系统虚拟化

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

虚拟化：进程 & 虚存抽象

- 动机：希望能在一个物理计算机上运行多个程序
  - 为此，每个程序都遵循统一的规范
    - 二进制接口(ABI)、操作系统API、操作系统规约
    - 程序可以在一个操作系统上作为进程执行

- 更进一步：能不能在一个物理计算机上运行多个操作系统?
  - 可能是相同的操作系统(但独立运行)
  - 可能是不同的操作系统
  - 甚至可能是不同的体系结构

为什么需要系统虚拟化?

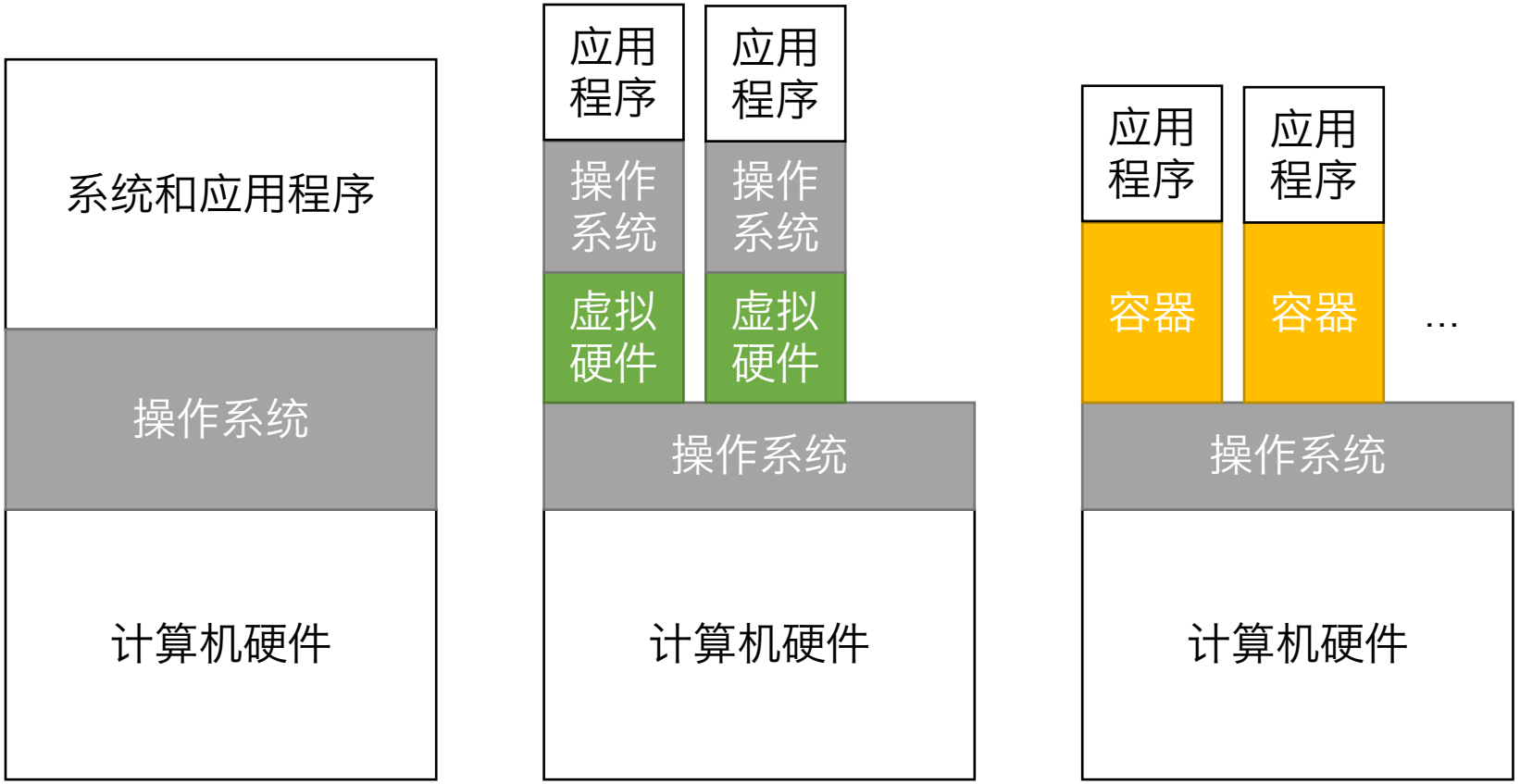
- 希望得到一个干净的运行环境
  - 安全漏洞、调试内核、玩怀旧游戏、……
- 希望复用硬件资源
  - 互联网：一般网站、课程网站、钓鱼网站、诈骗网站……
  - 从服务提供商的角度
    - 我买一台服务器，可以供100个网站用没问题
  - 从用户的角度
    - 我希望独占一台计算机——不要和其他程序互相干扰(例如/tmp)
    - 但租用一台计算机的成本又太高了

