口

- 模拟寄存器(中断等)
- 捕捉MMIO操作

- 硬件虚拟化的方式

- 硬件虚拟化捕捉PIO指令
- MMIO对应内存在第二阶段页表中设置为invalid

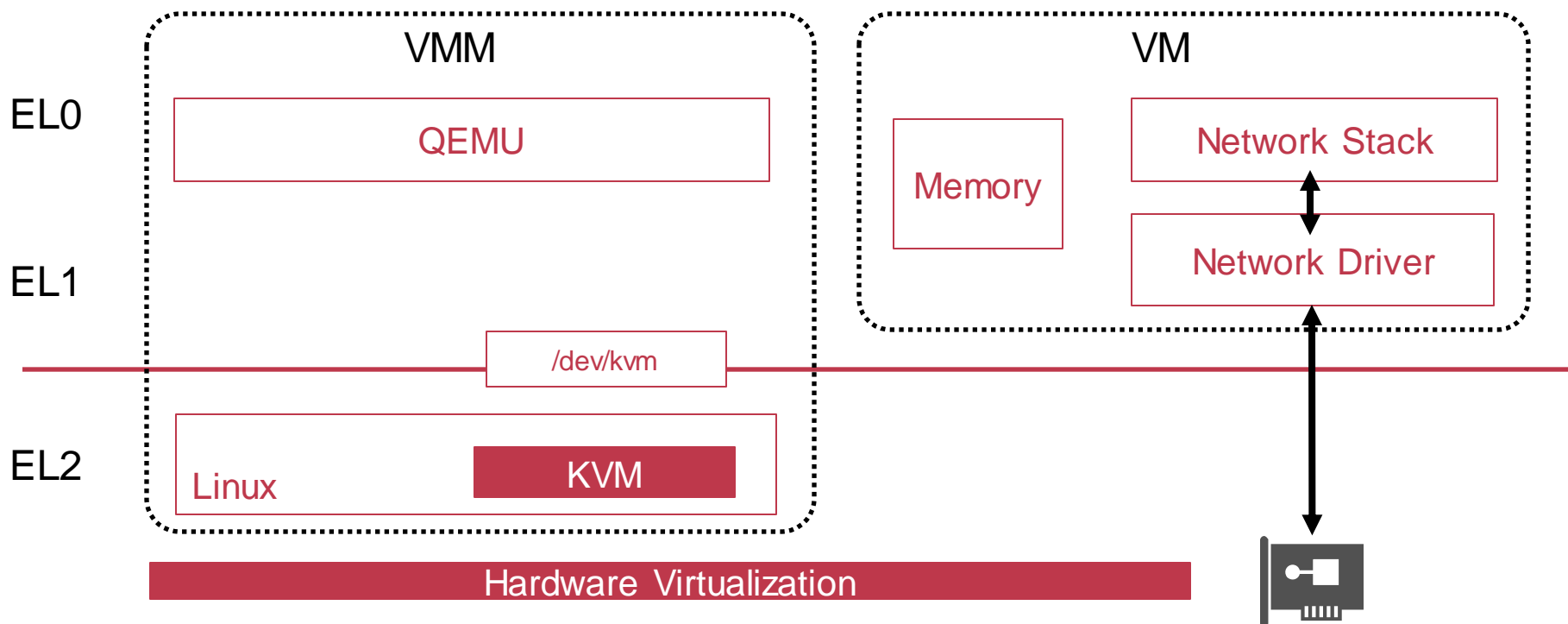


方法2：半虚拟化方式

- 协同设计
 - 虚拟机“知道”自己运行在虚拟化环境
 - 虚拟机内运行前端(front-end)驱动
 - VMM内运行后端(back-end)驱动
- VMM主动提供Hypercall给VM
- 通过共享内存传递指令和命令

方法3：设备直通

- 虚拟机直接管理物理设备



I/O虚拟化技术对比

| | 设备模拟 | 半虚拟化 | 设备直通 |
|---------------|------|-------|--------|
| 性能 | 差 | 中 | 好 |
| 修改虚拟机内核 | 否 | 驱动+修改 | 安装VF驱动 |
| VMM复杂度 | 高 | 中 | 低 |
| Interposition | 有 | 有 | 无 |
| 是否依赖硬件功能 | 否 | 否 | 是 |
| 支持老版本OS | 是 | 否 | 否 |

中断虚拟化

- **VMM在完成I/O操作后通知VM**
 - 例如在DMA操作之后
- **VMM在VM Entry时插入虚拟中断**
 - VM的中断处理函数会被调用
- **虚拟中断类型**
 - 时钟中断
 - 核间中断
 - 外部中断

RISC-V中断虚拟化的实现方法

- **打断虚拟机执行**

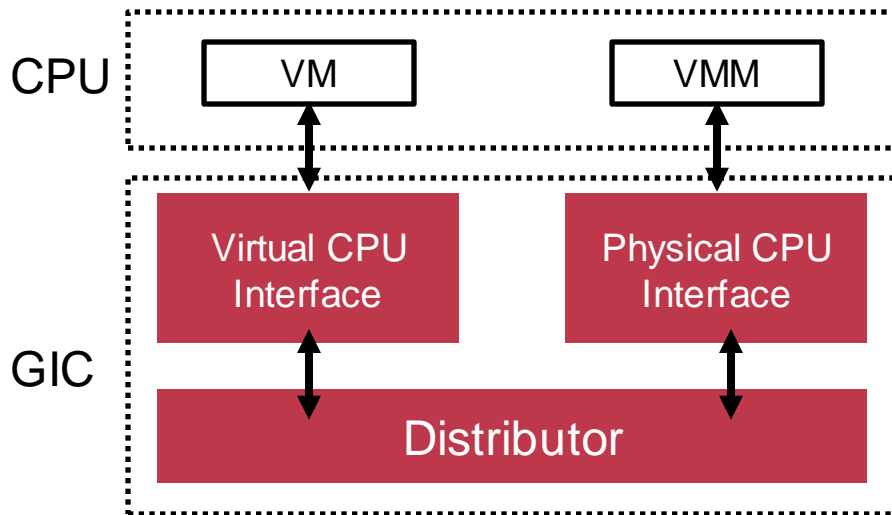
- 通过VSIP寄存器注入虚拟的中断
- 通过SBI接口注入核间中断和时钟中断不中断虚拟机执行

- **不中断虚拟机执行**

- 通过PLIC/VirtIO插入中断

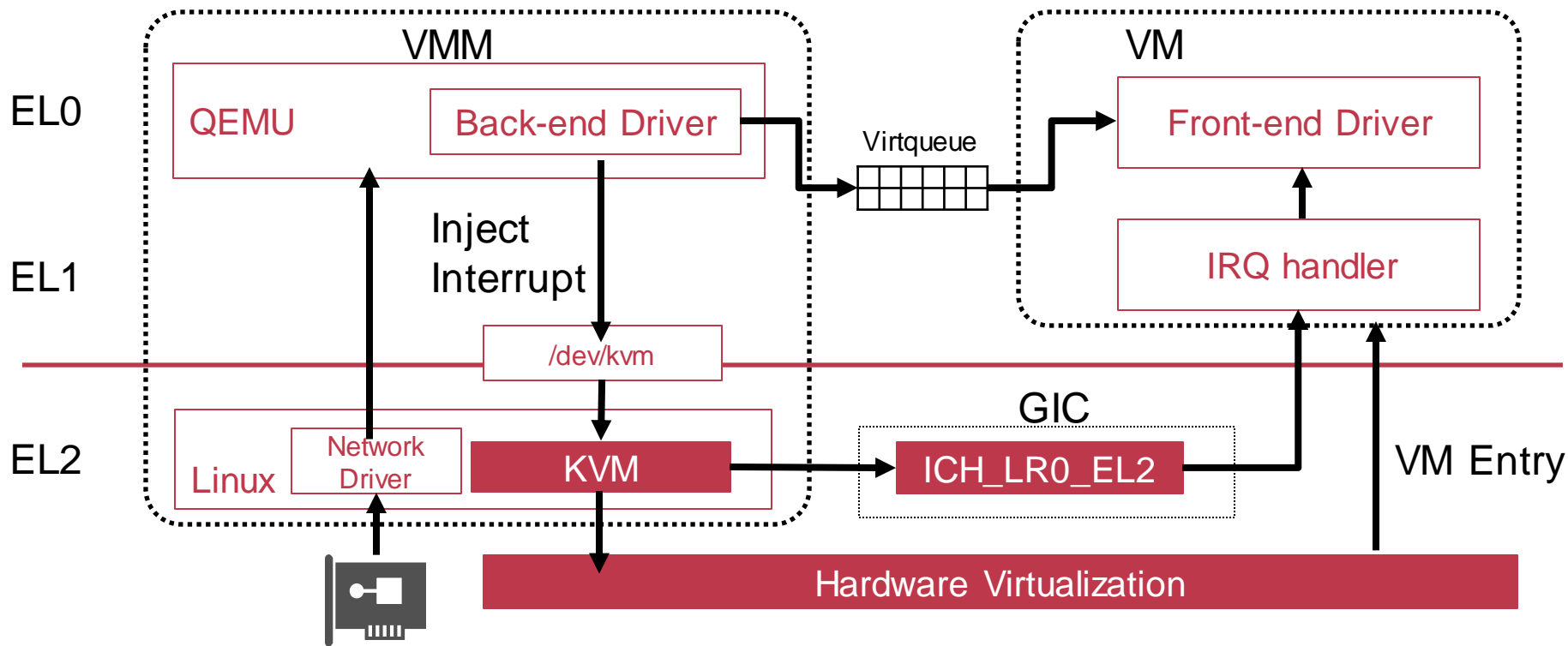
Virtual CPU Interface

- **GIC为虚拟机提供的硬件功能**
 - VM通过Virtual CPU Interface与GIC交互
 - VMM通过Physical CPU Interface与GIC交互



