中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

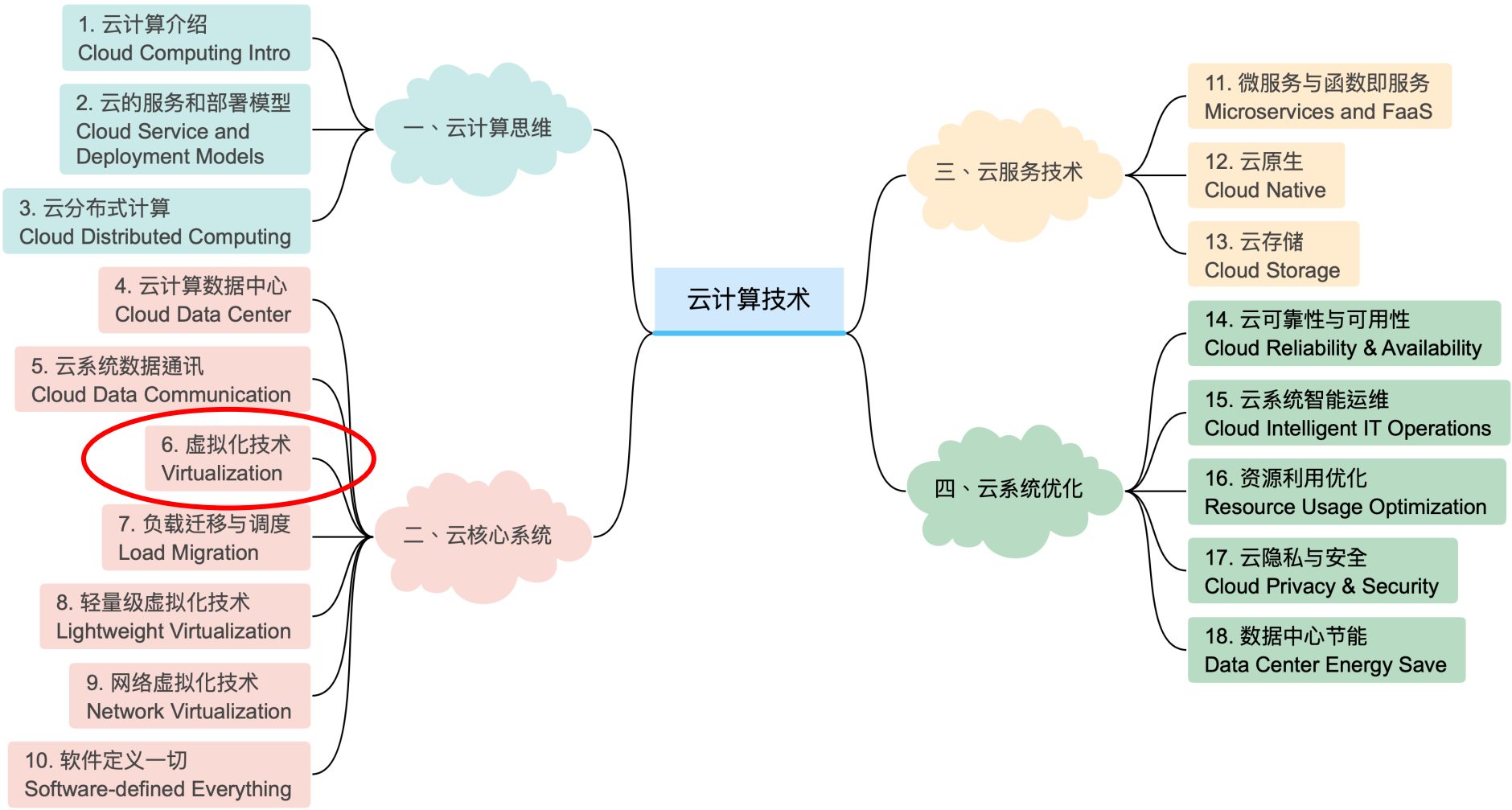
Lecture 06：虚拟化技术

SSE316：云计算技术
Cloud Computing Technologies

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn


云计算技术
一、云计算思维
1. 云计算介绍
Cloud Computing Intro
2. 云的服务和部署模型
Cloud Service and
Deployment Models
3. 云分布式计算
Cloud Distributed Computing
二、云核心系统
4. 云计算数据中心
Cloud Data Center
5. 云系统数据通讯
Cloud Data Communication
6. 虚拟化技术
Virtualization
7. 负载迁移与调度
Load Migration
8. 轻量级虚拟化技术
Lightweight Virtualization
9. 网络虚拟化技术
Network Virtualization
10. 软件定义一切
Software-defined Everything
三、云服务技术
11. 微服务与函数即服务
Microservices and FaaS
12. 云原生
Cloud Native
13. 云存储
Cloud Storage
四、云系统优化
14. 云可靠性与可用性
Cloud Reliability & Availability
15. 云系统智能运维
Cloud Intelligent IT Operations
16. 资源利用优化
Resource Usage Optimization
17. 云隐私与安全
Cloud Privacy & Security
18. 数据中心节能
Data Center Energy Save

Today’s topics

- 为什么需要虚拟化技术?
- 如何实现虚拟化
  - 操作系统中的接口层次
  - 可虚拟化原则
- 虚拟化技术

为什么需要虚拟化技术?

Why Virtualization?

一个典型的企业 IT 困境

小明的公司要上线三个系统：

❖官网（Web服务器）—— 需要 Linux + Apache

❖内部 OA（办公系统）—— 需要 Windows Server + .NET

❖数据分析平台 —— 需要 Linux + Python + GPU

传统方案：买三台物理服务器，每台 ¥50,000+

问题来了：

➢ 官网大部分时间 CPU 利用率只有 5%

➢ OA 系统只有工作日白天有人用

➢ 数据分析只在月底跑一次

花了 ¥150,000 买的服务器，90% 的时间都在 "睡觉"

别墅 vs 公寓楼：虚拟化的核心直觉

传统：每人一栋别墅

10个房间只用了1-2个
每栋都要独立维护水电煤
占地面积大，成本高

= 一台物理机跑一个应用
CPU利用率 10-15%

虚拟化：住进公寓楼

多户共享一栋楼
每户有独立隔间（隔离性）
物业统一管理（Hypervisor）

= 一台物理机跑多个VM
CPU利用率 60-80%

虚拟化的本质：抽象 + 隔离 + 共享

回顾：IaaS 的核心支撑技术是什么？

SaaS	软件即服务	Gmail, Office 365, 钉钉
PaaS	平台即服务	App Engine, 阿里云函数计算
IaaS	基础设施即服务	AWS EC2, 阿里云 ECS, Azure VM

答案：虚拟化技术（Virtualization）

什么是虚拟化？(Virtualization)

虚拟化（Virtualization）

通过软件在物理硬件之上创建一个抽象层，将一台物理计算机模拟为多台逻辑计算机。每台虚拟机拥有独立的操作系统、应用和资源，互不干扰。

更简洁地说：

虚拟化 = 用软件 "假装" 硬件

让一台电脑变成 "多台电脑"，每台都以为自己独占整台机器

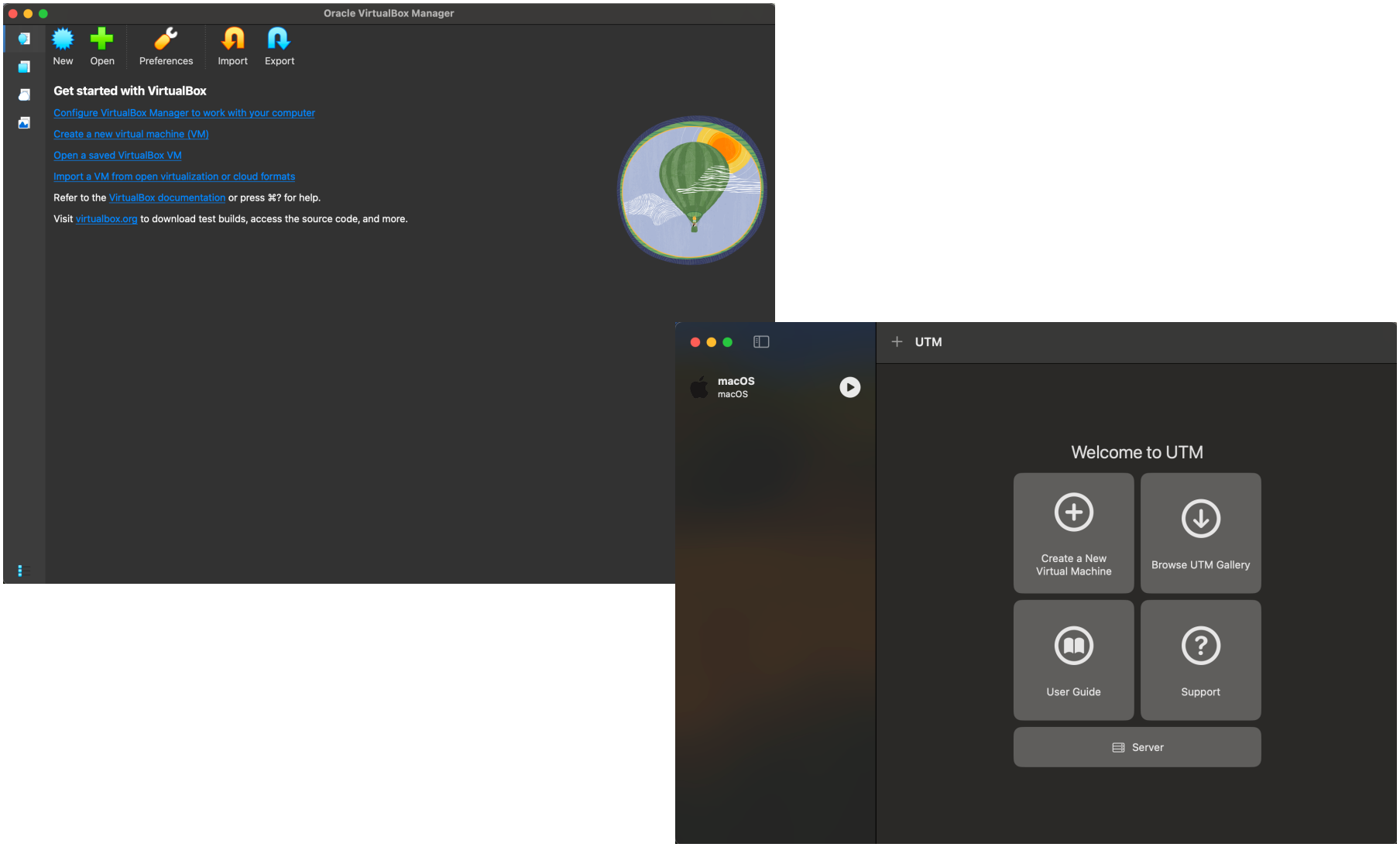
从程序的角度来看，虚拟机和真实的物理主机没有区别！

类比

就像一个演员同时分饰多个角色

或者一套房子被隔成多个独立的房间出租

什么是虚拟化？(Virtualization)



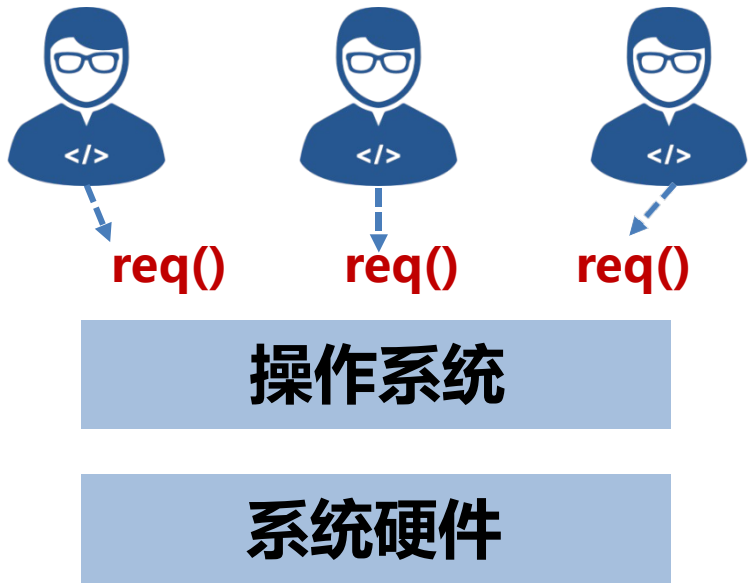
分时系统

- 在同一个硬件平台上**切换不同任务**，服务不同用户
- 用户需求增加则**创建更多进程**

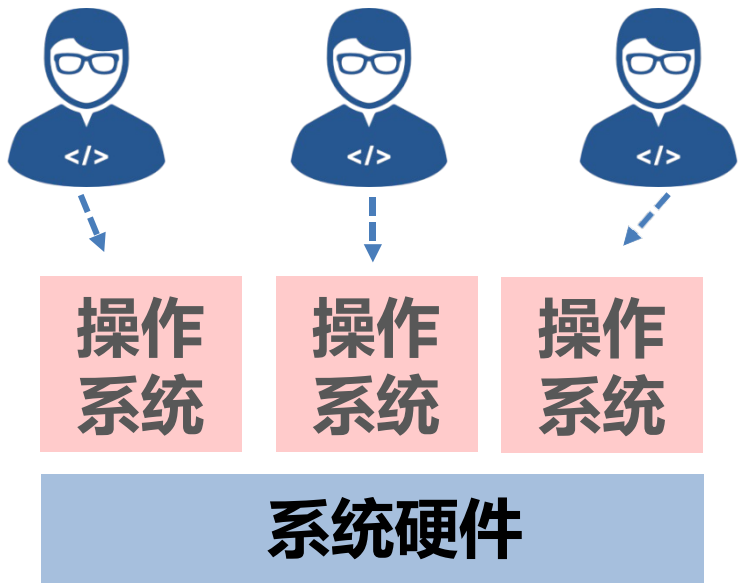
- 缺点
  - 任务、资源调度复杂
  - 硬件资源不能充分利用
  - 隔离性和安全性不足
  - 规模和性能受硬件平台限制

