IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

课程回顾

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

操作系统是在硬件和应用之间的软件层

操作系统和应用：

- 应用功能越来越多
- 操作系统沉淀越来越多功能、内涵与外延不断扩大

操作系统和硬件：

- 身体 vs. 灵魂

"操作系统是管理硬件资源、控制程序运行、改善人机界面和为应用软件提供支持的一种系统软件。"

[计算机百科全书(第2版)]

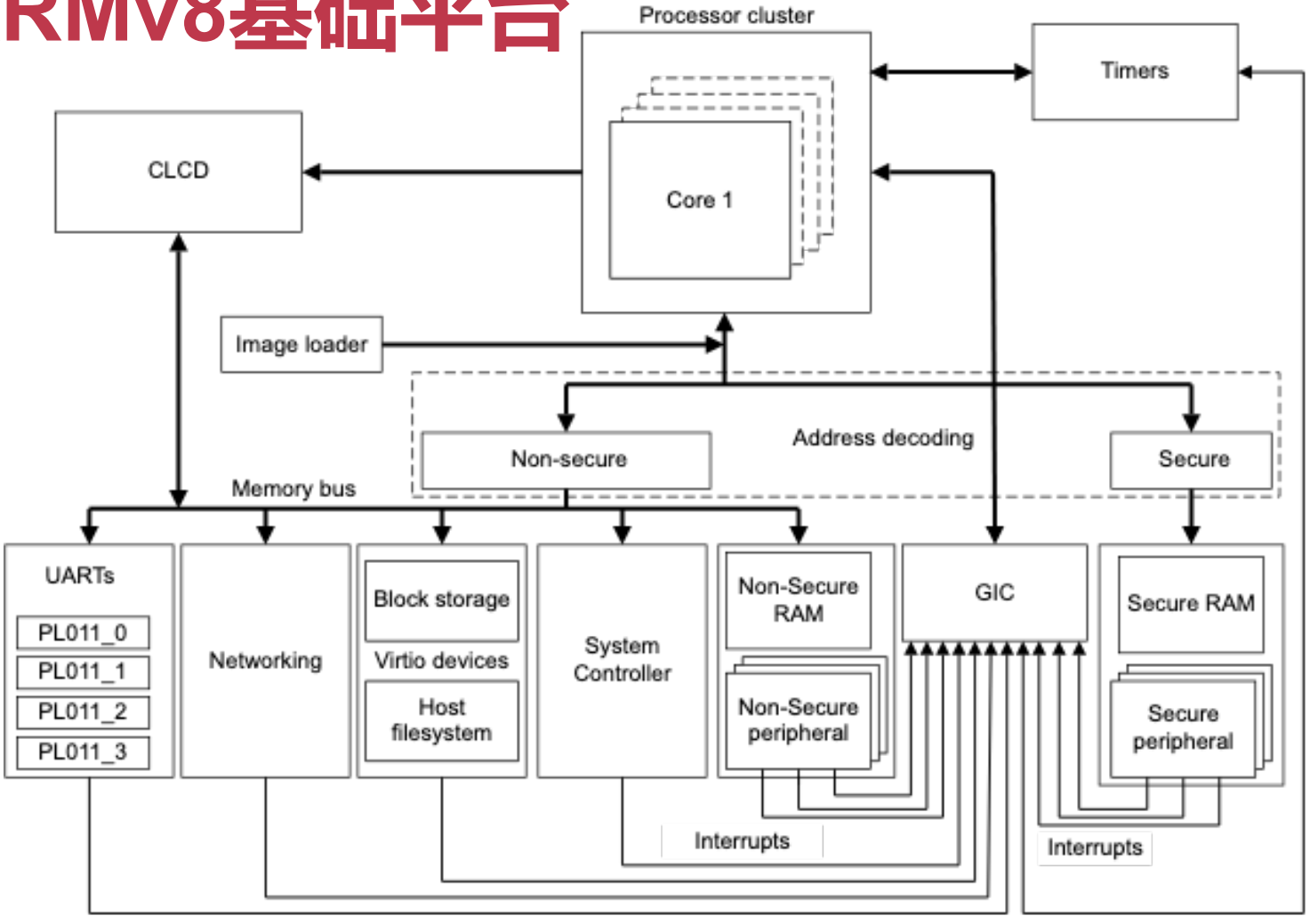
操作系统=管理+服务

- **管理和服务的目标有可能存在冲突**
  - 服务的目标：单个应用的运行效率最大化
  - 管理的目标：系统的资源整体利用率最大化
  - 例：单纯强调公平性的调度策略往往资源利用率低
    - 如细粒度的round-robin导致大量的上下文切换

操作系统的定义

- **操作系统的核心功能：**
  - 将有限的、离散的资源，高效地抽象为无限的、连续的资源
- **从软件角度的定义：**
  - 硬件资源虚拟化+管理功能可编程
- **从结构角度的定义：**
  - 操作系统内核+系统框架

ARMv8基础平台


Processor cluster
Timers
CLCD
Core 1
Image loader
Address decoding
Non-secure
Secure
Memory bus
UARTs
PL011_0
PL011_1
PL011_2
PL011_3
Networking
Block storage
Virtio devices
Host filesystem
System Controller
Non-Secure RAM
Non-Secure peripheral
GIC
Secure RAM
Secure peripheral
Interrupts
Interrupts

RISC vs CISC

	RISC (AArch64)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	增加通用寄存器数量	微码
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

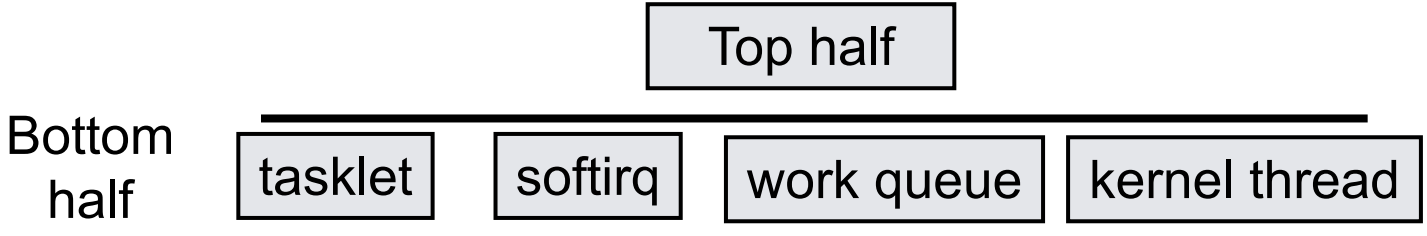
ChCore Bootloader

- **Bootloader和kernel放在同一个ELF文件中**
  - Bootloader位于.init段，并通过链接器设置入口
  - Kernel位于.text段
- **主CPU启动，其他次CPU等待**

```
BEGIN_FUNC(_start)
  mrs x8, mpidr_el1
  mov x9, #0xc1000000
  bic x8, x8, x9
  cbz x8, primary
```

Linux的中断处理理念

- **在中断处理中做尽量少的事**
- **推迟非关键行为**
- **结构：Top half & Bottom half**
  - Top half : 做最少的工作后返回
  - Bottom half ：推迟处理 (softirq, tasklets, 工作队列，内核线程)



操作系统复杂性与结构

- **操作系统中的"瓦萨号"**
  - 1991-1995年，IBM投入20亿美元打造Workspace操作系统
  - 目标过于宏伟，系统过于复杂，导致项目失败
  - 间接导致IBM全力投入扶植Linux操作系统
- **复杂系统的构建必须考虑其内部结构**
  - 不同目标之间往往存在冲突
  - 不同需求之间需要进行权衡

操作系统的不同目标

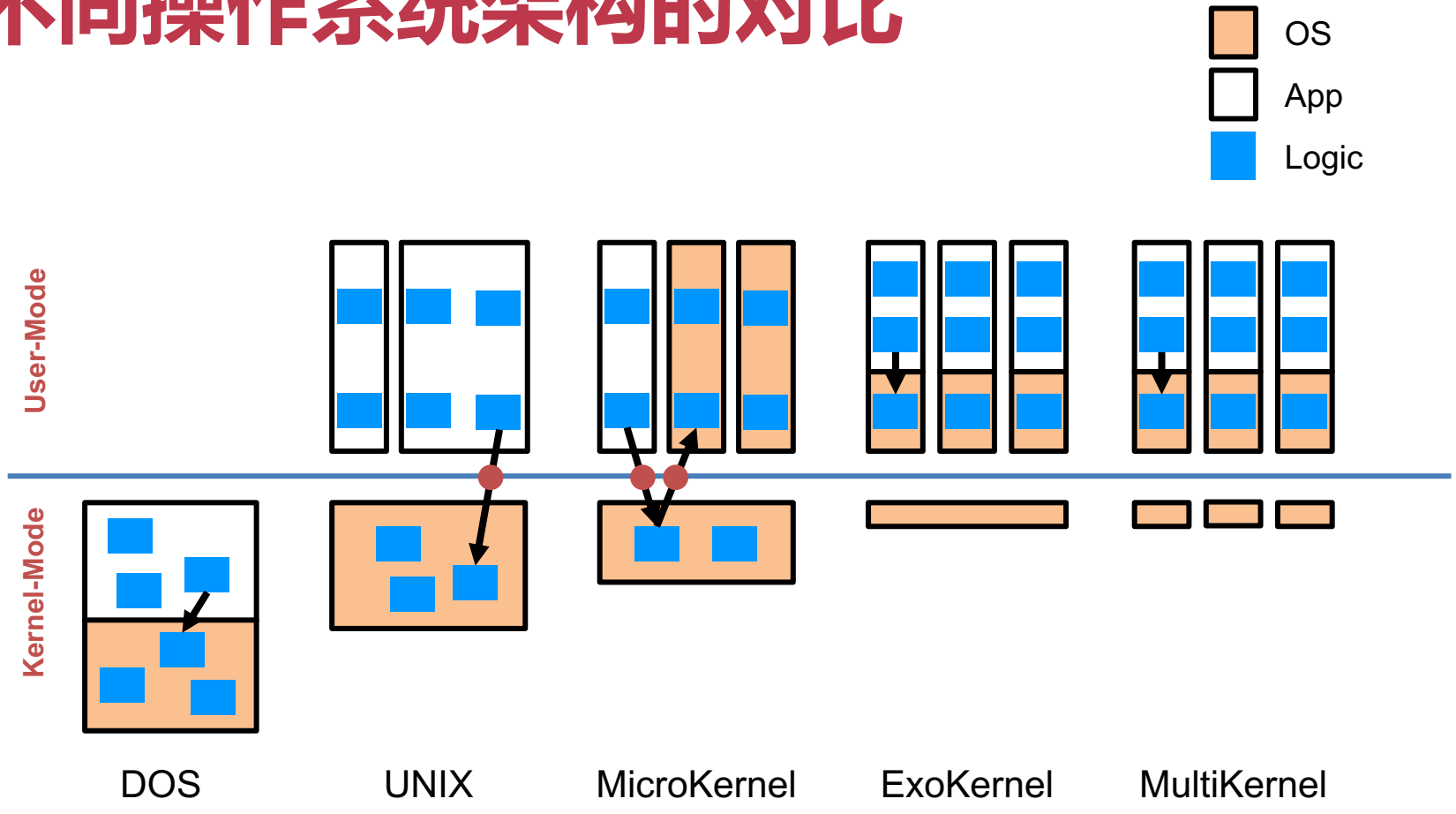
- **用户目标**
  - 方便使用
  - 容易学习
  - 功能齐全
  - 安全
  - 流畅
  - ……
- **系统目标**
  - 容易设计、实现
  - 容易维护
  - 灵活性
  - 可靠性
  - 高效性
  - ……

降低操作系统复杂性

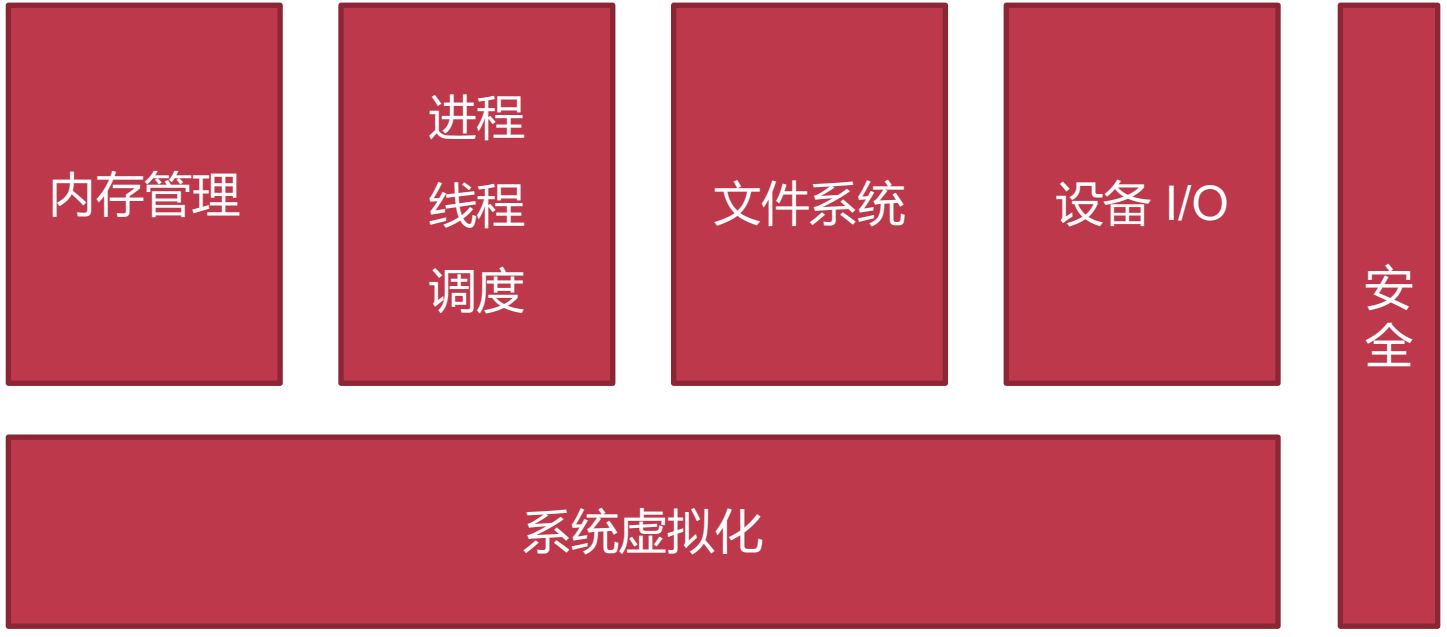
- **重要设计原则：策略与机制的分离**
  - 策略（Policy）：要做什么 —— 相对动态
  - 机制（Mechanism）：怎么做 —— 相对静态
  - 操作系统可仅通过调整策略来适应不同应用的需求

例子	策略	机制
登录	什么用户、以什么权限登录	输入处理、策略文件管理、桌面启动加载 ...
调度	调度算法：Round-robin、Earliest Deadline First ...	调度队列、调度实体（如线程）的表示、调度中断处理 ...

不同操作系统架构的对比

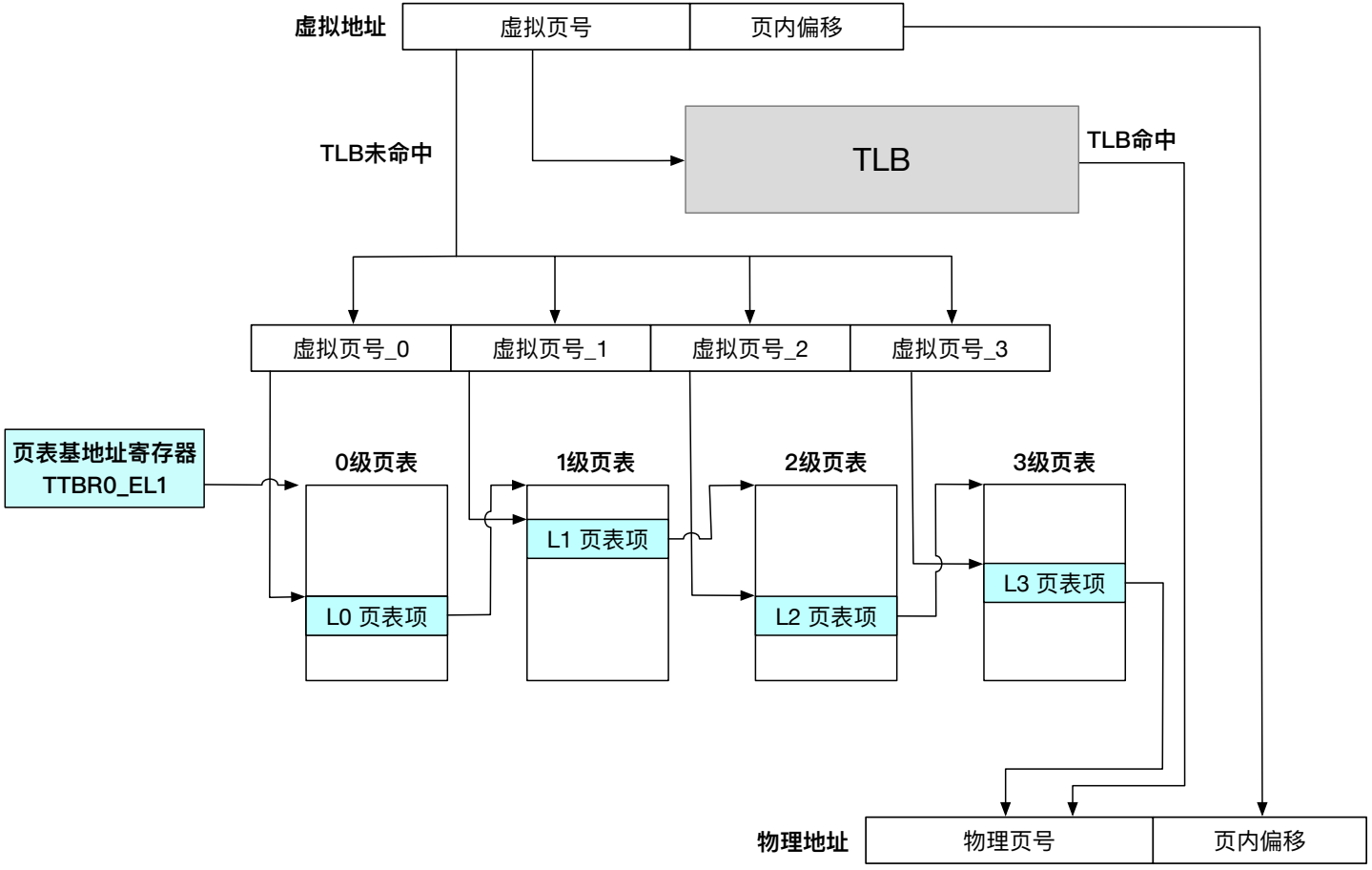


操作系统模块



内存管理

AARCH64的4级页表



TLB: Translation Lookaside Buffer

- **TLB 位于CPU内部**
  - 缓存了虚拟页号到物理页号的映射关系
  - **有限数目**的TLB缓存项

- **在地址翻译过程中，MMU首先查询TLB**
  - TLB命中，则不再查询页表（**fast path**）
  - TLB未命中，再查询页表

TLB刷新（TLB Flush）

- **TLB 使用虚拟地址索引**
  - 切换页表时需要全部刷新
- **AARCH64上内核和应用程序使用不同的页表**
  - 分别存在TTBR0_EL1和TTBR1_EL1
  - 系统调用过程不用切换
- **x86_64上只有唯一的基地址寄存器**
  - 内核映射到应用页表的高地址
  - 避免系统调用时TLB刷新的开销