插入虚拟中断：以半虚拟化举例

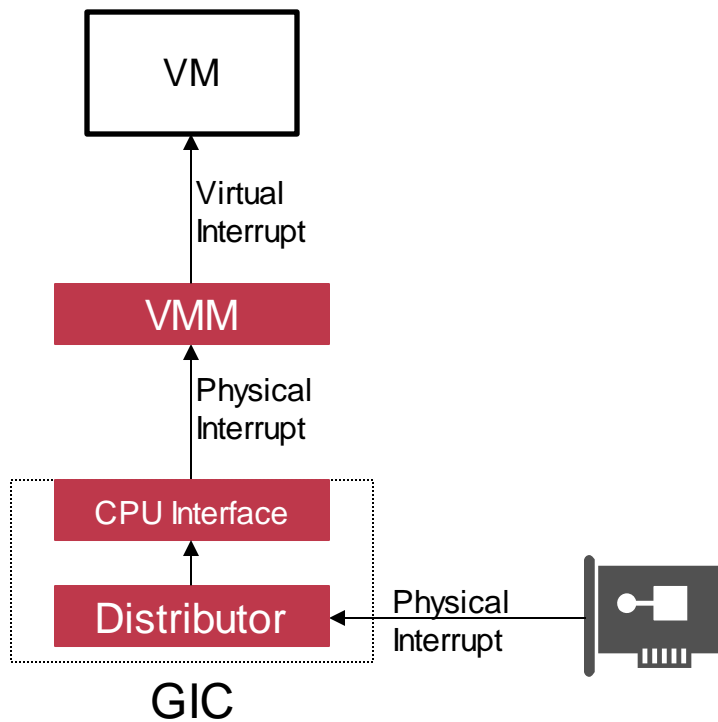
思考：这种方式有什么问题？



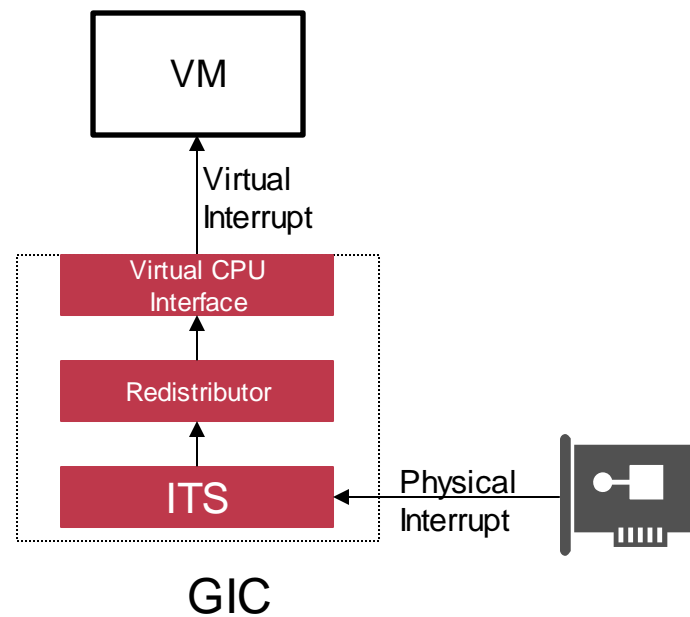
不打断虚拟机执行：GIC ITS

- **GIC第4版本推出了Direct injection of virtual interrupts**
 - 将物理设备的物理中断与虚拟中断绑定
 - 物理设备直接向虚拟机发送虚拟中断
- **VMM在运行VM前**
 - 配置GIC ITS (Interrupt Translation Service)
 - 建立物理中断与虚拟中断的映射
 - 映射内容
 - 设备与物理中断的映射
 - 分配虚拟中断号
 - 发送给哪些物理核上的虚拟处理器

虚拟中断的直接插入



GICv3中的物理中断插入



GICv4中的物理中断插入

Confidential Virtualization



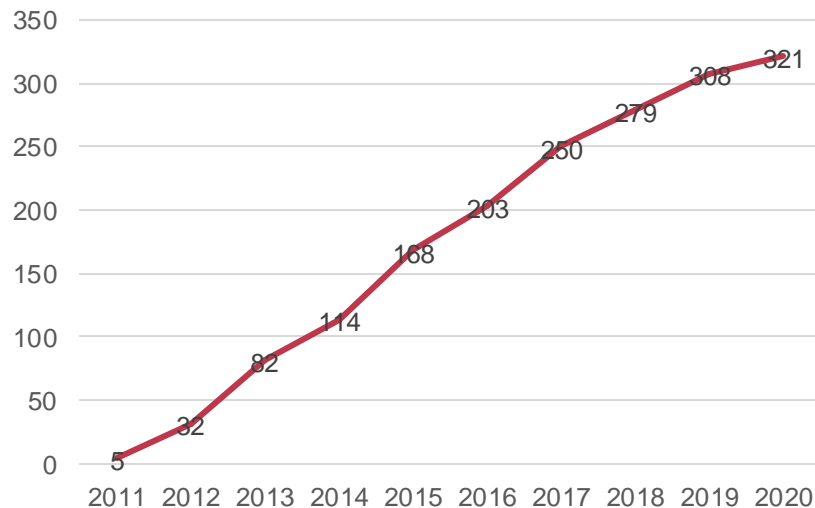
机密虚拟化

当虚拟机监控器不可信

- **系统的复杂性**
 - 软件：恶意软件，虚拟机监控器本身可能存在漏洞
 - 人：运维外包（如云计算等）导致接触计算机的人更复杂
- **不可信的虚拟机监控器**
 - 可以查看任意虚拟机的vCPU寄存器、内存内容、I/O数据
 - 可以修改虚拟机的状态
 - VMM控制着页表，可直接映射虚拟机的内存并读取数据
 - 可以控制虚拟机的行为
 - VMM控制着页表，可直接在应用内部新映射一段恶意代码
 - VMM可任意改变虚拟机的RIP，劫持其执行流

虚拟机监控器的漏洞数

- **Xen CVE is growing**
 - LoC: from 45K (v2.0) to 2,649K (v4.14.0)
 - 321 XSA
- **KVM and VMware**
 - KVM: 110+ CVE
 - VMware: 140+



<https://xenbits.xen.org/xsa/>

虚拟机监控器漏洞的危害

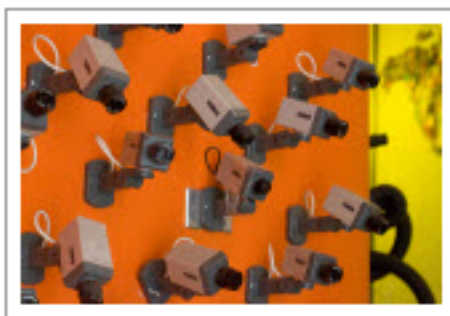
- 虚拟机提权(Privilege Escalation)
- 隐私数据泄露(Data Leakage)
- 拒绝服务攻击(Denial of Service)

| Hypervisor | Total | Host | | | Other | LoC |
|------------|-------|------|-----|----|-------|------|
| | | PE | DoS | DL | | |
| KVM | 127 | 17 | 53 | 10 | 47 | 57K |
| Xen | 423 | 109 | 189 | 25 | 100 | 302K |

来自管理员的威胁 (Insider Threats)

Google Fires Employee Accused Of Spying On Kids

By [Phil Villarreal](#) on September 16, 2010 9:15 AM



(RAWRZ!)

A Google

For a Google engineer who was fired in July, it apparently wasn't enough just to Google people in order to stalk them. Instead, he allegedly abused his access and violated the company's privacy policies to snoop on users.

Valleywag **reports** the man spied on four teenagers, peeking in on emails, chats and Google Talk call logs for several months before the company discovered what was going on.

..., peeking in on emails, chats and Google Talk call logs for several months

“

"We dismissed [redacted] for breaking Google's strict internal privacy policies. We carefully control the number of employees who have access to our systems, and we regularly upgrade our security controls—for example, we are significantly increasing the amount of time we spend auditing our logs to ensure those controls are effective. That said, a limited number of people will always need to access these systems if we are to operate them properly, which is why we take any breach so seriously."

一种新的威胁模型：安全处理器

- **不信任CPU外的硬件**
 - 包括内存（DRAM）、设备、网络
- **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）
 - 什么是“可信”？信什么呢？

Enclave/TEE：可信执行环境

- **Enclave/TEE的定义**

- Enclave，又称"可信执行环境"（TEE, Trusted Execution Environment），是计算机系统中一块通过底层软硬件构造的安全区域，通过保证加载到该区域的代码和数据的完整性和隐私性，实现对代码执行与数据资产的保护 —— *Wikipedia*

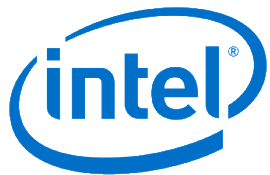
- **Enclave的两个主要功能**

- 远程认证：验证远程节点是否为加载了合法代码的Enclave
- 隔离运行：Enclave外无法访问Enclave内部的数据

- **Enclave带来的能力：限制访问数据的软件**

- 可保证数据只在提前被认证的合法节点间流动
 - 合法节点：部署了合法软件的节点

Enclave/TEE在工业界发展迅速



✧ Intel SGX/TDX



✧ AMD SEV



✧ ARM CCA



✧ Keystone, 蓬莱

- 多家云厂商利用TEE/Enclave保护数据

- 2018年, Microsoft Azure 基于Intel SGX推出Confidential Computing
- 2019年, Amazon 推出了Nitro Enclave保护用户的关键数据
- 2020年, Google Cloud 基于AMD SEV推出加密虚拟机Secure VM

- 2019年8月, 多家公司成立机密计算联盟

- 国内公司包括: 华为、阿里云、腾讯、百度、字节跳动等
- 国外公司包括: Arm、AMD、Intel、Redhat、Facebook、Google等



硬件Enclave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

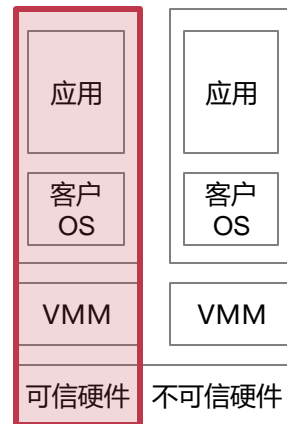


虚拟机抽象



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

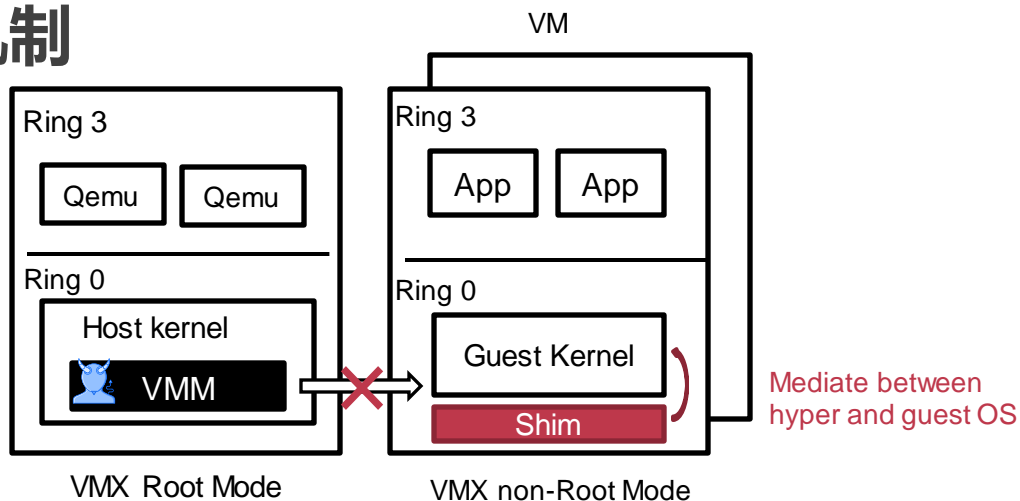
← 粒度细：内部简单，但交互复杂 ————— 粒度粗：交互简单，但内部复杂 →

机密虚拟化

- **一种通过基于硬件的可信执行环境来保护虚拟机数据的机制和方法**
 - 保护虚拟机免受不可信虚拟机监控器和管理员的威胁
- **关键技术**
 - 虚拟机内部与外部的隔离
 - 内存加密和完整性保护
 - 远程验证