虚拟化技术

从并发任务处理到并发执行环境



一台计算机可以为**多个用户**提供**并发任务处理**



一台计算机为**多个用户**提供**独立的操作环境**

虚拟化的三大核心价值

抽象 Abstraction

屏蔽底层硬件差异
应用看到的是"标准化"的虚拟硬件

隔离 Isolation

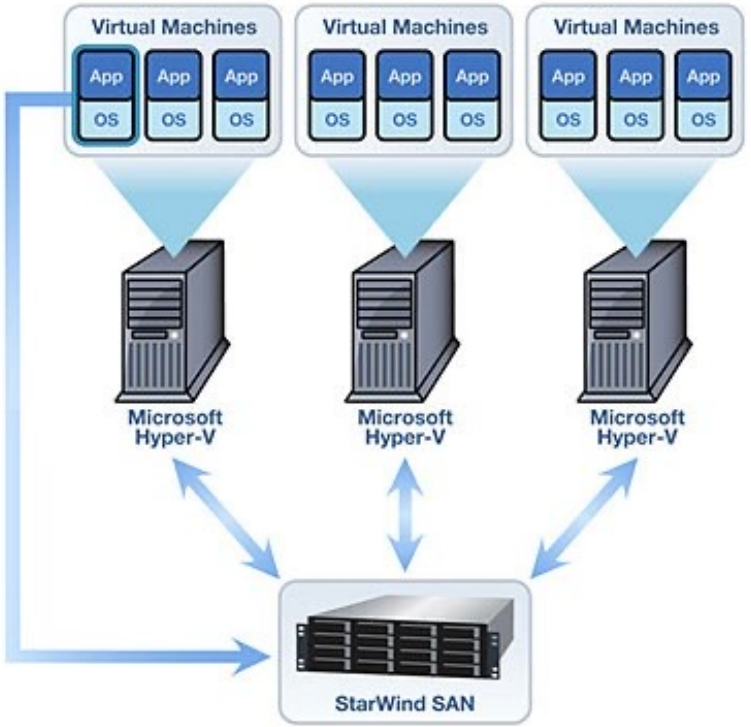
多个VM互不影响
一个崩溃不会拖垮其他

共享 Sharing

多个VM共享物理资源
CPU利用率从15%→70%+

虚拟化的优势1：整合服务器

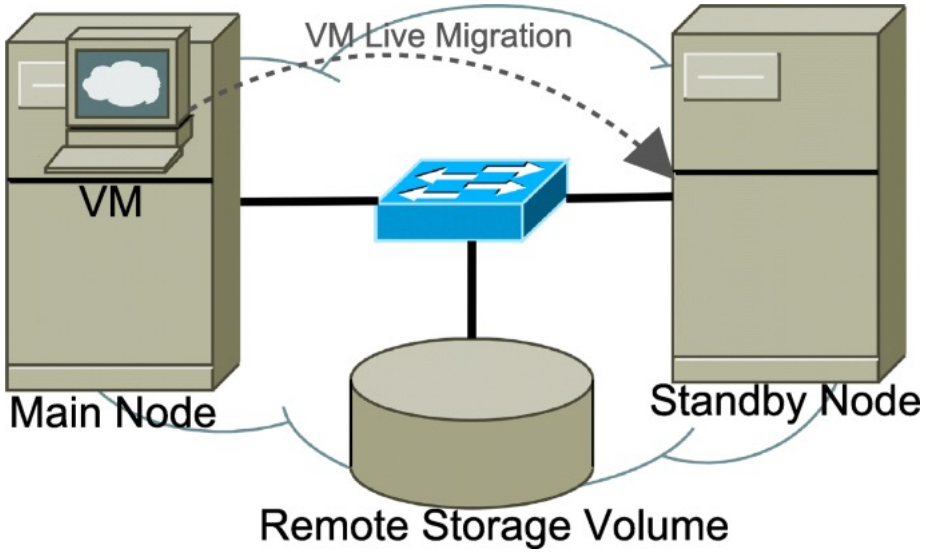
- 传统数据中心服务器资源利用率低
  - CPU 平均利用率仅为**约 20%**
- 提高物理机资源利用率
  - 一个物理机上整合多个虚拟机
- 降低云提供商成本
  - 基于用户**错峰使用**特性，"超售"系统资源



虚拟化的优势2：简化服务器管理

- 通过软件接口管理虚拟机
  - 创建、复制、备份、销毁

- 虚拟机热迁移
  - 无须停机或重启
  - 便于虚拟机升级和维护
  - 实现全局**负载均衡**



虚拟化的优势3：方便程序开发

- 调试操作系统
  - 单步调试操作系统
  - 查看当前虚拟硬件的状态
  - 随时修改虚拟硬件的状态

- 测试应用程序的兼容性
  - 可以在一台物理机上同时运行不同的操作系统
  - 测试应用程序在不同操作系统上的兼容性

虚拟化的起源 — 程序员"偷"大型机时间

1960年代，IBM 大型机价值数百万美元，是当时最昂贵的设备。

问题：一台机器一次只能跑一个程序，程序员们要"排队"等机时。
一个程序跑完可能要几小时，其他人只能干等。

于是 IBM 的工程师们想了个绝妙的办法：

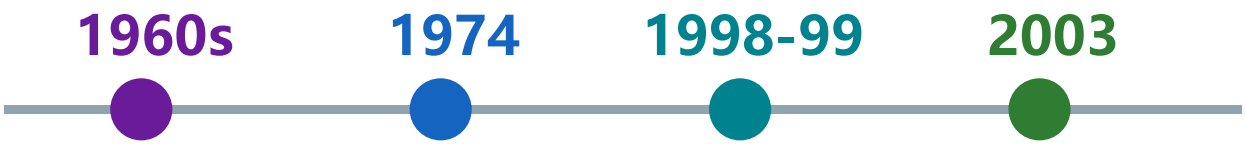
让大型机 "假装" 自己是好几台机器！

1967年，IBM 推出 CP/CMS 系统 — 世界上第一个虚拟机系统。
每个程序员都觉得自己独占了一台大型机，实际上是多人共享一台。

有趣的是，程序员们开始"偷"机时 — 在别人不用的时候多占一些资源

这反而提高了资源利用率！ IBM 发现这个"bug"其实是个 feature

虚拟化技术发展时间线



1960s

IBM CP/CMS

第一个虚拟机系统
大型机分时共享

1974

Popek-Goldberg

提出虚拟化
形式化条件

1998-99

VMware 成立

x86平台首次
实现虚拟化

2003

Xen 发布

开源半虚拟化
剑桥大学研究
项目

2005-06

VT-x/AMD-V

CPU硬件辅助
虚拟化支持

2007

KVM 进入 Linux

内核级虚拟化
成为主流方案

2010s+

云计算时代

AWS/Azure/
阿里云
虚拟化是基础

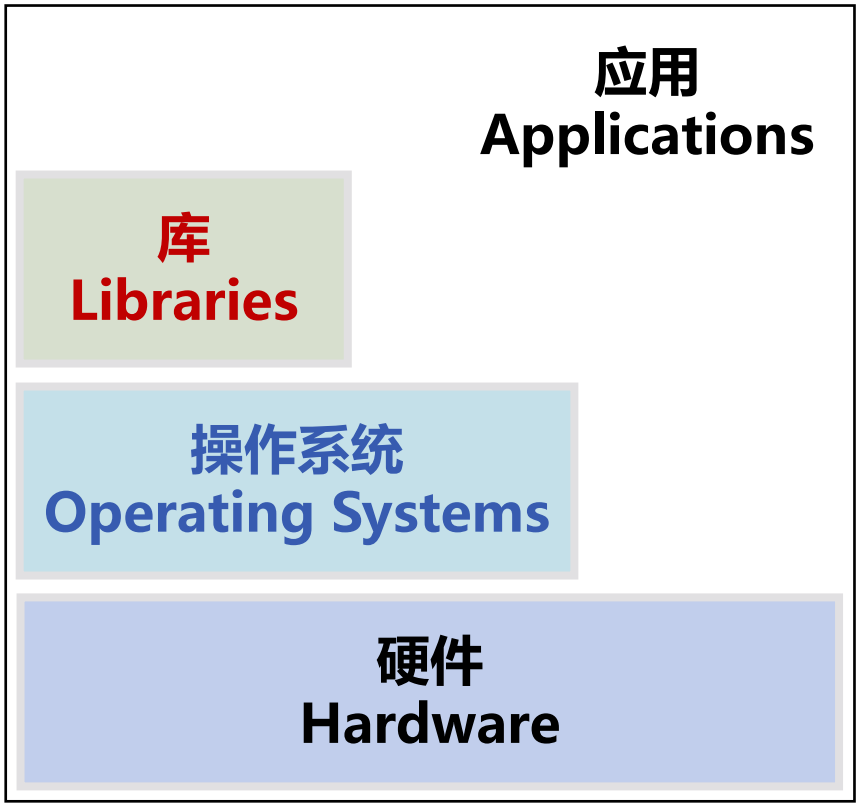
从大型机到云端，虚拟化走过了半个多世纪

操作系统中的接口层次

OS Interface Hierarchy

操作系统中的接口层次 – 提供抽象

- 定义了如何在计算机系统上高效开发和运行软件
  - 硬件和软件
  - 操作系统和应用程序
  - 库/服务和程序员



通过一个“代理”实现软件和硬件之间的交互和通信

接口层次：三个具体的例子

口同一个任务"打印 Hello World"，经过三层接口最终到达硬件

最上层

API

Application Programming Interface

库 / 服务 ↔ 程序员

例子：C 语言的 printf

```
printf("Hello\n");
```

你调用 printf()，只需要知道函数名和参数

不需要知道：

- 字符怎么编码
- 怎么写到显示器
- 具体是哪个操作系统

现实类比：

微信支付 API、高德地图 API、OpenAI API —— 你只管调用

🔷 **容器在这一层虚拟化**

中间层

ABI

Application Binary Interface

操作系统 ↔ 应用程序

例子：printf 内部调用 write 系统调用

```
mov eax, 4     ; 4 = sys_write
mov ebx, 1     ; stdout
int 0x80       ; 触发系统调用
```

编译器把 printf 翻译成这样的汇编

ABI 规定了：

- 系统调用号放在哪个寄存器
- 用哪条指令触发（int 0x80）
- 二进制格式 (ELF / PE)

现实类比：

Windows 的 .exe 在 Linux 上跑不了 —— 因为 ABI 不同

🔷 **Wine 在这一层"翻译"**

最底层

ISA

Instruction Set Architecture

硬件 ↔ 软件

例子：CPU 执行的机器指令

```
B8 04 00 00 00   ; mov eax, 4
BB 01 00 00 00   ; mov ebx, 1
CD 80            ; int 0x80
```

CPU 直接执行的二进制机器码

ISA 定义了：

- 有哪些指令 (mov/add/int)
- 有哪些寄存器 (eax/ebx)
- 每条指令的二进制编码

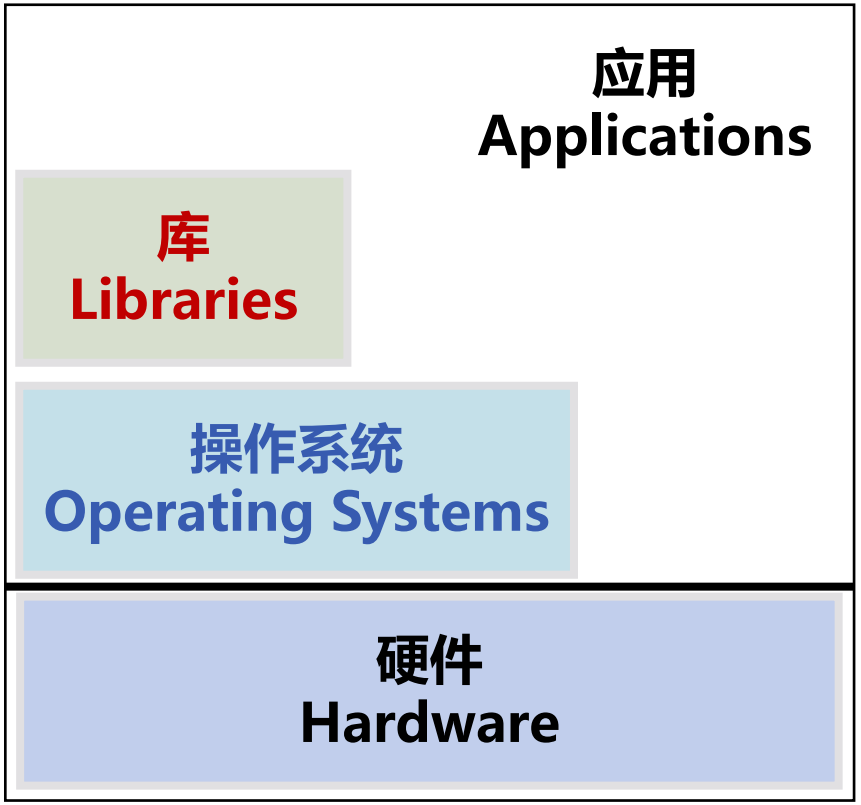
现实类比：

x86 / ARM / RISC-V 指令集互不兼容

🔷 **QEMU 在这一层模拟**

操作系统中的接口层次

- ISA (Instruction Set Architecture) 指令集架构层
  - **硬件和软件之间的接口，CPU 通过 ISA 告诉软件"我能执行哪些指令"，软件不需要知道电路怎么实现**
  - 定义了处理器支持的
    - 数据类型
    - 寄存器
    - 指令集
    - 地址模式等
  - 不同架构 CPU 的指令集不同
    - x86
    - ARM
    - RISC-V
    - MIPS

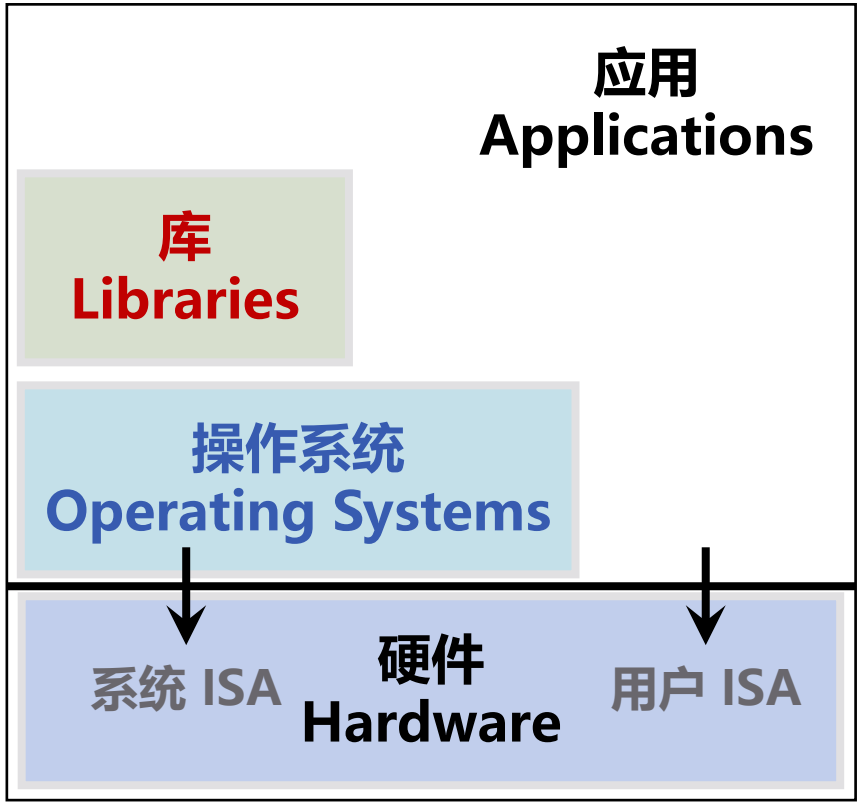


ISA 通俗理解：CPU 的 "语言"