- **刷TLB相关指令**
  - 清空全部
    - TLBI VMALLEL1IS
  - 清空指定ASID相关
    - TLBI ASIDE1IS
  - 清空指定虚拟地址
    - TLBI VAE1IS

如何降低TLB刷新的开销

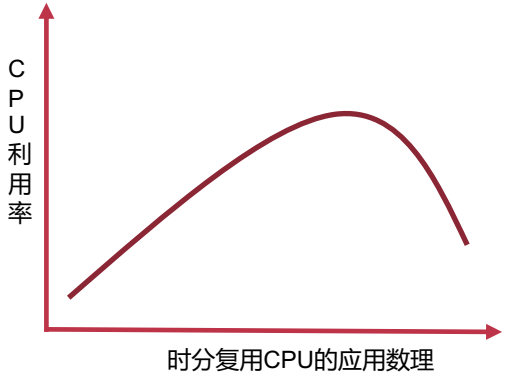
- **为不同的页表打上标签**
  - TLB缓存项都具有页表标签，切换页表不再需要刷新TLB
- **x86_64：PCID（Process Context ID）**
  - PCID存储在CR3的低位中
  - 在KPTI使用后变得尤为重要
    - Kernel Page Table Isolation
    - 即内核与应用不共享页表，防御Meltdown攻击
- **AARCH64：ASID（Address Space ID）**
  - OS为不同进程分配8/16 ASID，将ASID填写在TTBR0_EL1的高8/16位
  - ASID位数由TCR_EL1的第36位（AS位）决定

TLB与多核

- **使用了ASID之后**
  - 切换页表不再需要刷新TLB
  - 修改页表映射后，仍需刷新TLB
- **在多核场景下**
  - 需要刷新其它核的TLB吗？
  - 如何知道需要刷新哪些核？
  - 怎么刷新其它核

Thrashing Problem

- **直接原因**
  - 过于频繁的缺页异常（物理内存总需求过大）
- **大部分 CPU 时间都被用来处理缺页异常**
  - 等待缓慢的磁盘 I/O 操作
  - 仅剩小部分的时间用于执行真正有意义的工作
- **调度器造成问题加剧**
  - 等待磁盘 I/O导致CPU利用率下降
  - 调度器载入更多的进程以期提高CPU利用率
  - 触发更多的缺页异常、进一步降低CPU利用率、导致连锁反应



工作集模型（Working Set Model）

- **一个进程在时间t的工作集W(t, x) (Peter Denning)：**
  - 其在时间段（t - x, t)内使用的内存页集合
  - 也被视为其在未来（下一个x时间内）会访问的页集合
  - 如果希望进程能够顺利进展，则需要讲该集合保持在内存中

- **工作集模型：**
  - all-or-nothing模型
  - 进程工作集或者都在内存中，或者全都换出
  - 避免thrashing，提高系统整体性能表现

跟踪工作集w(t, x)

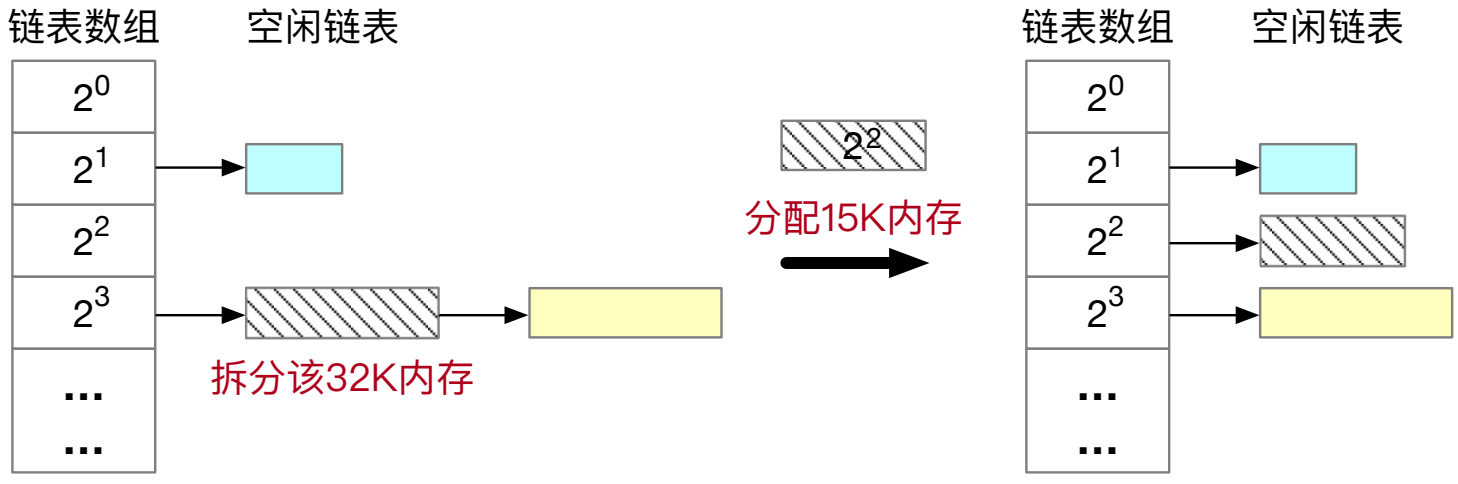
- **工作集时钟中断固定间隔发生，处理函数扫描内存页**
  - 访问位为1
    - 则说明在此次tick中被访问，
      记录上次使用时间为当前时间
  - 访问位为0（此次tick中未访问）
    - Age = 当前时间 – 上次使用时间
    - 若Age大于设置的x，则不在工作集
  - 将所有访问位清0
    - 注意：访问位（access bit）需要硬件支持

当前时间：2020

上次使用时间	访问位
2010	1
2000	1
1970	0
1990	0

伙伴系统例子

- 分配合适大小的块：什么是“合适”？

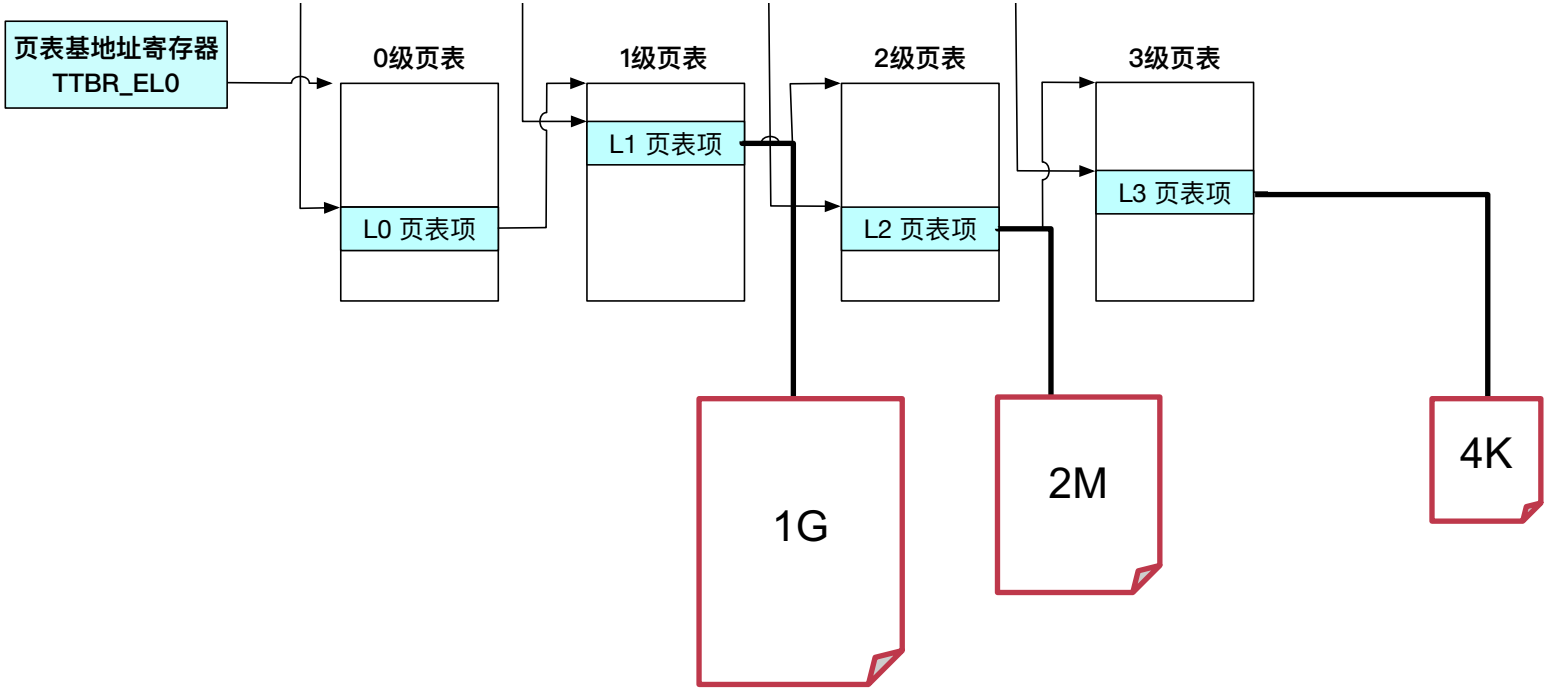


- **思考：分裂和合并都是级联操作，什么时候会级联？**

伙伴系统的巧妙之处

- **高效地找到伙伴块**
  - 互为伙伴的两个块的物理地址**仅有一位**不同
  - 一个是0，另一个是1
  - 块的**大小决定**是哪一位

大页


页表基地址寄存器
TTBR_EL0
0级页表
L0 页表项
1级页表
L1 页表项
2级页表
L2 页表项
3级页表
L3 页表项
1G
2M
4K

大页的利弊

- **好处**
  - 减少TLB缓存项的使用，提高 TLB 命中率
  - 减少页表的级数，提升遍历页表的效率
- **案例**
  - 提供API允许应用程序进行显示的大页分配
  - 透明大页（Transparent Huge Pages) 机制
- **弊端**
  - 未使用整个大页而造成物理内存资源浪费
  - 增加管理内存的复杂度

进程、线程与调度

进程创建：fork()

- **语义：为调用进程创建一个一模一样的新进程**
  - 调用进程为**父进程**，新进程为**子进程**
  - 接口简单，无需任何参数

- **fork后的两个进程均为独立进程**
  - 拥有不同的进程id
  - 可以并行执行，互不干扰（除非使用特定的接口）
  - 父进程和子进程会共享部分数据结构（内存、文件等）

线程：更加轻量级的运行时抽象

- **线程只包含运行时的状态**
  - 静态部分由**进程**提供
  - 包括了执行所需的**最小**状态（主要是寄存器和栈）

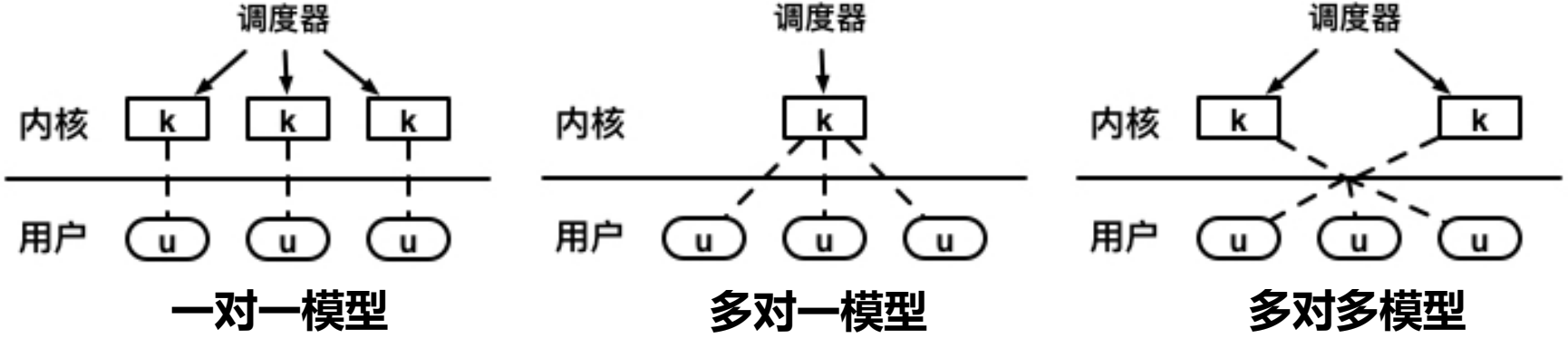
- **一个进程可以包含多个线程**
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行

用户态线程与内核态线程

- **根据线程是否受内核管理，可以将线程分为两类**
  - 内核态线程：内核可见，受内核管理
  - 用户态线程：内核不可见，不受内核直接管理
- **内核态线程**
  - 由内核创建，线程相关信息存放在内核中
- **用户态线程（纤程）**
  - 在应用态创建，线程相关信息主要存放在应用数据中

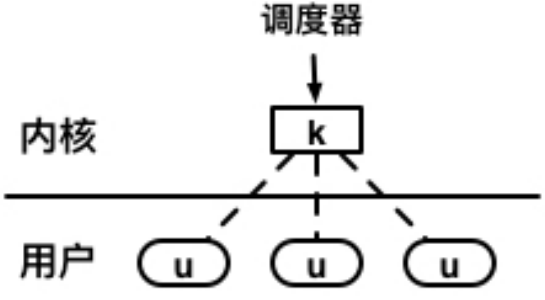
线程模型

- **线程模型表示了用户态线程与内核态线程之间的联系**
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程



多对一模型

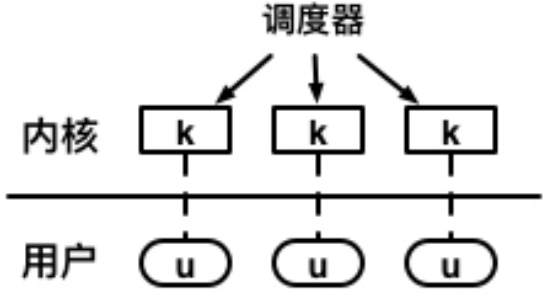
- **将多个用户态线程映射给单一的内核线程**
  - 优点：内核管理简单
  - 缺点：可扩展性差，无法适应多核机器的发展
- **在主流操作系统中被弃用**
- **用于各种用户态线程库中**



一对一模型

- **每个用户线程映射单独的内核线程**
  - 优点：解决了多对一模型中的可扩展性问题
  - 缺点：内核线程数量大，开销大

- **主流操作系统都采用一对一模型**
  - Windows、Linux、OS X……



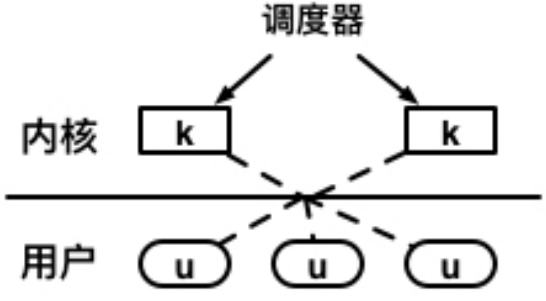
一对一线程模型的局限

- **复杂应用：对调度存在更多需求**
  - 生产者消费者模型：生产者完成后，消费者最好马上被调度
  - 内核调度器的信息不足，无法完成及时调度

- **“短命”线程：执行时间亚毫秒级（如处理web请求）**
  - 内核线程初始化时间较长，造成执行开销
  - 线程上下文切换频繁，开销较大

多对多模型（又叫Scheduler Activation）

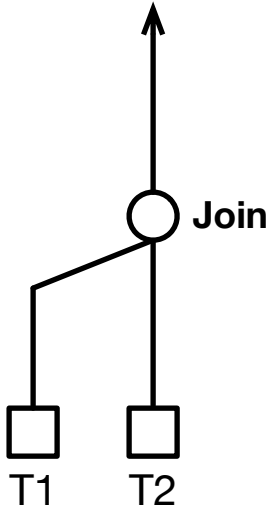
- **N个用户态线程映射到M个内核态线程（N > M）**
  - 优点：解决了可扩展性问题（多对一）和线程过多问题（一对一）
  - 缺点：管理更为复杂
  - Solaris在9之前使用该模型
    - 9之后改为一对一
- **在虚拟化中得到了广泛应用**
  - 内核：多个VCPU；用户：多个thread



线程的基本操作：以*pthreads*为例

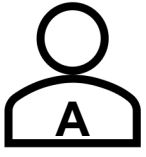
- **创建：pthread_create**
  - 内核态：创建相应的内核态线程及内核栈
  - 应用态：创建TCB、应用栈和TLS

- **合并：pthread_join**
  - 等待另一线程执行完成，并获取其执行结果
  - 可以认为是fork的“逆向操作”

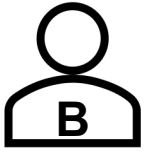


Round Robin (时间片轮转)

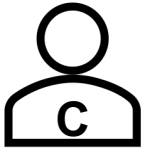
公平起见
每人轮流一分钟！



感觉多等了好久…

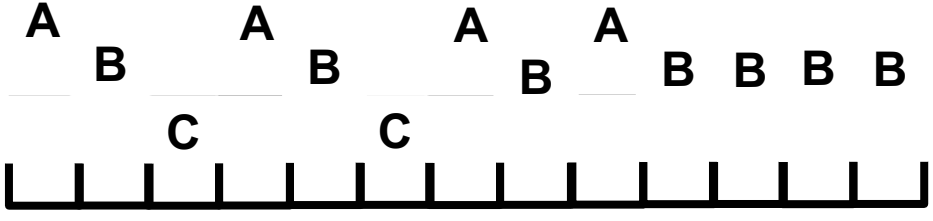


学霸的响应时间短了好多



学霸的响应得更快了

问题	到达时间	解答时间（工作量）
A	0	4
B	1	7
C	2	2



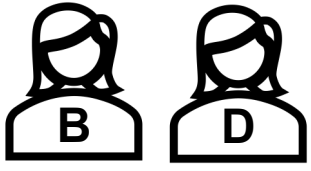
轮询：公平、平均响应时间短

问题：牺牲周转时间

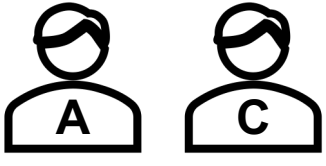
Multi-level Queue

先回答B,D
然后再回答A,C

优先级0（高）



优先级1（低）



多级队列：

1）维护多个优先级队列

2）高优先级的任务优先执行

3）同优先级内使用Round Robin调度

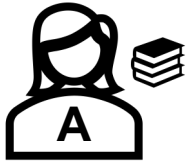
问题：优先级反转

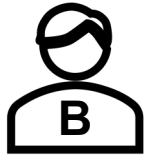
- **高、低优先级任务都需要独占共享资源**
  - 共享资源
    - 存储
    - 硬件
    - "秘籍"
    - …
  - 通常使用信号量、互斥锁实现独占
- **低优先任务占用资源 -> 高优先级任务被阻塞**