系统虚拟化

- 虚拟机：虚拟完整的计算机系统
  - 每个虚拟机都是一台完全独立的计算机

- 容器：虚拟操作系统的实例(容器)
  - 每个虚拟操作系统都管理独立的进程/设备/文件

系统虚拟化 (cont'd)


系统和应用程序
操作系统
计算机硬件
应用程序
应用程序
操作系统
操作系统
虚拟硬件
虚拟硬件
操作系统
计算机硬件
应用程序
应用程序
容器
容器
...
操作系统
计算机硬件

一些例子

- 虚拟机：全系统虚拟化
  - NEMU, LiteNES ← 简单的模拟器
  - VirtualBox, QEMU, VMWare, ...
- 容器：操作系统实例虚拟化
  - chroot jails ← 操作系统的机制
  - Docker
  - Windows Subsystem for Linux

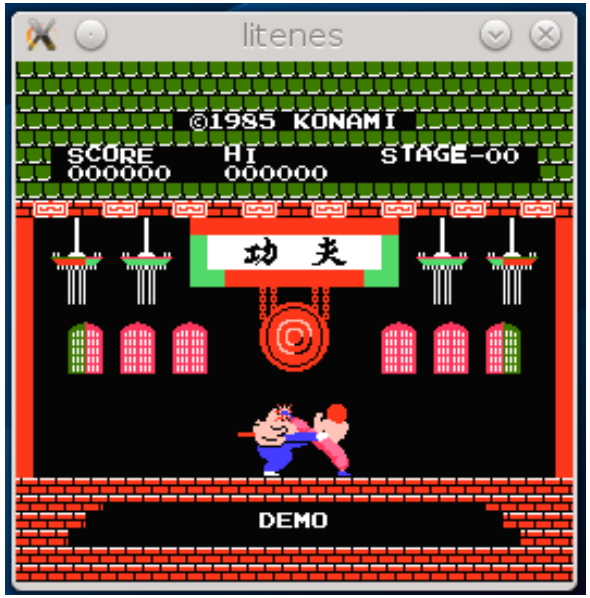
虚拟机：实现

虚拟机 (Virtual Machine)

- An efficient, isolated duplicate of a real computer machine

- 一个完全模拟出的计算机
  - 计算机系统 ≈ 逻辑门
  - 模拟时钟驱动下各个系统部件的状态
    - 处理器、定时器、显示控制器……
    - 行为是明确定义的

- So far, so good



模拟执行：性能

- 实际执行的代码有很多“热循环”

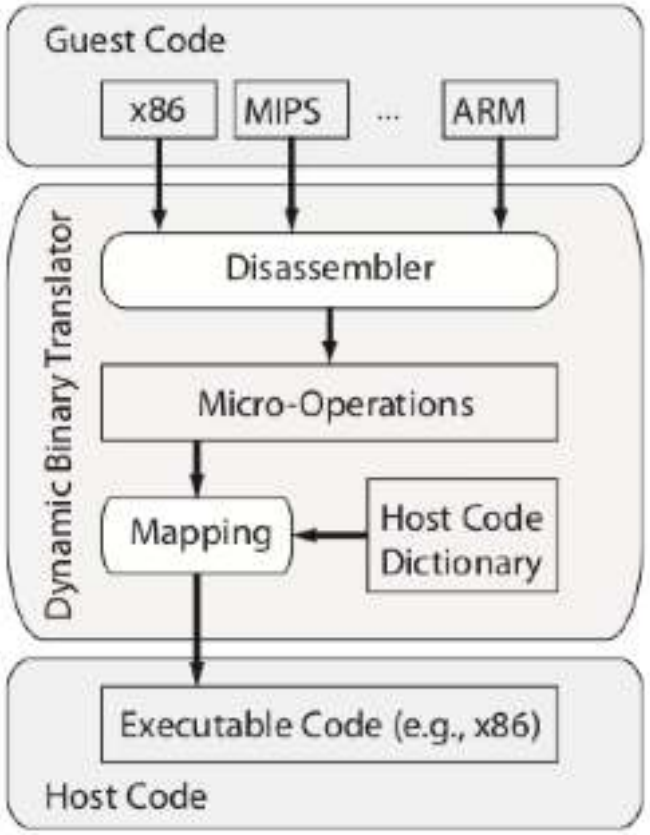
```
// there is a global sum
for (int i = 0; i < n; i++) {
  int volatile tmp = sum;
  tmp++;
  sum = tmp;
}
```