- **类比：ISA 就像人类的语言**
  - **x86 指令集 → 好比 "普通话"**
    - 大部分笔记本电脑、台式机、服务器都 "说" 这种语言
    - Intel, AMD 的 CPU 都属于 x86 家族
  - **ARM 指令集 → 好比 "英语"**
    - 你的手机、平板、Apple M 系列芯片都 "说" 这种语言
    - 功耗低，适合移动设备
  - **RISC-V 指令集 → 好比 "法语"**
    - 开源的新兴指令集，谁都可以免费使用和修改
- **关键问题：为什么你不能把手机 App 直接拷到电脑上运行?**
  - 因为手机 App 是用 ARM "语言" 写的，电脑 CPU 只懂 x86 "语言"！
  - 这就像给只会说普通话的人一本英文书，看不懂！
- **虚拟化的一个作用就是 “翻译” —让一种 CPU 能跑另一种 CPU 的程序**
  - *例如 QEMU 可以在 x86 电脑上模拟运行 ARM 程序*

操作系统中的接口层次

- 用户 ISA（处理器大部分指令集）
  - **决定如何编写程序**
  - **用户态和内核态程序均可用**
  - 算数运算、逻辑运算、数据传输
  - mov x0, sp
  - add x0, x0, #1
- 系统 ISA（处理器小部分指令集）
  - **决定如何编写操作系统**
  - **仅内核态程序可用**
  - 中断处理、内存管理、设备控制
  - msr vbar_el1, x0

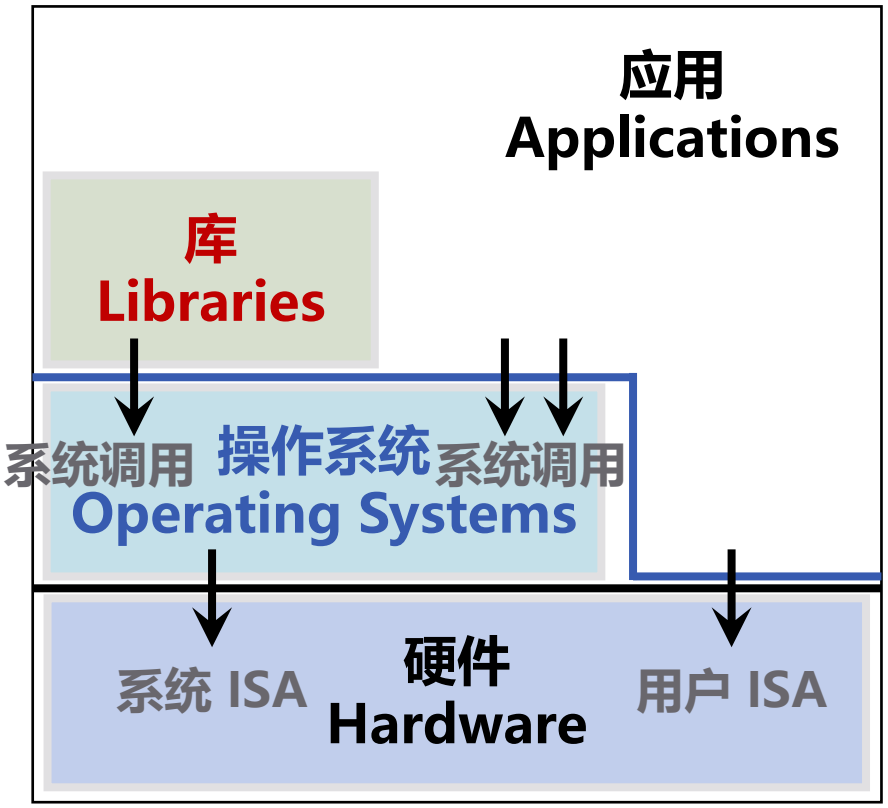


用户 ISA vs 系统 ISA：餐厅的比喻

- **想象一家餐厅：**
  - **用户 ISA = 顾客可以做的事：看菜单、点菜、吃饭、买单**
    - CPU 中：加减乘除、比较大小、读写普通内存
    - 示例：mov x0, sp（把数据从一个地方搬到另一个地方）
    - 任何程序都可以用这些指令，很安全
  - **系统 ISA = 员工才能做的事：进出厨房、操作燃气灶、管理食材库存**
    - CPU 中：管理内存页表、控制中断、操作硬件设备
    - 示例：msr vbar_el1, x0（设置异常向量表，控制 CPU 行为）
    - 只有操作系统内核（Ring 0）才能执行，普通程序用了会报错！
- **为什么这样设计？**
  - 如果让顾客随便进厨房 → 有人可能偷食材、关掉别人的灶
  - 如果让普通程序执行系统ISA → 可能修改内存映射、搞崩整个系统
- **这也是虚拟化的核心挑战：Guest OS 需要执行系统 ISA，但它不在 Ring 0！**

操作系统中的接口层次

- ABI (Application Binary Interface) 应用二进制接口层
  - **操作系统和应用程序之间的接口，规定了编译后的程序如何与 OS 交互（系统调用方式、二进制格式等）**
  - 包含用户 ISA 和系统调用
  - 定义了
    - 数据类型的大小和对齐
    - 函数调用的约定
    - 系统库和对象文件的格式
    - …



ABI 通俗理解：程序的 "护照"

- **类比：ABI 就像一个国家的标准插座**
  - 中国用 220V / A型插座
  - 美国用 120V / B型插座
  - 同个电器（程序），去不同国家（操作系统）需要转换插头（重新编译）

- **生活中的 ABI 不兼容：**
  - **同样是 x86 CPU，为什么 Windows 的 .exe 在 Linux 上跑不了?**
  - 因为 ABI 不同!
    - Windows ABI：系统调用用 int 0x2e 或 syscall，文件格式是 PE (.exe)
    - Linux ABI：系统调用用 int 0x80 或 syscall，文件格式是 ELF
    - 参数传递方式、函数调用约定也不一样

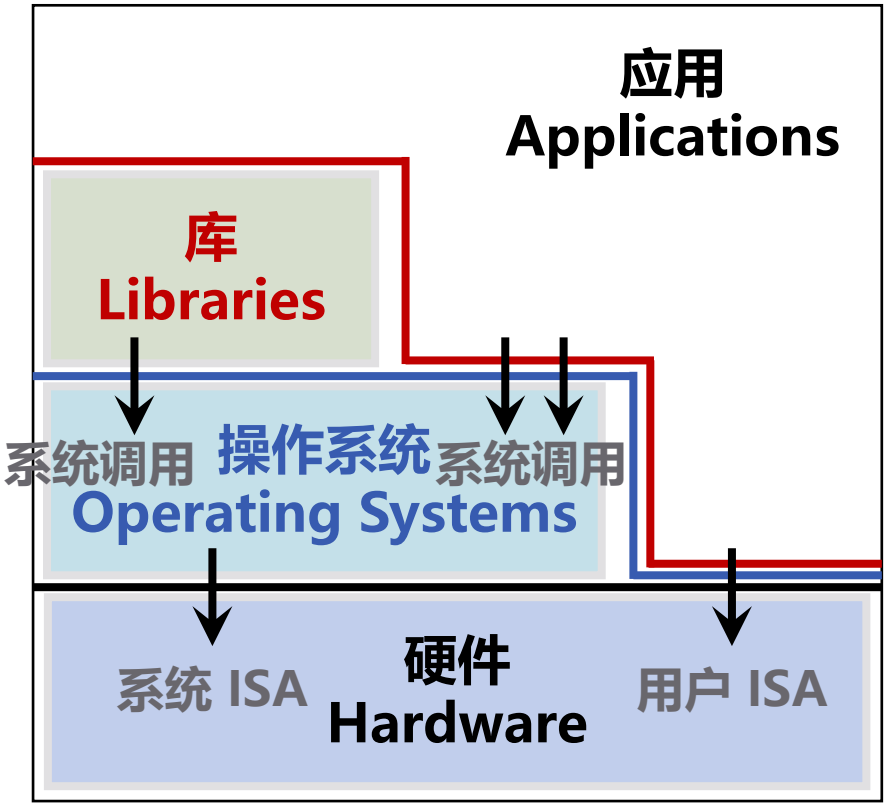
- **Wine 是怎么让 Linux 跑 Windows 程序的?**
  - 它在 ABI 层做 "翻译"：把 Windows 系统调用转成 Linux 系统调用
  - 这就是一种 ABI 层的虚拟化!

- *同理，Android模拟器也是在ABI层做翻译，让ARM的App跑在x86上*

操作系统中的接口层次

□API (Application Programming Interface) 层

- 用户态库或服务提供给程序员的接口，无需关心函数实现细节
- 用于让不同的软件系统、应用程序或组件之间进行通信和交互
- 包含库的接口和用户 ISA
- 便于程序员快速开发软件



API 通俗理解：不用会做菜也能吃到美食

❑类比：API 就像餐厅的菜单

- 你只需要说 "来一份番茄炒蛋" → 厨师帮你做好端上来
- 你不需要知道：用了几个鸡蛋、放了多少盐、炒了几分钟

❑编程中的 API 也一样：

- `printf("Hello World")` → 在屏幕上输出文字
- 你不需要知道：字符编码、显存地址、像素点怎么亮的
- `requests.get("https://api.weather.com")` → 获取天气数据
- 你不需要知道：TCP 三次握手、DNS 解析、HTTP 报文格式

❑你每天都在用的 API：

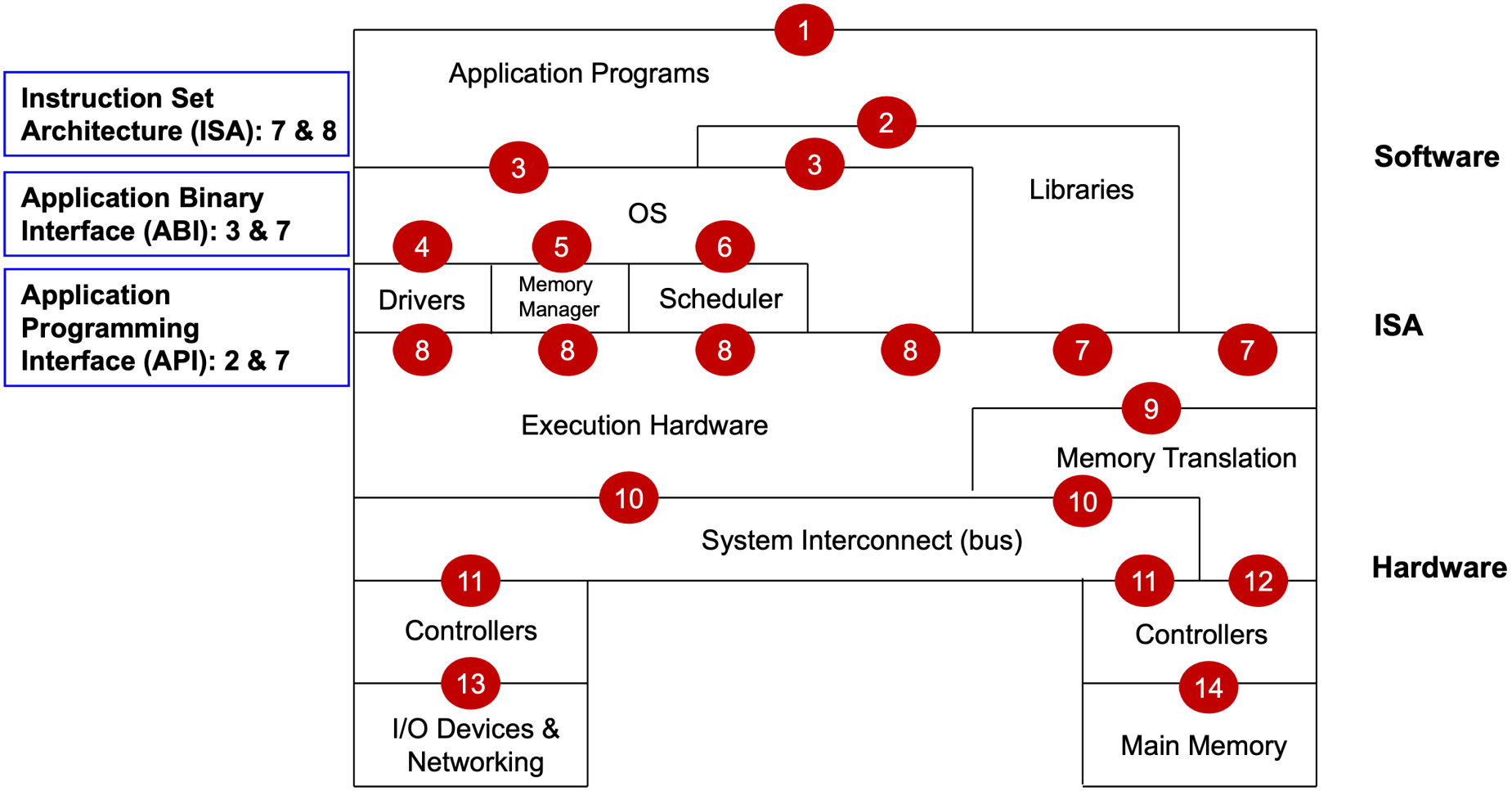
- 微信小程序调用微信支付 API → 不用自己实现支付系统
- App 调用高德地图 API → 不用自己写地图引擎
- OpenAI API → 调用 GPT 模型，不用自己训练