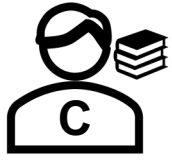
问题：优先级反转

优先级：A>B>C

问题：
A被C占有的资源阻塞
优先级较低的B先于A学习







1.申请秘籍成功

2. 抢占C
申请秘籍失败
等待

3.B优先级高于C
可以向学霸学习

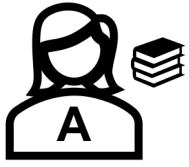
解决方法：优先级继承

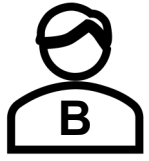
优先级：A>B>C

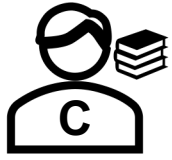
解决方案：

A暂时将优先级转移给C

让C尽快归还秘籍







1.申请秘籍成功

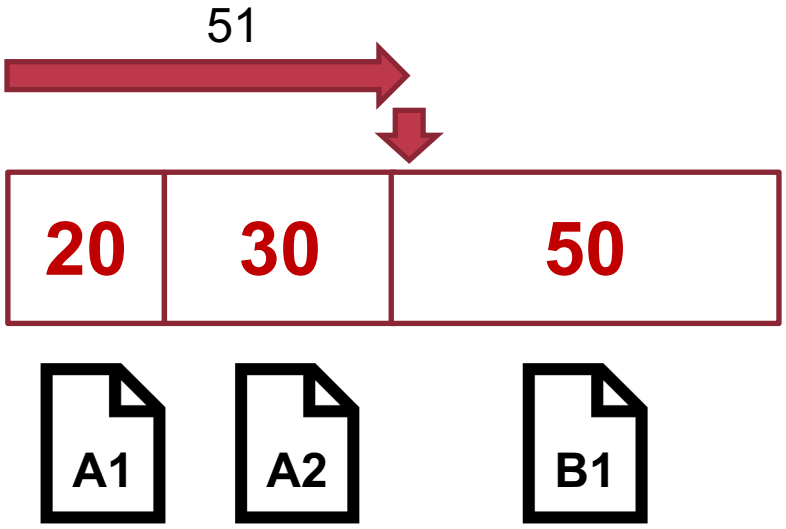
2. 抢占C
转移优先级给C

3.归还秘籍
返回优先级

4. 申请秘籍成功
继续学习

一种公平共享的实现：Lottery Scheduling

- 每次调度时，生成随机数$R \in [0, T)$
- 根据R，找到对应的任务
  - R=51 → 调度B1




```
R = random(0, T)
sum = 0
foreach(task in task_list) {
    sum += task.ticket
    if (R < sum) {
        break
    }
}
schedule()
```


Lottery Scheduling: Flexible Proportional-Share Resource Management. https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf

IPC：进程间通信

共享内存

- **系统内核为两个进程映射共同的内存区域**
  - 快递员和小明的快递桌

- **挑战: 做好同步**
  - 发送者不能覆盖掉未读取的数据 (新快递把旧的快递挤下桌)
  - 接收者不能读取没有准备好的数据 (小明拿错了快递)

共享内存的问题

- **轮询导致资源浪费**
  - 小明时不时就得下楼检查一下快递桌子
  - 快递员需要等待桌子有空闲空间
  - 一天大部分时间都花在了上下楼和检查快递上了

- **固定一个检查时间，时延长**
  - 小明每天晚上检查一下有没有新的快递过来
  - 早上到达的快递要晚上才能拿到

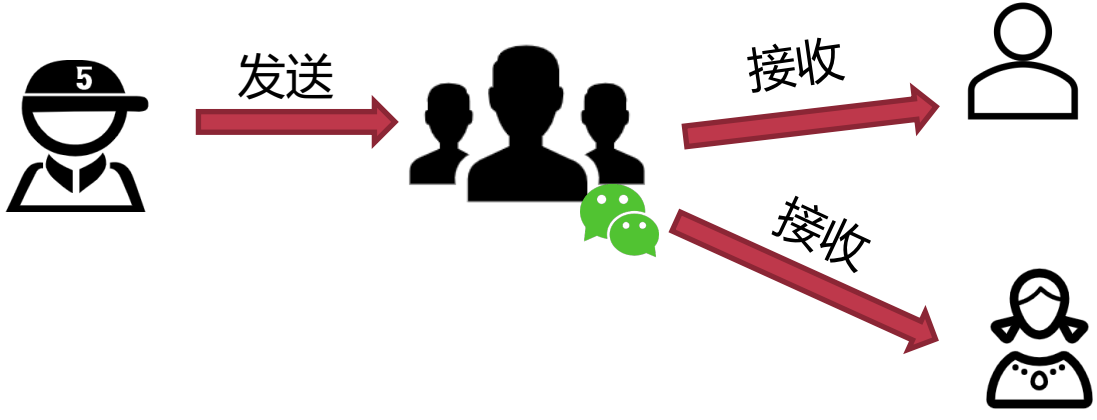
消息传递

- **基本操作:**
  - 发送消息*Send(message)*
  - 接收消息*Recv(message)*

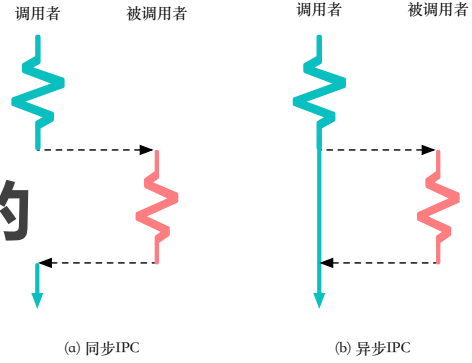
- **如果两个进程*P*和*Q*希望通过消息传递进行通信，需要:**
  - 建立一个通信连接
  - 通过Send/Recv接口进行消息传递

间接通信: 用聊天群发布快递信息

- **消息的发送和接收需要经过一个“信箱”**
  - 聊天群 (所有在群内的人都可以接收消息)
  - 每个“信箱”有自己唯一的标识符 (这里的群号)
  - 发送者往 “信箱”发送消息，接收者从“信箱”读取消息



消息传递的同步与异步



- **消息的传递可以是阻塞的，也可以是非阻塞的**
- **阻塞通常被认为是同步通信**
  - 阻塞的发送/接收: 发送者/接收者一直处于阻塞状态，直到消息发出/到来
  - 同步通信通常有着更好的时延和易用的编程模型 (不会被投诉)
- **非阻塞通常被认为是异步通信**
  - 发送者/接收者不等待操作结果，直接返回
  - 异步通信的带宽一般更高 (快递员可以送更多的快递)

Unix 管道

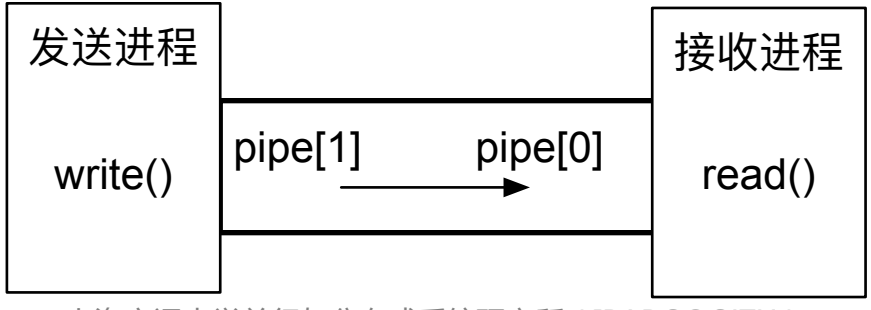
- **管道是Unix等宏内核系统中非常重要的进程间通信机制**
- **管道(Pipe): 两个进程间的一根通信通道**
  - 一端向里投递，另一端接收
  - 管道是间接消息传递方式，通过共享一个管道来建立连接
- **例子: 我们常见的命令 ls | grep**

```
➜  os-textbook git:(master) ls | grep ipc
ipc.tex
```

Unix 管道

```
int fd[2];
pipe(fd);
fd[0]; // read side
fd[1]; // write side
```

- **管道的特点:**
  - 单向通信，当缓冲区满时阻塞
  - 一个管道有且只能有两个端口: 一个负责输入 (发送数据)，一个负责输出 (接收数据)
  - 数据不带类型，即字节流
  - 基于Unix的文件描述符使用



扩展: Sleep/Wakeup通信机制

- Xv6管道实现中依赖于sleep和wakeup两个接口
- Xv6中的sleep和wakeup是经典的进程间等待(wait)和通知(notify)的机制
- 信道(Channel)是等待和通知的媒介
- 一个进程可以通过sleep接口将自己等待在一个信道上
- 另外一个进程可以通过wakeup将等待在某个信道上的进程唤醒

多核与同步

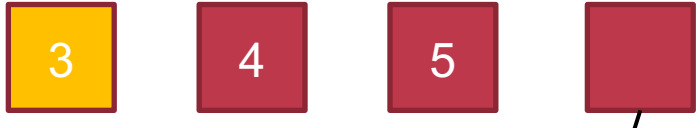
排号锁（Ticket Lock）

通过遵循竞争者到达的**顺序**来传递锁。

owner：表示当前在吃的食客　　next：表示目前放号的最新值

海底捞

假设只有一桌...



2.等待叫号
while(owner != my_ticket);

owner = 3
next = 6

1. 拿号 => 6号
my_ticket = atmoic_FAA(&next, 1)

排号锁（Ticket Lock）

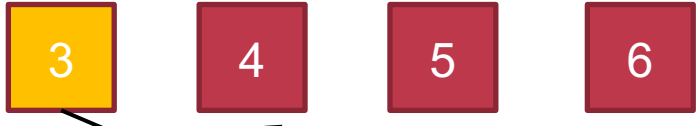
通过遵循竞争者到达的**顺序**来传递锁。

owner：表示当前在吃的食客　　next：表示目前放号的最新值

```
while(owner != my_ticket);
```

海底捞

假设只有一桌...



2. 叫下个人进来

```
owner += 1
```

1. 吃完了，买单

```
owner = 3
next = 7
```

排号锁（Ticket Lock）

通过遵循竞争者到达的**顺序**来传递锁。

owner：表示当前的持有者 next：表示目前放号的最新值

lock操作

```
1. my_ticket = atomic_FAA(
       &lock->next, 1);
2. while(lock->owner !=
       my_ticket)
       /* waiting */;
```

拿号

等号

unlock操作

```
1. lock->owner ++;
```

叫号

读写锁的使用示例

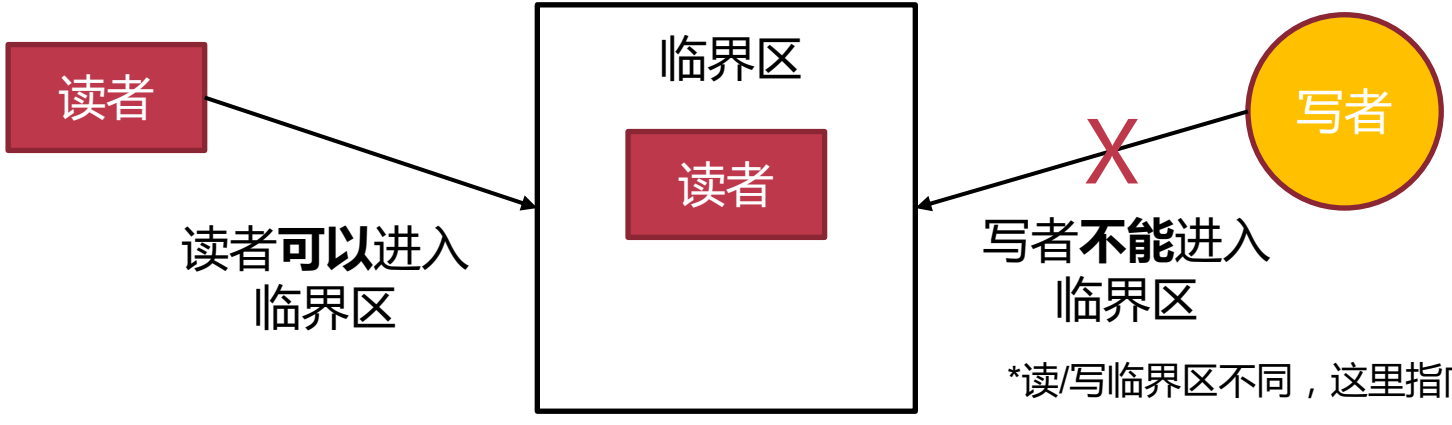
```
struct rwlock *lock;
char data[SIZE];

void reader(void) {
        lock_reader(lock);
        read_data(data)
        unlock_reader(lock);
}

void writer(void) {
        lock_writer(lock);
        update_data(data);
        unlock_writer(lock);
}
```

读写锁

互斥锁：所有的进程均互斥，同一时刻**只能有一个进程**进入临界区

对于部分只读取共享数据的进程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥



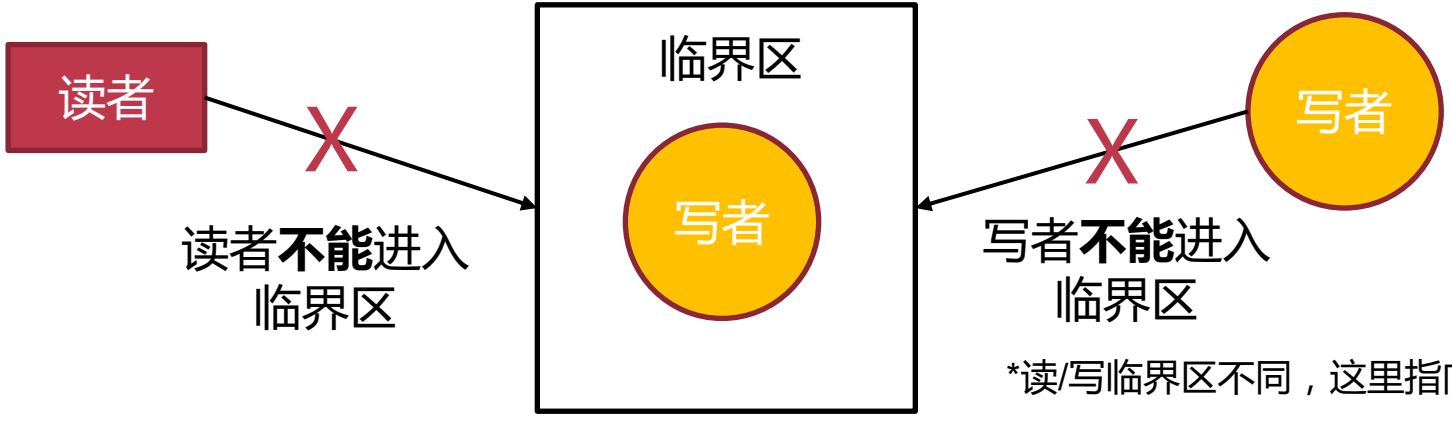
*读/写临界区不同，这里指向一个为示意

读写锁

互斥锁：所有的进程均互斥，同一时刻只能有一个进程进入临界区

对于部分只读取共享数据的进程过于严厉

读写锁：区分读者与写者，允许读者之间并行，读者与写者之间互斥

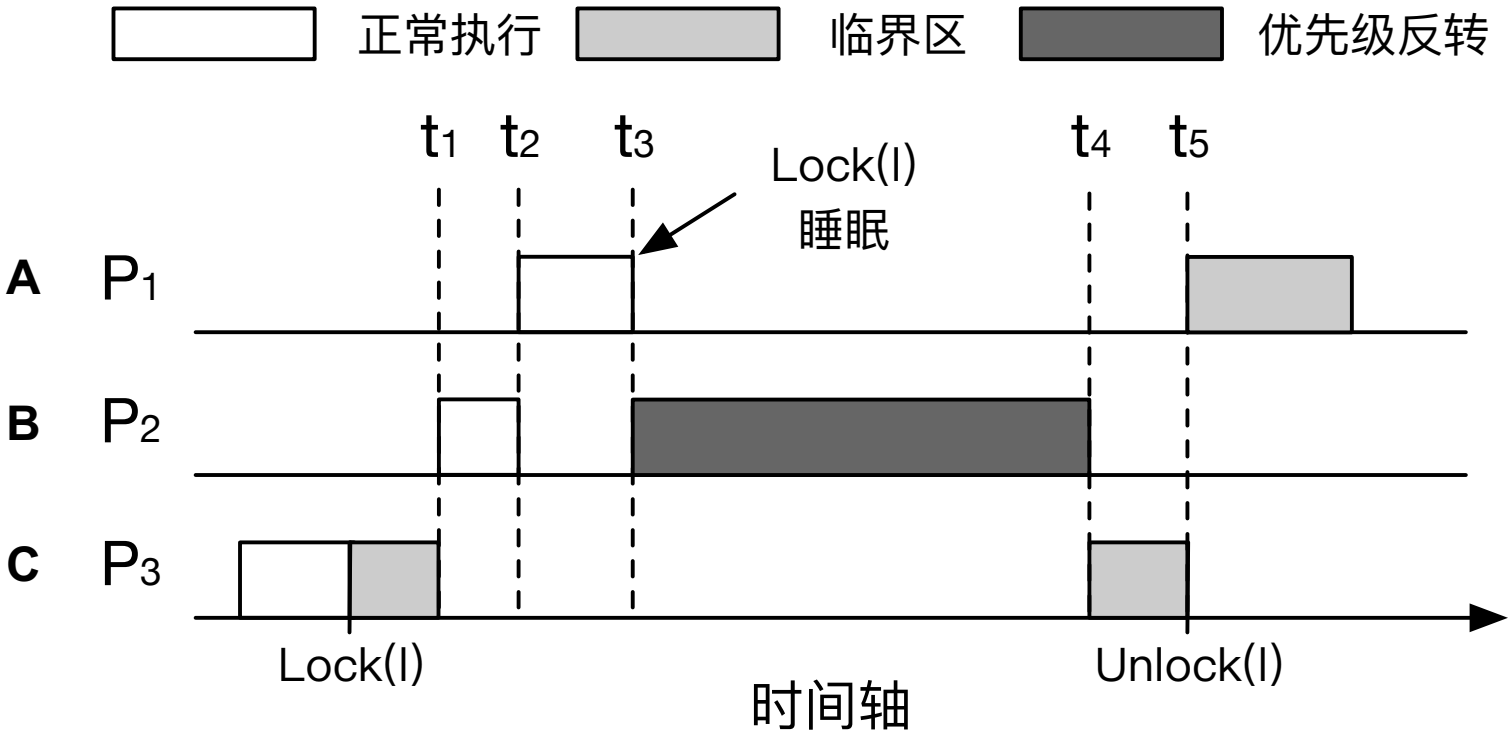


*读/写临界区不同，这里指向一个为示意

读写锁的偏向性

- **考虑这种情况：**
  - `t0`：有读者在临界区
  - `t1`：有新的写者在等待
  - `t2`：另一个读者能否进入临界区？
- **不能：偏向写者的读写锁**
  - 后序读者必须等待写者进入后才进入 **更加公平**
- **能：偏向读者的读写锁**
  - 后序读者可以直接进入临界区 **更好的并行性**