- 在NEMU中，每一条指令都经历了漫长的过程
  - 取指令、译码、执行(寄存器, 内存, I/O, ...)、异常、中断
  - 相比native, 数百倍的性能损失

更好的实现：Dynamic Binary Translation

- 每个指令都有行为定义
  - 参见x86手册

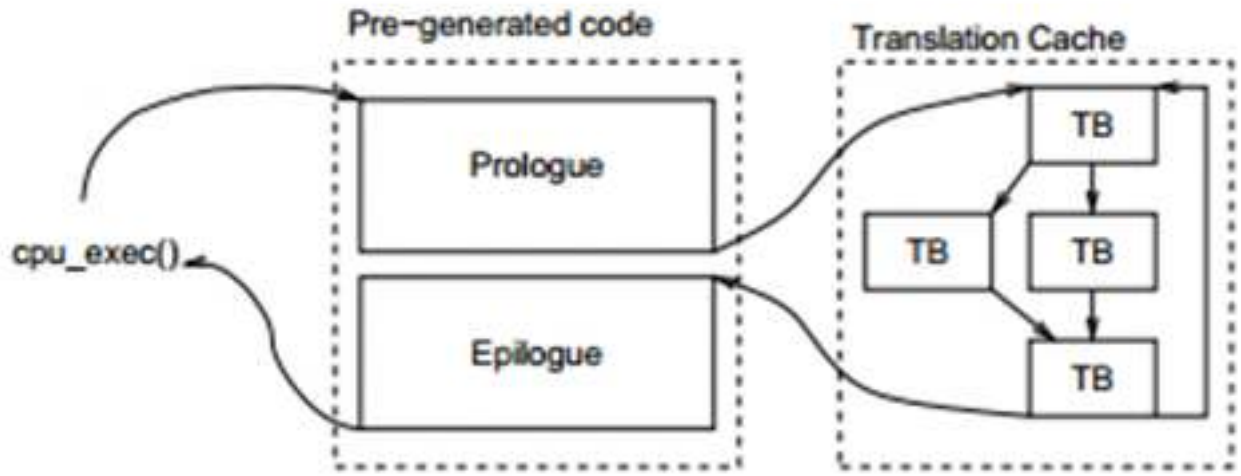
```
IF OperandSize = 16 THEN
  ESP ← ESP - 2;
  (SS:ESP) ← (SOURCE);
  (* word assignment *)
ELSE
  ESP ← ESP - 4;
  (SS:ESP) ← (SOURCE);
  (* dword assignment *)
FI
```



- 把指令序列翻译成中间代码
  - 再优化、编译成本地代码
  - 早期版本QEMU是编译成C代码(gcc优化)

QEMU：执行代码

- 耗时很长的循环不会调用任何prologue/epilogue
  - 所以QEMU性能非常出众



- 问题：如果代码恰好是个while (1); 怎么模拟中断?