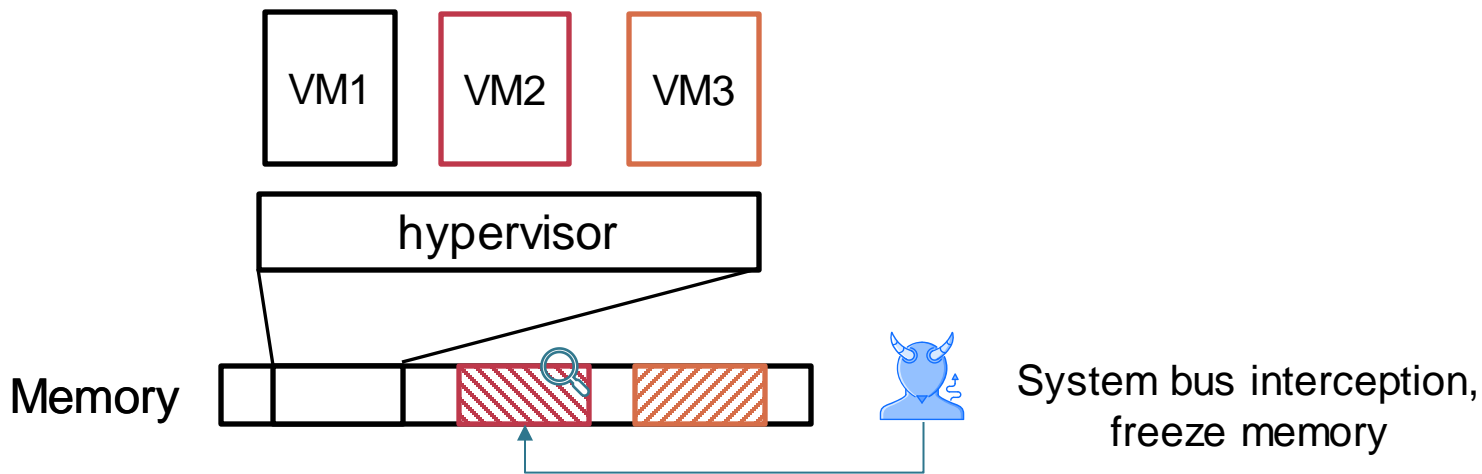
可信固件

- 由CPU厂商开发的可信软件，比虚拟机监控器的特权级更高
- 可信固件能够访问所有物理资源，对系统进行配置并提供保护机制



内存隔离机制

- 内存隔离：机密虚拟机的内存范围不能被其他虚拟机和虚拟机监控器访问



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

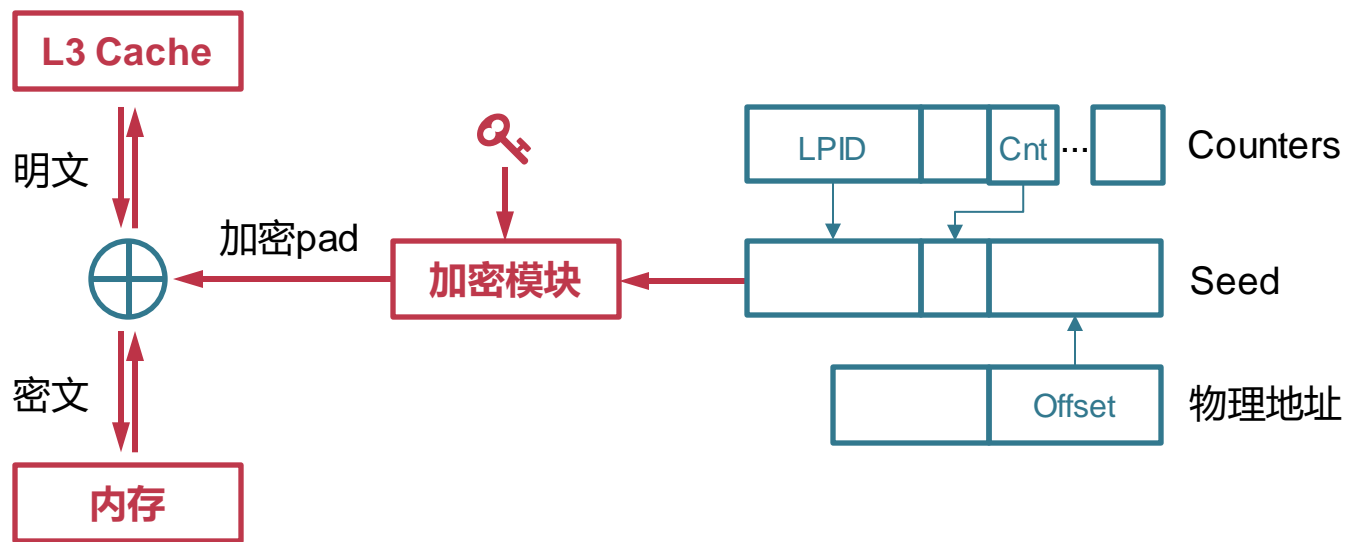
- **硬件Merkle Tree保护完整性**

- 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

硬件内存加密

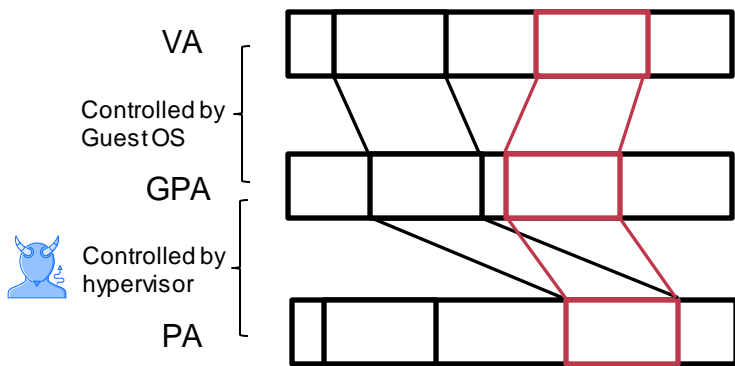
- **方法一：单密钥加密**
 - 缺点：同样的明文会产生同样的密文
- **方法二：多密钥加密**
 - 缺点：如何保存这些密钥？CPU内部放不下
- **方法三：单密钥 + 多 seed**
 - 为每个cache line单独生成一个seed，用密钥加密后，对数据进行异或

生成seed用于加密

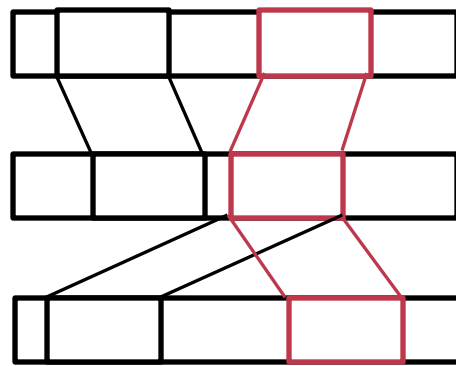


内存完整性保护

- 内存数据的完整性保护：机密虚拟机的内存数据不可被恶意修改
- 内存映射的完整性保护：机密虚拟机的第二阶段页表映射不可被恶意改变



Memory aliasing



Memory remapping

内存完整性保护

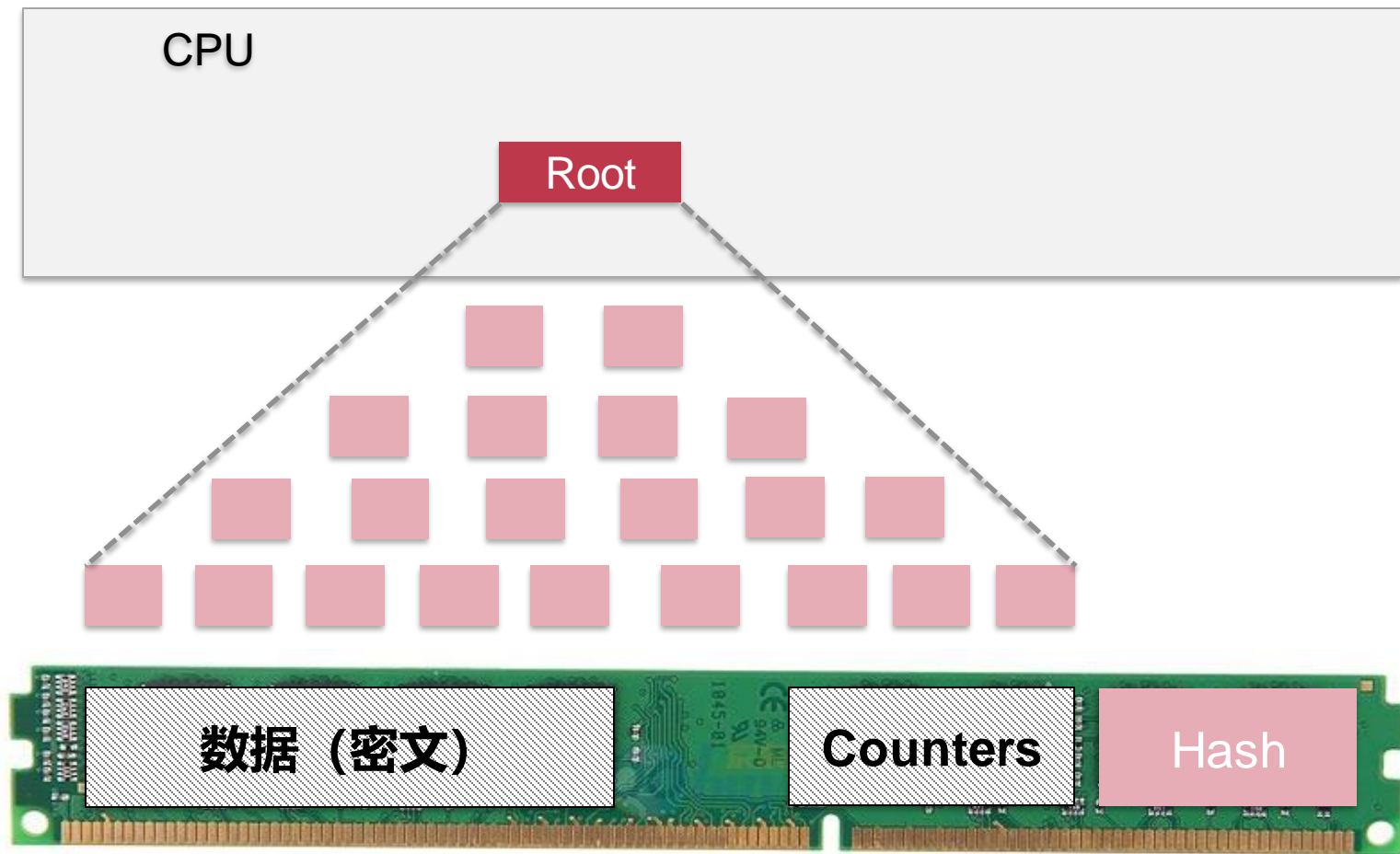
- **Merkle hash Tree**

- 可以保证内存不会受到拼接和欺骗攻击
 - 不知道hash key无法计算对应的mac
- 无法防御回放攻击
 - 攻击者可以将mac和data同时替换成老版本

- **将root hash (mac) 存储在CPU中**

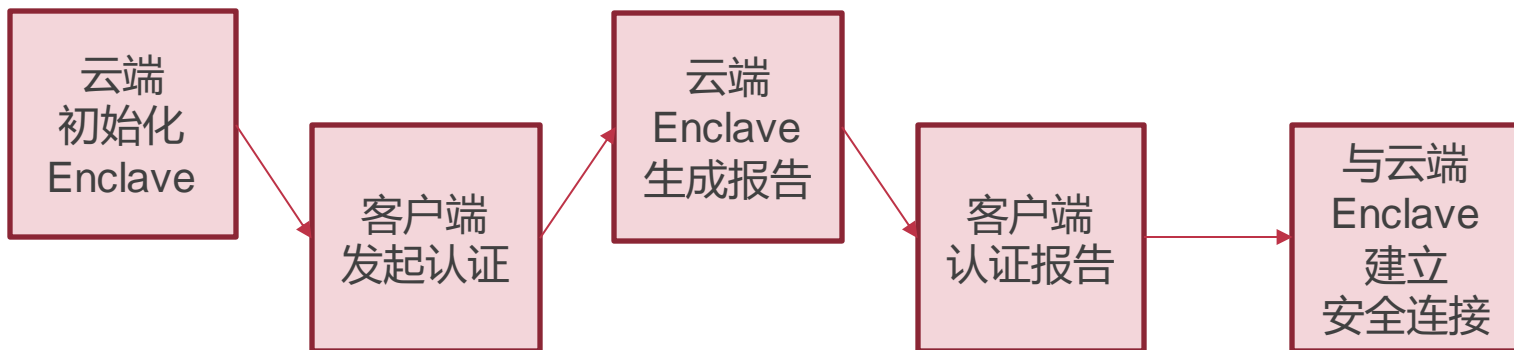
- 防御回放攻击
 - 攻击者无法修改root mac的值

内存完整性保护：Merkle Hash Tree



远程验证 (Remote Attestation)