优先级反转（again!）


正常执行
临界区
优先级反转
t1
t2
t3
t4
t5
Lock(l)
睡眠
A
P1
B
P2
C
P3
Lock(l)
Unlock(l)
时间轴

优先级反转解决方案

思考：为什么会出现优先级反转？

操作系统：基于优先级调度

锁：按照锁使用的策略进行调度

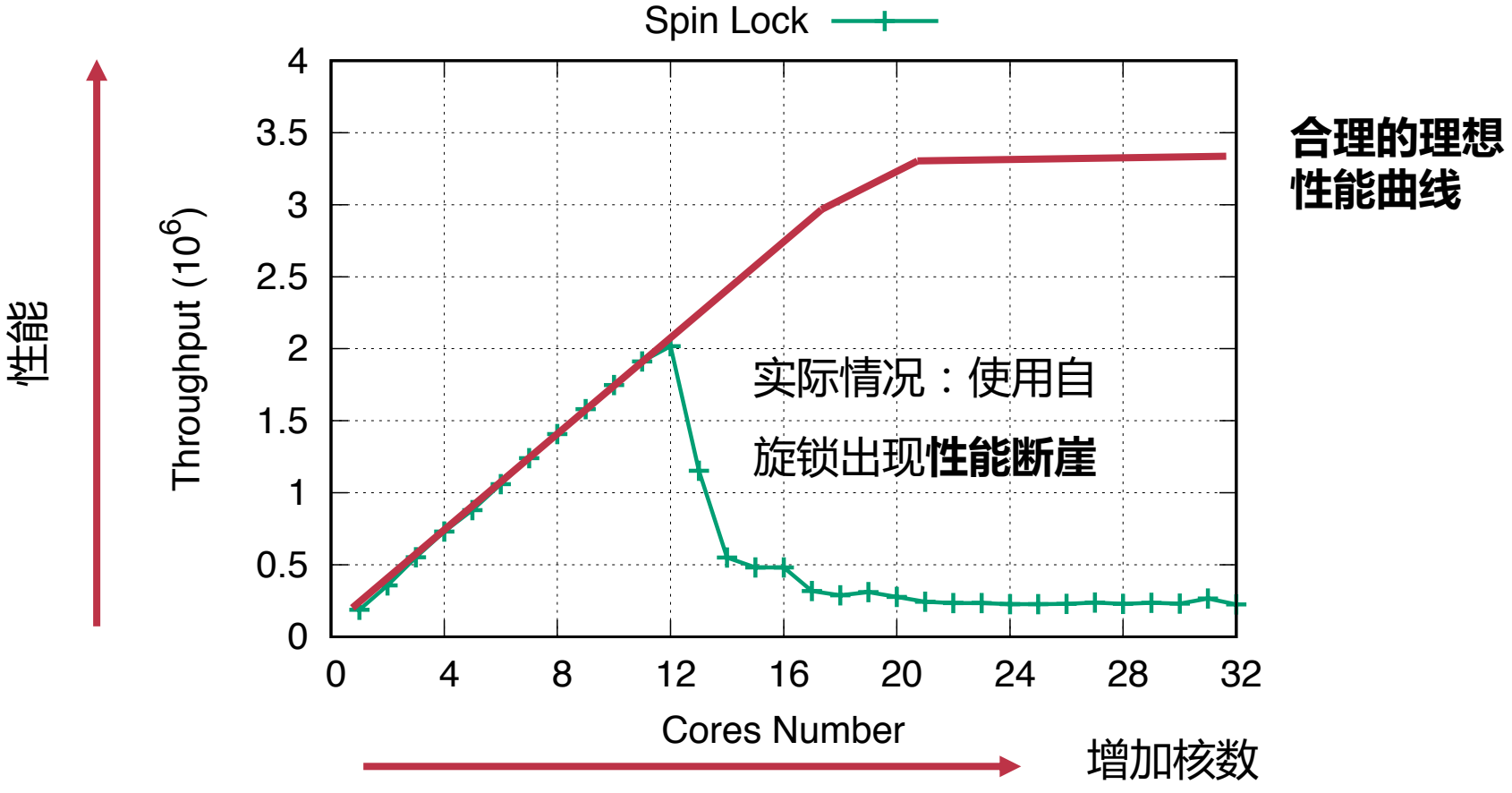
根本原因：双重调度不协调

如何解决？打通两重调度，给另一个调度hint

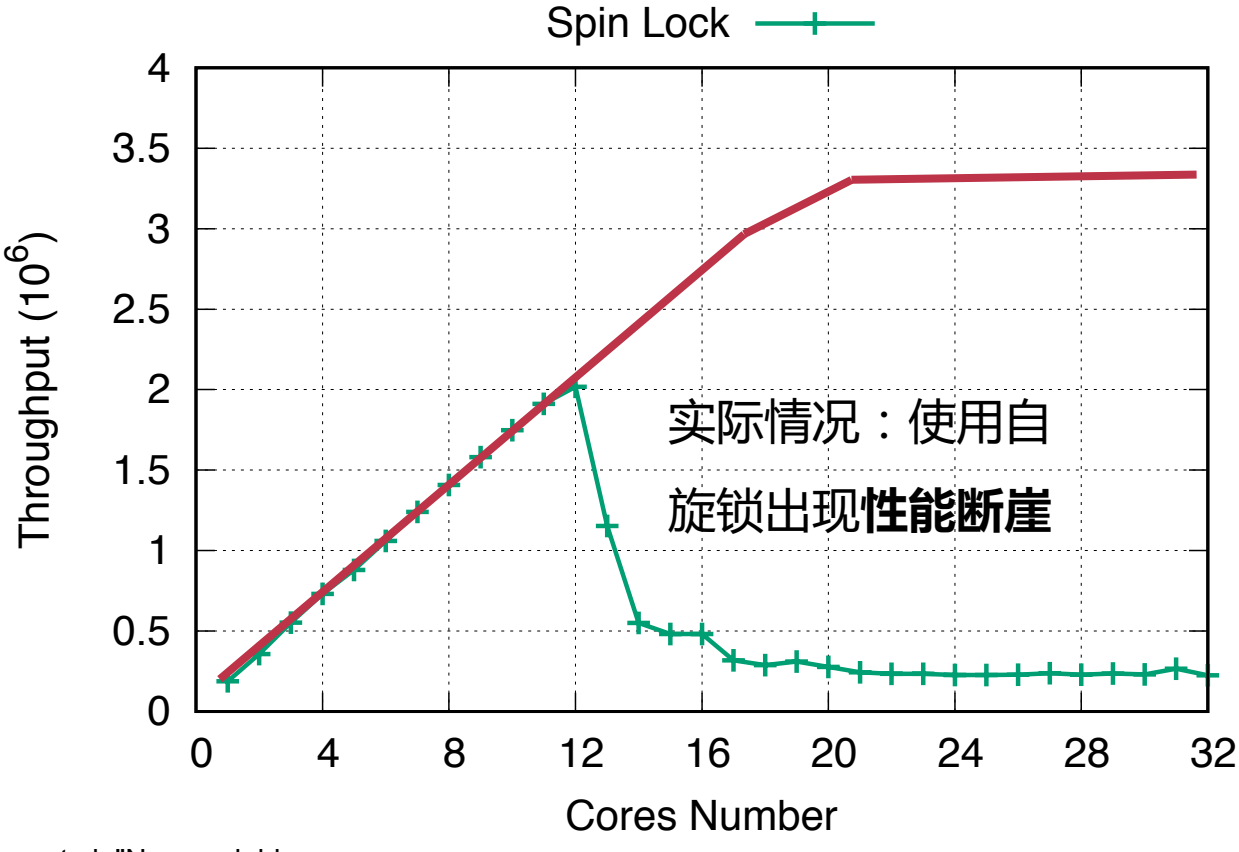
- 不可打断临界区协议 (Non-preemptive Critical Sections, NCP)
- 优先级继承协议 (Priority Inheritance Protocol, PIP)
- 即时优先级置顶协议 (Immediate Priority Ceiling Protocols, IPCP)
- 原生优先级置顶协议 (Original Priority Ceiling Protocols, OPCP)

高优先级进程被阻塞时，给锁持有者该锁竞争者中最高优先级：锁给操作系统调度hint

可扩展性断崖


Spin Lock
性能
Throughput (10^6)
4
3.5
3
2.5
2
1.5
1
0.5
0
0
4
8
12
16
20
24
28
32
Cores Number
增加核数
实际情况：使用自
旋锁出现**性能断崖**
合理的理想
性能曲线

Non-scalable Locks are Dangerous!*



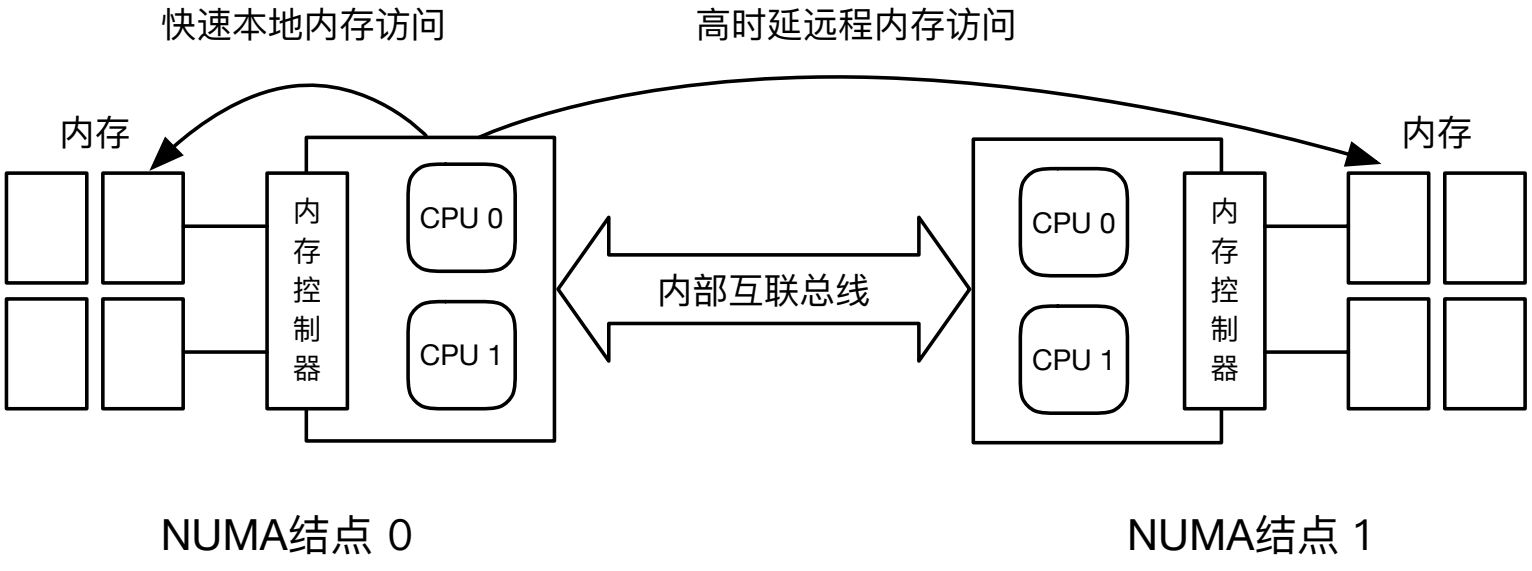
鲲鹏服务器上**互斥锁微基准测试**

合理的理想性能曲线

为什么会出现这种情况？

* Boyd-Wickizer, Silas, et al. "Non-scalable locks are dangerous." *Proceedings of the Linux Symposium*. 2012.

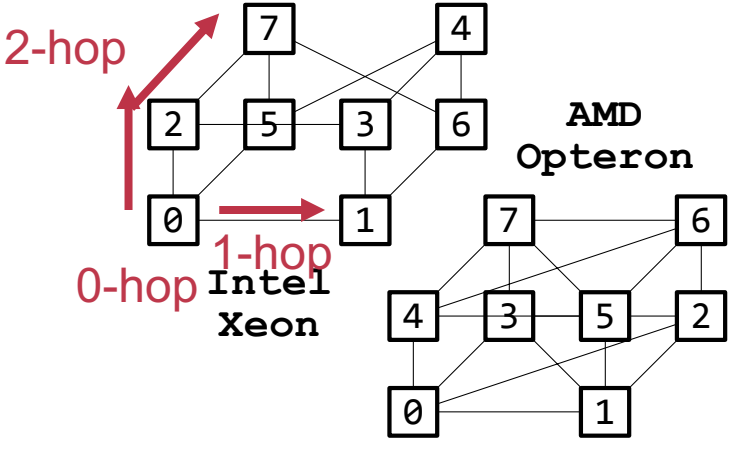
非一致内存访问（NUMA）



避免单内存控制器成为瓶颈，减少内存访问距离

常见于多处理器（多插槽）机器　单处理器众核系统也有可能使用，如Intel Xeon Phi

Intel与AMD的NUMA系统架构与特性



Intel与AMD多插槽NUMA架构

结构复杂

Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

Intel与AMD NUMA访存时延特性

跳数(hop)越多，延迟越高

Intel与AMD的NUMA系统架构与特性

Access	0-hop	1-hop	2-hop	Interleaved
80-core Intel Xeon machine				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64-core AMD Opteron machine				
Sequential	3241	2806/2406	1997	2509
Random	533	509/487	415	466

Intel与AMD NUMA访存带宽特性（MB/s）

跳数越多，带宽受限

NUMA环境中新的挑战

Challenge: 锁**不知道**临界区中需要访问的内容！

除了锁的元数据，
主要是临界区中
访问的共享数据

```
while(TRUE) {
```

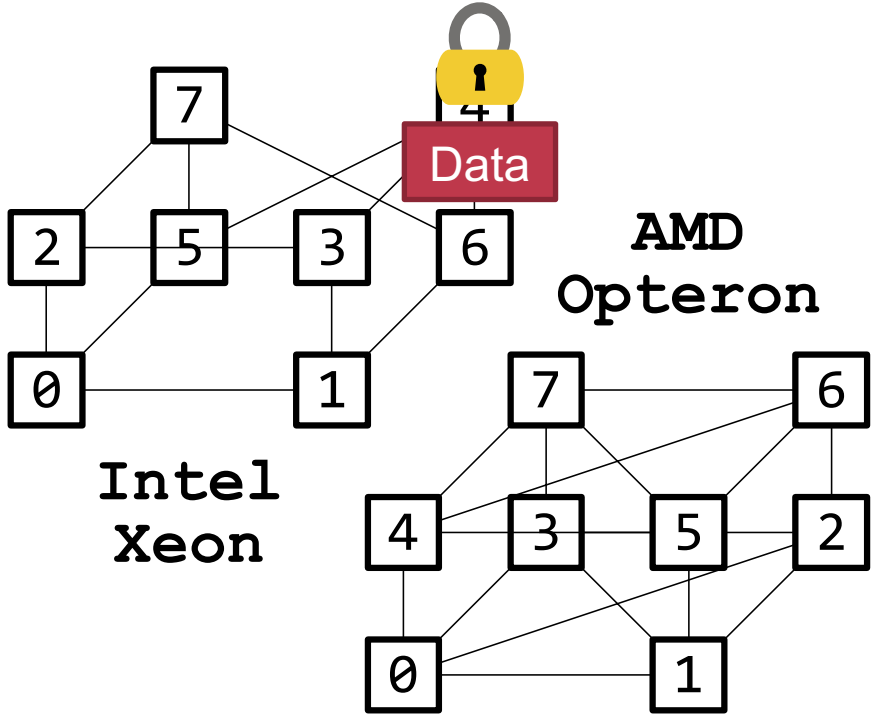
申请进入临界区

临界区部分

通知退出临界区

其他代码

```
}
```



即使在cc-NUMA中没有出现缓存失效 **跨结点的缓存一致性协议开销巨大**

NUMA-aware设计：以cohort锁*为例

核心思路：在一段时间内将访存**限制在本地**

先获取**每结点**本地锁

再获取全局锁

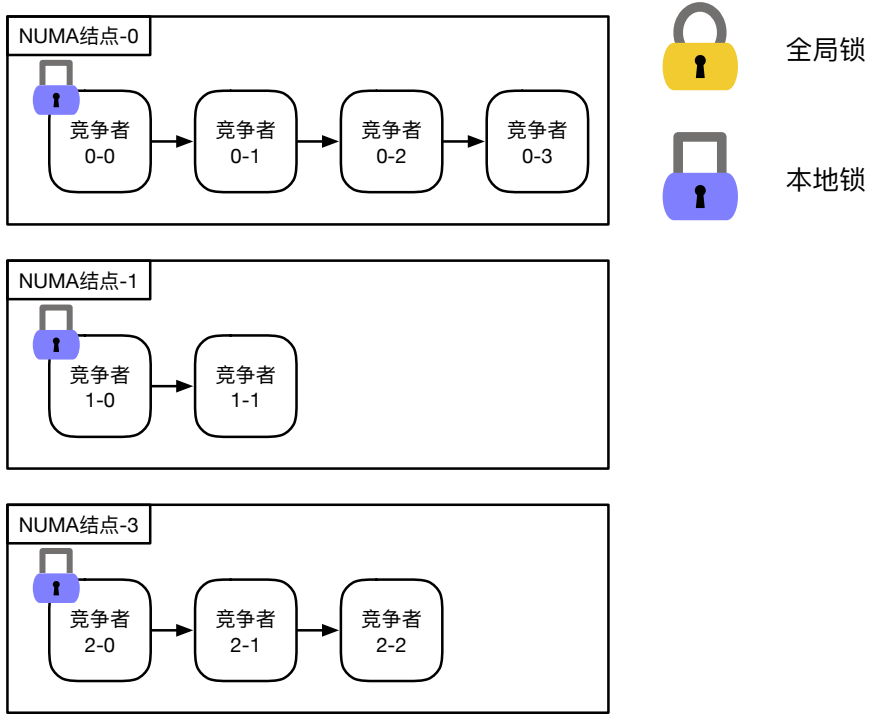
成功获取全局锁

释放时将其传递给

本地等待队列的下一位

全局锁在一段时间内

只在一个结点内部传递

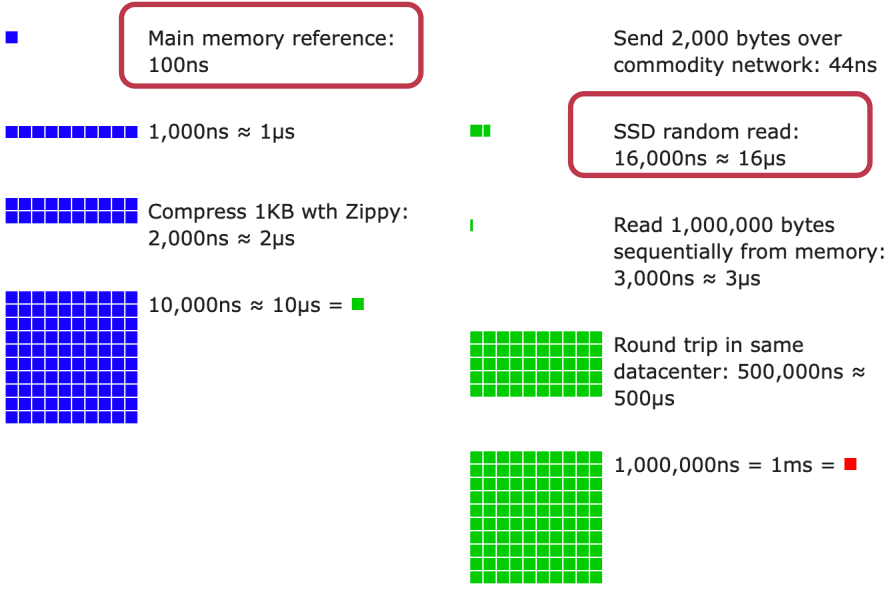


*Dice, David, Virendra J. Marathe, and Nir Shavit. "Lock cohorting: a general technique for designing NUMA locks."