QEMU：内存和I/O设备

- SoftMMU
  - 负责地址映射的解析
  - 并且有Guest虚拟地址到Host虚拟地址映射的TLB
  - 内存是个很大的性能瓶颈(IF、MEM都需要TLB lookup)

- I/O设备
  - 经过SoftMMU (由IOTLB检查是否是mmio)
  - I/O也是个很大的瓶颈

真正的问题：软/硬件协作提高虚拟机性能

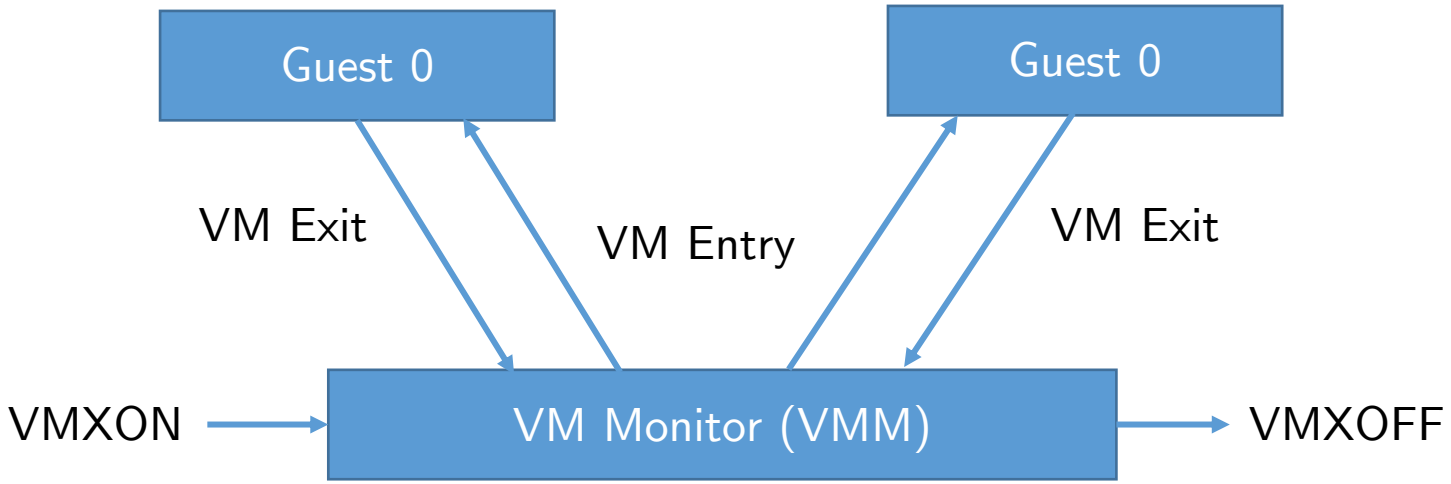
- DBT无法解决的问题
  - mov %eax, (%ebx) ← 内存可能是*任何东西*
  - mov %cr0, $CR0_FLAGS ← 行为复杂，必须翻译
  - int $0x80 ← 行为复杂，必须翻译

- 一旦翻译就有绕不过去的瓶颈
  - 内存、I/O……

- 解决办法：硬件提供一个专供虚拟机执行的模式

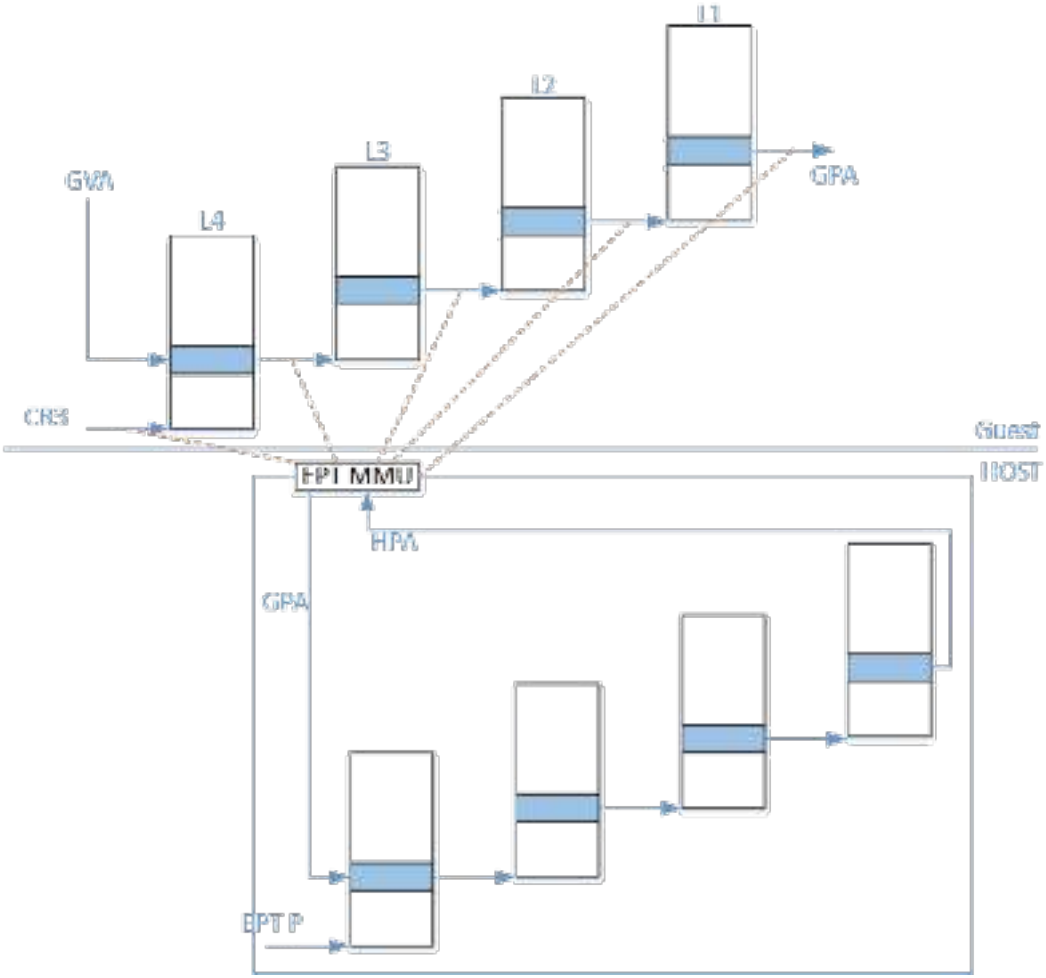
处理器：增加一个模式 (VMM)

- 处理器尽全力执行所有指令
  - 只有实在处理不了才触发VM Exit (类似于异常)，由VMM处理



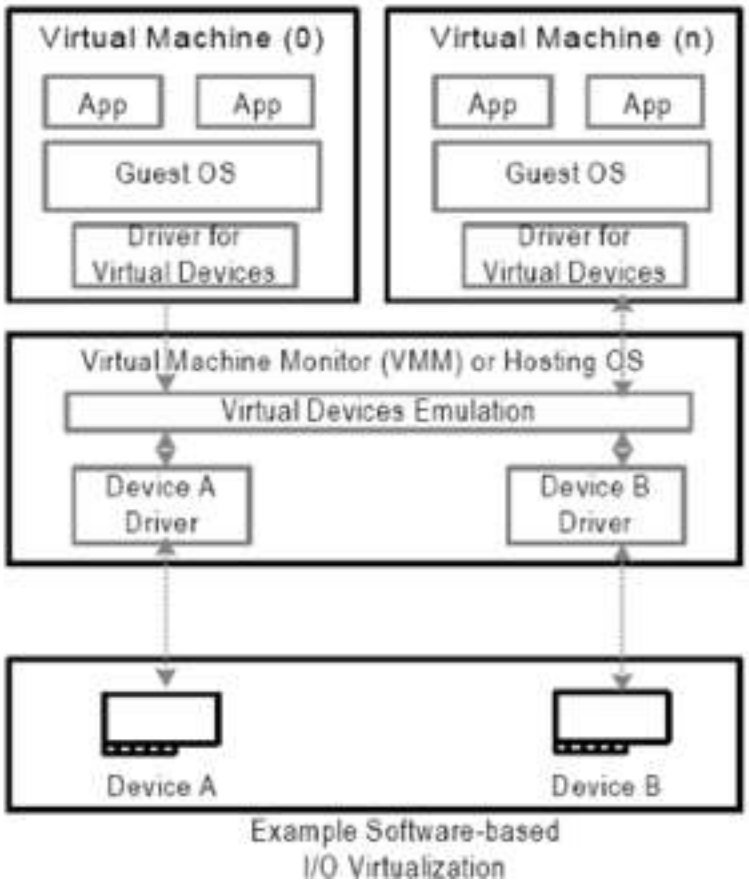
地址翻译：EPT (VT-x)

- 实现GPA到HPA的自动转换
  - 转换中EPT缺页产生VM-Exit
  - Guest CR3缺页不会VM-Exit
- 只有Guest访问没有分配的PA时才会VM-Exit

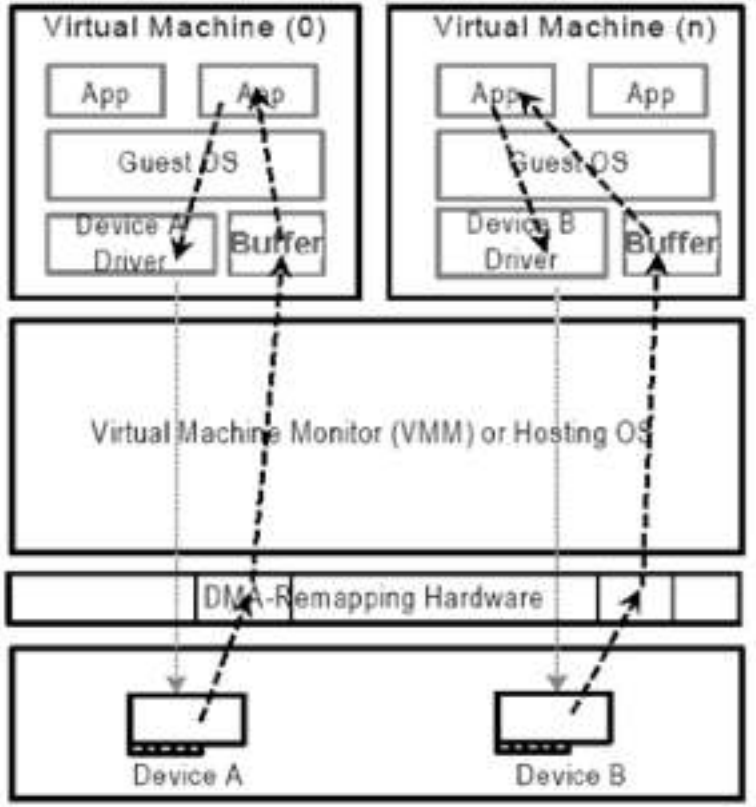


I/O：重映射 (VT-d)

- IOMMU允许中断/DMA到VM的直接映射



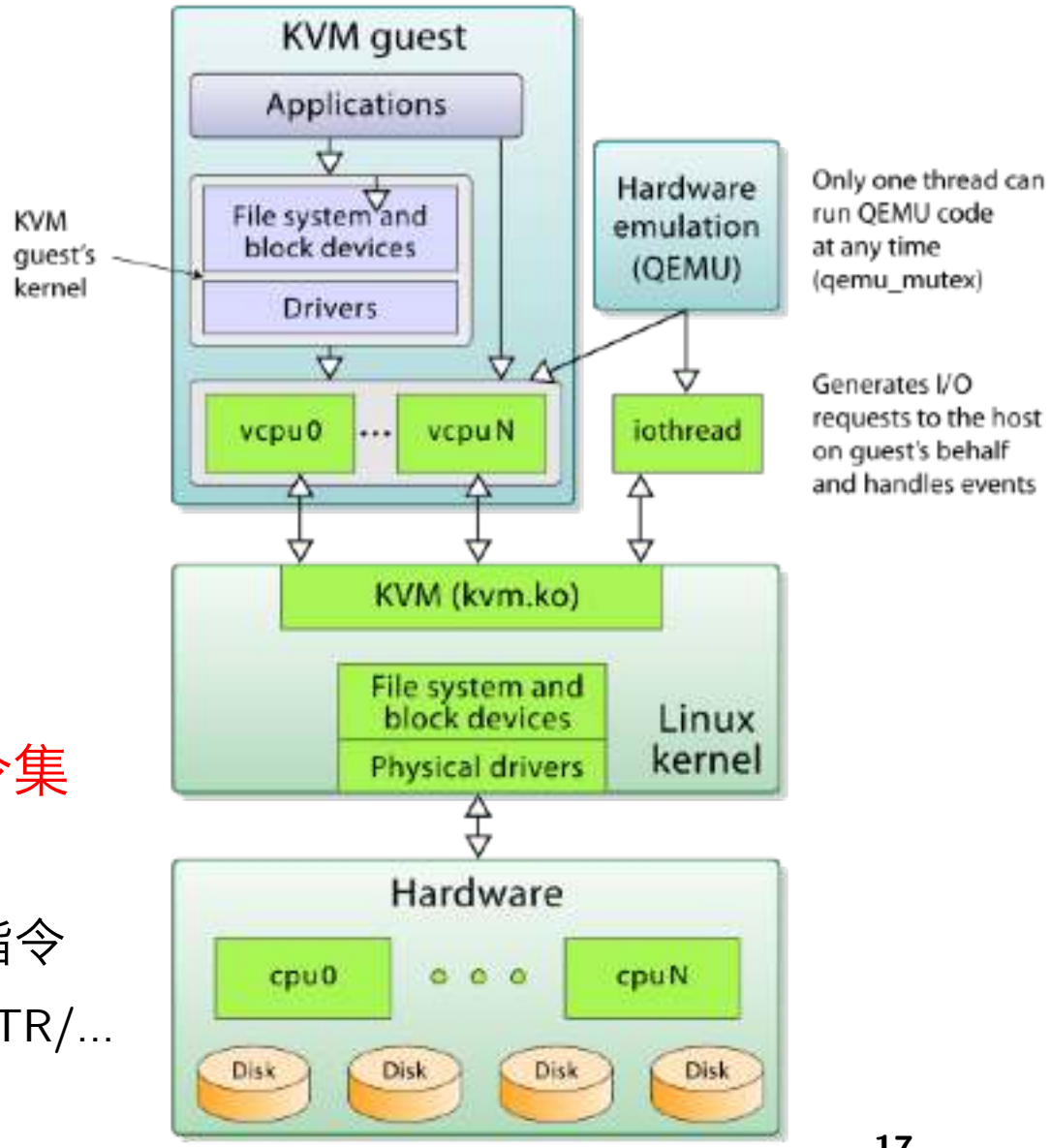
Example Software-based I/O Virtualization



Direct Assignment of I/O Devices

QEMU & KVM

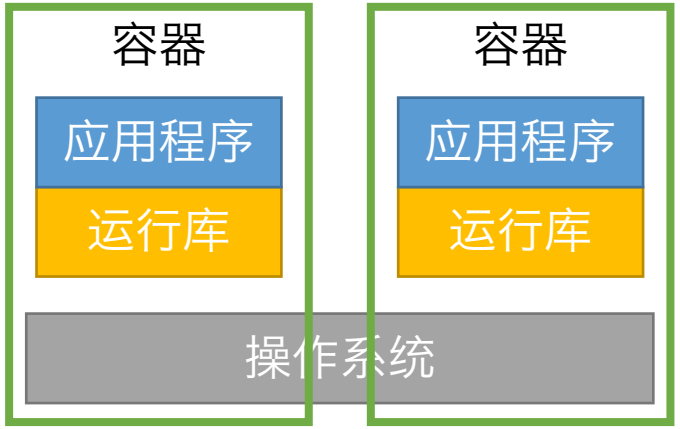
- Guest的执行
  - Guest在虚拟化模式执行
  - VMM能处理
    - HLT/PAUSE指令……
  - VMM不能处理
    - I/O；中断……
- KVM实现了一个虚拟指令集
  - 通过/dev/kvm访问
  - ioctl(fd, KVM_RUN)取指令
    - KVM_EXIT_MMIO/INTR/…



容器：实现

容器：虚拟操作系统实例

- 如果在同一个操作系统内创建两组进程
  - 互相“看不见”对方
  - 各自看到一份(不同的)文件系统和设备
- 也就实现了虚拟机的功能
  - 省去了“模拟指令集”里的各种麻烦事



应用程序眼中的操作系统

- 是一组API和API行为的规约

- 操作系统的一些具体约定
  - 系统启动执行“init程序”(systemd)
  - 文件系统的根是“/”
  - fork()会分配一个系统唯一的pid
  - bind()会绑定到系统的某个端口

- 把操作系统中的对象虚拟化即可

称不上容器的chroot

- chroot - change root directory (1980s)
  - 容器的“原始祖先”

```c
#include <unistd.h>
int chroot(const char *path);
```

- 把path作为当前进程文件系统的“根”
  - 在路径解析时，将“/”解析为path
  - 当已经位于“/”时将“..”解析为“.”

- 思考题：chroot如何实现？

称不上容器的chroot (cont'd)

- 打开了一个目录“/abc”
  - 这个目录被移动到文件系统的其他部分了
  - 然后`chdir ..`就走出了chroot的根

- chroot(2)
  - In particular, it is not intended to be used for any kind of security purpose, neither to fully sandbox a process nor to restrict filesystem system calls

容器：操作系统内核支持

- namespaces (man 7 namespaces)
  - 虚拟化内核中的对象
    - IPC – 进程间通信对象
    - Network – 网络设备、协议栈、端口……
    - Mount – 挂载点
    - PID – 进程号
    - User – 用户和组
    - UTS – Hostname、域名
  - 我们写OSLab时，这些资源(比如pid)是操作系统全局共享的
    - namespace相当于为这些资源做“命名”

容器：操作系统内核支持 (cont'd)

- cgroups (control groups) 控制组
  - 一组进程，并且资源受到控制
    - 内存不能超过指定限制
    - CPU和I/O的占有率控制
    - 按组统计资源使用

- namesapces + cgroups + chroot ≈ 虚拟操作系统
  - 创建一个虚拟操作系统
    - 创建namespace
    - 用cgroup管理其中进程

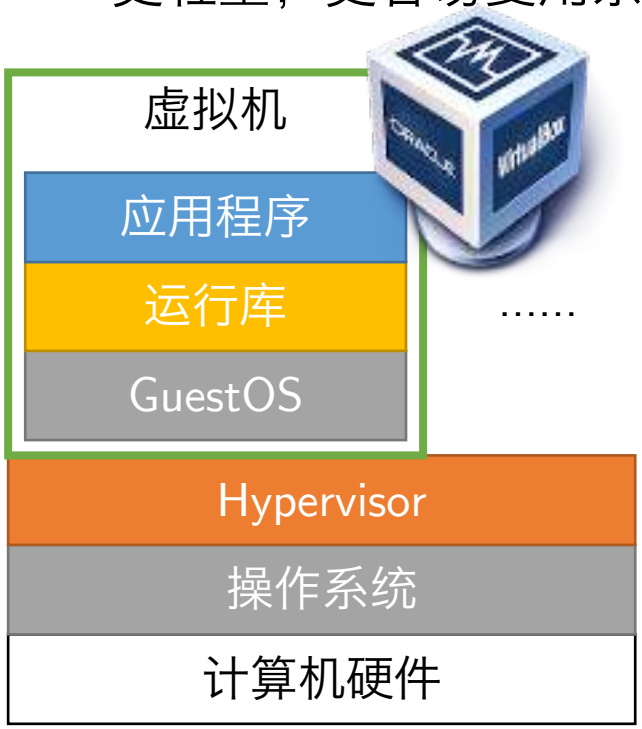
LXC：非常简洁的API (类似于fork/execve)

```
struct lxc_container {
  // 管理
  bool (*create)(struct lxc_container *c, char *t, ...);
  bool (*start)(c, int useinit, char *argv[]);
  bool (*stop)(c);
  bool (*reboot)(c);
  bool (*shutdown)(c, int timeout);
  // 在容器运行
  int (*attach)(c, lxc_attach_exec_t exec_function, ...);
  // 设备管理
  bool (*add_device_node)(c, char *src_path, char *dest_path);
  bool (*attach_interface)(c, char *dev, char *dst_dev);
  // 快照
  int (*snapshot)(c, char *commentfile);
  bool (*checkpoint)(c, char *directory, bool stop, bool verbose);
  ...
}
```

系统虚拟化

虚拟机 vs 容器

- 更多隔离 ← 虚拟机
  - 更安全；兼容性更好
- 更多共享 ← 容器
  - 更轻量；更容易复用系统内资源



系统虚拟化：云计算的技术支撑