❑与虚拟化的关系：容器 (Docker) 就是在 API 层做虚拟化

- 共享同一个 OS 内核，只隔离文件系统、网络等 API 层资源

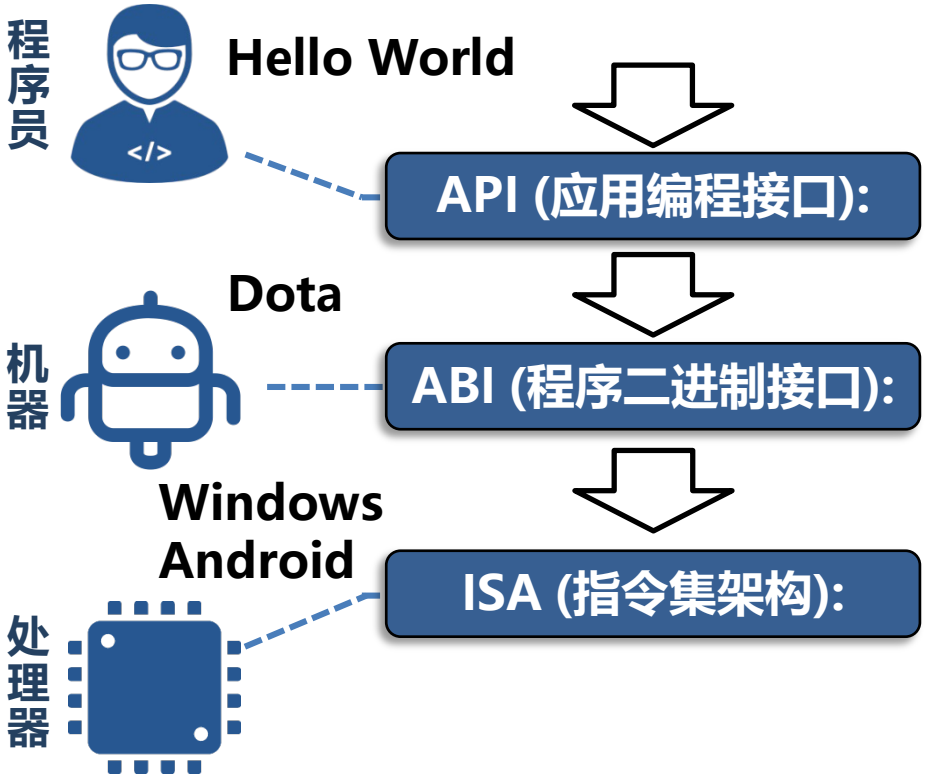
操作系统中的接口层次

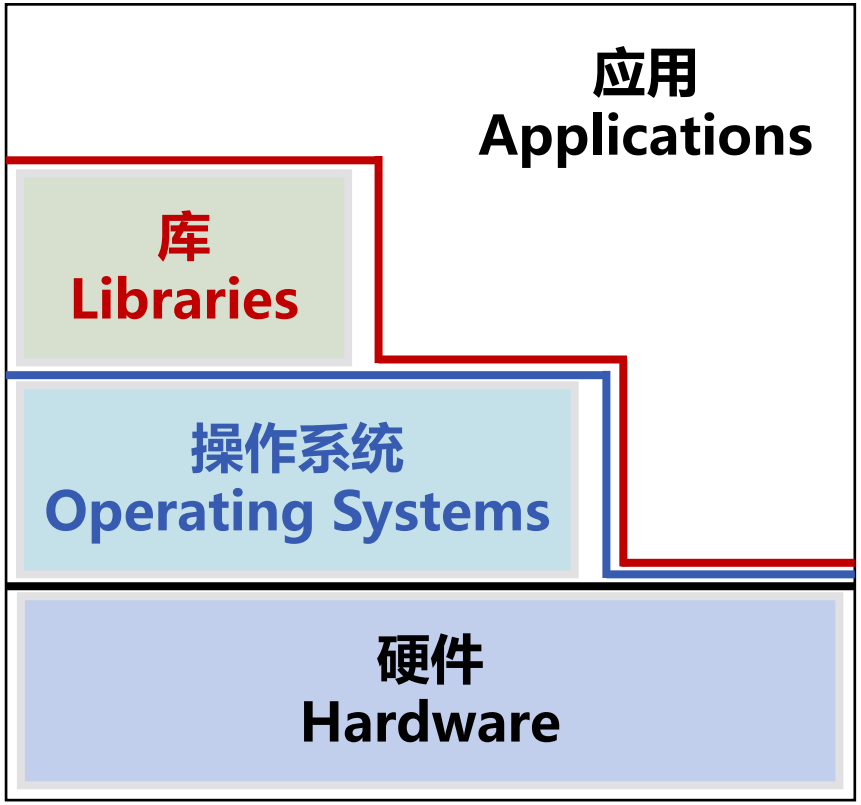
- 多样化的接口提供了不同的执行环境视角，丰富了 对 “机器” 的定义



不同接口层次面对的对象

关键资源下沉，逐层通过 接口（Interface） 交互





接口层级与虚拟化的关系

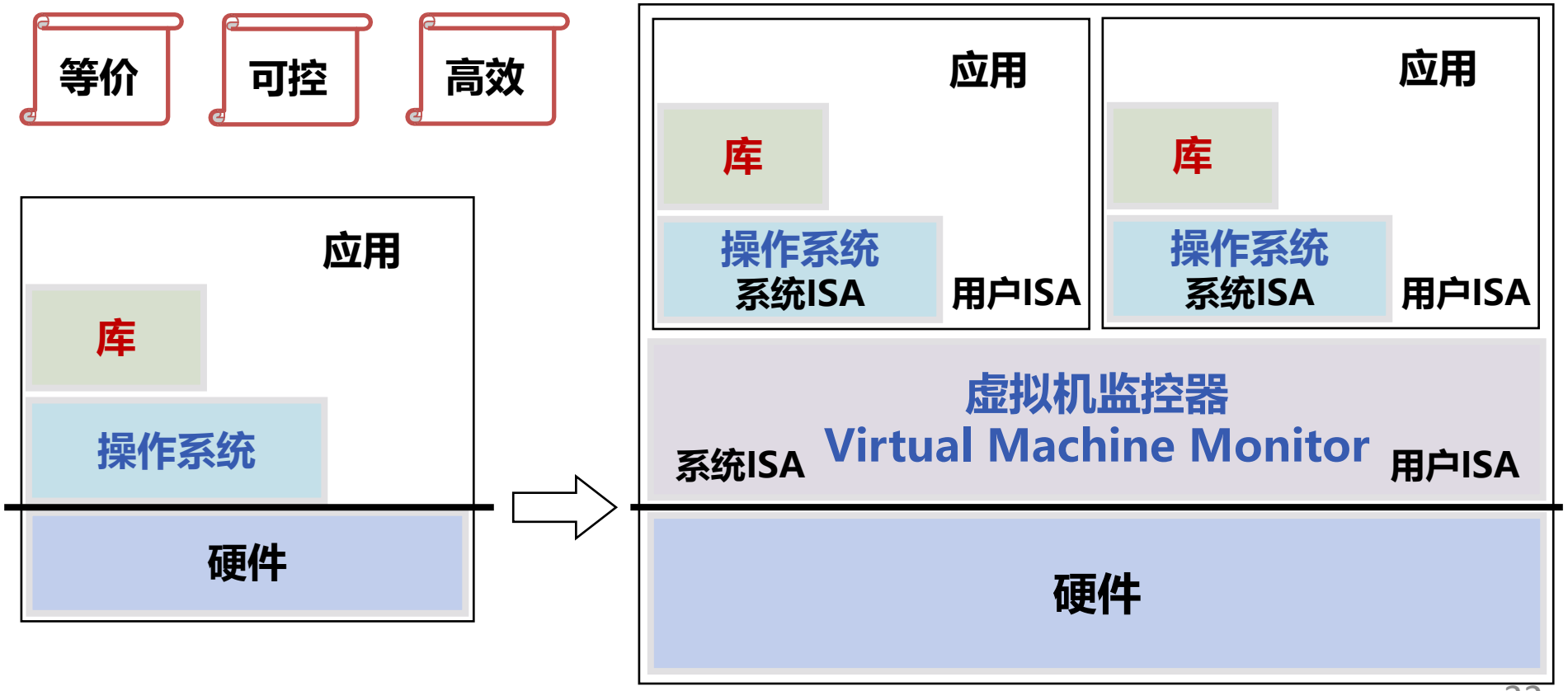
虚拟化 = 在某个接口层创建一个 "假的" 实现

不同的虚拟化方式，对应不同的接口层级：

ISA 层虚拟化 模拟整套硬件指令集	QEMU（模拟ARM on x86）
ABI 层虚拟化 拦截系统调用	Wine（在Linux上运行Windows程序）
API 层虚拟化 提供兼容的API	容器（Docker）、JVM

高效虚拟化的三个特征

- 通过模拟 ISA，为虚拟机内程序提供与硬件完全一致的接口
- 虚拟机监控器控制所有物理资源：CPU、内存、设备
- 高效虚拟化仅比硬件的性能略差 (大多数指令避免VMM干预)



Poppek and Goldberg. Formal Requirements for Virtualizable Third Generation Architecture. 1974

虚拟机类型与 Hypervisor

VM Types & Hypervisor Architecture

虚拟机的两大类别

进程虚拟机 (Process VM)

为单个进程提供虚拟运行环境

典型例子：

- JVM (Java Virtual Machine)
- .NET CLR
- Python 解释器
- Wine（在Linux上运行.exe）

特点：只虚拟化应用层
跨平台、轻量级

系统虚拟机 (System VM)

虚拟化整个计算机系统

典型例子：

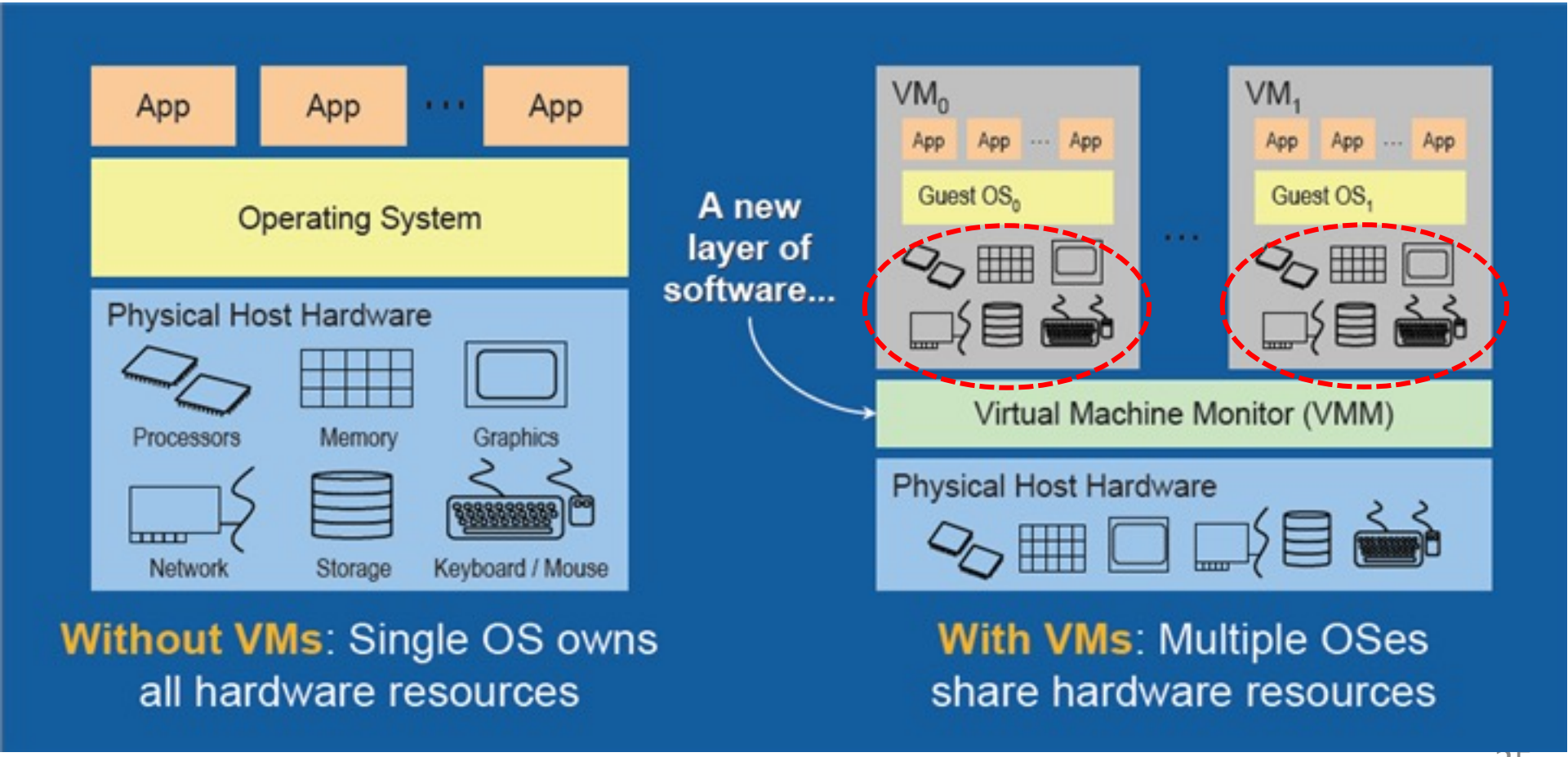
- VMware ESXi / Workstation
- KVM + QEMU
- VirtualBox
- Hyper-V

特点：虚拟化到硬件层
可运行完全不同的 OS

本节课重点关注系统虚拟机（System VM）

虚拟机监控器 VMM

- 引入一个新的软件层，虚拟机监控器（帮助隔离，帮助翻译）
- 每个虚拟机拥有完整的“硬件资源”



Hypervisor（虚拟机监控器 VMM）

Hypervisor = Virtual Machine Monitor (VMM)
负责创建、管理和调度虚拟机，是虚拟化的 "大管家"

Hypervisor 的核心职责：

- CPU 调度：决定哪个 VM 在何时使用物理 CPU
- 内存管理：维护每个 VM 的内存映射和隔离
- I/O 管理：管理虚拟设备与物理设备的映射
- 安全隔离：确保 VM 之间互不干扰

类比：Hypervisor 就像公寓楼的物业管理公司

虚拟机监控器的类型

- Type-1 型 VMM
  - 直接运行在硬件之上，控制硬件资源
  - 充当操作系统的角色
  - 实现调度、内存管理、驱动的功能
  - 性能损失较小

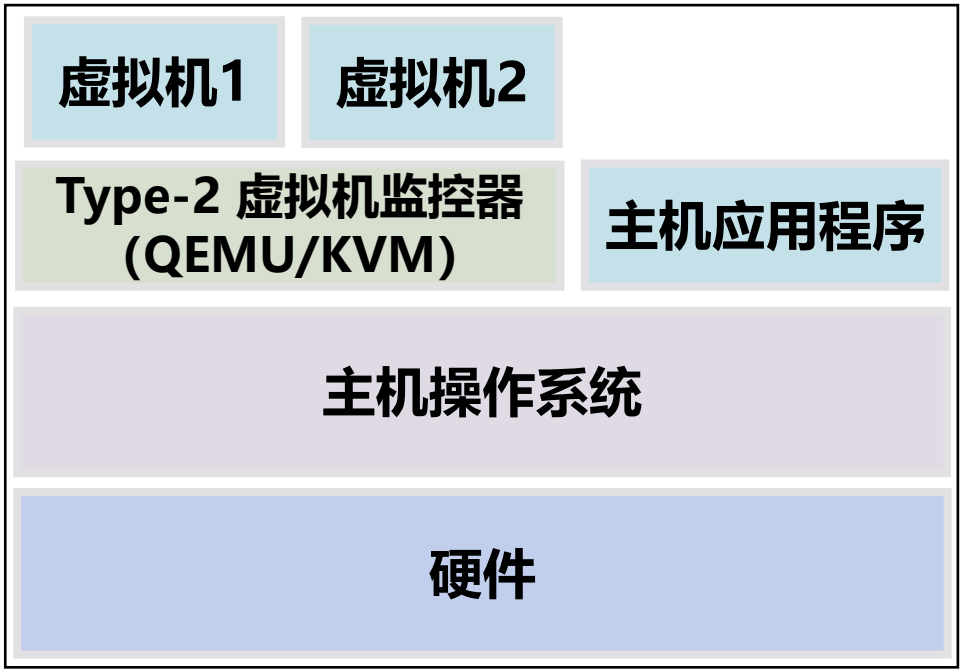
虚拟机监控器的类型

- Type-2 型 VMM
  - 依托于主机操作系统，由主机操作系统管理物理资源
  - VMM 以进程/内核模块的形态运行
  - 复用 OS 大部分功能：文件系统、处理器调度、内存管理
  - 性能损失相对较大

VMware Fusion

Parallel Desktop

VirtualBox



主流 Hypervisor 产品一览

产品	类型	开源	典型场景	特点
VMware ESXi	Type 1	否	企业数据中心	功能最全，生态成熟
KVM	Type 1*	是	云平台 (OpenStack)	Linux 内核模块，主流开源方案
Xen	Type 1	是	AWS 早期, Citrix	半虚拟化先驱
Hyper-V	Type 1	否	Windows Server/Azure	微软生态深度集成
VirtualBox	Type 2	是	个人开发/测试	免费跨平台，上手简单
VMware Workstation	Type 2	否	桌面虚拟化	功能强大的桌面方案

** KVM 虽作为 Linux 内核模块运行，但它直接接管 CPU 虚拟化，属于 Type 1 架构*

KVM 到底是 Type 1 还是 Type 2?