文件系统

页缓存

- 存储访问非常耗时



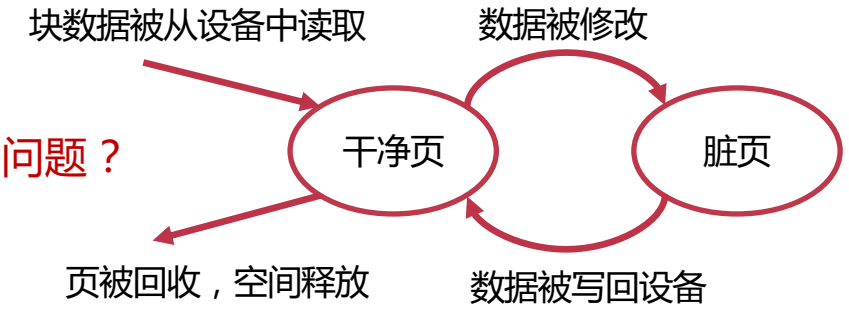
- 文件访问具有时间局部性
  - 一些目录/文件的数据块会被频繁的读取或写入

https://colin-scott.github.io/personal_website/research/interactive_latency.html

页缓存

- 通过缓存提升文件系统性能
  - 在一个块被读入内存并被访问完成后，并不立即回收内存
    - 将块数据暂时缓存在内存中，下一次被访问时可以避免磁盘读取
  - 在一个块被修改后，并不立即将其写回设备
    - 将块数据暂时留在内存中，此后对于该数据块的写可直接修改在此内存中
  - 定期或在用户要求时才将数据写回设备

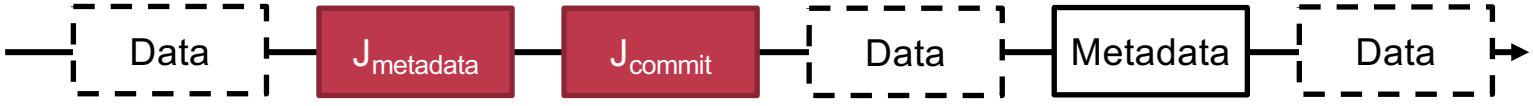
问题：数据不及时写回，会造成什么问题？



Ext4用JBD2实现的三种日志模式

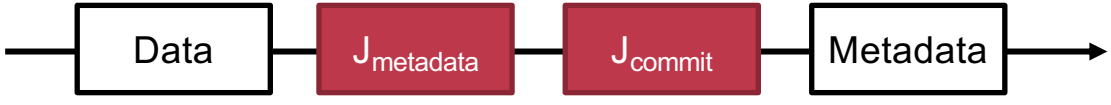
Writeback Mode：日志只记录元数据

最快，但是一致性最差！



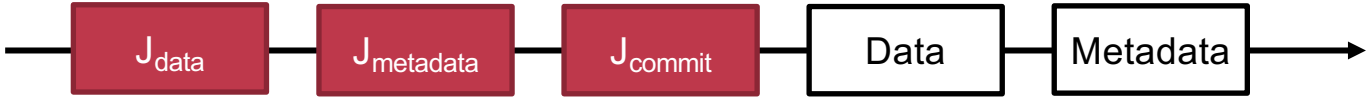
Ordered Mode：日志只记录元数据+数据块在元数据日志前写入磁盘

默认模式



Journal Mode：元数据和数据均使用日志记录

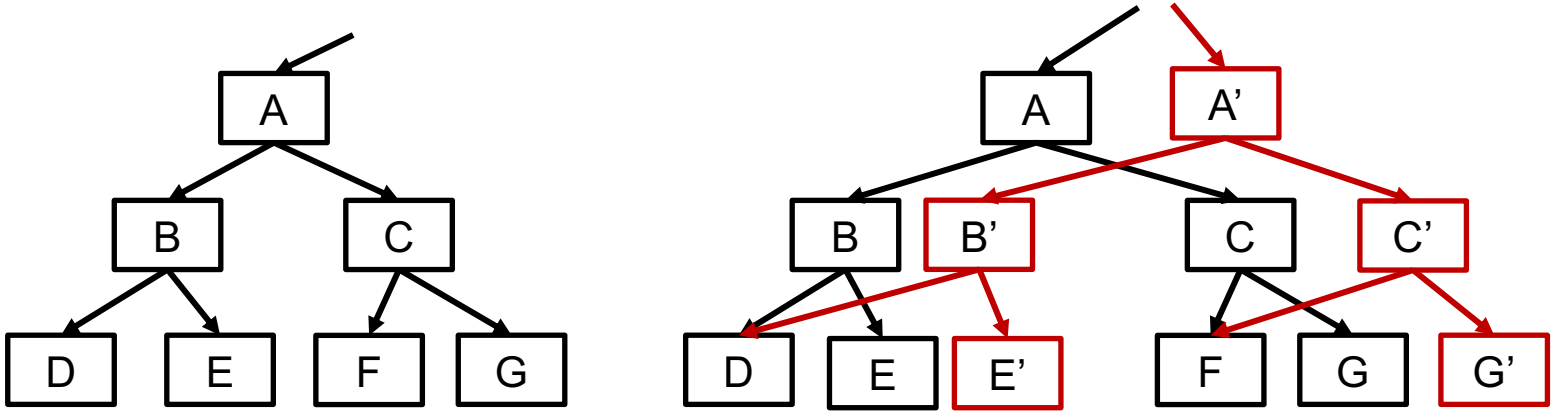
一致性最好，但数据写入两次！



思考一下：三种模式各自有何问题和优势？

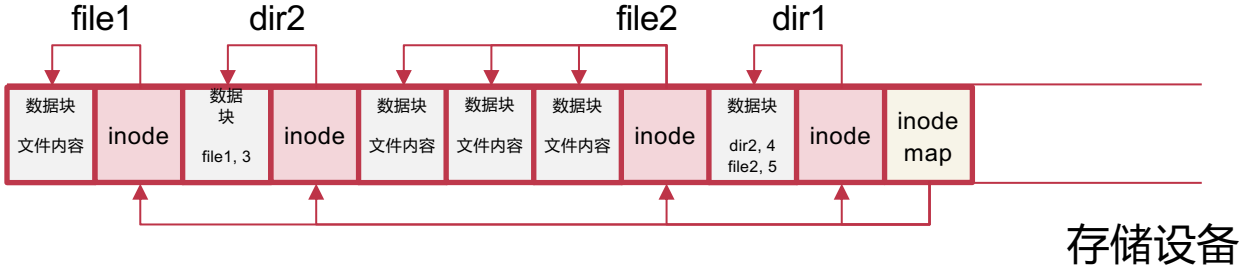
写时复制（Copy-on-Write）

- 在修改多个数据时，不直接修改数据，而是将数据复制一份，在复制上进行修改，并通过递归的方法将修改变成原子操作
- 常用于树状结构



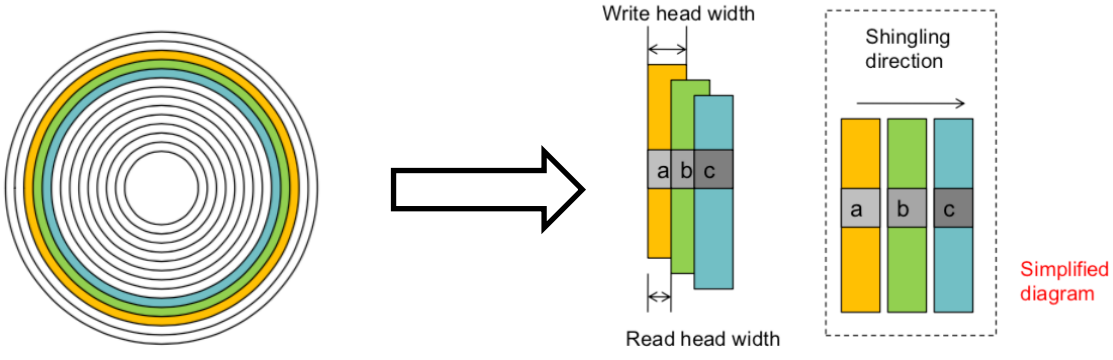
日志文件系统（Log-structured FS）

- 假设：文件被缓存在内存中，文件读请求可以被很好的处理
  - 于是，文件写成为瓶颈
- 块存储设备的顺序写比随机写速度很块
  - 磁盘寻道时间
- 将文件系统的修改以日志的方式**顺序写入**存储设备



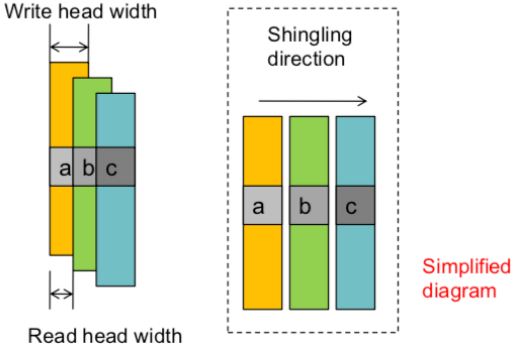
瓦式磁盘

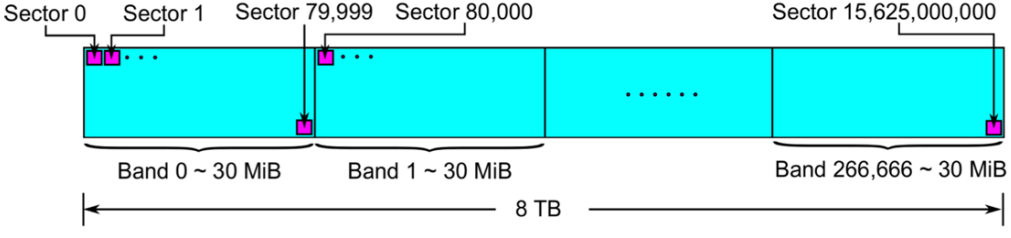
- 传统磁盘密度难以提升
  - 写磁头的宽度难以减小
- 瓦式磁盘将磁道重叠，提升存储密度
  - 减小读磁头的宽度



瓦式磁盘的问题：随机写

- 随机写会覆盖后面磁道的数据
  - 只能顺序写入
- 避免整个磁盘只能顺序写入
  - 磁盘划分成多个Band，Band间增大距离
  - 每个Band内必须顺序写入

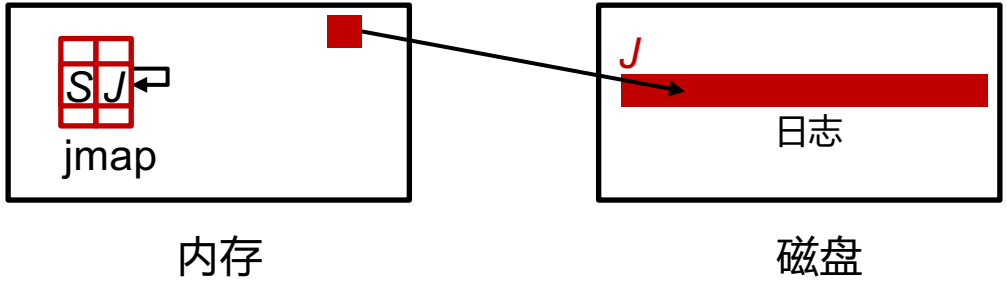




Logical view of an 8 TB statically mapped drive-managed SMR disk

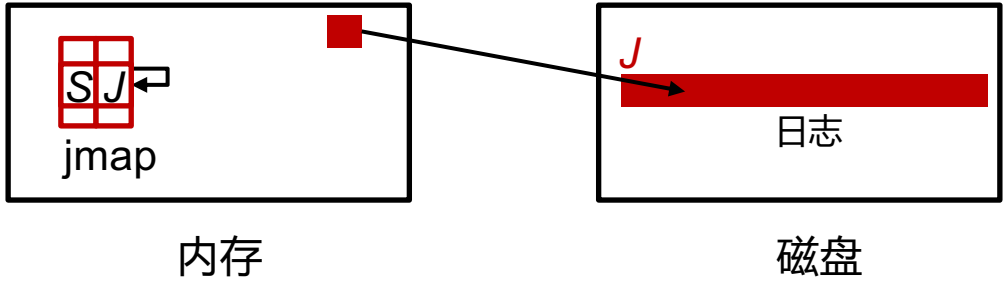
Ext4上的元数据分散

- 修改磁盘布局需要大规模修改Ext4
  - 人力成本、新增Bug、破坏原有功能……
- 怎么办？
- 引入Indirection：以LFS形式增加一个元数据缓存



方法：以LFS形式增加一个元数据缓存

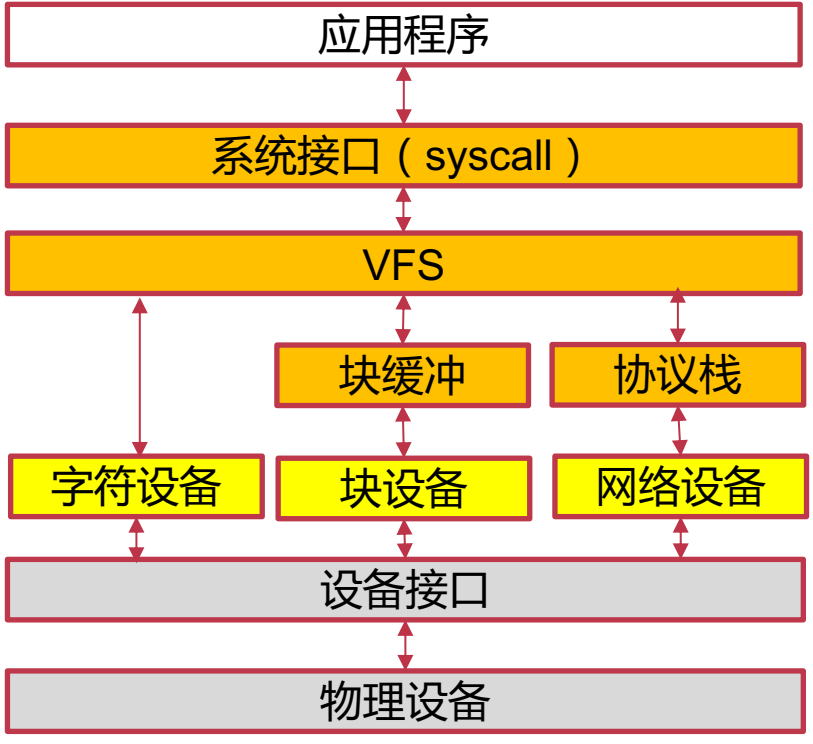
- 以LFS形式维护10GB日志空间作为元数据缓存
  1. JBD2首先将元数据写入日志区域*J*，将元数据标记为clean（无需写回）
  2. JBD2在内存中的jmap中将*S*映射到*J*
- Indirection: 元数据访问需要通过jmap进行一次地址转换