- 要解决的问题：如何远程判断某个主体是机密虚拟机？
 - 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



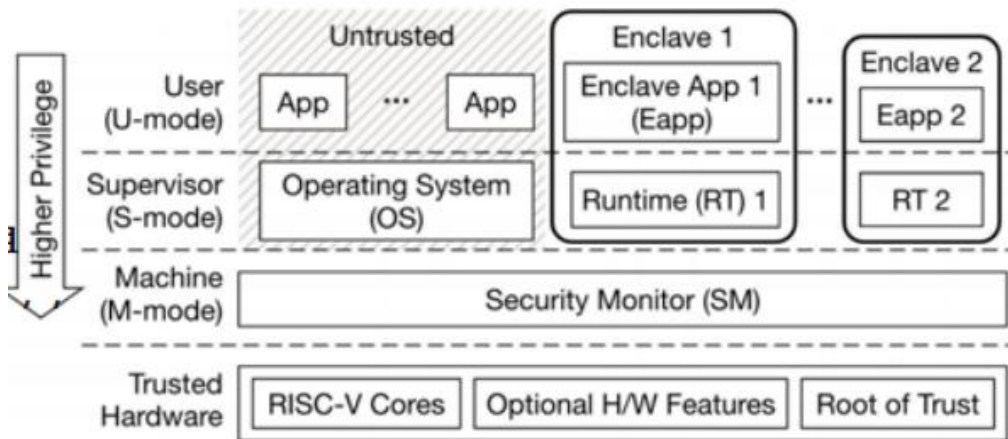
机密虚拟化案例分析：RISC-V KEYSTONE

RISC-V PMP技术

- **RISC-V Priv 1.10标准引入了物理内存保护（PMP）功能。**
 - 一个很强大的原语
- **PMP通过一组控制和状态寄存器（CSR）来实现。**
 - 在U模式下，只能访问那些由PMP配置为允许访问的物理内存区域。
 - 在S模式下，可以访问所有未被禁止的物理内存区域。
 - 然而，只有M模式下可以配置和更改PMP寄存器的状态。
- **Keystone需要PMP来实现enclave的内存隔离。**

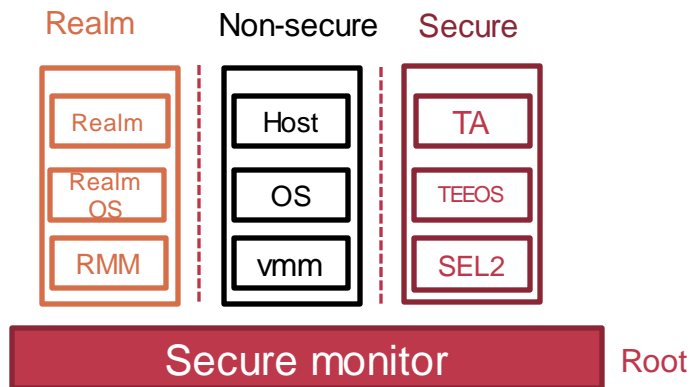
RISC-V Keystone

- RISC-V社区开源的框架
 - 旨在构建自定义的可信环境（TEE）
 - 更加灵活和可定制
- 同时创建多个互相隔离的飞地（Enclave）
 - 每个飞地都有自己的运行时环境、应用程序和数据
 - 提高了系统的可靠性



世界状态World State和内存隔离

- 4个世界状态: Non-secure, Realm, secure, root
- 4个Physical Address Space (PAS)
 - Non-secure PAS
 - Secure PAS
 - Realm PAS
 - Root PAS



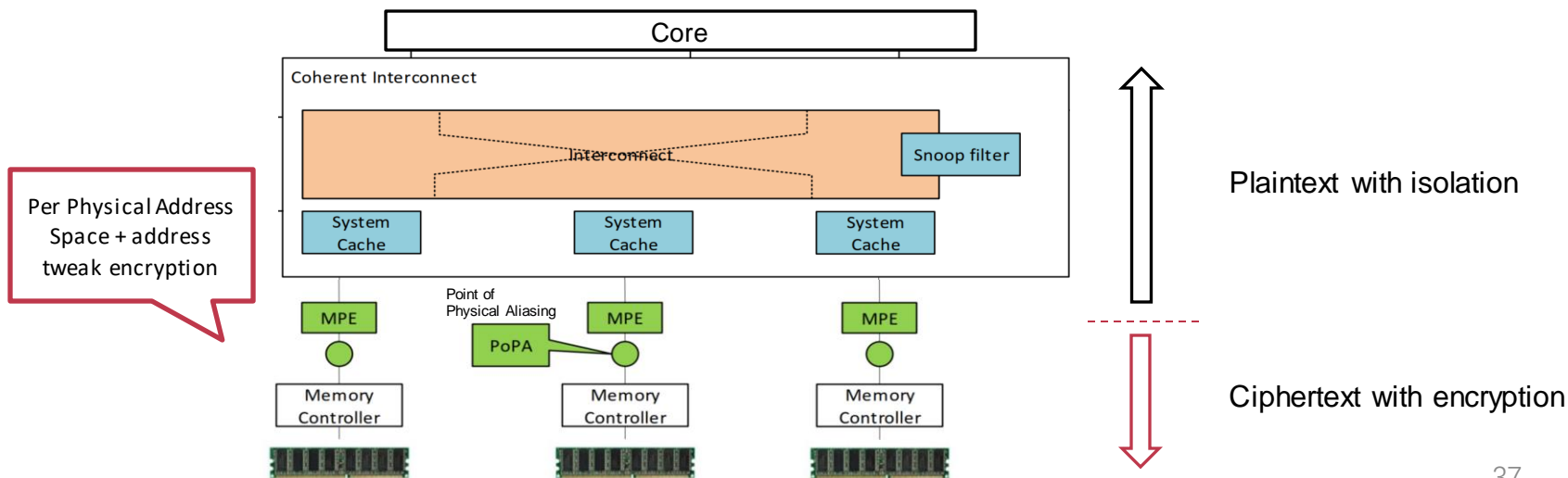
内存访问权限表

| | Non-secure | Realm | Secure | Root |
|------------|------------|-------|--------|------|
| Non-secure | ✓ | ✗ | ✗ | ✗ |
| Realm | ✓ | ✓ | ✗ | ✗ |
| Secure | ✓ | ✗ | ✓ | ✗ |
| Root | ✓ | ✓ | ✓ | ✓ |

内存加密

- 内存保护引擎 (MPE)

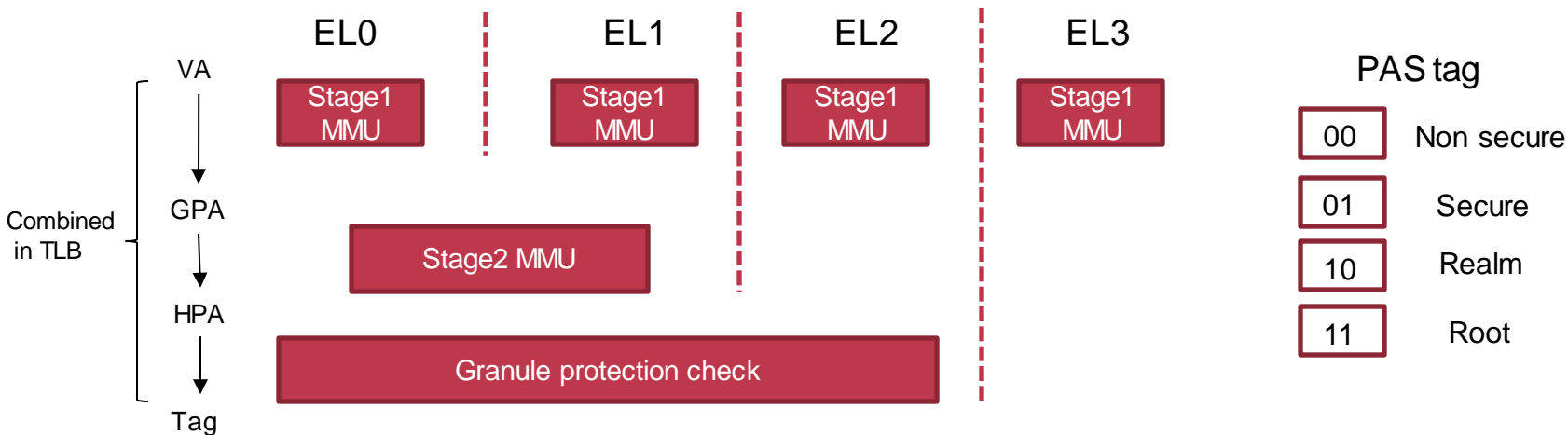
- MPE为不同PAS提供不同的密钥或Tweak
- Non-secure以外的PAS必须经过内存加密
- Monitor 可以配置MPE
- 硬件完整性保护非必须



物理内存检查模块Granule Protection Check (GPC)

- Granule protection check

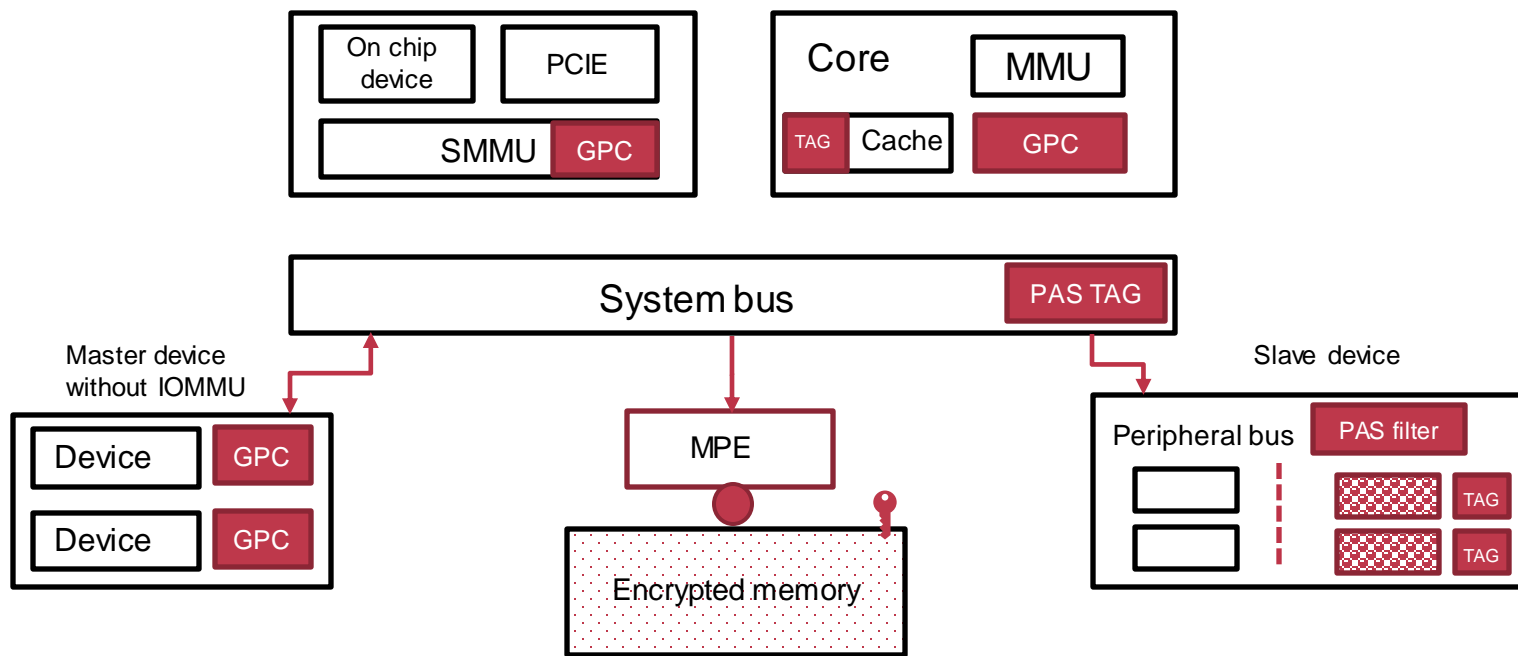
- 2-level index table for PAS tag
- PA->Tag translation