❑这是一个经典的学术争议，面试也常问！

❑看起来像 Type 2 的理由：

- KVM 是 Linux 内核的一个模块，Linux 既是 Host OS 又充当 Hypervisor
- 用户需要在 Linux 系统上启动 KVM

❑实际上是 Type 1 的理由：

- KVM 作为内核模块直接使用 VT-x/AMD-V 硬件支持
- VM 执行时，CPU 进入 Guest 模式，KVM 直接管理硬件
- Linux 内核本身变成了 Hypervisor 的一部分

业界主流观点：KVM 是 Type 1 (Bare-metal) Hypervisor

更准确的说法是 "混合型"：Linux + KVM 共同构成了 Hypervisor 层

如何实现系统虚拟化?

System-level Virtualization Implementation

如何实现虚拟化?

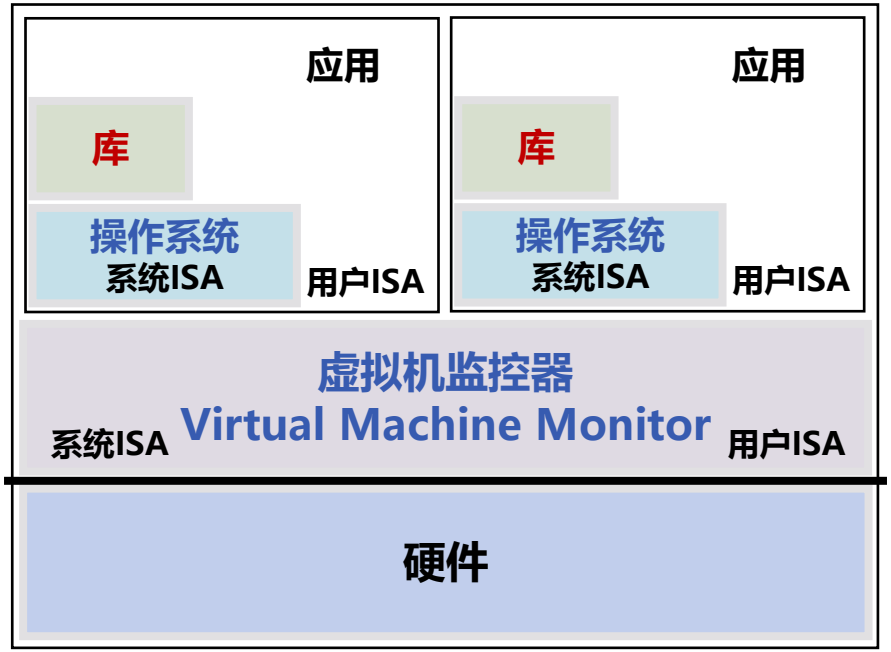
□用户 ISA 虚拟化

- 操作系统通过**进程**这一抽象，支持多个应用运行在同一个 CPU
- MOV, ADD, SUB

□系统 ISA 虚拟化（**关键步骤！**）

- 系统 ISA 会影响整个系统，进而影响其他进程或虚拟机
- 读写敏感寄存器
- 控制处理器行为
- 控制虚拟/物理内存
- 控制外设

原先仅针对单一操作系统，
若多个操作系统同时控制上
述资源则会造成混乱

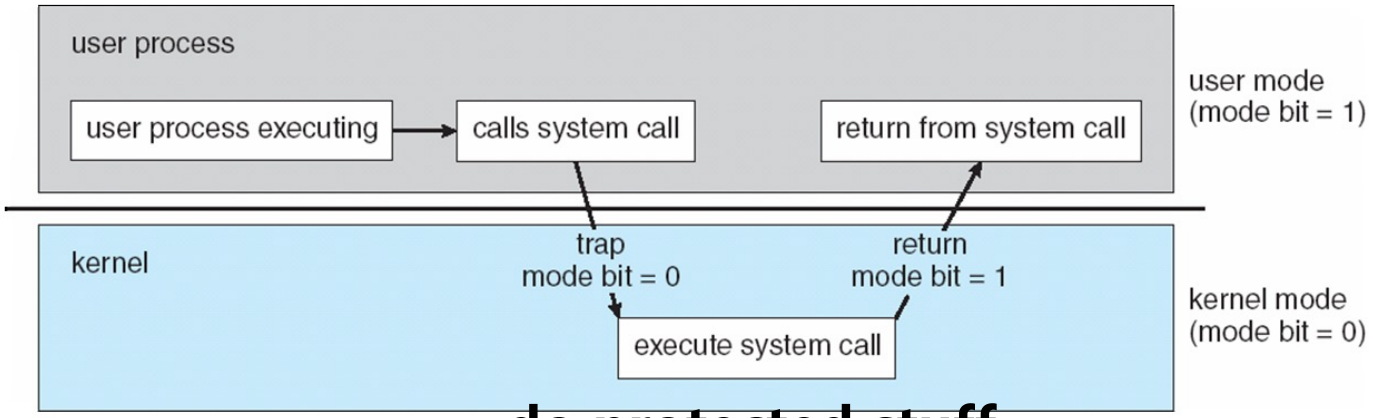


如何捕捉系统 ISA?

- CPU 在**用户态**执行系统 ISA 时，便会**下陷 (Trap)** 到内核处理
- 在**内核态**，操作系统对计算机的硬件和软件资源拥有完全的控制权，可执行任何系统级任务
- 典型下陷场景
  - 执行系统调用 (文件操作/网络通信/进程控制): svc (supervisor call)
  - 指令触发异常，如除以零、缺页
  - 外设中断信息，如键盘、鼠标

system call/page fault/enter on interrupt

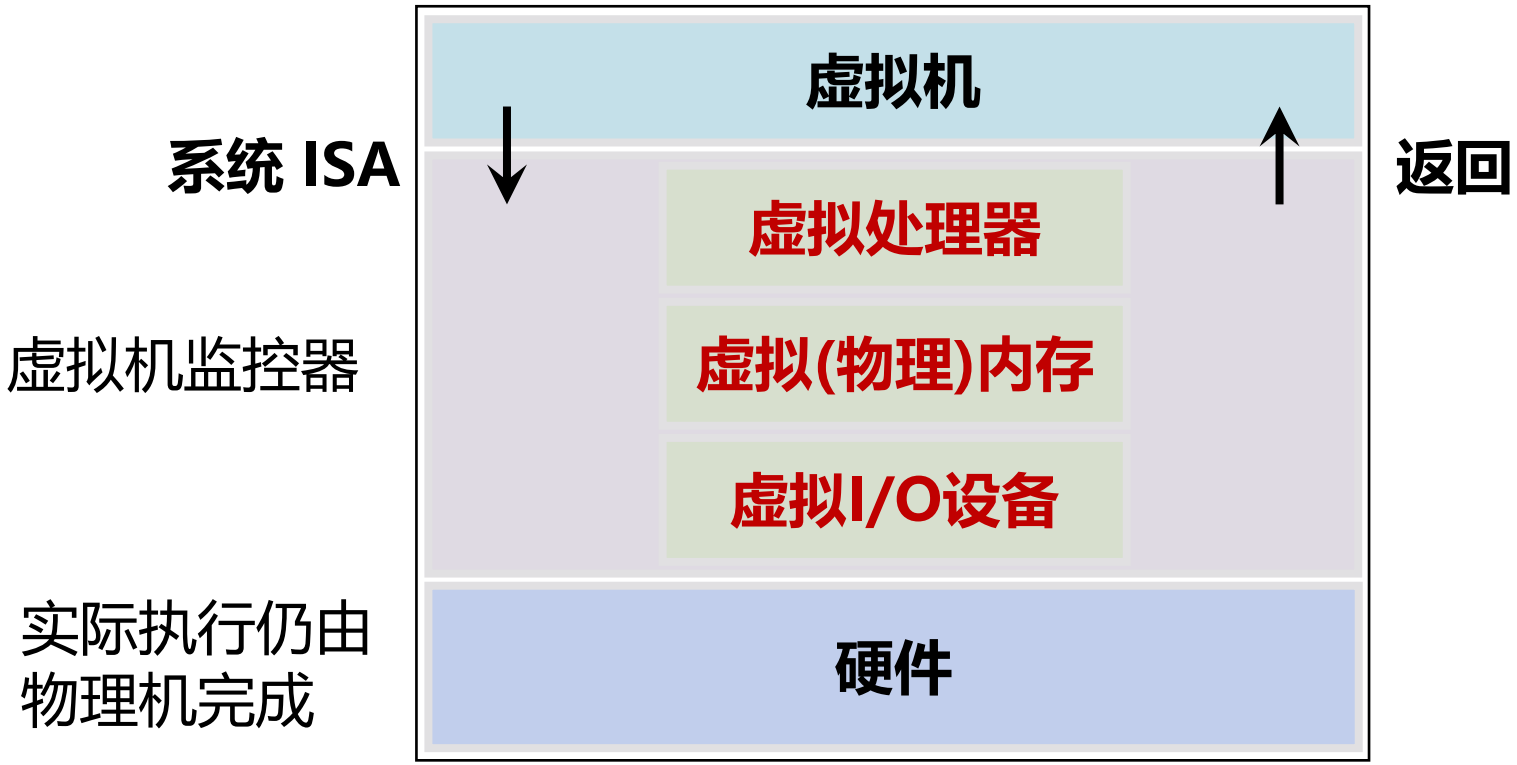
exit back to user program



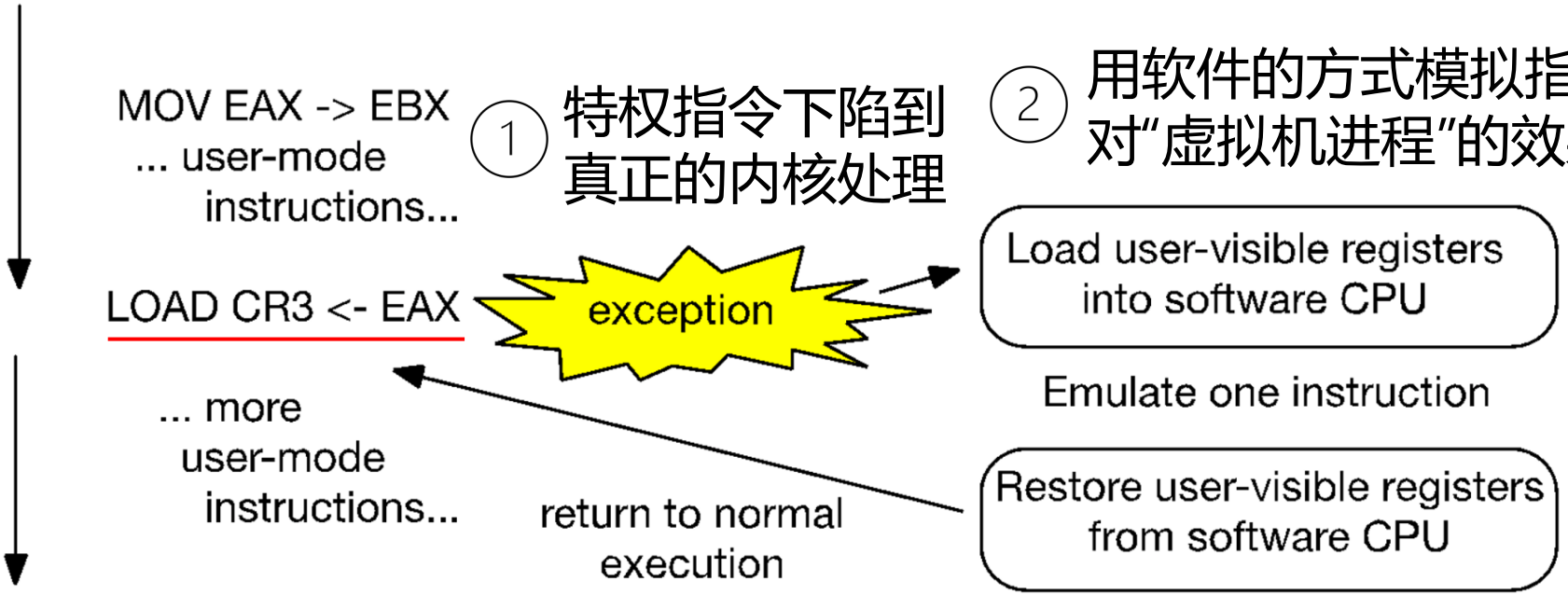
do protected stuff

系统虚拟化流程

- **下陷 – 模拟（Trap-and-Emulate)**
  - Step 1：捕捉所有系统 ISA 下陷
  - Step 2：模拟出指令的效果，实现**系统 ISA 虚拟化**
    - 虚拟处理器、虚拟内存、虚拟设备
  - Step 3：回到虚拟机继续执行



下陷 – 模拟过程



① 特权指令下陷到真正的内核处理

② 用软件的方式模拟指令对“虚拟机进程”的效果

③ 将结果加载到 CPU 里，然后返回用户态的下一条指令

EAX (Extended Accumulator Register)：x86架构中的一个32位寄存器，用于算数运算
EBX (Extended Base Register)：x86架构中的一个32位寄存器，用作基址寄存器，进行内存寻址
CR3 (Control Register 3)：x86架构中用于存储页目录的物理地址，用户态程序不可访问

这个过程有什么问题?