设备与I/O

案例：Linux常见设备分类

- 字符设备
- 块设备
- 网络设备



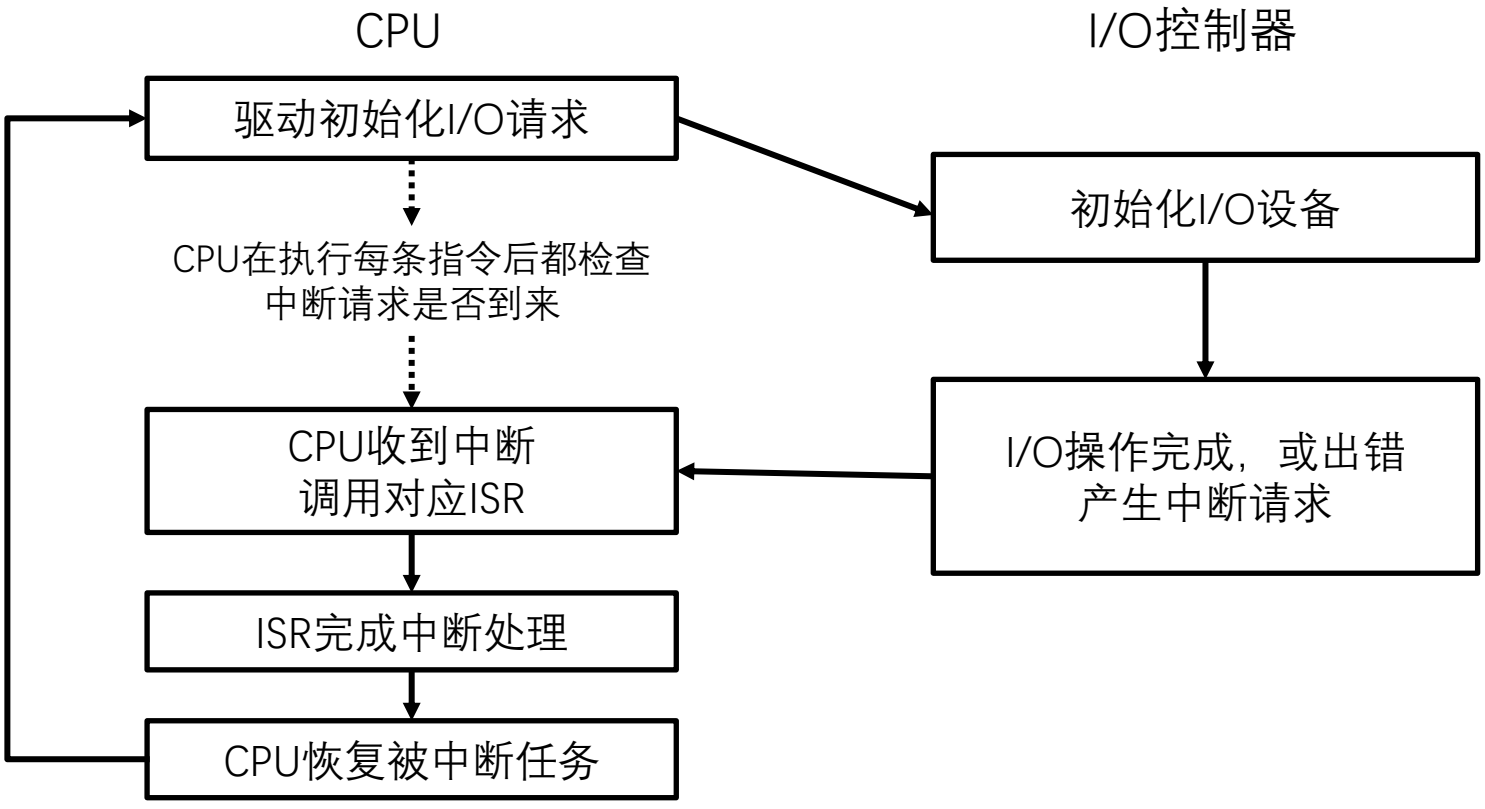
CPU与外设的数据交互

- **可编程 I/O（Programmable I/O）**
  - 通过CPU in/out 或 load/store 指令
  - 消耗CPU时钟周期和数据量成正比
  - 适合于简单小型的设备
- **直接内存访问（DMA）**
  - 外设可直接访问总线
  - DMA与内存互相传输数据，传输不需要CPU参与
  - 适合于高吞吐量I/O

可编程 I/O

- **PIO (Port IO)**
  - IO设备具有独立的地址空间
  - 使用特殊的指令（如x86中的in/out指令）
- **MMIO (Memory-mapped IO)**
  - 将设备映射到连续物理内存中
  - 使用内存访问指令
  - 行为与内存不完全一样，读写会有副作用

CPU中断处理流程


CPU
I/O控制器
驱动初始化I/O请求
初始化I/O设备
CPU在执行每条指令后都检查
中断请求是否到来
CPU收到中断
调用对应ISR
I/O操作完成，或出错
产生中断请求
ISR完成中断处理
CPU恢复被中断任务

ARM中断的生命周期

① **Generate**：外设发起一个中断

② **Distribute**：Distributor对收到的中断源进行仲裁，然后发送给对应的CPU Interface

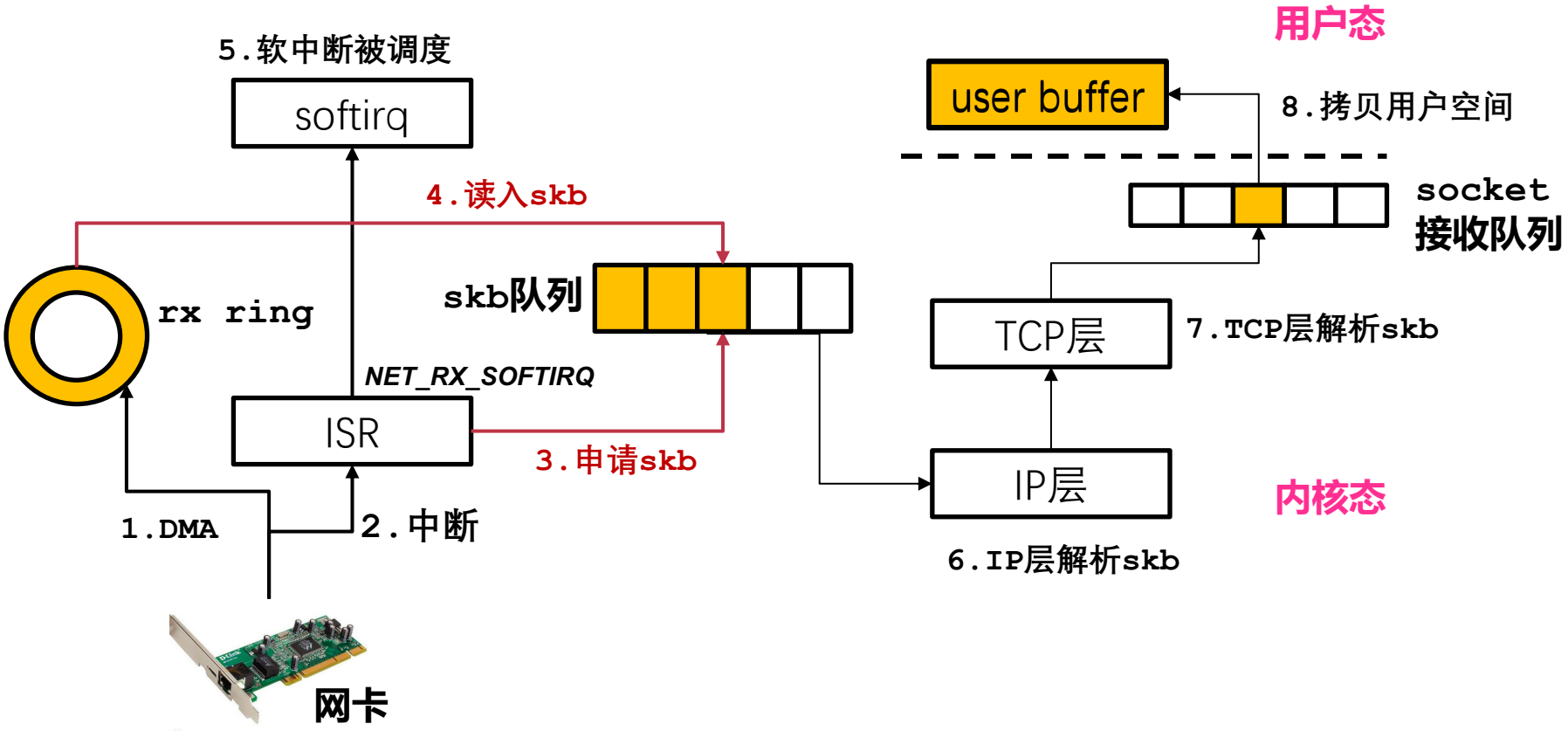
③ **Deliver**：CPU Interface将中断传给core

④ **Activate**：core读 GICC_IAR 寄存器，对中断进行确认

⑤ **Priority drop:** core写 GICC_EOIR 寄存器，实现优先级重置

⑥ **Deactivate**：core写 GICC_DIR 寄存器，来无效该中断

Linux收包过程


用户态
5.软中断被调度
softirq
user buffer
8.拷贝用户空间
4.读入skb
socket
接收队列
rx ring
skb队列
NET_RX_SOFTIRQ
TCP层
7.TCP层解析skb
ISR
3.申请skb
IP层
内核态
1.DMA
2.中断
6.IP层解析skb
网卡

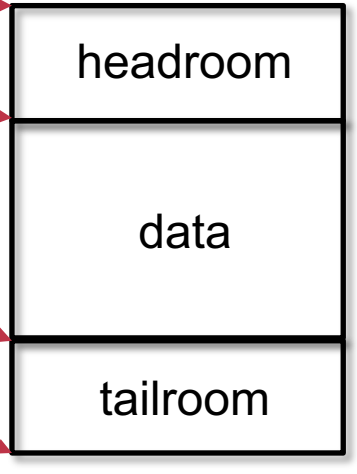
网络包的管理

- **要求高效地处理分层**
  - 发包时需要不断添加新的头部，收包则相反
- **避免连续存放网络包**
  - 移动过程中数据拷贝会有很大开销
- **Linux数据结构：sk_buff（简称skb）**
  - 让分层的处理变得高效：零拷贝
  - 快速申请和释放内存：防止内存碎片

sk_buff

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff     * next;
    struct sk_buff     * prev;

    // 真正指向的数据buffer
    struct sock  *sk;

    // 缓冲区的头部
    unsigned char *head;
    // 实际数据的头部
    unsigned char *data;
    // 实际数据的尾部
    unsigned char *tail;
    // 缓冲区的尾部
    unsigned char *end;
};
```

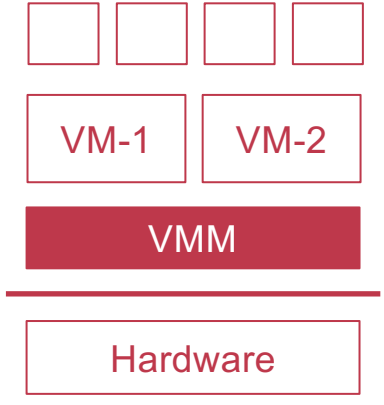
- **sk_buff本身不存储报文**
  - 通过指针指向真正的报文内存空间
- **在各层传递时**
  - 只需调整指针相应位置即可



系统虚拟化

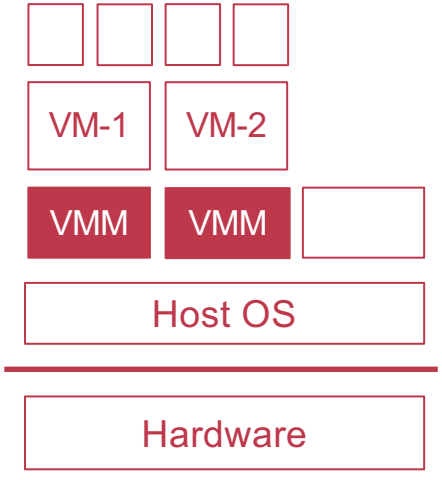
Type-1虚拟机监控器

- **直接运行在硬件之上**
  - 充当操作系统的角色
  - 直接管理所有物理资源
    - 实现调度、内存管理、驱动等功能
- **性能损失较少**
- **例如Xen, VMware ESX Server**



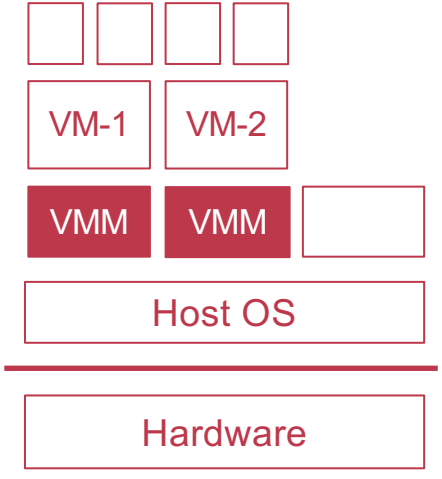
Type-2虚拟机监控器

- **依托于主机操作系统**
  - 主机操作系统管理物理资源
  - 虚拟机监控器以进程/内核模块的形态运行
- **易于实现和安装**
- **例如QEMU/KVM**
- **思考：**
  - Type-2类型有什么优势？



Type-2的优势

- **在已有的操作系统之上将虚拟机当做应用运行**
- **复用主机操作系统的大部分功能**
  - 文件系统
  - 驱动程序
  - 处理器调度
  - 物理内存管理

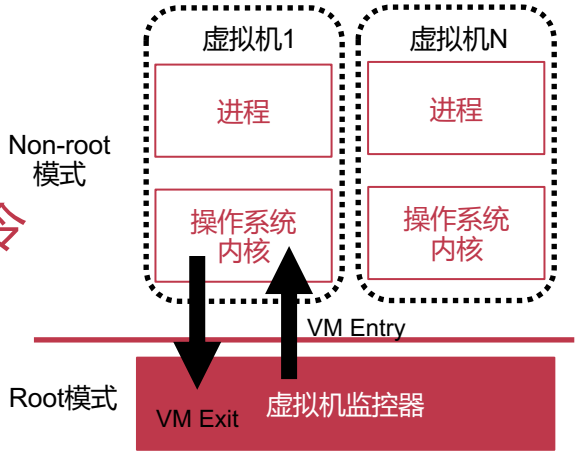


Virtual Machine Control Structure (VMCS)

- **VMM提供给硬件的内存页（4KB）**
  - 记录与当前VM运行相关的所有状态
- **VM Entry**
  - 硬件自动将当前CPU中的VMM状态保存至VMCS
  - 硬件硬件自动从VMCS中加载VM状态至CPU中
- **VM Exit**
  - 硬件自动将当前CPU中的VM状态保存至VMCS
  - 硬件自动从VMCS加载VMM状态至CPU中

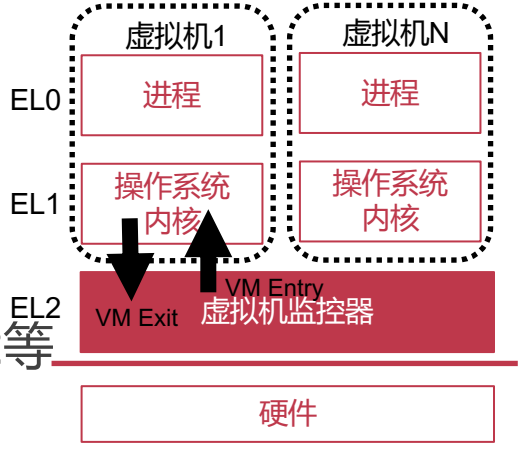
x86中的VM Entry和VM Exit

- **VM Entry**
  - 从VMM进入VM
  - 从Root模式切换到Non-root模式
  - 第一次启动虚拟机时使用VMLAUNCH指令
  - 后续的VM Entry使用VMRESUME指令
- **VM Exit**
  - 从VM回到VMM
  - 从Non-root模式切换到Root模式
  - 虚拟机执行敏感指令或发生事件(如外部中断)



ARM的VM Entry和VM Exit

- **VM Entry**
  - 使用ERET指令从VMM进入VM
  - 在进入VM之前，VMM需要主动加载VM状态
    - VM内状态：通用寄存器、系统寄存器、
    - VM的控制状态：HCR_EL2、VTTBR_EL2等
- **VM Exit**
  - 虚拟机执行敏感指令或收到中断等
  - 以Exception、IRQ、FIQ的形式回到VMM
    - 调用VMM记录在vbar_el2中的相关处理函数
  - 下陷第一步：VMM主动保存所有VM的状态



ARM硬件虚拟化的新功能

- **ARM中没有VMCS**
- **VM能直接控制EL1和EL0的状态**
  - 自由地修改PSTATE(VMM不需要捕捉CPS指令)
  - 可以读写TTBR0_EL1/SCTRL_EL1/TCR_EL1等寄存器
- **VM Exit时VMM仍然可以直接访问VM的EL0和EL1寄存器**

- **思考题1：为什么ARM中可以不需要VMCS？**
- **思考题2：ARM中没有VMCS，对于VMM的设计和实现来说有什么优缺点？**

VT-x和VHE对比

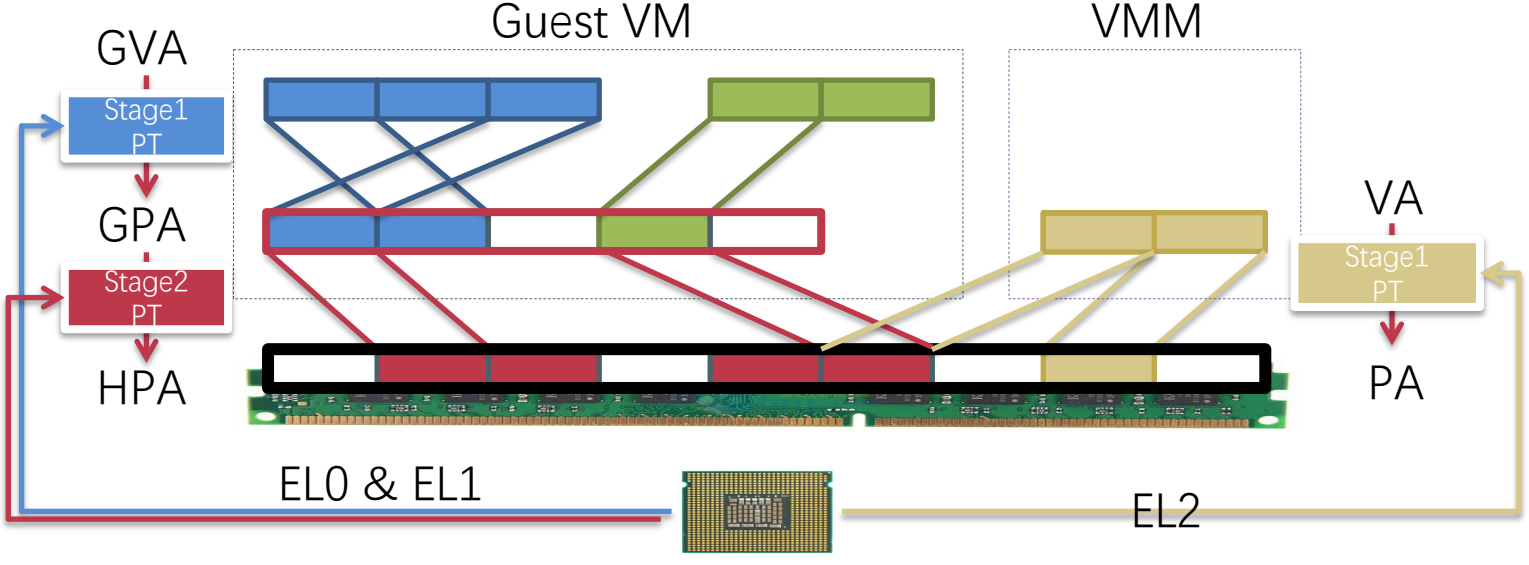
	VT-x	VHE
新特权级	Root和Non-root	EL2
是否有VMCS？	是	否
VM Entry/Exit时硬件自动保存状态？	是	否
是否引入新的指令？	是(多)	是(少)
是否引入新的系统寄存器?	否	是(多)
是否有扩展页表(第二阶段页表)?	是	是

硬件虚拟化对内存翻译的支持

- **Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- **新的页表**
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

第二阶段页表

- 第一阶段页表：虚拟机内虚拟地址翻译（GVA->GPA）
- 第二阶段页表：虚拟机客户物理地址翻译（GPA->HPA）



TLB：缓存地址翻译结果

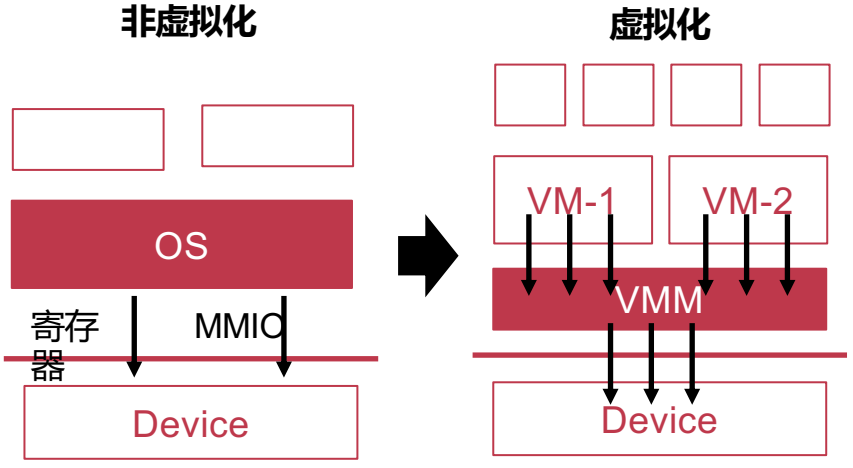
- **回顾：TLB不仅可以缓存第一阶段地址翻译结果**
- **TLB也可以第二阶段地址翻译后的结果**
  - 包括第一阶段的翻译结果(GVA->GPA)
  - 包括第二阶段的翻译结果(GPA->HPA)
  - 大大提升GVA->HPA的翻译性能：不需要24次内存访问
- **切换VTTBR_EL2时**
  - 理论上应将前一个VM的TLB项全部刷掉

如何处理缺页异常

- **两阶段翻译的缺页异常分开处理**
- **第一阶段缺页异常**
  - 直接调用VM的Page fault handler
  - 修改第一阶段页表**不会**引起任何虚拟机下陷
- **第二阶段缺页异常**
  - 虚拟机下陷，直接调用VMM的Page fault handler

方法1：设备模拟

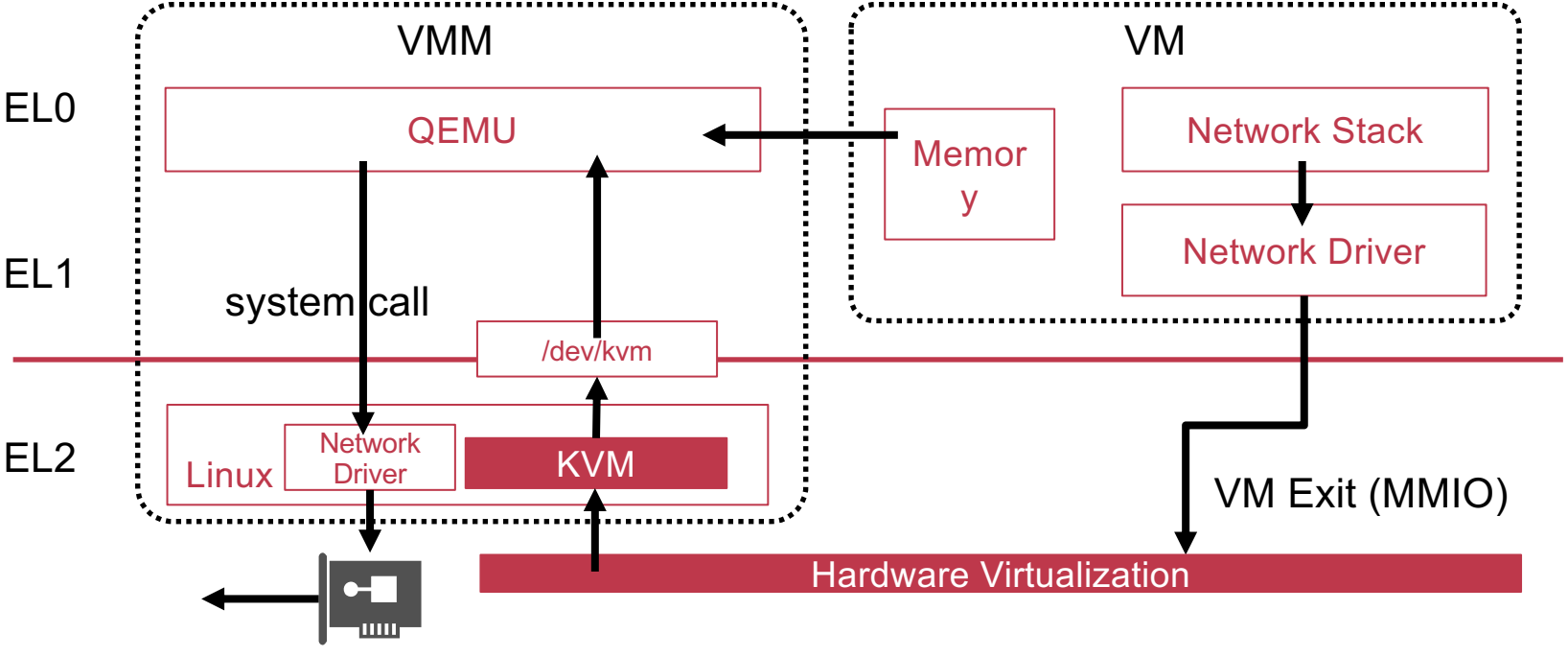
- **OS与设备交互的硬件接口**
  - 模拟寄存器(中断等)
  - 捕捉MMIO操作



- **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

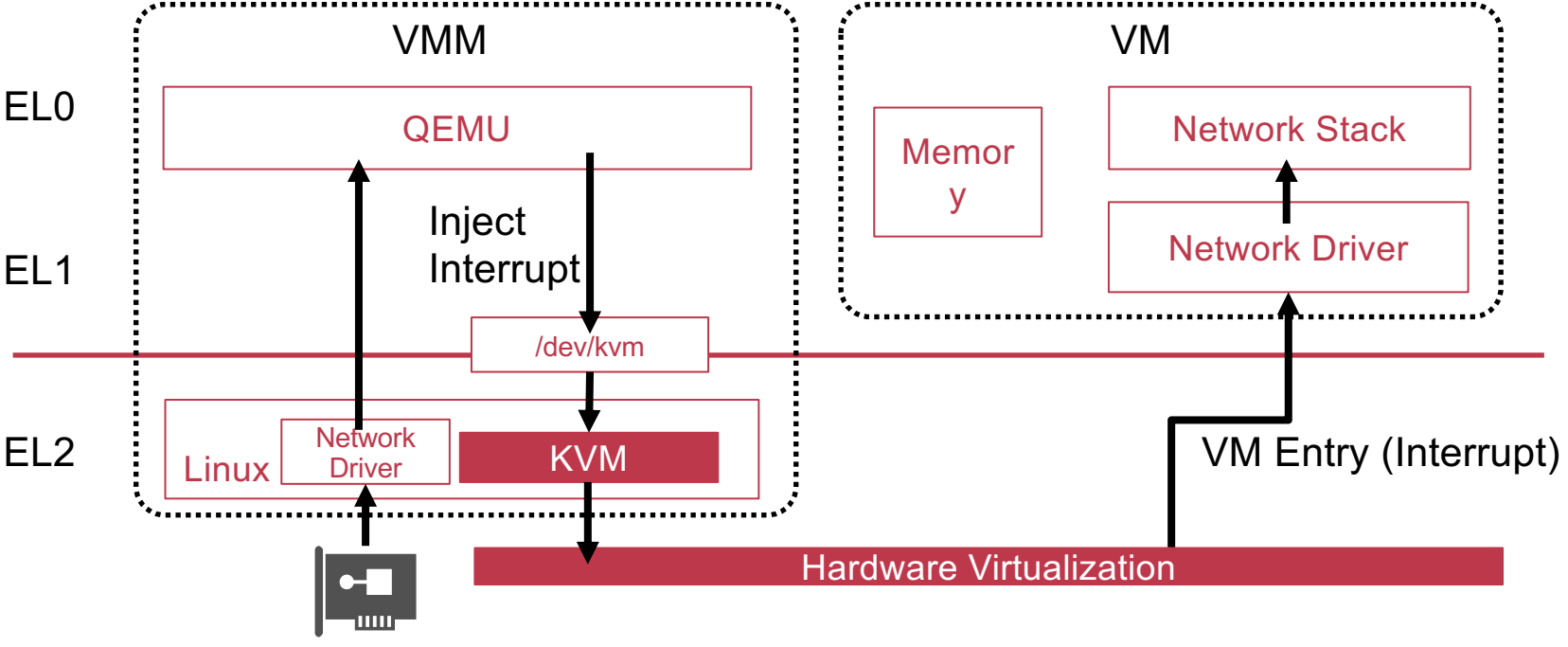
例：QEMU/KVM设备模拟1

- 以虚拟网卡举例——发包过程



例：QEMU/KVM设备模拟2

- 以虚拟网卡举例——收包过程

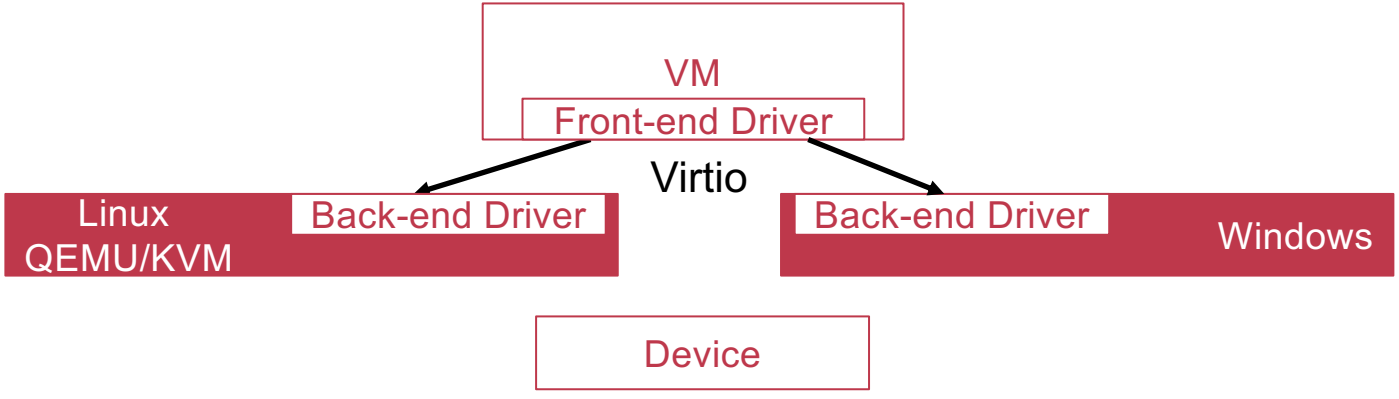


方法2：半虚拟化方式

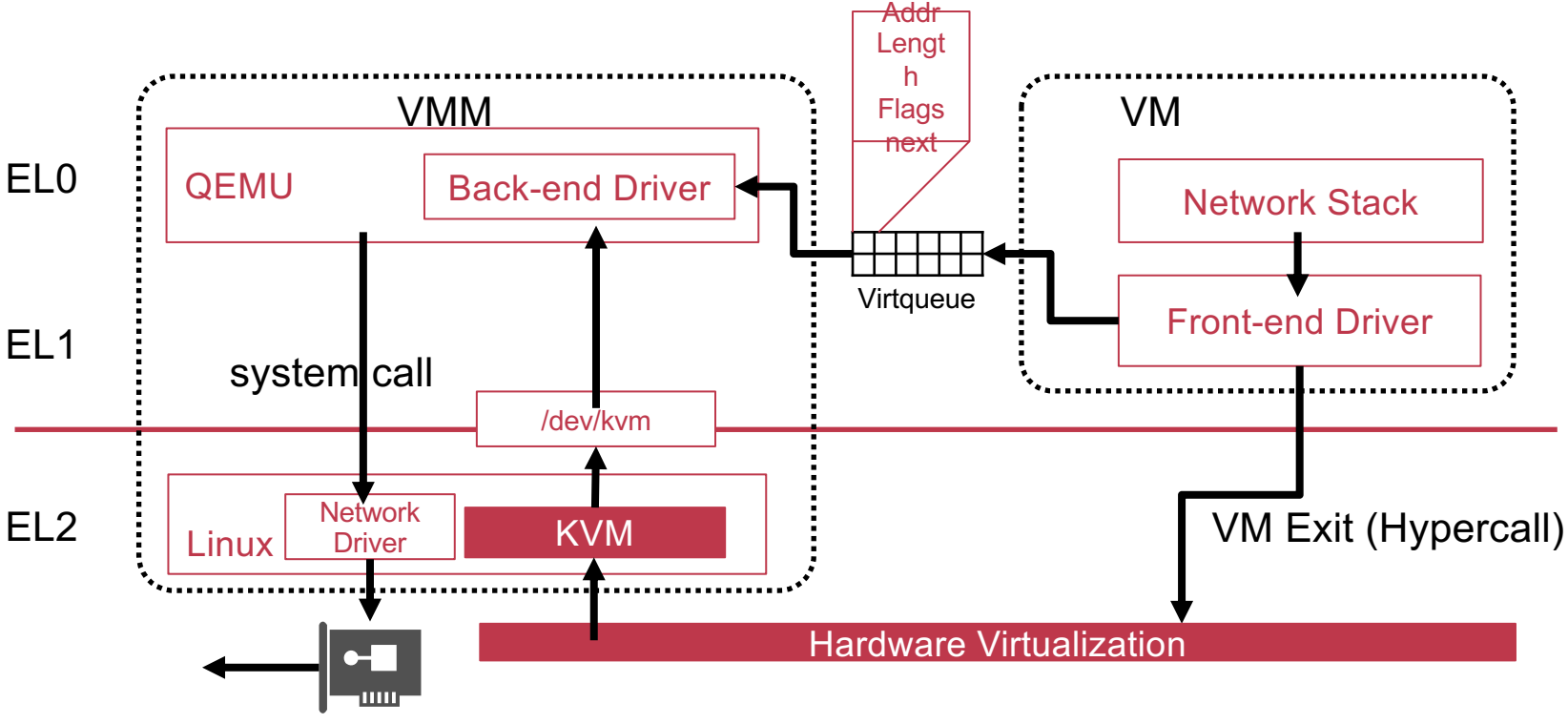
- **协同设计**
  - 虚拟机“知道”自己运行在虚拟化环境
  - 虚拟机内运行前端(front-end)驱动
  - VMM内运行后端(back-end)驱动
- **VMM主动提供Hypercall给VM**
- **通过共享内存传递指令和命令**

VirtIO: Unified Para-virtualized I/O

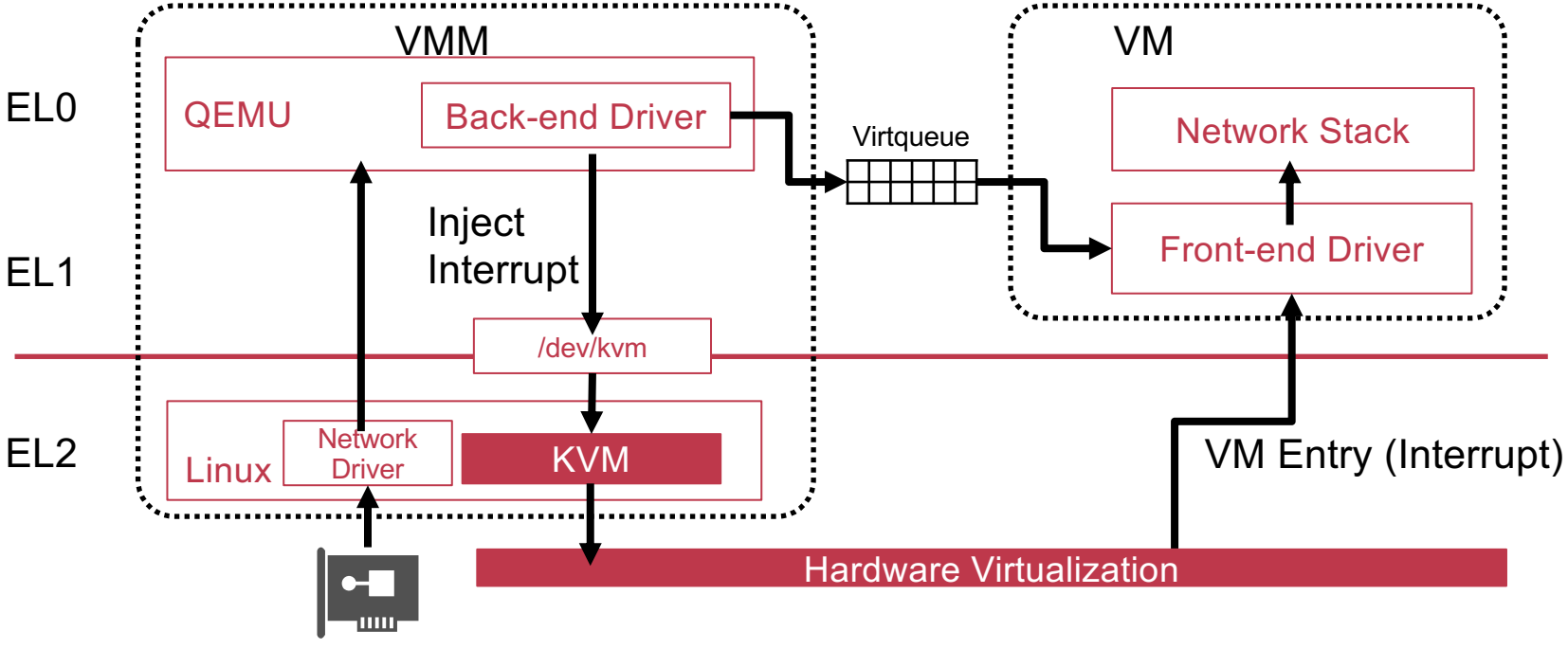
- **标准化的半虚拟化I/O框架**
  - 通用的前端抽象
  - 标准化接口
  - 增加代码的跨平台重用



例：QEMU/KVM半虚拟化1

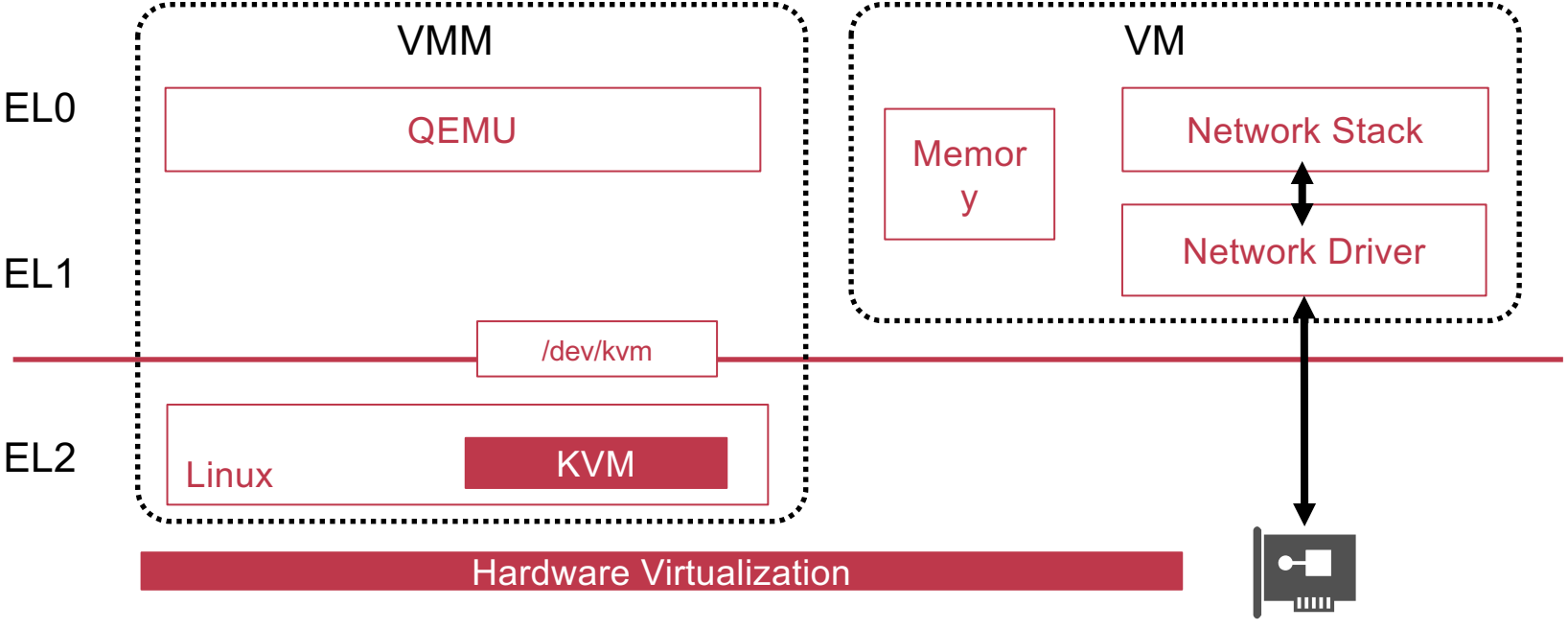


例：QEMU/KVM半虚拟化2



方法3：设备直通

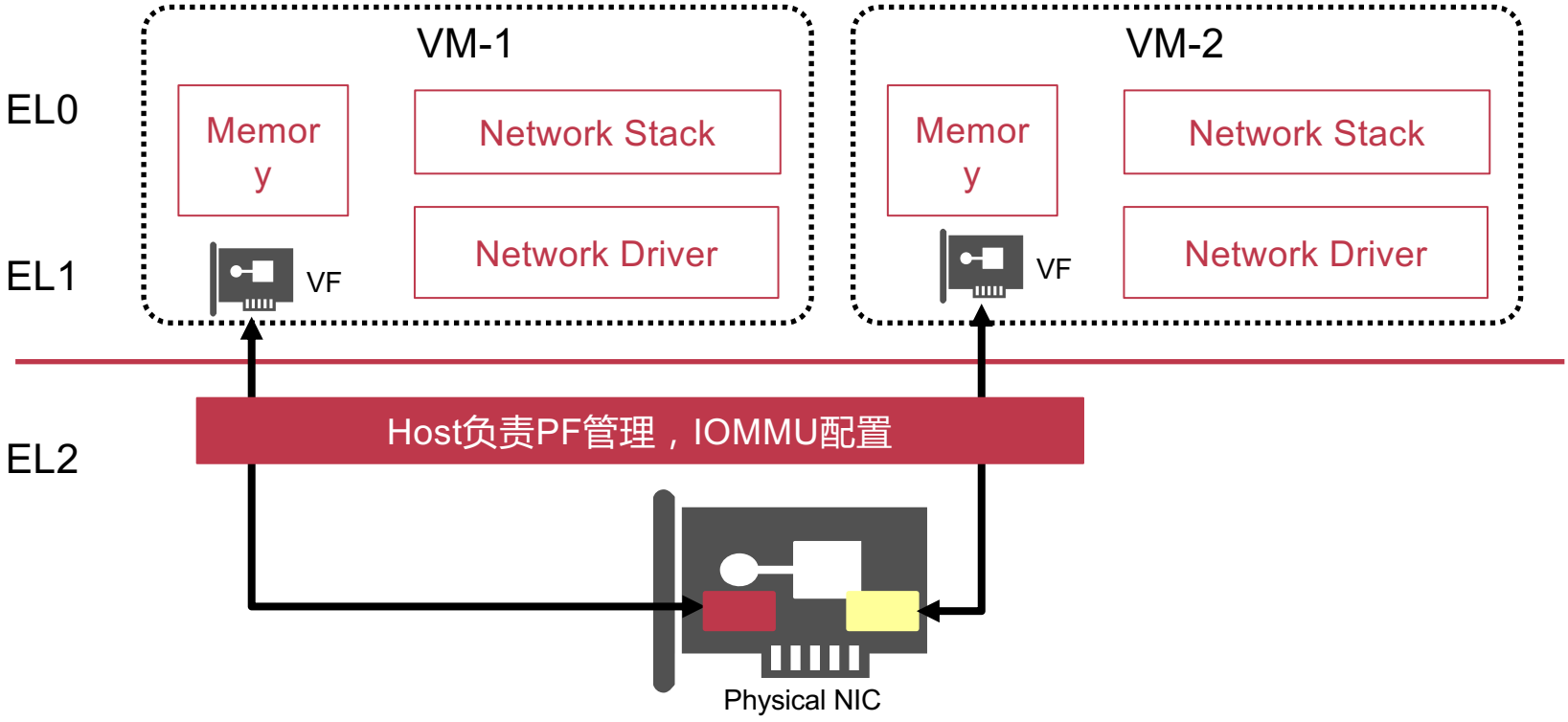
- 虚拟机直接管理物理设备



Single Root I/O Virtualization (SRIOV)

- **SR-IOV是PCI-SIG组织确定的标准**
- **满足SRIOV标准的设备，在设备层实现设备复用**
  - 能够创建多个Virtual Function(VF)，每一个VF分配给一个VM
    - 负责进行数据传输，属于数据面（Data-plane）
  - 物理设备被称为Physical Function(PF)，由Host管理
    - 负责进行配置和管理，属于控制面（Control-plane）
- **设备的功能**
  - 确保VF之间的数据流和控制流彼此不影响

SRIOV的使用

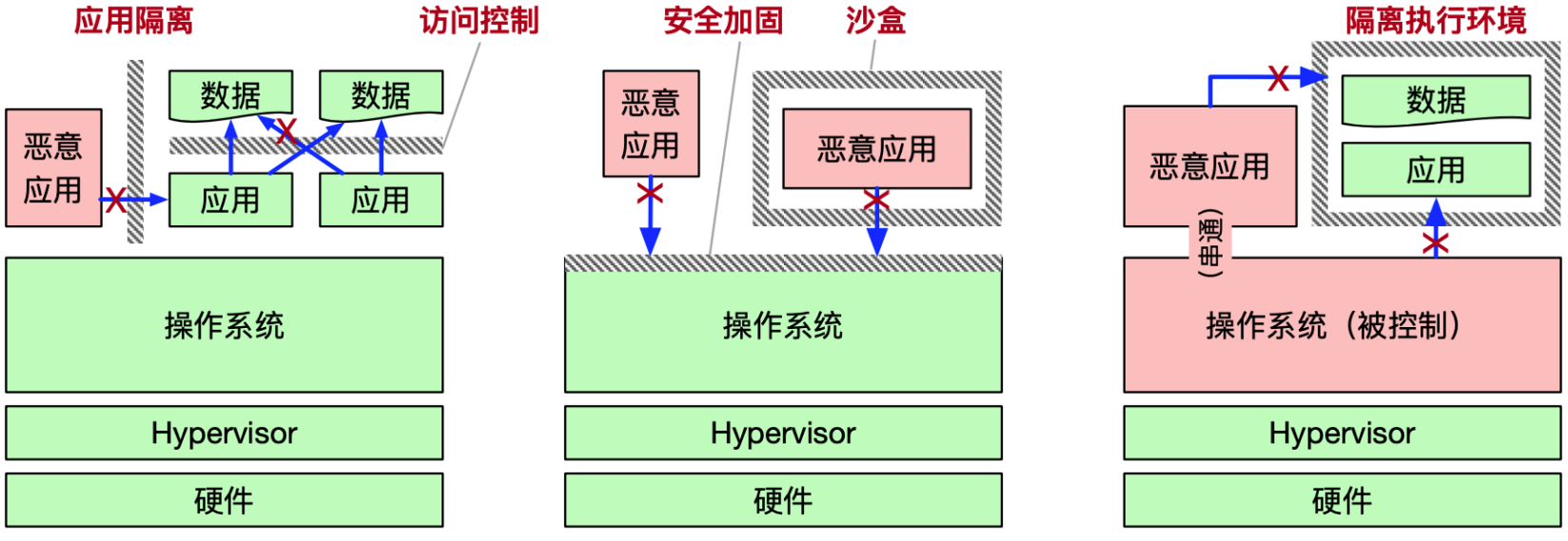

VM-1
VM-2
EL0
EL1
EL2
Memor y
Network Stack
Network Driver
VF
Memor y
Network Stack
Network Driver
VF
Host负责PF管理，IOMMU配置
Physical NIC

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

操作系统安全

操作系统安全的三个层次

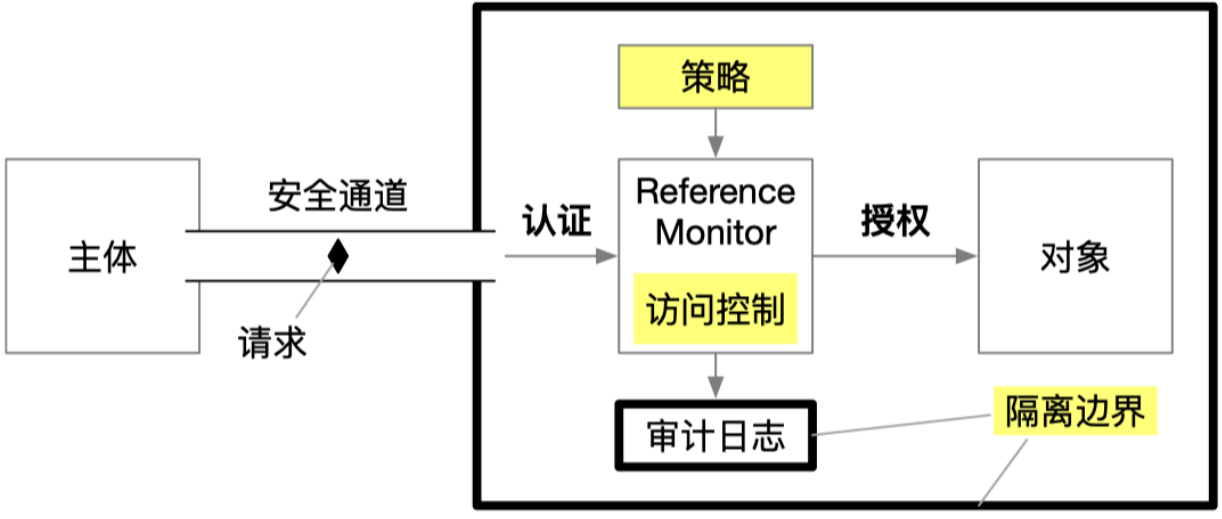


(a) 基于操作系统的应用隔离与访问控制

(b) 操作系统对恶意应用的隔离与防御

(c) 操作系统不可信时对应用的保护

引用监视器（Reference Monitor）机制



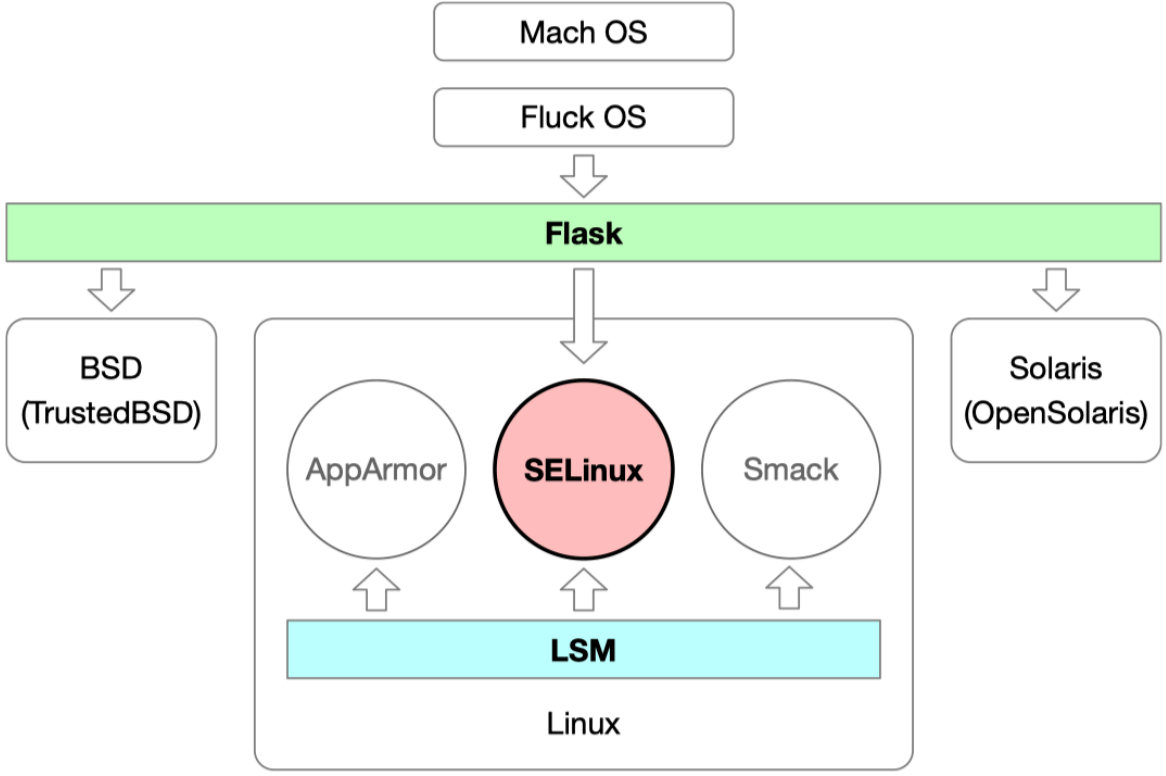
Reference Monitor 负责两件事：

1. **认证（Authentication）**：确定发起请求实体的**身份**
2. **授权（Authorization）**：确定实体确实拥有访问资源的**权限**

DAC与MAC

- **自主访问控制（DAC: Discretionary Access Control）**
  - 指一个对象的拥有者有权限决定该对象是否可以被其他人访问
  - 例如，文件系统就是一类典 型的 DAC，因为文件的拥有者可以设置文件如何被其他用户访问
- **强制访问控制（MAC: Mandatory Access Control）**
  - 什么数据能被谁访问，完全由底层的系统决定
  - 例如，在军队中，如果某个文件设置为机密，那么就算是指挥官也不能把这个文件给没有权限的人看——这个规则是由军法（系统）规定的

SELinux、Flask与LSM



侧信道与隐秘信道的关系

- **侧信道与隐秘信道很类似**
  - 两者都使用类似的方式进行数据的传递
- **侧信道攻击和隐秘信道攻击的不同**
  - 隐秘信道攻击：两方是互相串通的，其目的就是为了将信息从一方传给另一方
  - 侧信道攻击：一方是攻击者，另一方是被攻击者，攻击者窃取被攻击者的数据
    - 即被攻击者无意通过侧信道泄露了自己的数据

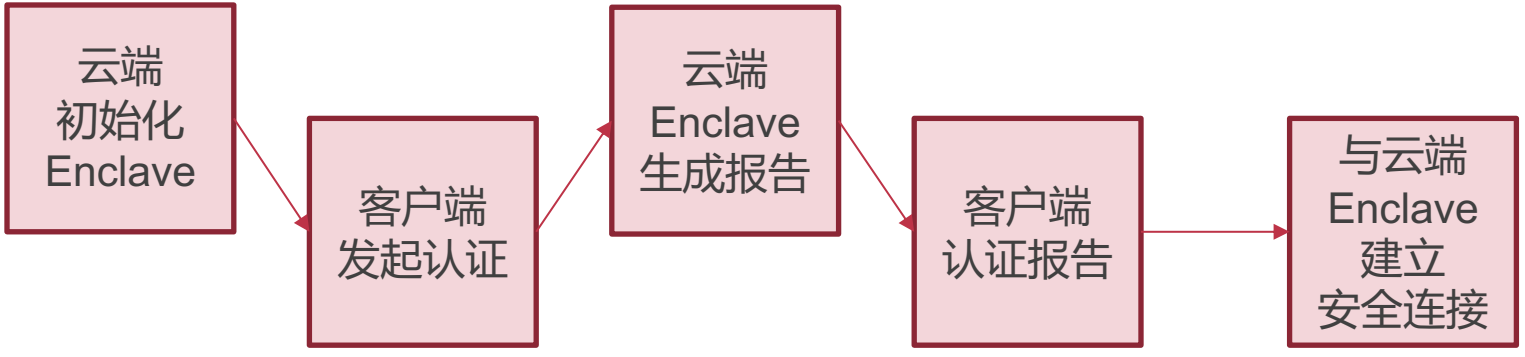
常量时间（Constant Time）算法

- **算法的运行时间与输入无关**
  - 无法通过运行时间得到与输入相关的任何信息
  - 代码执行没有分支跳转
- **常见的实现方法：cmov**
  - Conditional MOV
- **缺点：计算变得更慢**
  - 需要做两份运算

```
/* 传统实现方式 */
if (secret == 0)
    x = a + b;
else
    x = a / b;

/* 常量时间实现方式 */
v1 = a + b;
v2 = a / b;
cond = (secret == 0)
x = cmov(cond, v1, v2)
```

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



控制系统复杂性

- **Enclave的抽象是一种简化**
  - 对威胁模型和信任关系的简化
    - 例如：Intel SGX将对软硬件环境的信任规约到对Intel的信任
  - 这种简化有可能带来新的问题：Single-point of Failure
- **Enclave的主要技术**
  - 保护技术：基于权限的隔离与基于加密的控制
  - 远程验证：对密钥的管理

Enclave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

2020，在线的一学期

祝大家取得好成绩！