物理内存检查模块Granule Protection Check (GPC)

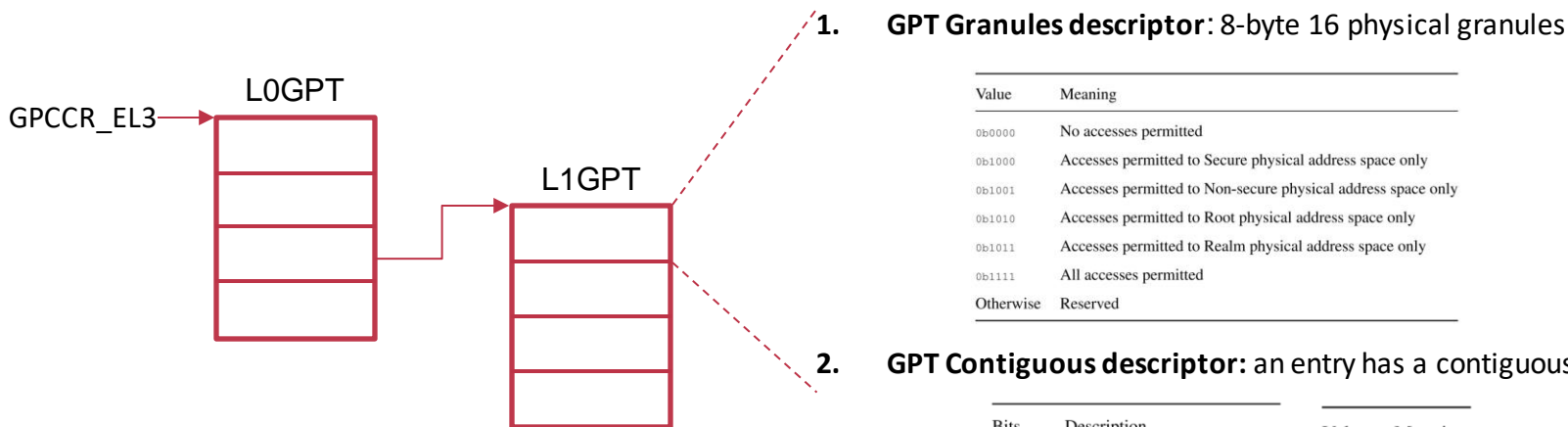
- 在MMU和I/OMMU中添加GPC模块

- PAS标签在SoC中不断进行传递



GPT的格式

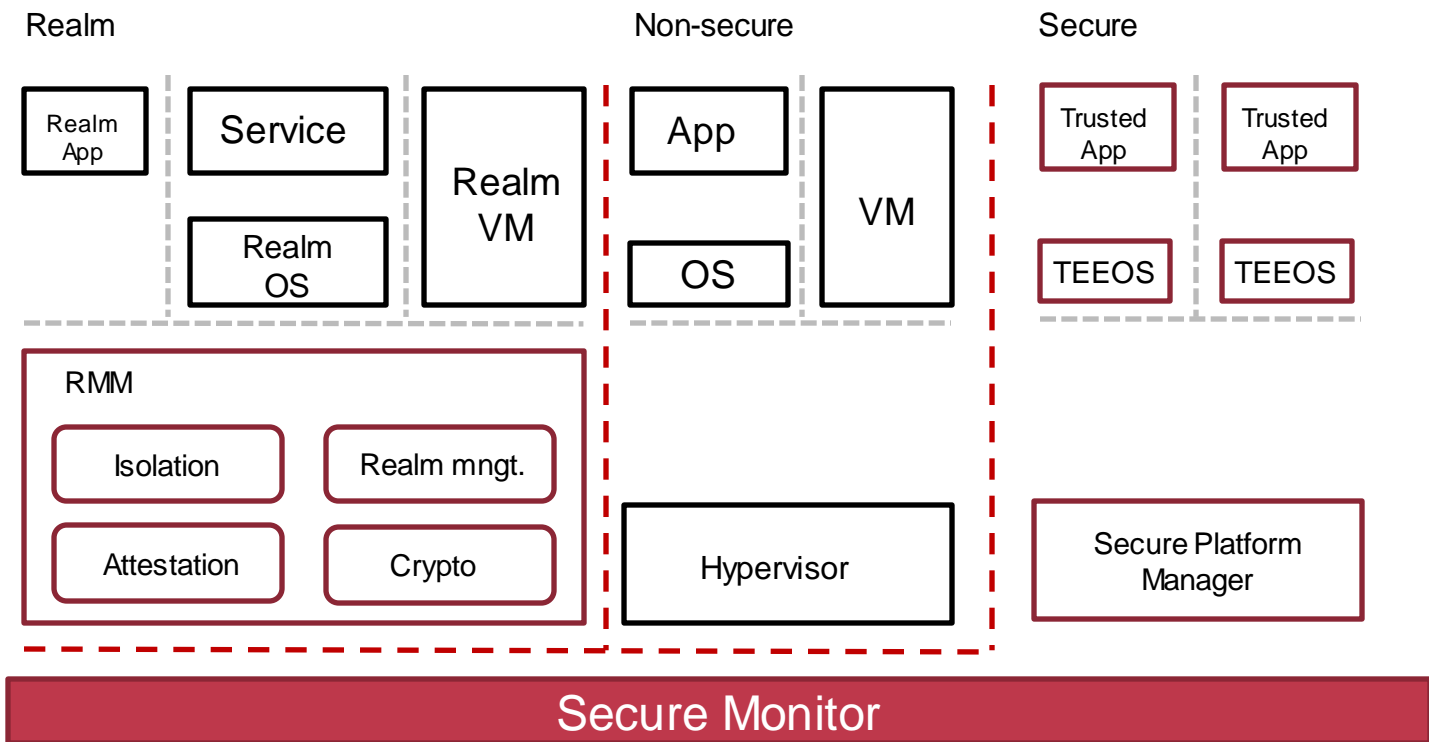
- 两层页表，存在不同的页表项格式
 - L0GPT Entry size: 1G, 16G, 64G, 512G
 - L1GPT Entry size: 4K, 16K, 64K, 2M, 32M, 512M



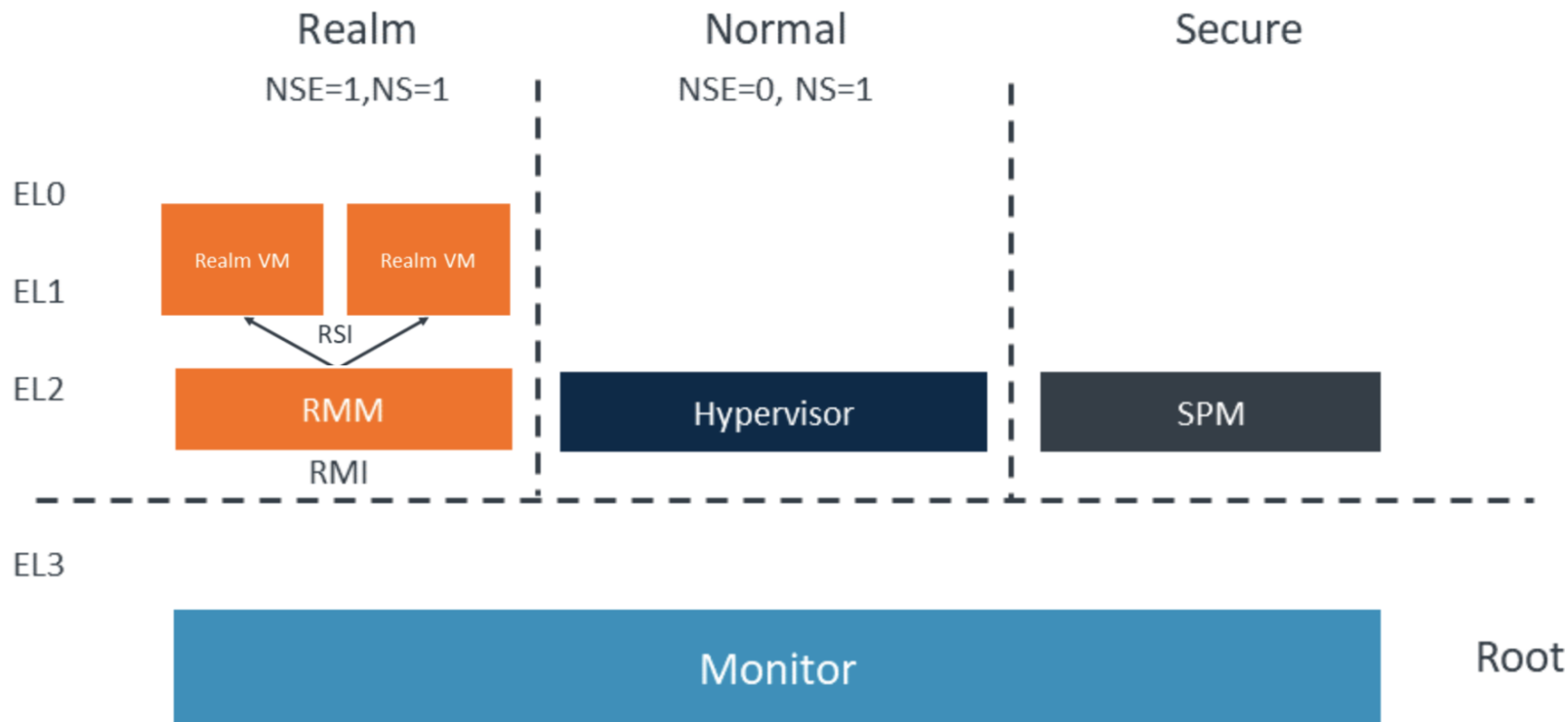
2. GPT Contiguous descriptor: an entry has a contiguous block

| Bits | Description | Value | Meaning |
|---------|--------------------------------|-------|----------|
| [63:10] | Reserved, RES0 | 0b00 | Reserved |
| [9:8] | Contig | 0b01 | 2MB |
| [7:4] | GPI | 0b10 | 32MB |
| [3:0] | 0b0001 (Contiguous descriptor) | | |

可信软件固件



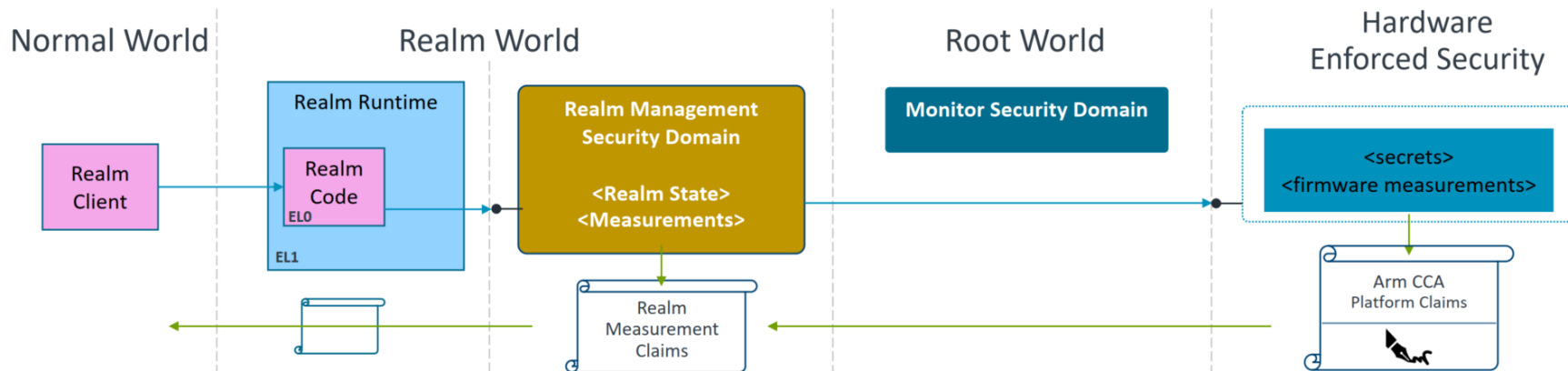
RMM接口：RMI和RSI



远程验证

• 验证报告的控制流

- Monitor获得CCA平台报告，并返回给RMM
- RMM将Realm的度量信息添加到报告中，并返回给Realm



其他机密虚拟化方案

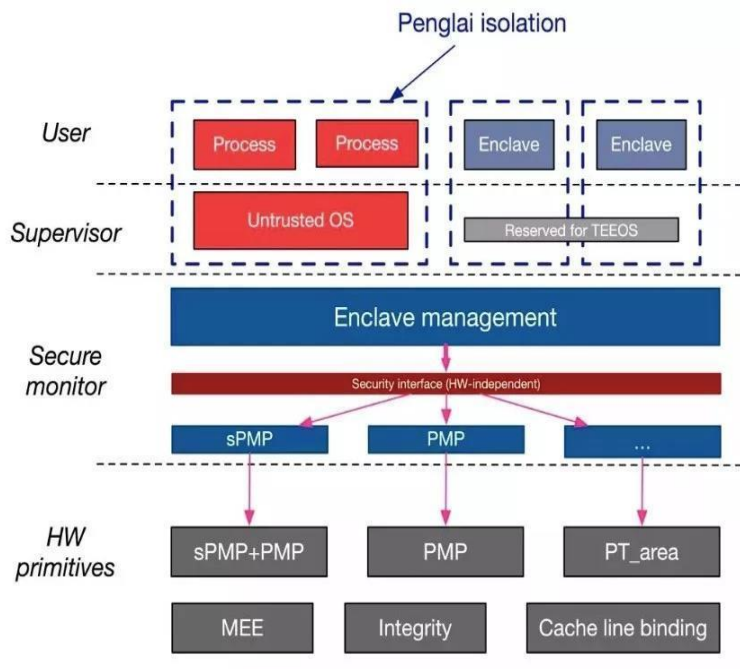
RISC-V Penglai-Enclave

- 简介

- 蓬莱可扩展TEE系统是基于RISC-V的安全执行环境。
- 扩展现有硬件原语，通过软硬件协同支持可扩展的隔离环境。

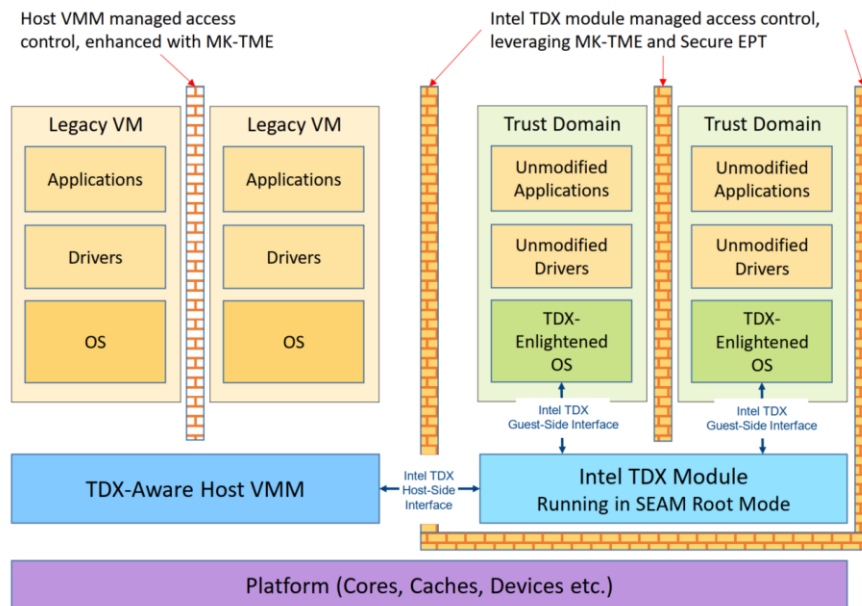
- 蓬莱架构

- 设计了一层"安全原语"接口，实现软件可信基的通用性。
- 管理逻辑可实现在通用接口上，无需关心具体的硬件隔离和保护机制



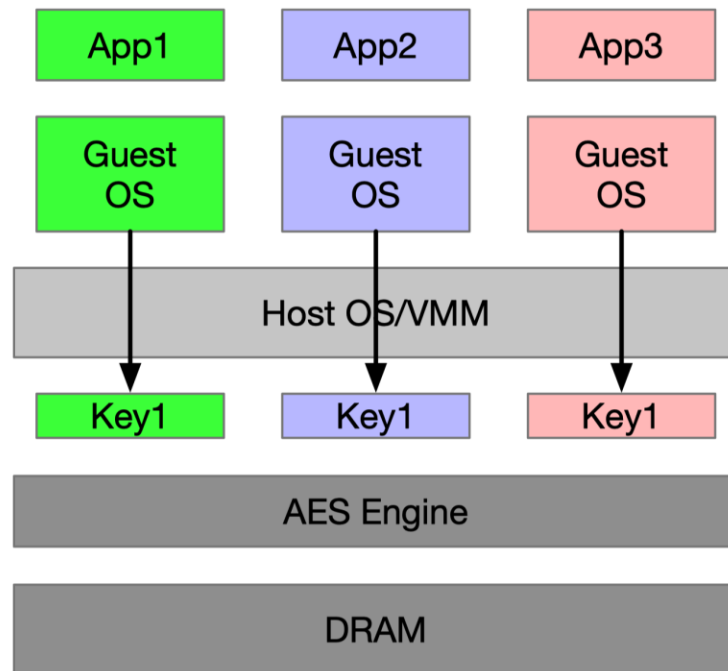
Intel Trusted Domain Extensions (TDX)

- Intel TDX is designed to isolate VMs (TD) from the hypervisor and any other non-TD software
 - Virtual Machine Extensions (VMX)
 - Multi-key, total memory-encryption (MKTME) technology



AMD Secure Encrypted Virtualization (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





容器

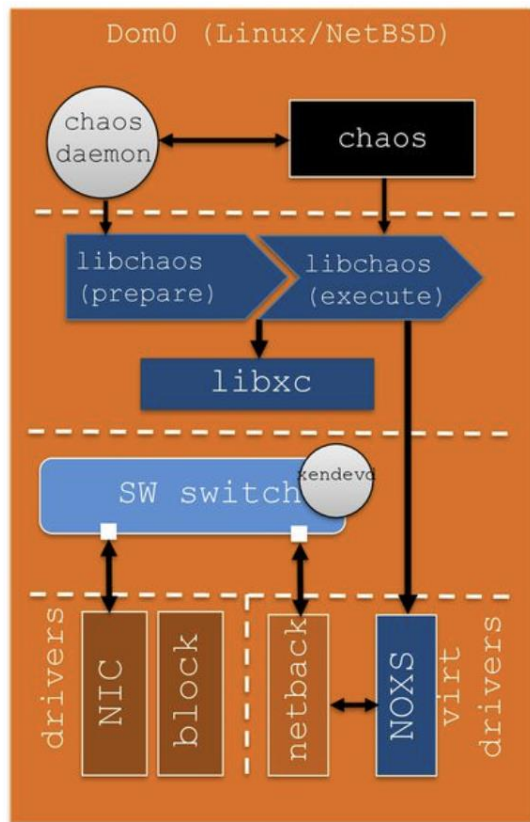
Hypervisor

- **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) (?)

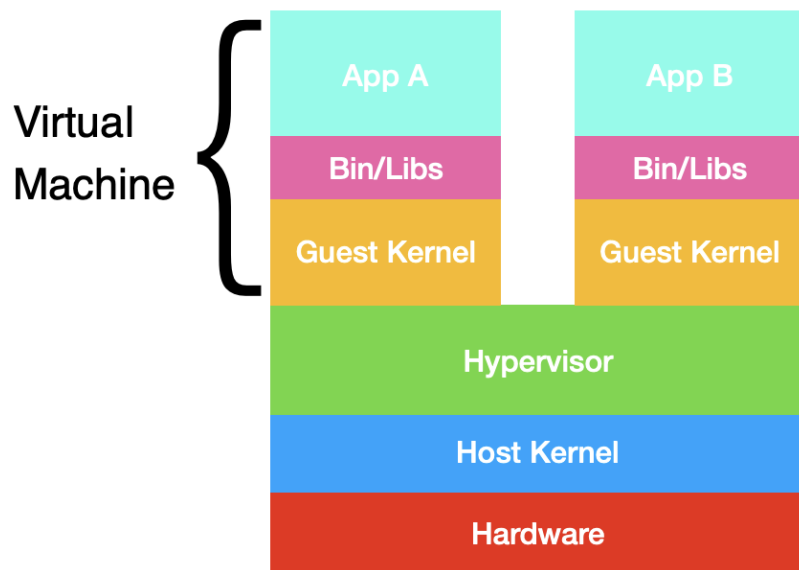


1. **Chaos** – toolstack optimized for paravirtualized guests
2. **Split functionality**
3. **Noxs** - no XenStore

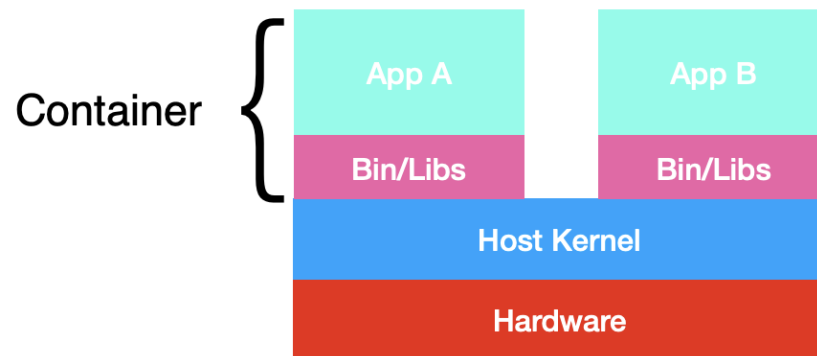
15

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 then 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

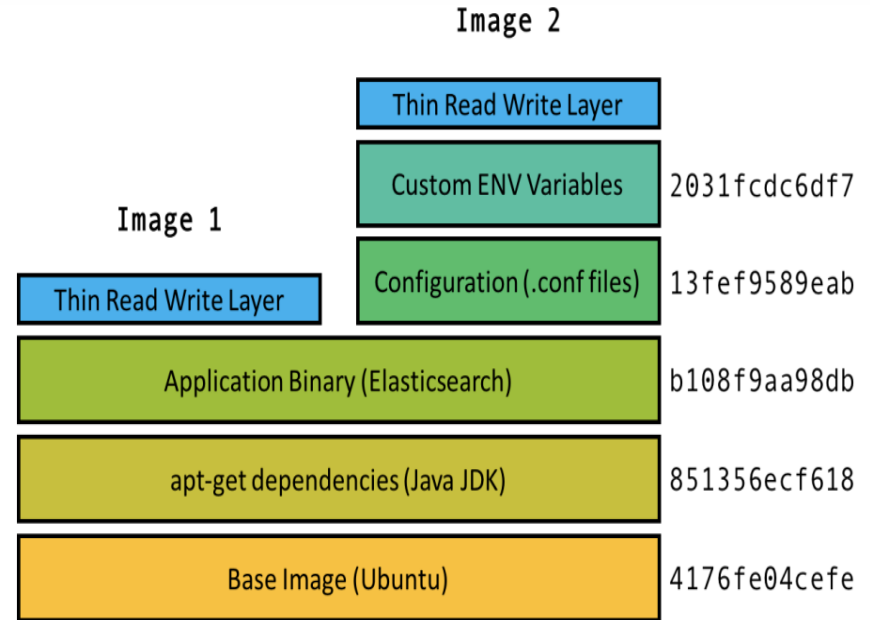
WORKDIR ~/

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
```

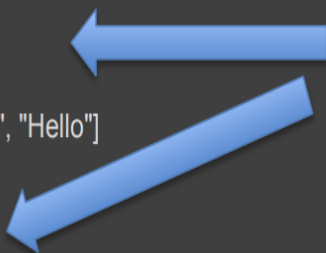
```
Step 4/4 : CMD ["world"]
```

```
---> Using cache
```

```
---> a89172ee2876
```

```
Successfully built a89172ee2876
```

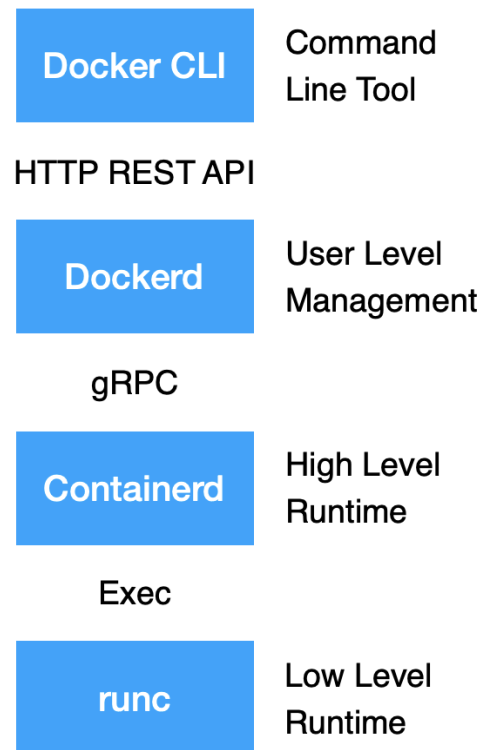
```
Successfully tagged hello_world_cmd:latest
```



Have seen this before. Use
cache

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**

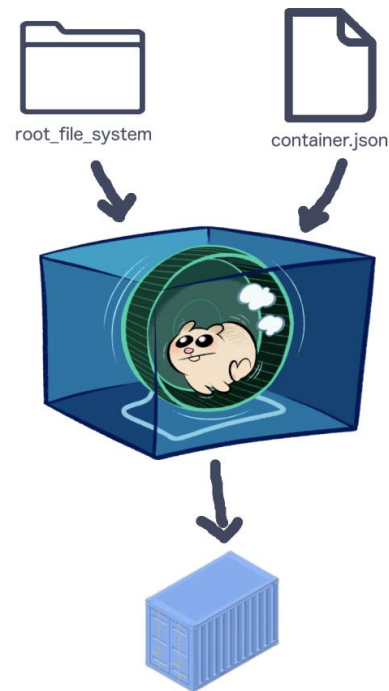


Image source: Cameron Lonsdale, runc

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 = 1000000000 $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

```
$ cgcreate -g cpu,memory:$UUID /Users/cs162/bin/ls
$ cgset -r memory.limit_in_bytes = 1000000000 $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**

App / Container

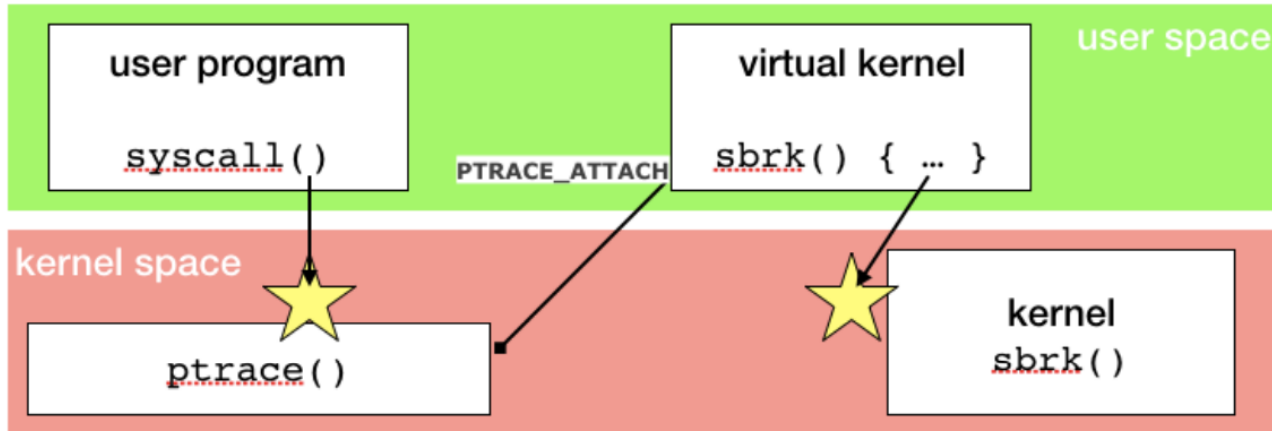
System Calls

gVisor

Limited System Calls

Host Kernel

gvisor



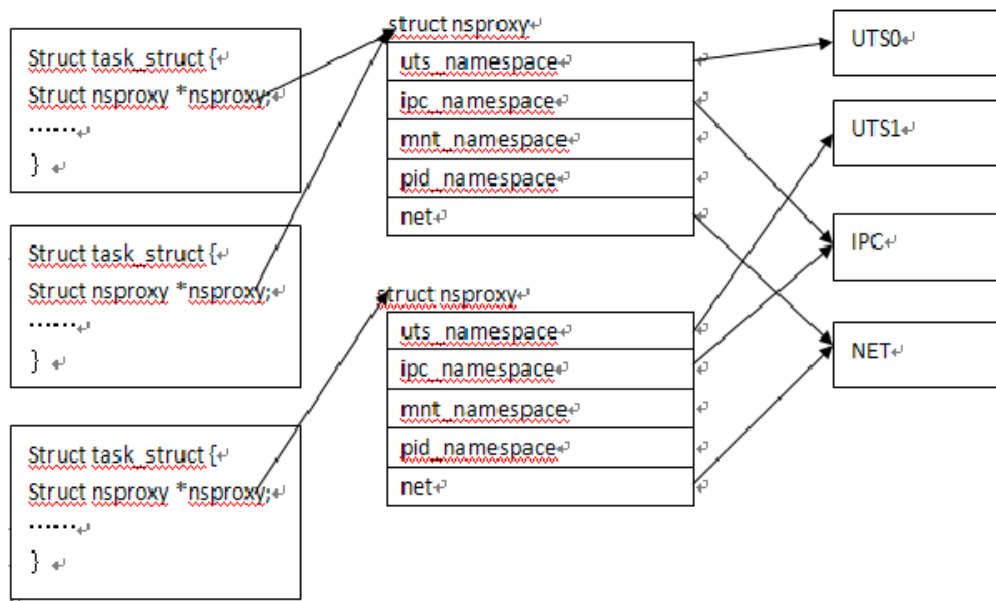
- **Intercepts syscalls through ptrace**
 - Linux syscall that allows one process to “control” another
 - gdb uses it to step through instructions, gVisor to intercept syscalls
- **Can also use kernel based virtual machines (KVM)**
 - Experimental feature that requires hardware support
 - Pushes the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

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]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]

# 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**

UTS (UNIX Time-sharing System) namespace

- UTS namespace 提供了主机名和域名的隔离，这样每个容器就拥有独立的主机名和域名了，在网络上就可以被视为一个独立的节点，在容器中对 hostname 的命名不会对宿主机造成任何影响。
 - struct utsname里的nodename和domainname两个字段。
 - 不同Namespace中可以拥有独立的主机名和域名。
- **在创建新的uts namespace时，需要设置单独的hostname，从而不影响到原进程**
 - Sethostname()

IPC (Interprocess Communication) namespace

- 容器中进程间通信采用的方法
 - 管道
 - 信号量
 - 信息队列
 - 共享内存
- 容器间通信，即为相同pid namespace中的IPC
 - 全局唯一标识符

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**

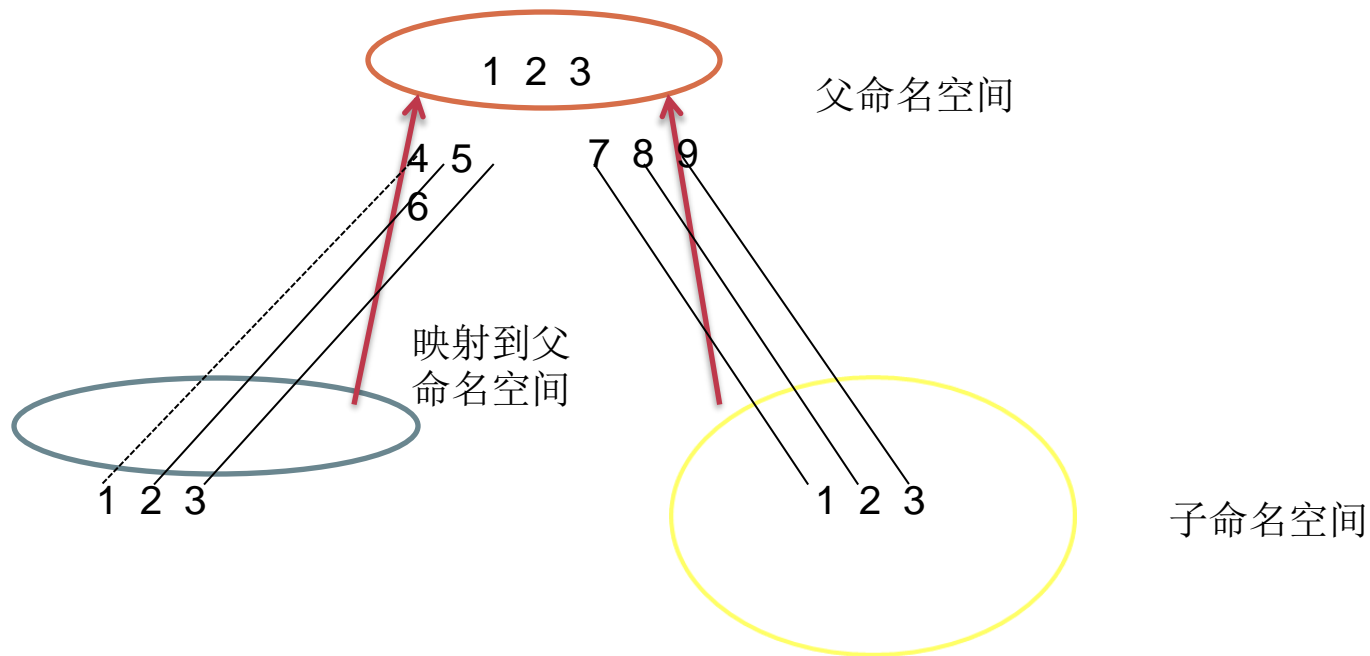
```
struct pid_namespace {  
    //...  
    unsigned int level;  
    struct pid_namespace *parent;  
    //...  
};
```