- 捕捉**下陷指令**并处理是为了防止影响整个系统
- 然而，并不是所有对系统有影响的指令均会下陷!

特权指令 Privileged instructions

在用户态执行时会触发下陷的指令，包括主动触发下陷的指令和不允许在用户态执行的指令。

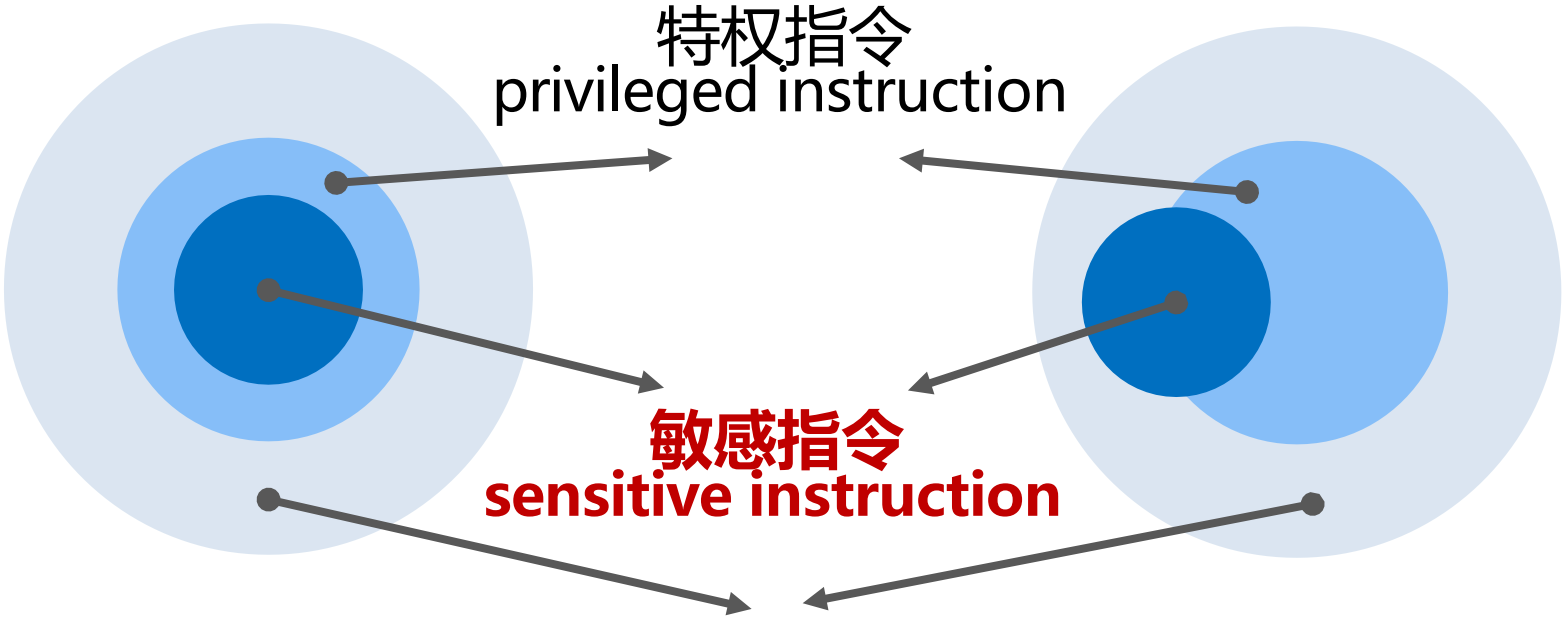
敏感指令 Sensitive instructions

管理系统物理资源或者更改CPU状态的指令：读写特殊寄存器、读写敏感内存、I/O指令。

当所有敏感指令都是特殊指令，在非特权级执行时都会发生下陷时，才为可虚拟化架构

虚拟化原则

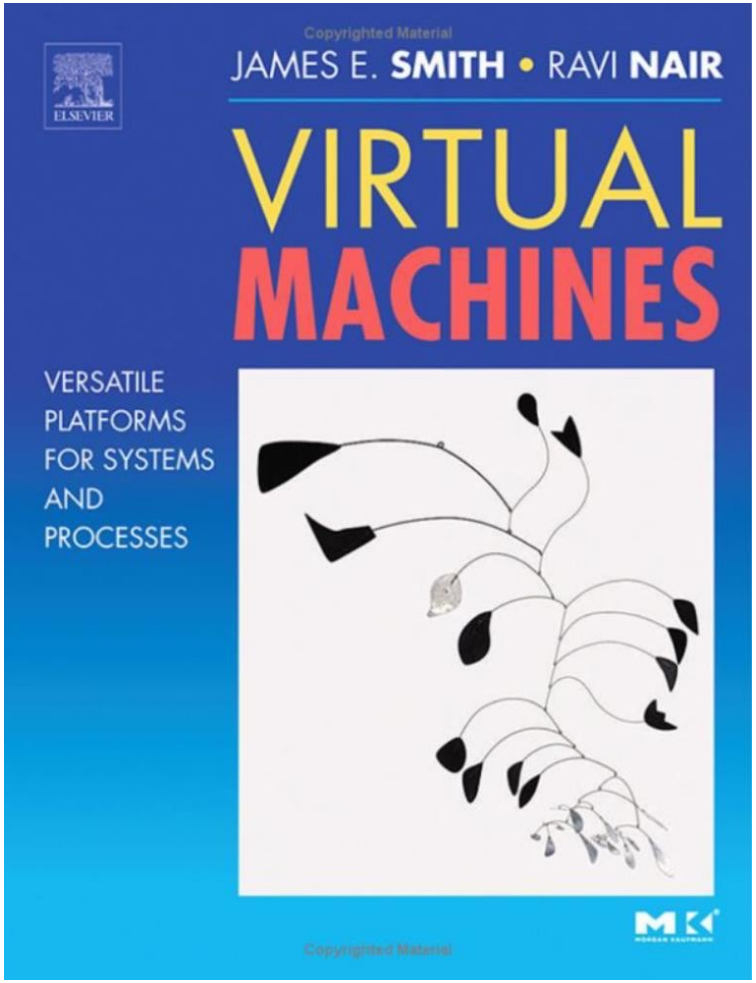
Conditions for ISA Virtualizability

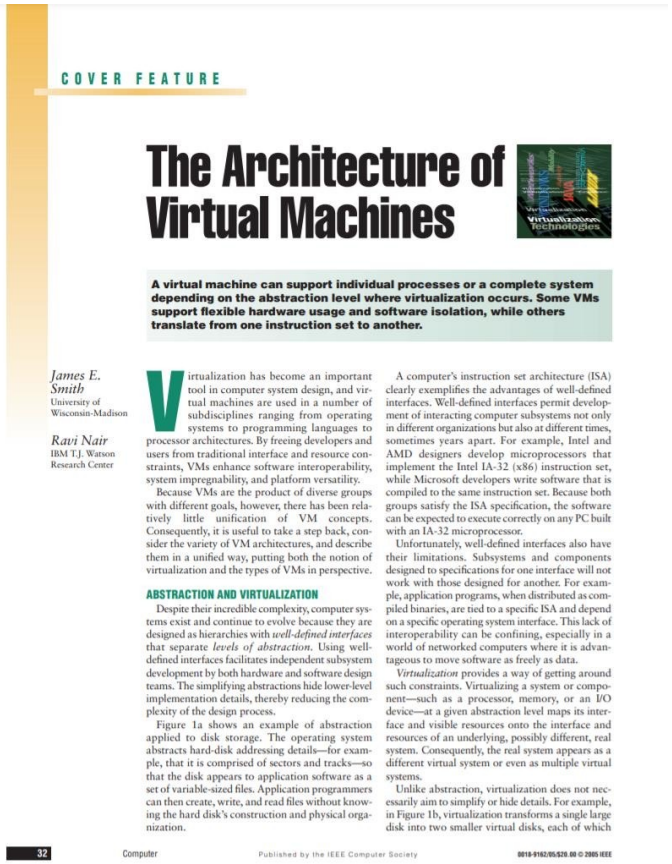


可虚拟化的架构
(如 RISC 架构)

不可虚拟化的架构
(如 ARM, x86 架构)

虚拟化原则





COVER FEATURE

The Architecture of Virtual Machines

A virtual machine can support individual processes or a complete system depending on the abstraction level where virtualization occurs. Some VMs support flexible hardware usage and software isolation, while others translate from one instruction set to another.

James E. Smith
University of Wisconsin-Madison

Ravi Nair
IBM T.J. Watson Research Center

Virtualization has become an important tool in computer system design, and virtual machines are used in a number of subdisciplines ranging from operating systems to programming languages to processor architectures. By freeing developers and users from traditional interface and resource constraints, VMs enhance software interoperability, system impregnability, and platform versatility.

Because VMs are the product of diverse groups with different goals, however, there has been relatively little unification of VM concepts. Consequently, it is useful to take a step back, consider the variety of VM architectures, and describe them in a unified way, putting both the notion of virtualization and the types of VMs in perspective.

ABSTRACTION AND VIRTUALIZATION

Despite their incredible complexity, computer systems exist and continue to evolve because they are designed as hierarchies with *well-defined interfaces* that separate *levels of abstraction*. Using well-defined interfaces facilitates independent subsystem development by both hardware and software design teams. The simplifying abstractions hide lower-level implementation details, thereby reducing the complexity of the design process.

Figure 1a shows an example of abstraction applied to disk storage. The operating system abstracts hard-disk addressing details—for example, that it is comprised of sectors and tracks—so that the disk appears to application software as a set of variable-sized files. Application programmers can then create, write, and read files without knowing the hard disk's construction and physical organization.

A computer's instruction set architecture (ISA) clearly exemplifies the advantages of well-defined interfaces. Well-defined interfaces permit development of interacting computer subsystems not only in different organizations but also at different times, sometimes years apart. For example, Intel and AMD designers develop microprocessors that implement the Intel IA-32 (x86) instruction set, while Microsoft developers write software that is compiled to the same instruction set. Because both groups satisfy the ISA specification, the software can be expected to execute correctly on any PC built with an IA-32 microprocessor.

Unfortunately, well-defined interfaces also have their limitations. Subsystems and components designed to specifications for one interface will not work with those designed for another. For example, application programs, when distributed as compiled binaries, are tied to a specific ISA and depend on a specific operating system interface. This lack of interoperability can be confining, especially in a world of networked computers where it is advantageous to move software as freely as data.

Virtualization provides a way of getting around such constraints. Virtualizing a system or component—such as a processor, memory, or an I/O device—at a given abstraction level maps its interface and visible resources onto the interface and resources of an underlying, possibly different, real system. Consequently, the real system appears as a different virtual system or even as multiple virtual systems.

Unlike abstraction, virtualization does not necessarily aim to simplify or hide details. For example, in Figure 1b, virtualization transforms a single large disk into two smaller virtual disks, each of which

Published by the IEEE Computer Society 0018-9162/05/$20.00 © 2005 IEEE

x86 的困境

核心问题：x86 有 17 条“敏感但非特权”指令

这些指令会影响系统状态（敏感），但在用户态执行时不会触发 Trap（非特权）
VMM 完全不知道 Guest 执行了这些指令 → 虚拟化语义被破坏！

典型的问题指令：

SGDT / SIDT — 读取全局/中断描述符表地址
Guest 读到的是 Host 的真实地址，而非虚拟值

POPF — 弹出标志寄存器
在用户态静默忽略敏感位的修改，不触发 Trap

PUSHF — 压入标志寄存器
暴露真实的特权级别，Guest 可能发现自己不在 Ring 0

导致的后果：

Guest OS 可能崩溃或行为异常
VMM 无法完全控制虚拟机
x86 不满足 Popek-Goldberg 定理

→ 所以需要其他技术来弥补！

这就催生了四种不同的 **CPU 虚拟化解决方案**

虚拟化原则

- ARM 也不是可虚拟化架构
- Change Process State (Interrupt Disable, Interrupt Enable)

CPSID **CPSIE**

分别用于打开和关闭中断

- 内核态执行：PSTATE.{A, I, F} 可以被 CPS 指令修改
  - PSTATE 是 ARM 架构中的程序状态寄存器
  - A（Asynchronous Abort Mask）：异步异常屏蔽位
  - I（IRQ Mask）：中断请求屏蔽位
  - F（FIQ Mask）：快速中断请求屏蔽位
- 用户态执行：CPS 被当作 NOP 指令，不产生任何效果
- 因此不是特权指令

https://developer.arm.com/documentation/dui0489/i/Cihfdbhd

CPU 虚拟化核心技术

CPU Virtualization Techniques

CPU 虚拟化：四种主要技术路线

解释执行 Interpretation

逐条翻译执行
每条指令都经 VMM
最慢但最通用

QEMU (纯软件模拟)

二进制翻译 Binary Translation

动态替换敏感指令
大部分指令直接执行
较好的兼容性

VMware (早期)

半虚拟化 Paravirtualization

修改Guest OS
用 Hypercall替换
需要源码配合

Xen (早期)