- **怎么监控Docker中运行的程序状态？**

进程的命名空间

- 每当在内核中新建一个进程时，都会在全局内创建一个唯一的描述符来描述该进程。在没有命名空间的情况下，用一个单项递增的数字作为进程号即可。为了实现进程隔离，引入层次化的命名空间。由于子命名空间中的所有进程在父命名空间中都是可见的，导致在子命名空间中新建一个进程会出现在往上的每层命名空间都有一个唯一的进程号描述。这样，一个进程在整个系统中可能有多个进程描述符，**在一般的使用时，显示的进程号往往是该进程对应到顶层命名空间的进程号。**每个容器包括一定独立的资源，隔离开来进行管理。

PID namespace



PID namespace

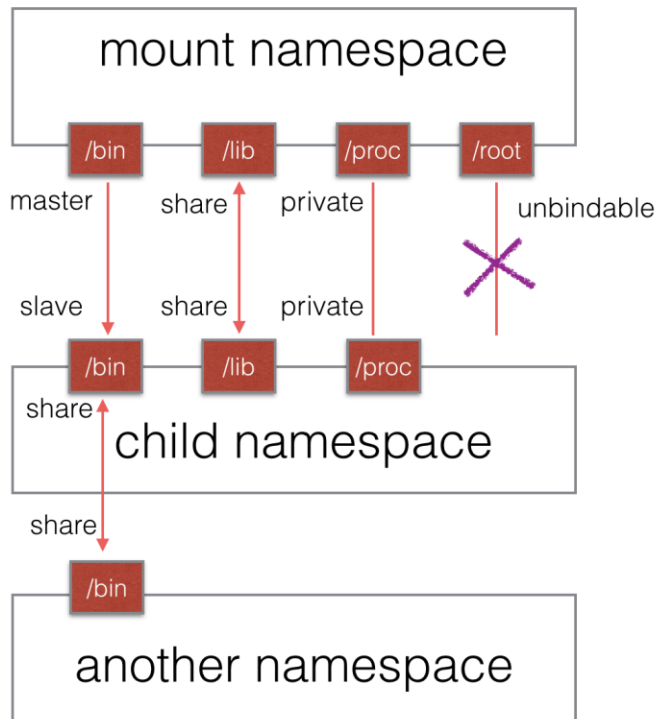
- PID namespace 中的 init 进程
 - 创建的第一个pid namespace
 - Ini进程会负责回收资源和监控、维护后续启动进程的运行状态
- 信号与init进程
 - 信号屏蔽：如果init中没有写处理某个信号的代码逻辑，那么则与init在同一个pid namespace中的进程发送的该信号会被屏蔽
 - 父节点pid namespace中进程发送的信号会被忽略吗？
 - 一旦init进程被销毁，同一pid namespace中的其他进程呢？
- Unshare()和setns()
 - 调用者进程不进入新的pid namespace，而是随后创建的子进程进入
 - 与其他namespace不同的原因？

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]/mountstats` ——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。

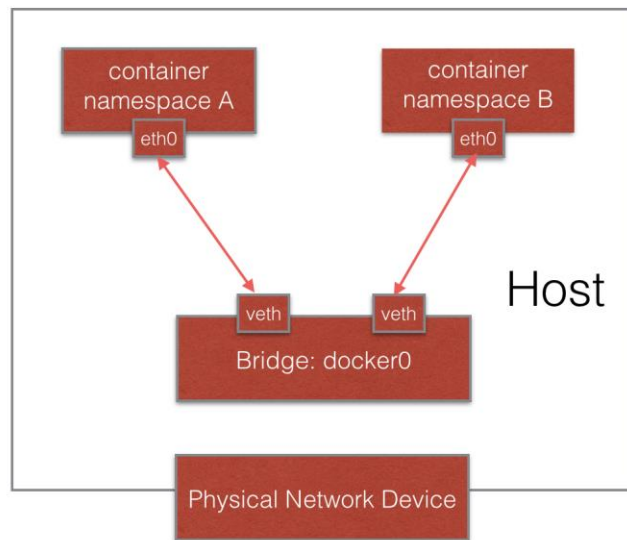
Mount namespace

- 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



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；另一端放在原先 namespace 中连接物理网络设备，在通过网桥把别的设备连接尽量或进行路由转发



User namespace

- **隔离目标：安全相关的标识符和属性**
 - 用户 ID、用户组 ID、root 目录、key（指密钥）以及特殊权限
- **Linux中用非root的用户来创建一个容器，其创建的容器进程属于拥有超级权限的用户，可行吗？**

3、cgroup机制和LXC

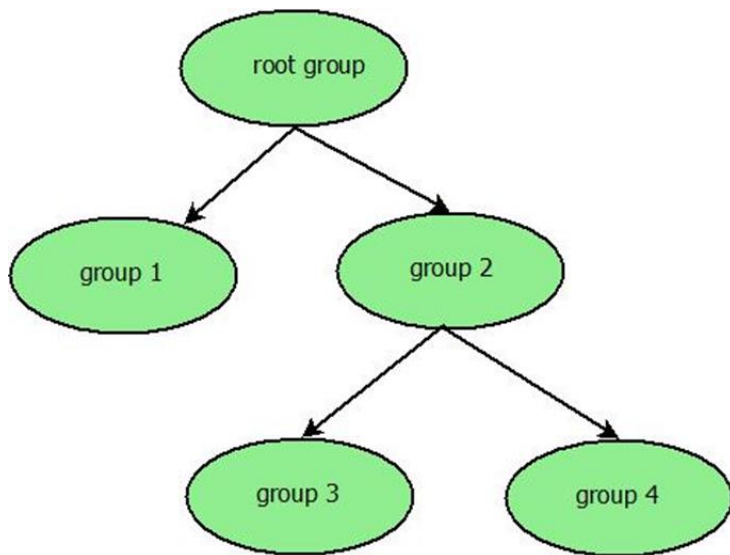
- **Cgroups是什么？**
- Cgroups是control groups的缩写，是Linux内核提供的一种可以限制、记录、隔离进程组所使用的物理资源（如：cpu,memory,IO等等）的机制。最初由google的工程师提出，后来被整合进Linux内核。Cgroups也是LXC为实现虚拟化所使用的资源管理手段，可以说没有cgroups就没有LXC。
- Cgroups以分组的形式对进程使用系统资源的行为进行管理和控制。也就是说，用户通过cgroup对所有进程进行分组，再对该分组整体进行资源的分配和控制。

3、cgroup机制和LXC

- **Cgroups可以做什么？**
- Cgroups最初的目标是为资源管理提供一个统一的框架，既整合现有的cpuset等子系统，也为未来开发新的子系统提供接口。现在的cgroups适用于多种应用场景，从单个进程的资源控制，到实现操作系统层次的虚拟化。**Cgroups提供了以下功能：**
- 1.限制进程组可以使用的资源数量。比如：memory子系统可以为进程组设定一个memory使用上限，一旦进程组使用的内存达到限额再申请内存，就会出发OOM。
- 2.进程组的优先级控制。比如：可以使用cpu子系统为某个进程组分配特定cpu share。
- 3.记录进程组使用的资源数量。比如：可以使用cpuacct子系统记录某个进程组使用的cpu时间
- 4.进程组隔离。比如：使用ns子系统可以使不同的进程组使用不同的namespace，以达到隔离的目的，不同的进程组有各自的进程、网络、文件系统挂载空间。
- 5.进程组控制。比如：使用freezer子系统可以将进程组挂起和恢复。

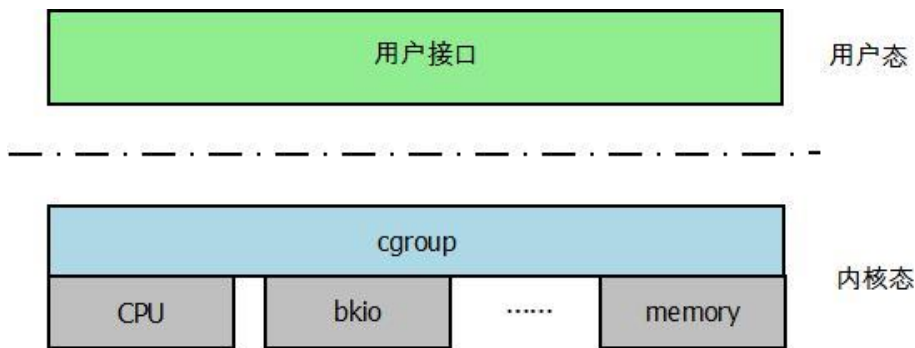
3、cgroup机制和LXC

- **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。



3、cgroup机制和LXC

- cgroup是一种对进程资源管理和控制的统一框架，它提供的是一种机制，而具体的策略是通过子系统来完成的。每个子系统都代表一种类型的资源，如CPU子系统，memory子系统。



- 由图可以看出，cgroup在用户态提供统一的用户接口，而每个子系统对资源的控制功能则通过其钩子函数实现。这样使得cgroup在上层是一个统一的框架，而下层则可以实现多种资源的控制。
- cgroup在Linux内核中是以文件系统的形式存在的，不过cgroup对应的这种文件系统与proc文件系统类似，都是只存在于内存中的“虚拟”文件系统。

3、cgroup机制和LXC

- **Cgroups子系统介绍**

- **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。
- **ns** -- 名称空间子系统。

3、cgroup机制和LXC

- **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

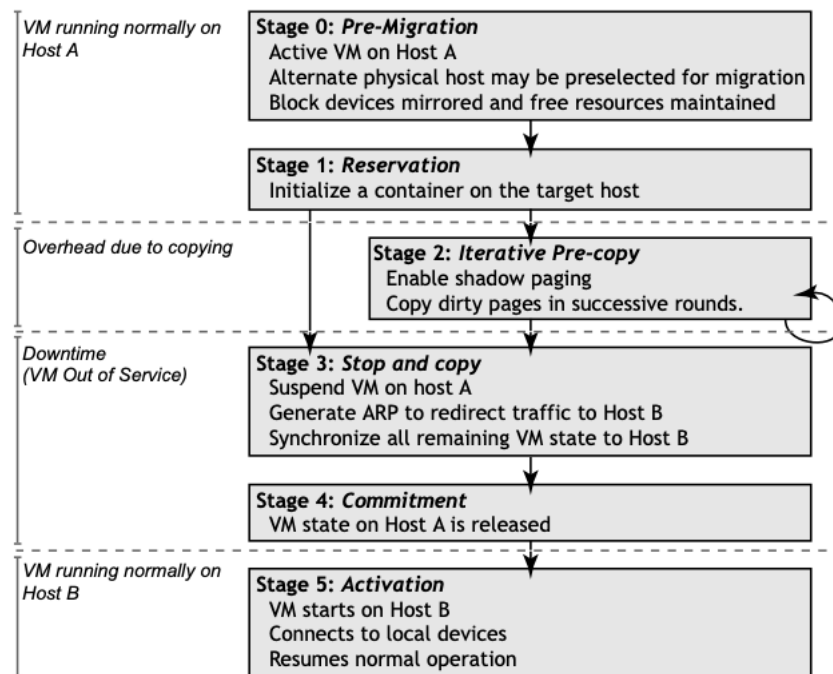


Figure 1: Migration timeline

Live Migration of Virtual Machines [2005, NSDI]