硬件辅助 HW-Assisted

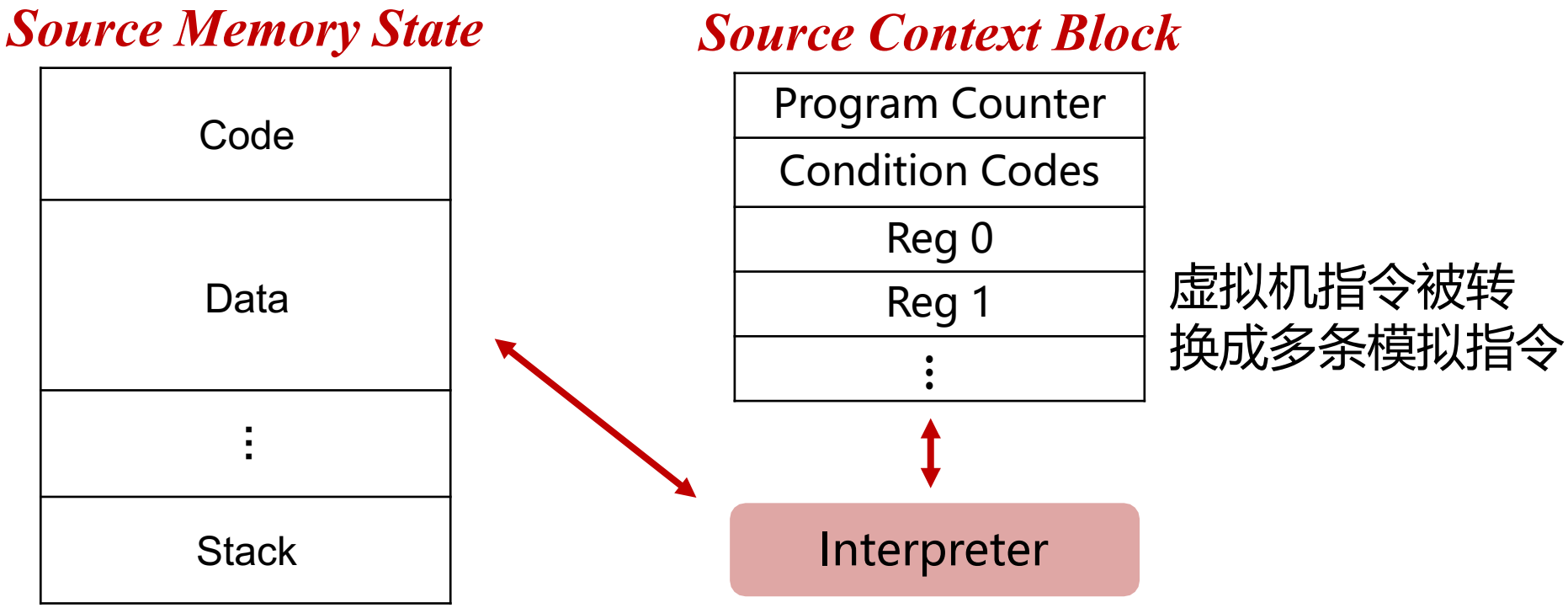
CPU原生支持
VMX root/non-root
当今主流方案

KVM + VT-x
VMware + VT-x

性能提升 ——→ 当今主流

解释执行 (Interpretation)

最简单也最慢的虚拟化方式
VMM 逐条取出 Guest 的每条指令，分析其类型和操作数，然后模拟执行



An interpreter manages the complete architected state of a machine implementing the source ISA

解释执行 (Interpretation)

最简单也最慢的虚拟化方式
VMM 逐条取出 Guest 的每条指令，分析其类型和操作数，然后模拟执行

工作流程：
1. 取指（Fetch）：从 Guest 内存中取出一条指令
2. 解码（Decode）：分析指令类型、寄存器、操作数
3. 模拟（Emulate）：在 VMM 中**用软件模拟该指令的效果**
4. 写回（Writeback）：更新 Guest 的虚拟寄存器和内存

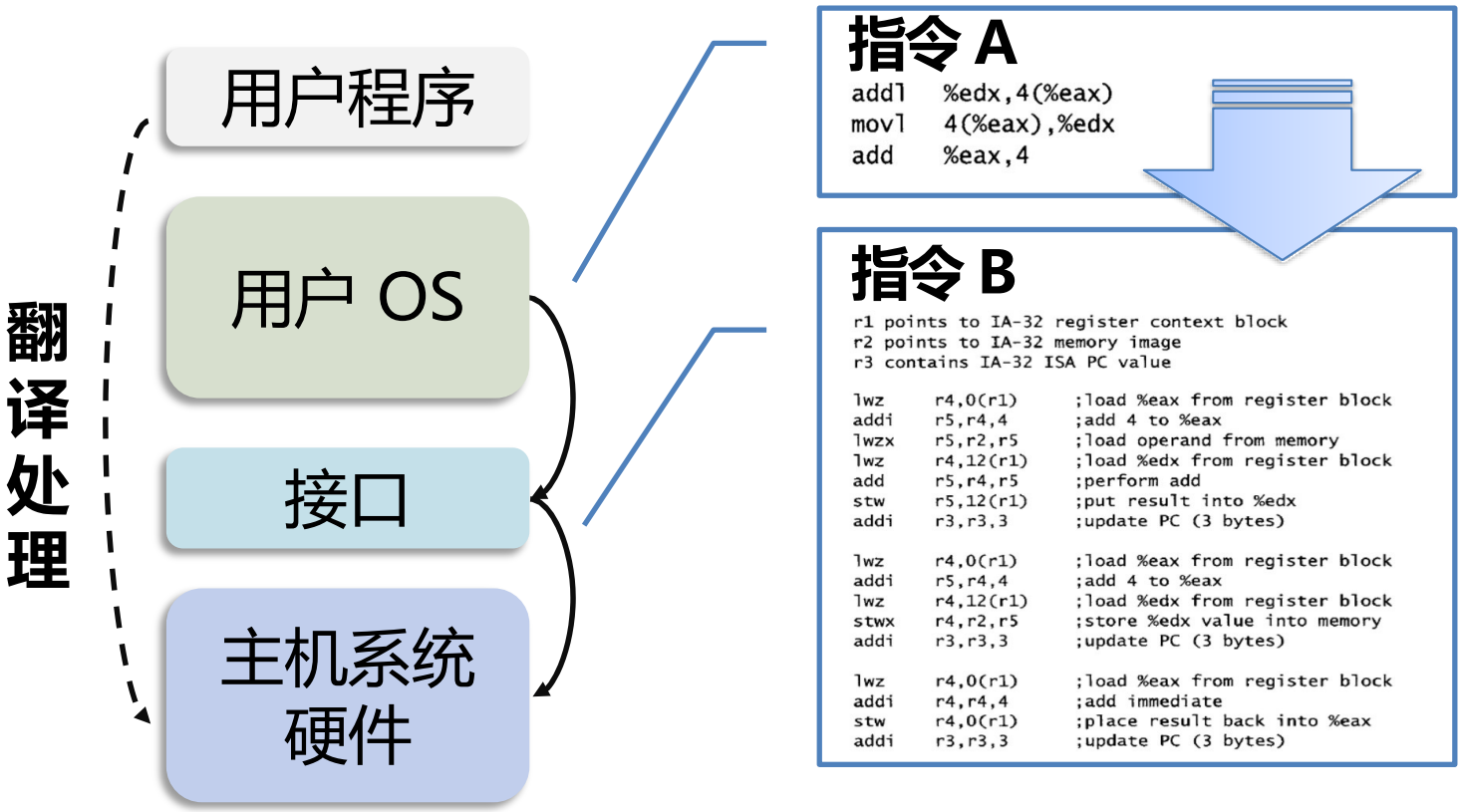
代表：QEMU（不启用 KVM 时）

优点：不依赖下陷，可以模拟任意架构（在 x86 上跑 ARM/MIPS）
缺点：比原生执行慢 100-1000 倍！

二进制翻译 (Binary Translation)

VMware 的 "魔法"：不修改 Guest OS，但能安全运行

扫描 Guest 代码块，动态替换其中的敏感指令为安全的等效代码



二进制翻译 (Binary Translation)

VMware 的 "魔法"：不修改 Guest OS，但能安全运行
扫描 Guest 代码块，动态替换其中的敏感指令为安全的等效代码。

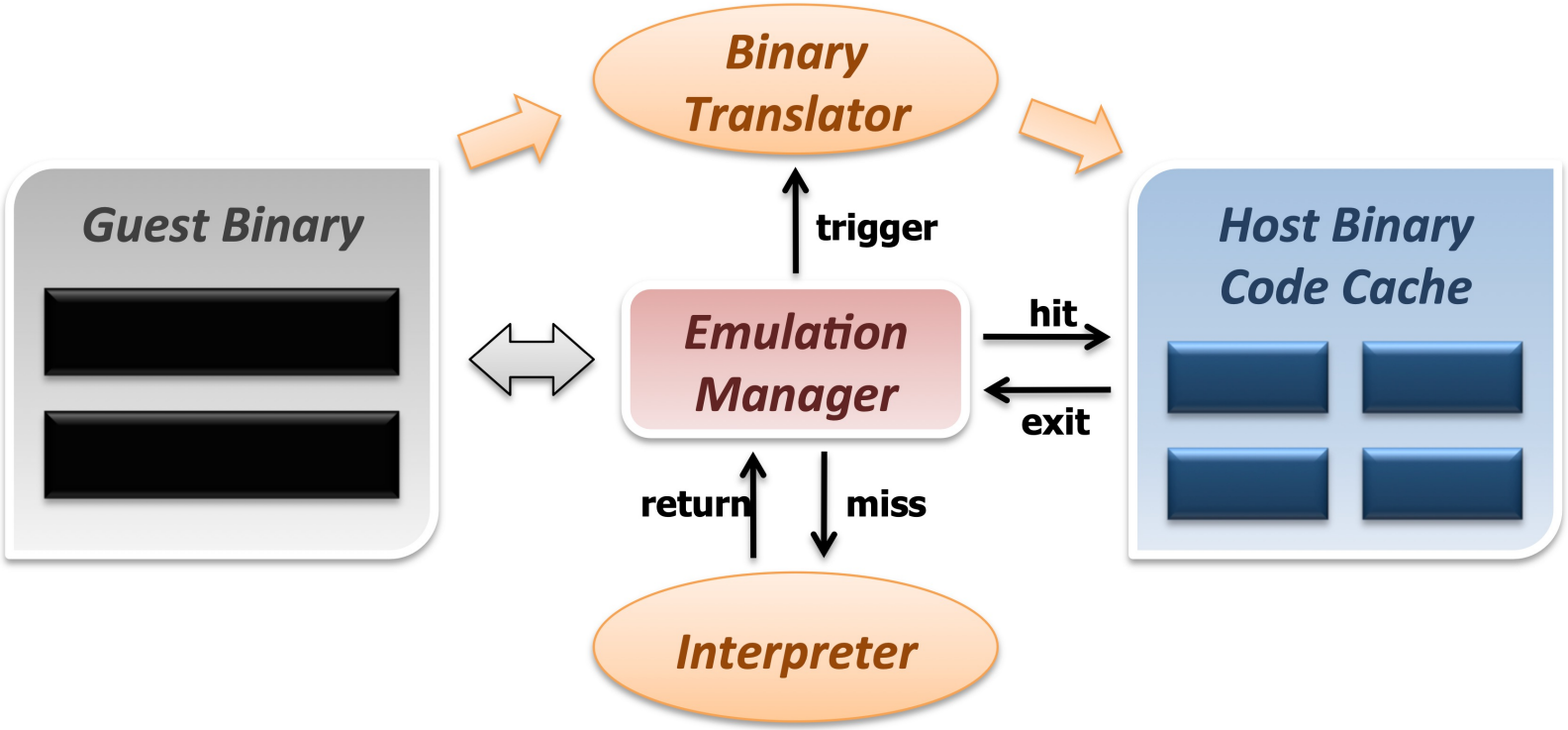
原理：

1. VMM 拦截 Guest OS 的代码执行流
2. 将代码分成基本块（Basic Block）
3. 找到敏感指令（如 SGDT）→ 替换为安全的 VMM 调用
4. **翻译后的代码缓存起来，下次直接用（Translation Cache）**

优点：不需要修改 Guest OS，支持 Windows 等闭源系统
缺点：翻译有开销（约 5-20% 性能损失），实现非常复杂，同时中断粒度变大，只能在基本块边界插入虚拟中断

二进制翻译 (Binary Translation)

- 相比解释执行，动态二进制翻译采用**批量翻译**与**缓冲**提高性能
- 缓存翻译好的代码块（如循环或热点函数），便于后续重用
- 翻译代码块前先检查其是否在缓存中



二进制翻译 (Binary Translation)

1. Guest OS 原始代码（在硬盘上或刚加载到内存时）

假设 Guest OS 的某段二进制代码中包含以下指令：

```
MOV EAX, 1      ; 普通指令：把 1 放入 EAX 寄存器
ADD EBX, 2      ; 普通指令：EBX 加上 2
SGDT [Memory]   ; 危险敏感指令！读取真实 GDT 地址到内存中
CMP EAX, EBX    ; 普通指令：比较操作
```

2. VMM 的“查找与替换”动作

当 VMM 准备让这段代码在 CPU 上运行时，它会先翻译这段代码，将其缓存在翻译缓存（Translation Cache）中。翻译后的代码如下：

```
MOV EAX, 1              ; 安全，直接保留
ADD EBX, 2              ; 安全，直接保留
CALL VMM_Emulate_SGDT   ; <--- 核心！SGDT 被替换成了一个安全调用！
CMP EAX, EBX            ; 安全，直接保留
```

GDT (Global Descriptor Table) 是 x86 架构中一个非常核心的数据结构。它像一张“大表”，记录了内存分段的信息（比如哪段内存是代码，哪段是数据，它们的起始地址和权限是什么）

二进制翻译 (Binary Translation)

3. 执行过程

- 物理 CPU 直接飞速执行 MOV 和 ADD 指令
- 当执行到第三条指令时，CPU 并没有执行 SGDT，而是执行了 CALL VMM_Emulate_SGDT，程序主动跳转到了 VMM 预先写好的一段 C/汇编代码中

4. VMM 模拟代码内部发生的事情（伪代码）：

```
void VMM_Emulate_SGDT() {
  // VMM 知道当前是哪个虚拟机在请求
  VM_Context* current_vm = get_current_vm();

  // VMM 不去读真实的硬件 GDT，而是取出该虚拟机的“伪造 GDT 地址”
  uint32_t fake_gdt_address = current_vm->virtual_gdt_base;

  // 把这个伪造的地址，安全地写回到 Guest OS 指定的内存位置 [Memory] 中
  write_to_guest_memory(Guest_Memory_Address, fake_gdt_address);

  // 模拟完成，返回 Guest OS 的下一条指令 (CMP EAX, EBX)
  return;
}
```

VMware 的车库传奇与 x86 虚拟化

1998年，斯坦福大学的 Mendel Rosenblum 教授和他的妻子 Diane Greene 在加州 Palo Alto 创办了 VMware。

当时，学术界普遍认为 x86 架构 "不可虚拟化"：

x86 有 17 条 "敏感指令" 不会触发陷阱（trap），违反了 Popek-Goldberg 条件。

VMware 的天才之处：用 "二进制翻译"（Binary Translation）绕过了这个限制！

在运行时动态替换那些 "不听话" 的指令。

1999年 VMware Workstation 发布，震惊了整个行业。

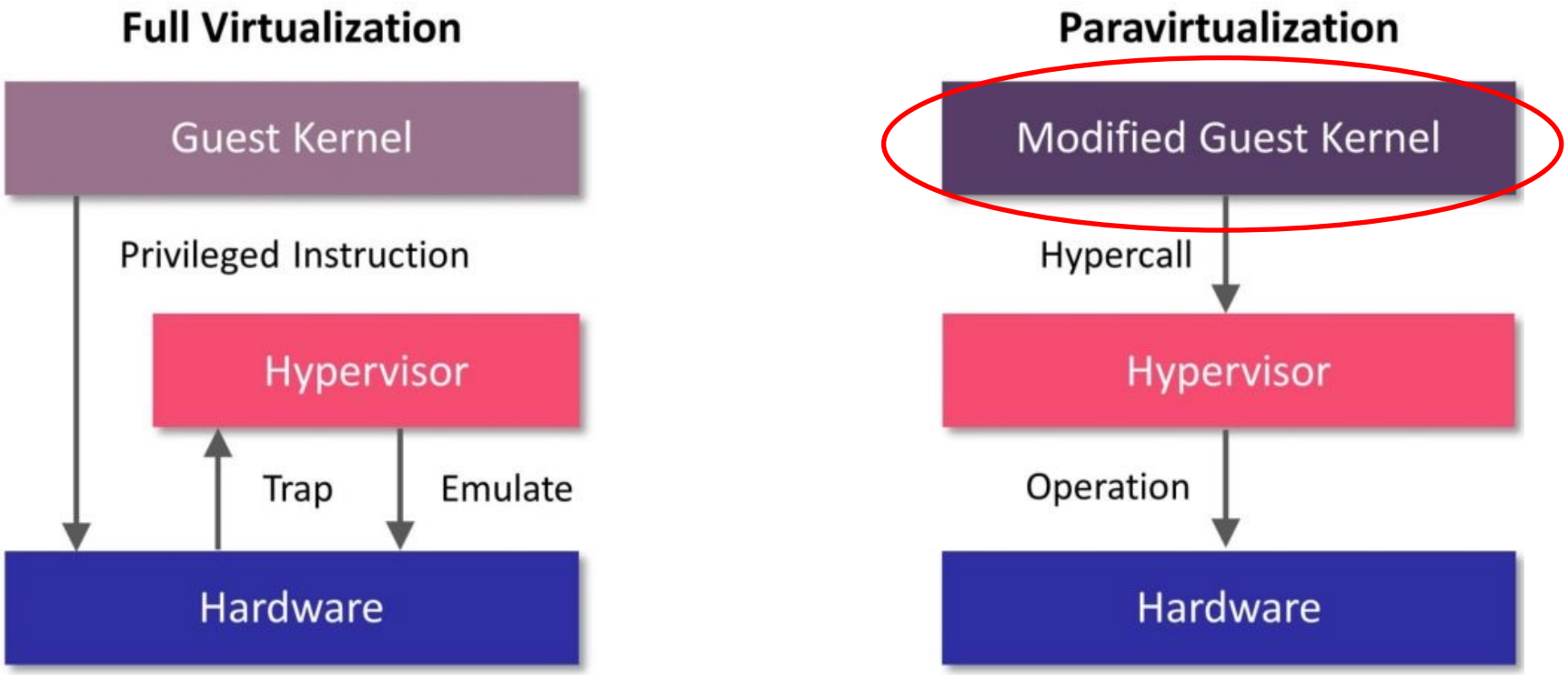
2004年 被 EMC 以 $6.35 亿收购，后来市值一度超过 $600 亿！

教训：当所有人都说 "不可能" 的时候，也许只是方法还没找对。

半虚拟化 (Paravirtualization)

换一个思路：既然 Guest OS 的指令有问题，不如直接改 Guest OS!

把不能引起下陷的敏感指令替换为 Hypercall (类似系统调用，但调用的是 VMM)



半虚拟化 (Paravirtualization)

换一个思路：既然 Guest OS 的指令有问题，不如直接改 Guest OS！

把 "不听话" 的指令替换为 Hypercall（类似系统调用，但调用的是 VMM）

Xen 的做法（2003）：

修改 Linux 内核约 1-2% 的代码
将特权指令替换为 Hypercall，虚拟机主动调用 Hypercall
Guest OS "知道" 自己在虚拟机里（不假装是物理机）

System Call vs Hypercall 类比：
System Call：应用程序 → 操作系统（"帮我打开文件"）
Hypercall：Guest OS → Hypervisor（"帮我分配内存页"）

优点：性能接近原生（开销仅 2-5%）
缺点：需要修改 Guest OS 源码，Windows 等闭源 OS 无法使用

x86 特权级模型：Ring 0 ~ Ring 3

□CPU 硬件提供的 4 个保护环 — 权限由内向外递减

每个 Ring 的职责

Ring 0 — 操作系统内核
执行特权指令、管理内存页表、控制中断、访问硬件

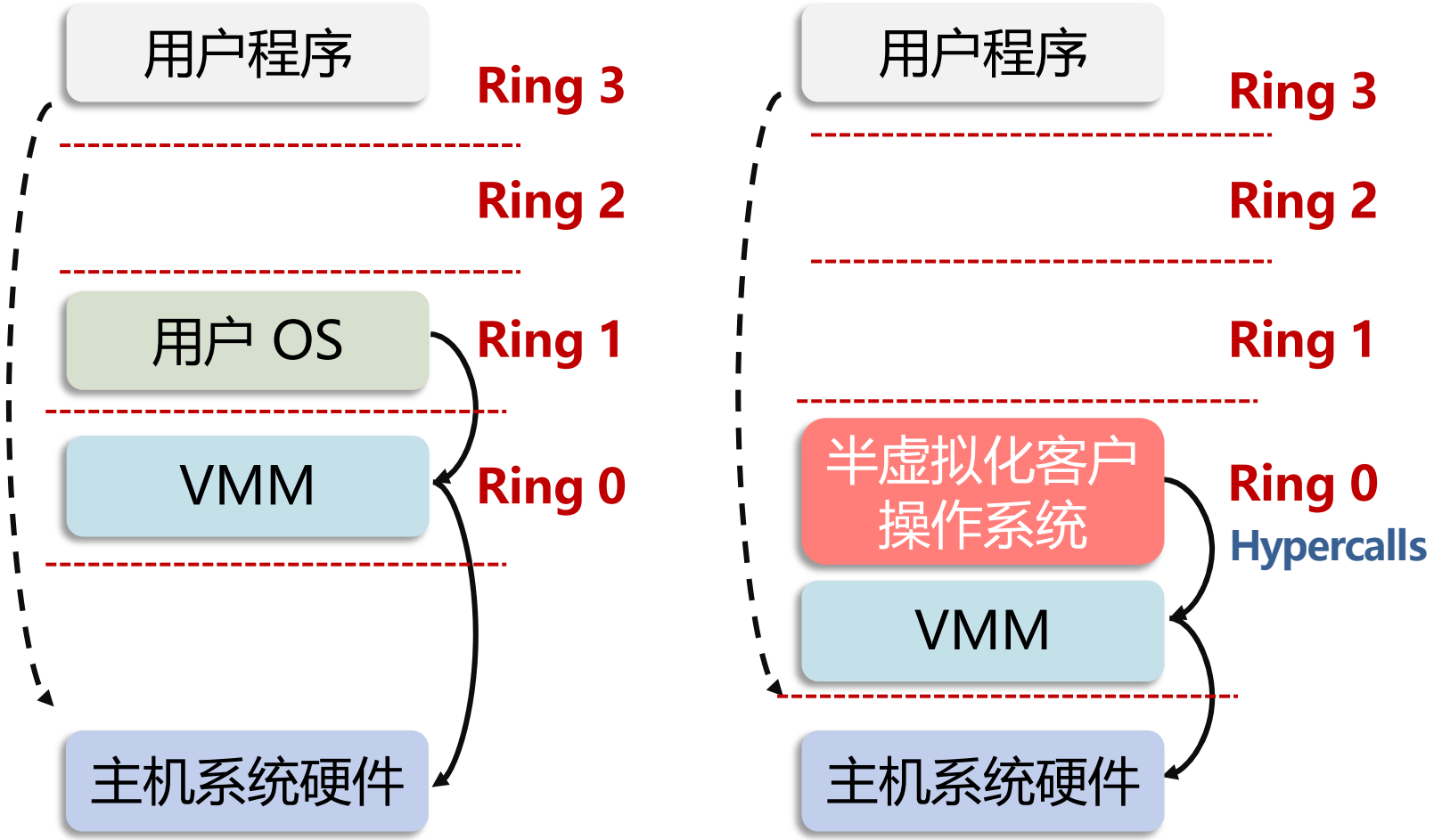
Ring 1 — 设备驱动（历史上）
理论设计，现代 OS 几乎不用

Ring 2 — 系统服务（历史上）
理论设计，现代 OS 几乎不用

Ring 3 — 用户应用程序
普通程序（如浏览器、Office、游戏、微信）运行于此，不能直接访问硬件

半虚拟化 (Paravirtualization)

- 半虚拟化客户操作系统与虚拟监控器**密切协作**
- **客户 OS 知道自己运行在虚拟环境中**，并依此优化自己



硬件辅助虚拟化 (Hardware-Assisted)

终极方案：让 CPU 自己支持虚拟化！
Intel VT-x (2005) / AMD-V (2006)：在 CPU 中新增虚拟化扩展指令

VMX Root Mode
(Hypervisor 运行在此)
拥有完全硬件控制权

→ VM Entry

← VM Exit

VMX Non-Root Mode
(Guest OS 运行在此)
受控的执行环境

Guest OS 不需要任何修改，也不需要二进制翻译
CPU 自动捕获敏感操作并触发 VM Exit → 当今几乎所有虚拟化方案的基础

硬件辅助虚拟化

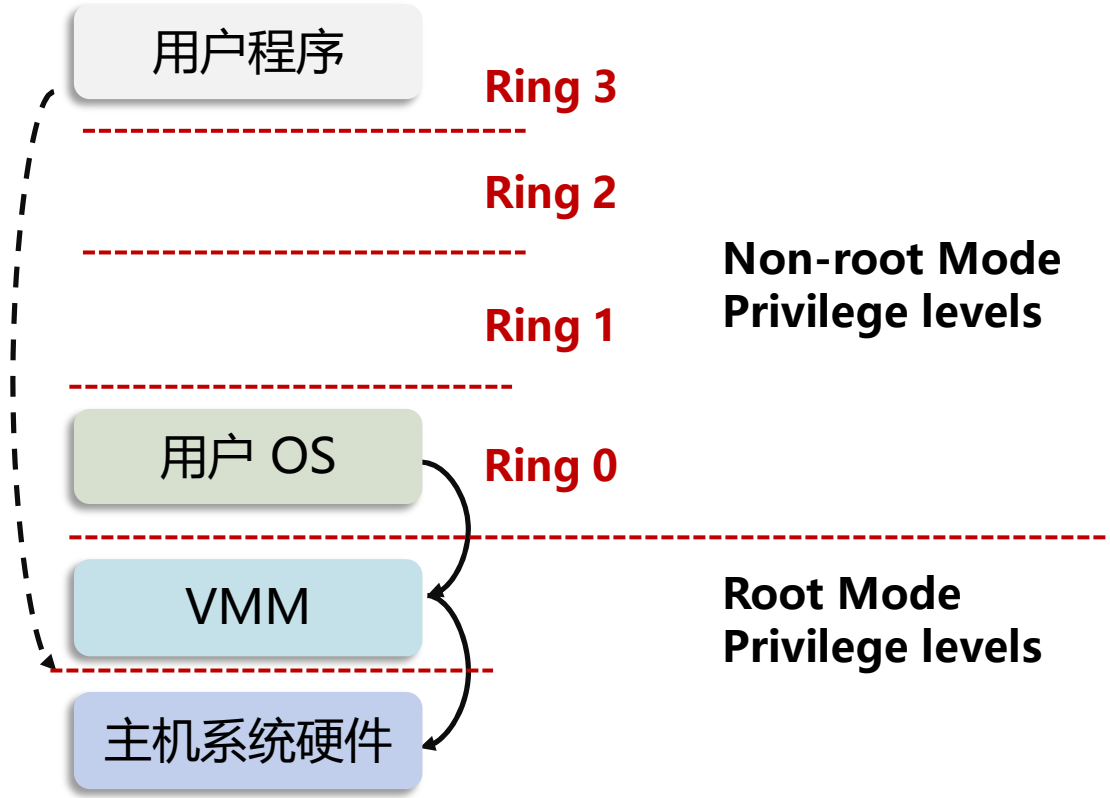
- 传统虚拟化技术中，VMM 和客户 OS **共享同一个特权级别**，客户 OS 特权指令直接操作硬件，导致**资源冲突**或**逃逸漏洞**(如虚拟机破坏宿主机)

- **硬件辅助虚拟化，由处理器硬件提供架构支持以创建 VMM，因此允许客户机 OS 不加修改直接运行**
  - first introduced on the IBM System/370 in 1972

- 2005/2006年间，Intel 和 AMD 两家公司各自独立推出了解决方案，即提供**处理器扩展** (CPU extensions)
  - Intel VT-x (formerly known as Vanderpool)
  - AMD-V (formerly known as Pacifica)

硬件辅助虚拟化

- ✓在已有 CPU 权级下增加根模式（root mode）和非根模式（non-root mode）
- ✓Intel VT-x 为每一个虚拟机提供虚拟机控制结构 (virtual machine control structure, VMCS)
- ✓在硬件层面实现了对敏感指令的监控和拦截，都能被 VMM 捕获



客户 OS 运行在非根模式，表面上可以执行特权指令，但实际受硬件限制，只能在VMM 的控制下运行

VMM 运行在根模式，拥有对硬件的完全控制权

硬件辅助虚拟化

硬件虚拟化的作用

- CPU 在非根模式下维护一个敏感指令列表
- 当客户机尝试执行这些指令时，CPU 硬件会直接拦截，无需依赖软件模拟或修改客户机代码

示例

- 当客户机执行 MOV CR3 (切换页表) 时，CPU 触发 VM Exit，Hypervisor 会更新扩展页表 (EPT) 以隔离客户机内存
- 当客户机执行 IN EAX, DX (读取端口) 时，CPU 触发 VM Exit，Hypervisor将 I/O 请求转发给虚拟设备，如 QEMU 模拟的磁盘

Trap & Emulate 与四种技术的关系

Trap & Emulate = 虚拟化的理想蓝图

大部分指令直接执行，敏感指令触发 Trap 由 VMM 处理

x86 有敏感指令不会 Trap ↓

绕开	修补	改造	回归
解释执行	**二进制翻译**	**半虚拟化**	**硬件辅助**
放弃原生执行 100% 软件模拟 不依赖 Trap	提前替换 问题指令 其余直接执行	改 Guest OS 主动 Hypercall 不靠被动 Trap	CPU 重新设计 所有敏感操作 → VM Exit
与 T&E 的关系： 彻底绕开问题	**与 T&E 的关系：** 部分利用 Trap & Emulate	**与 T&E 的关系：** 用主动调用替代 Trap	**与 T&E 的关系：** 让 Trap & Emulate 完美实现

越靠右 → 越接近 Trap & Emulate 的理想模型 → 性能越好

各种虚拟化技术的比较

	全虚拟化	**硬件辅助**	**半虚拟化**
技术	二进制翻译	虚拟化扩展	Hypercall
兼容性	好	好	差
性能	较好	好	分场合

中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn