

# 内存管理

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

# 版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

# Review: 外核架构 ( Exokernel )

- **Exokernel 不提供硬件抽象**
  - "只要内核提供抽象，就不能实现性能最大化"
  - 只有应用才知道最适合的抽象 ( end-to-end 原则 )
- **Exokernel 不管理资源，只管理应用**
  - 负责将计算资源与应用的绑定，以及资源的回收
  - 保证多个应用之间的隔离

内核态 : Exokernel

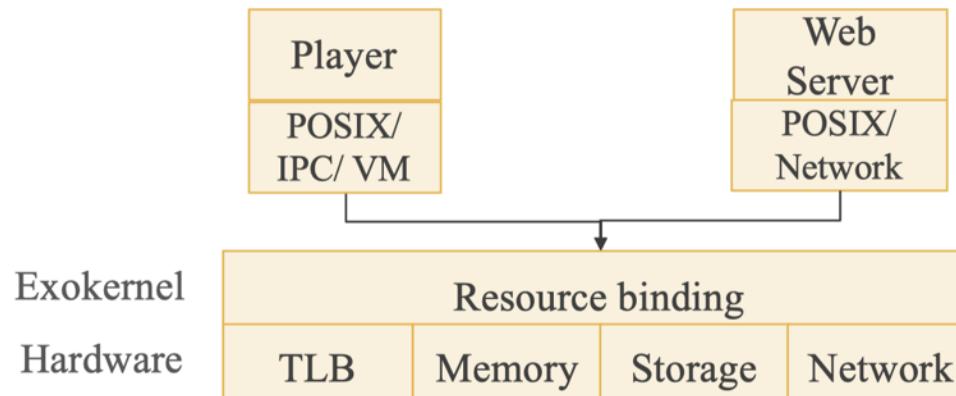
用户态 : libOS

- **回顾：操作系统 = 服务应用 + 管理应用**

# Review: Exokernel + LibOS

- 库OS ( LibOS )

- 策略与机制分离：将对硬件的抽象以库的形式提供
- 高度定制化：不同应用可使用不同的LibOS，或完全自定义
- 更高性能：LibOS与应用其他代码之间通过函数调用直接交互



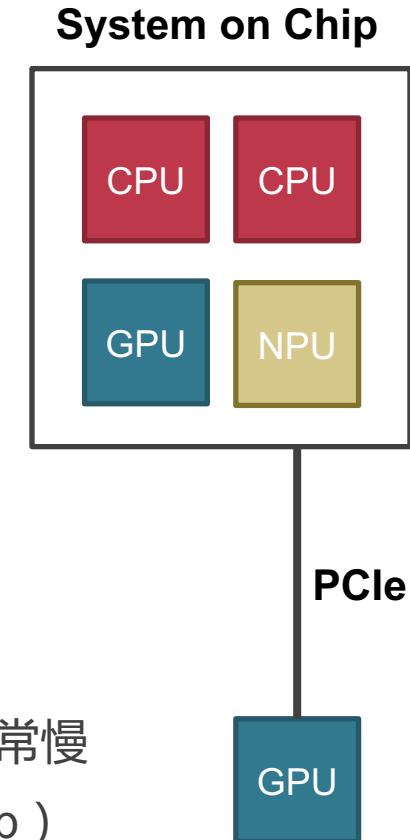


# 多内核/复内核 ( MULTI-KERNEL )

# 多内核/复内核（ Multikernel ）

- **背景：多核与异构**

- OS内部维护很多共享状态
  - Cache一致性的保证越来越难
  - 可扩展性非常差，核数增多，性能不升反降
- GPU等设备越来越多
  - 设备本身越来越智能——设备有自己的CPU
  - 通过PCIe连接，主CPU与设备CPU之间通信非常慢
  - 通过系统总线连接，异构SoC（ System on Chip ）



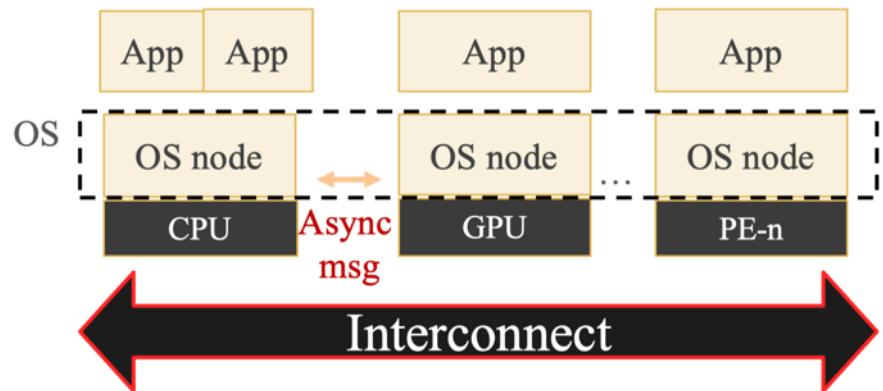
# Multikernel的设计

- **Multikernel的思路**

- 默认的状态是划分而不是共享
  - 维持多份状态的copy而不是共享一份状态
  - 显式的核间通信机制

- **Multikernel的设计**

- 在每个core上运行一个小内核
    - 包括CPU、GPU等
  - OS整体是一个分布式系统
  - 应用程序依然运行在OS之上



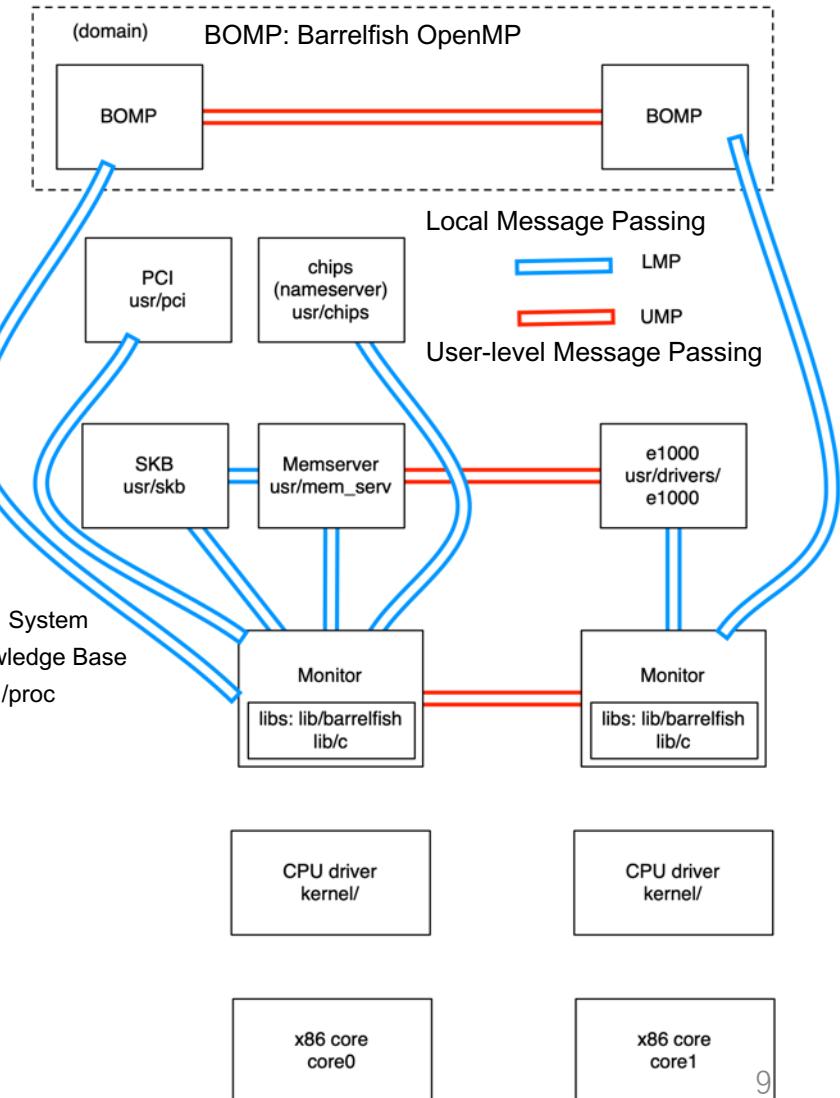
# Barrelfish Multikernel

- **Barrelfish操作系统**
  - 来自ETH Zurich和微软研究院
  - 支持异构CPU
  - 在CPU核与节点之间提供通用异构消息抽象
  - 大约10,000行C，500行汇编代码



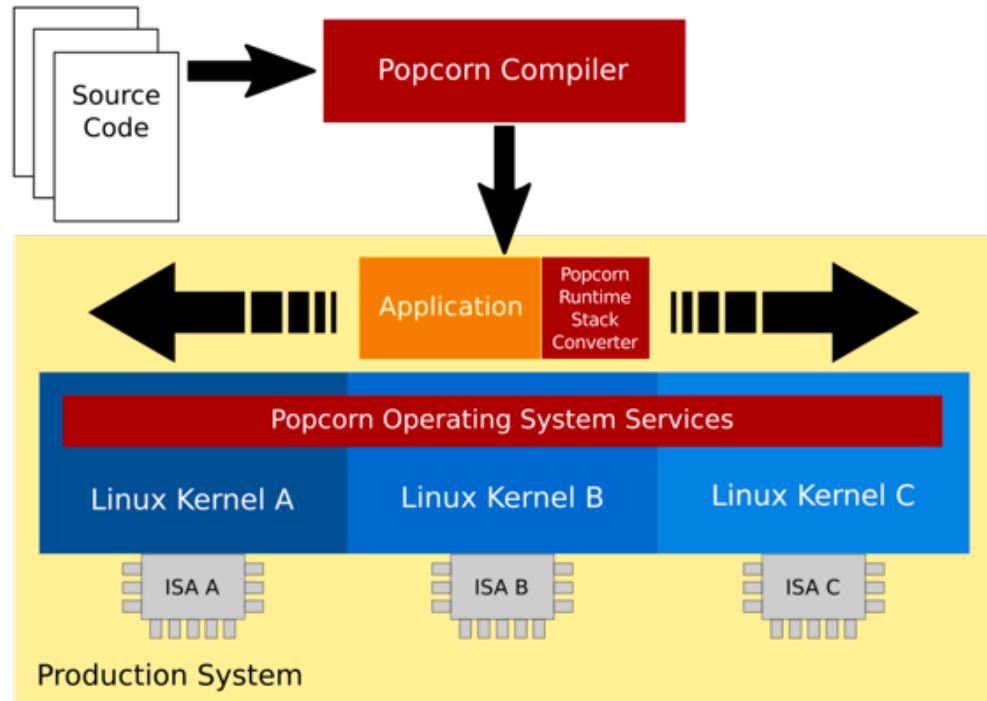
# Barrelfish Structure

- 内核：每个core对应一个
  - 类似"CPU驱动"，适应不同CPU
  - 负责执行系统调用，处理中断/异常
  - 事件触发，单线程，不可中断
  - 内核调度并运行"Dispatcher"
- Dispatcher
  - 类似线程
  - 多个Dispatcher组成一个Domain
- Domain
  - 类似进程



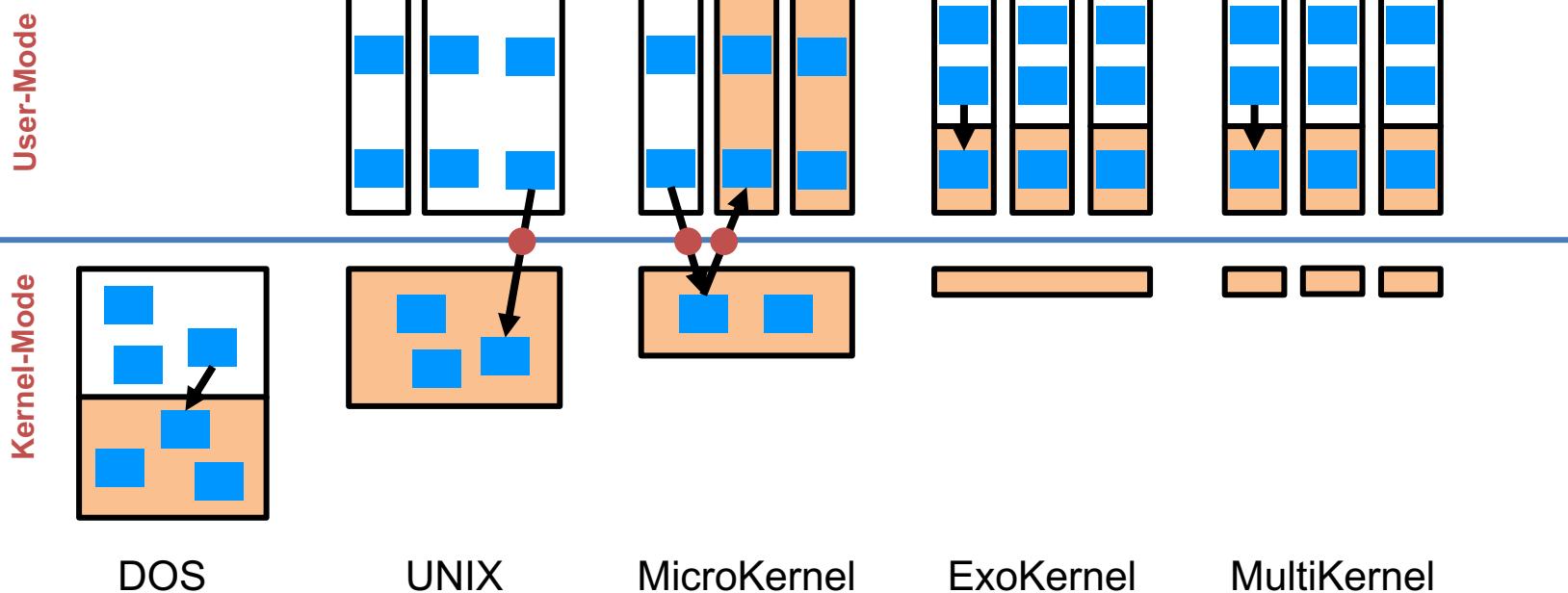
# Popcorn Linux

- 支持异构体系结构
  - ARM、x86等
- 多个Linux内核副本
  - 一套代码编译不同副本
  - 不同ISA不同副本
  - 多个副本同时向上提供OS服务



# 不同操作系统架构的对比

OS  
App  
Logic



# 操作系统结构的演进与生态

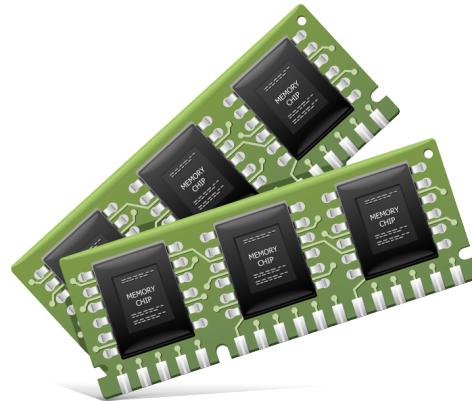
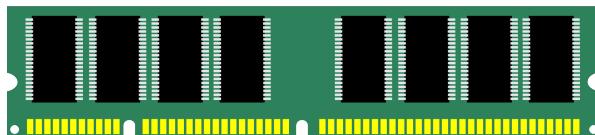
- 系统软件需要一条演进之路
  - 尽可能集成现有的POSIX API/Linux ABI
  - 避免棘手的系统调用（如fork）
  - 避免不可扩展的POSIX API
- 系统软件一直在不断演化
  - 例：Linux Userspace I/O (UIO)，向微内核近了一步
  - 单节点下也存在更多的分布式、低时延的可编程设备
  - 非易失性内存的出现可能推动存储层次在OS中的完全改革



# 物理内存

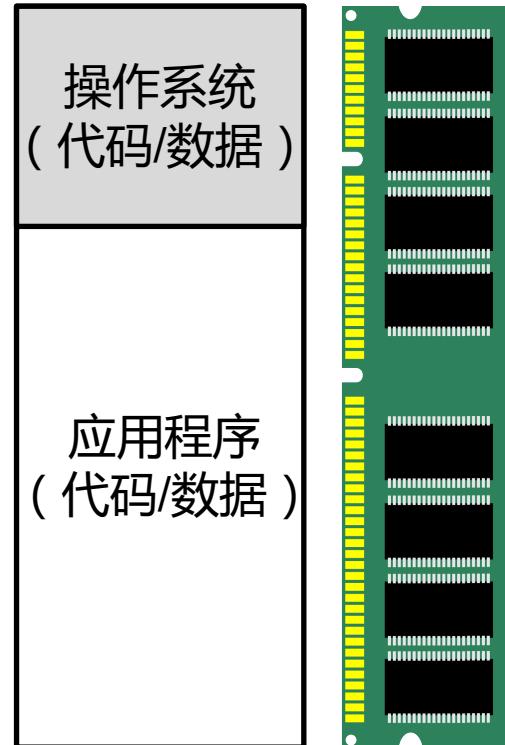
# 物理内存

- 常说的"内存条"就是指物理内存
- 数据从磁盘中加载到物理内存后，才能被CPU访问
  - 操作系统的代码和数据
  - 应用程序的代码和数据



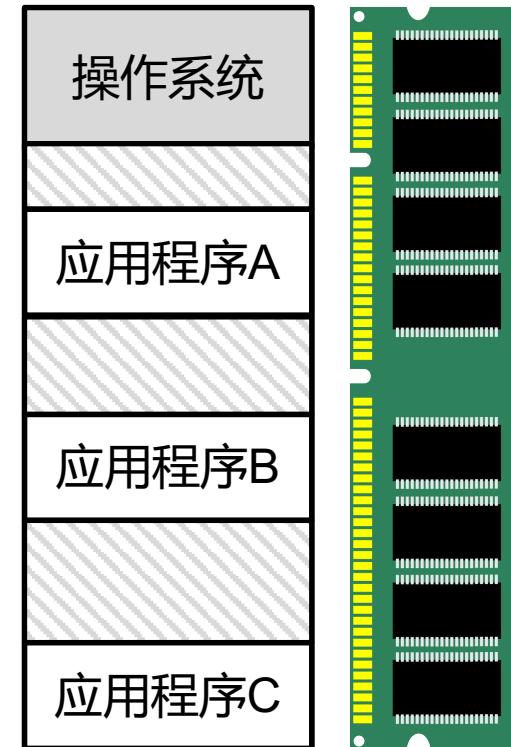
# 最早期的计算机系统

- 硬件
  - 物理内存容量小
- 软件
  - 单个应用程序 + (简单) 操作系统
  - 直接面对物理内存编程
  - 各自使用物理内存的一部分



# 多重编程时代

- **多用户多程序**
  - 计算机很昂贵，多人同时使用（远程连接）
- **分时复用CPU资源**
  - 保存恢复寄存器速度很快
- **分时复用物理内存资源**
  - 将全部内存写入磁盘开销太高
- **同时使用、各占一部分物理内存**
  - 没有安全性（隔离性）



如何让OS与不同的应用程序都高效又安全地使用物理内存资源？

# IBM 360的内存隔离：Protection Key

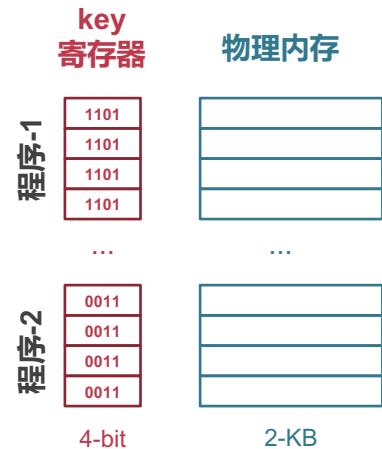
- Protection key机制

- 内存被划分为一个个大小为2KB的内存块 ( Block )
- 每个内存块有一个4-bit的key，保存在寄存器中
- 1MB内存需要256个保存key的寄存器，占256-Byte
  - 内存变大怎么办？需要改CPU以增加key寄存器...
- 每个进程对应一个key
  - CPU用另一个专门的寄存器，保存当前运行进程的key
  - 不同进程的key不同
- 一个进程访问一块内存时
  - CPU检查进程的key与内存的key是否匹配



# Protection Key机制的挑战

- **应用加载与隔离**
    - 不同应用被加载到不同的物理地址段
    - 不同应用的key不同，以保证隔离
  - **问题**
    - 同一个二进制文件，程序-1加载到0000-1000地址段，程序-2加载到5000-6000地址段
    - "JMP 42"，程序-1能执行，程序-2会出错
  - **解决方法**
    - 代码中所有地址在加载过程中都需要增加一个偏移量，如改为："JMP 5042"
    - 新的问题：
      - 加载过程变得更慢
      - 如何在代码中定位所有的地址？如 "MOV REG1, 42"，其中的42是地址还是数据？



# 使用物理地址的缺点

- **物理地址对应用是可知的，导致：**
  - 一个应用会因其他应用的加载而受到影响
  - 一个应用可通过自身的内存地址，猜测出其他应用的加载位置
- **是否可以让应用看不见物理地址？**
  - “看不见”，指应用对物理地址不可知
  - 一个进程不用关心其他进程占了什么地址，不受其他进程的影响
  - 看不见其他进程的信息，带来更强的隔离能力

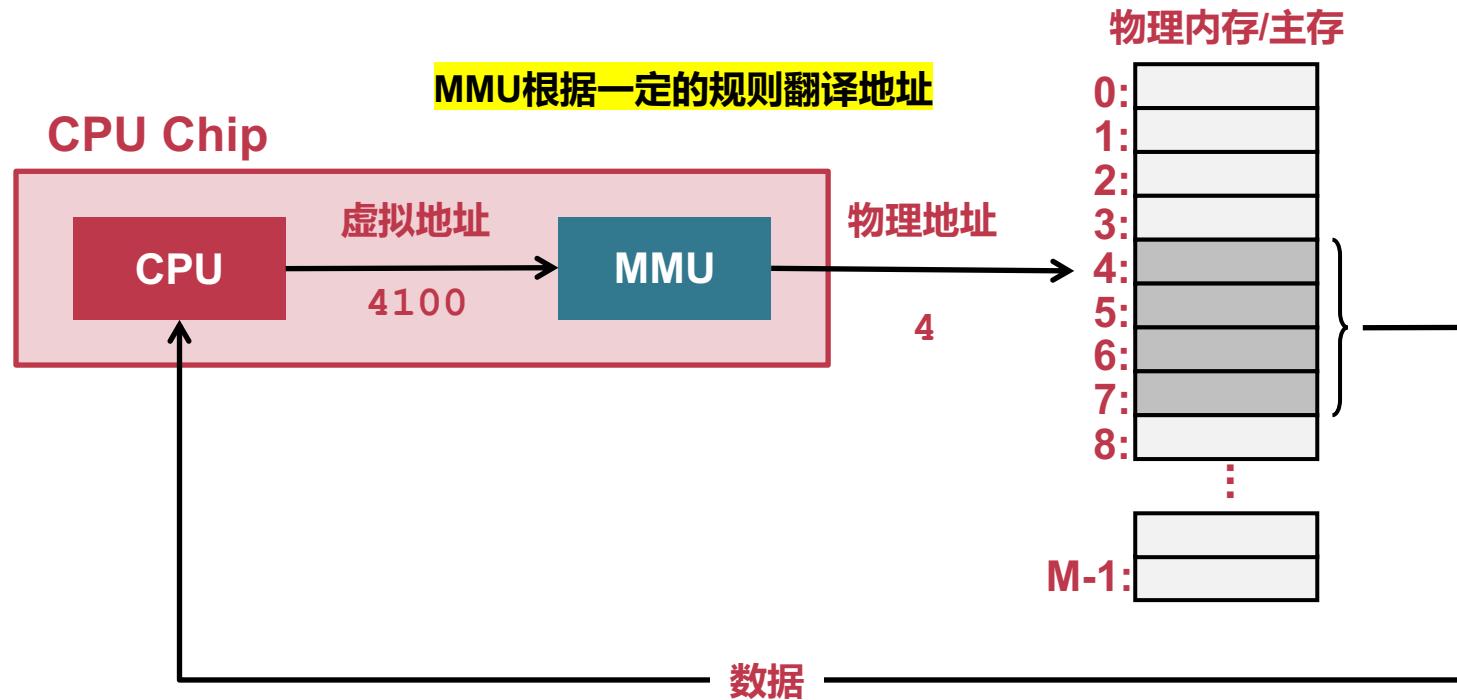
# 虚拟内存抽象

- "*All problems in computer science can be solved by another level of indirection*"      --- David Wheeler
- **以虚拟内存抽象为核心的内存管理**
  - CPU : 支持虚拟内存功能，新增了虚拟地址空间
  - 操作系统 : 配置并使能虚拟内存机制
  - 所有软件 ( 包括OS ) : 均使用虚拟地址，无法直接访问物理地址

# 虚拟地址

- **虚拟内存抽象下，程序使用虚拟地址访问主存**
  - 虚拟地址会被硬件"自动地"翻译成物理地址
- **每个应用程序拥有独立的虚拟地址空间**
  - 应用程序认为自己独占整个内存
  - 应用程序不再看到物理地址
  - 应用加载时不用再为地址增加一个偏移量

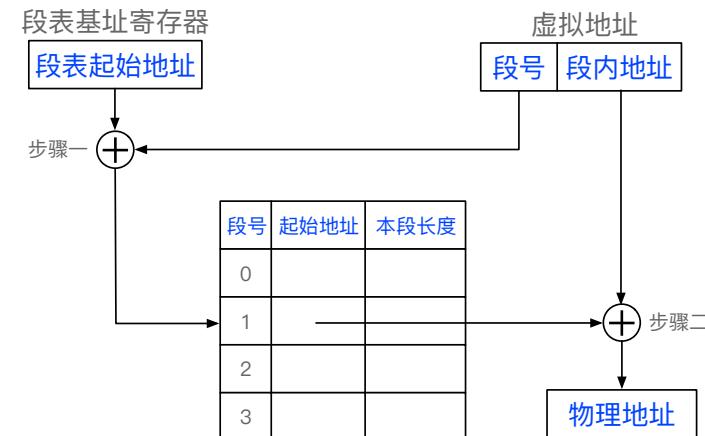
# 地址翻译过程



翻译规则取决于虚拟内存采用的组织机制，包括：分段机制和分页机制

# 分段机制

- **虚拟地址空间分成若干个不同大小的段**
  - 段表存储着分段信息，可供MMU查询
  - 虚拟地址分为：段号 + 段内地址（偏移）
- **物理内存也是以段为单位进行分配**
  - 虚拟地址空间中相邻的段，对应的物理内存可以不相邻
- **存在问题**
  - 分配的粒度太粗，外部碎片
  - 段与段之间留下碎片空间，降低主存利用率



# 分页机制

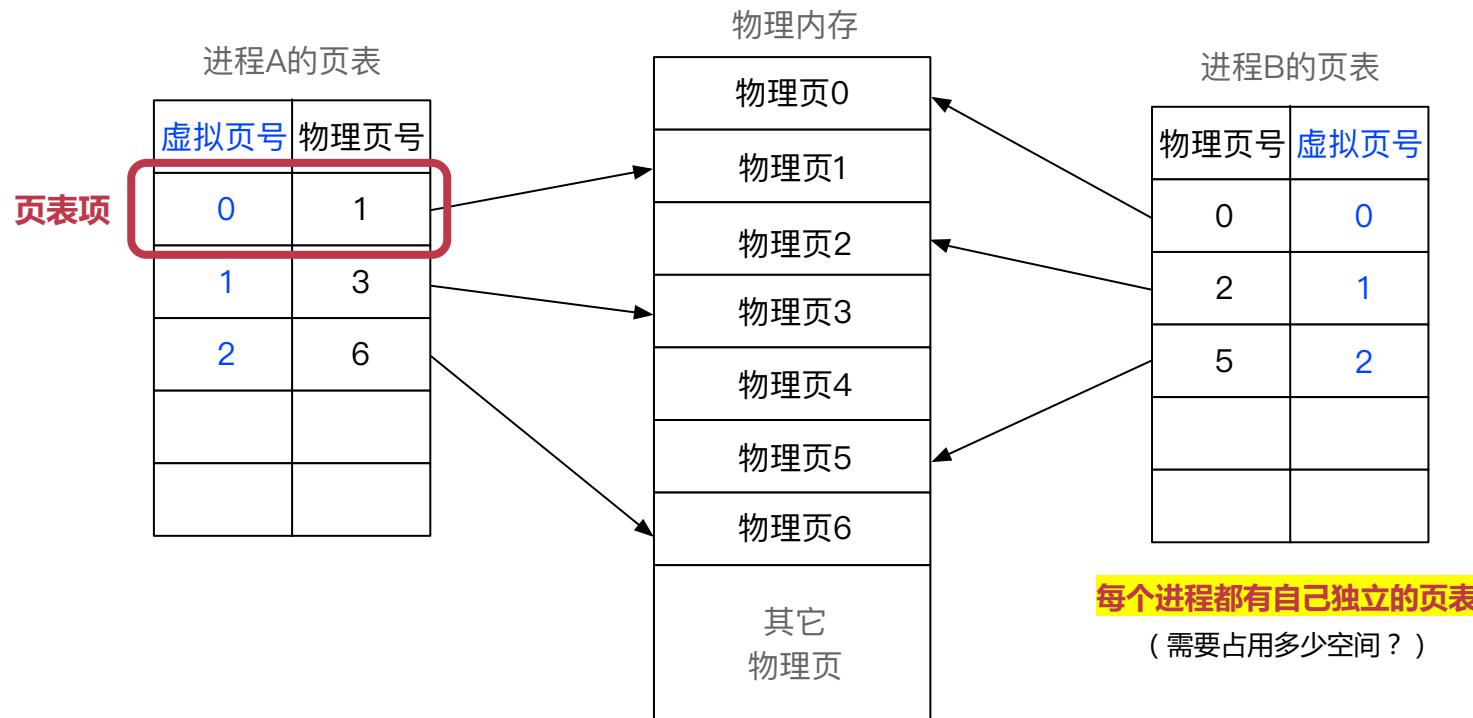
- **更细粒度的内存管理**
  - 物理内存也被划分成连续的、等长的物理页
  - 虚拟页和物理页的页长相等
  - 任意虚拟页可以映射到任意物理页
  - 大大缓解分段机制中常见的外部碎片
- **虚拟地址分为：**
  - 虚拟页号 + 页内偏移
- **主流CPU均支持分页机制，可替换分段机制**

进程虚拟地址空间



# 页表：分页机制的核心数据结构

- 页表包含多个页表项，存储虚拟页到物理页的映射



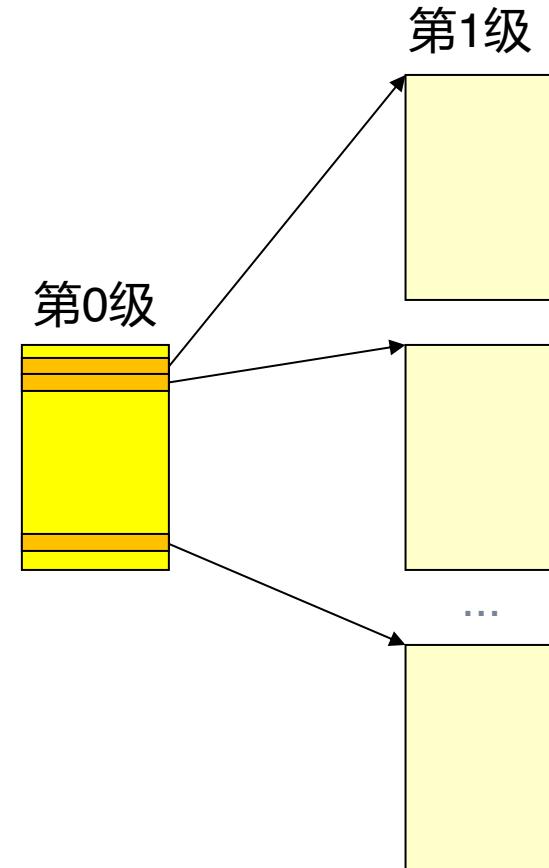
# 单级页表的问题

- 若使用单级页表结构，一个页表有多大？

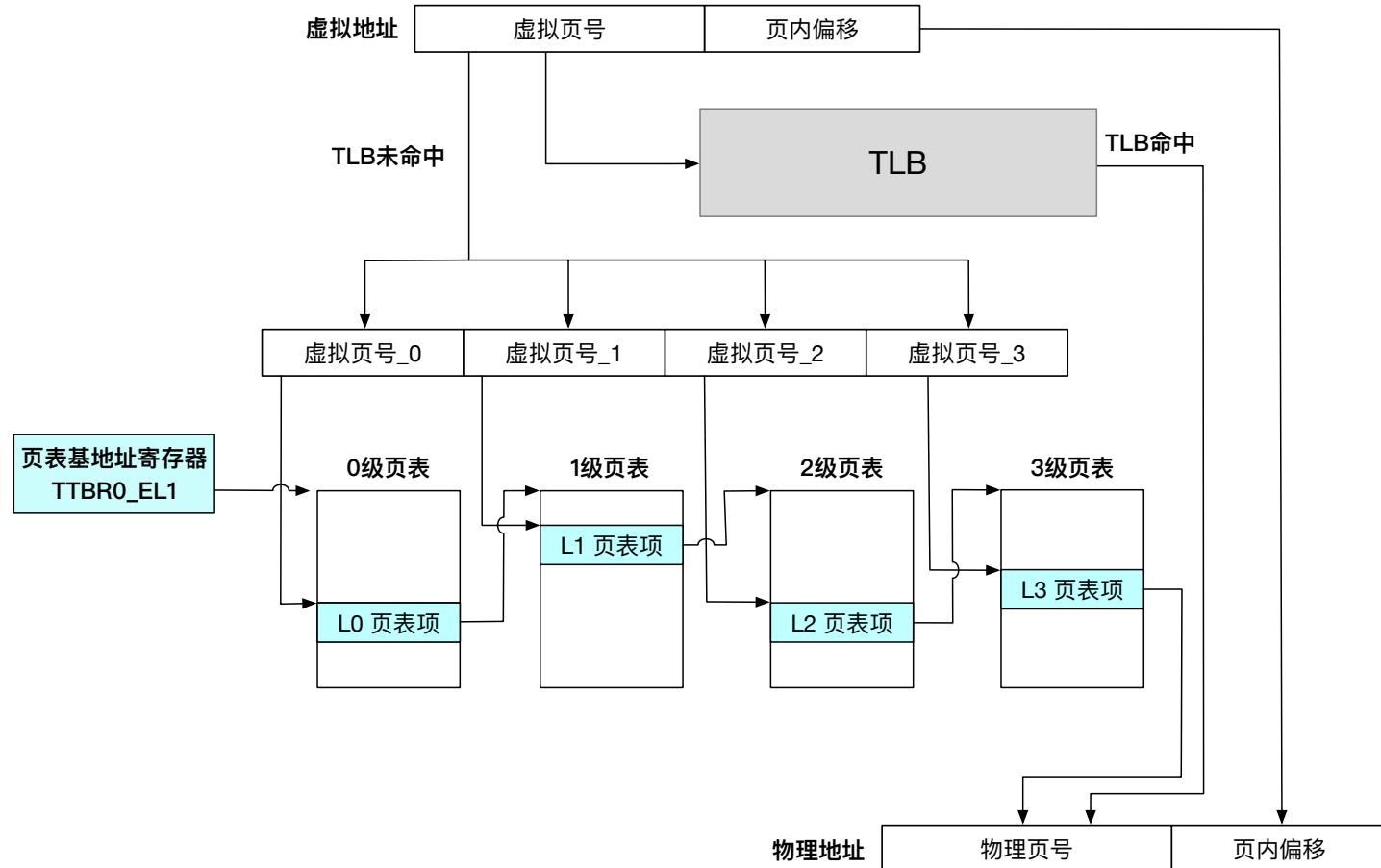
- 32位地址空间，页4K，页表项4B，  
页表大小： $2^{32} / 4K * 4 = 4MB$
- 64位地址空间，页4K，页表项8B，  
页表大小： $2^{64} / 4K * 8 = 33,554,432 GB$

- 使用多级页表减少空间占用

- 若某级页表中的某条目为空，那么对应的下一级页表无需存在
- 实际应用的虚拟地址空间大部分都未被使用，因此无需分配页表
- 减少空间的原因：允许页表中出现“空洞”



# AARCH64的4级页表



# 64位虚拟地址翻译

- 「63:48」
  - 必须全是0或者全是1（一般应用程序地址选择0）
  - 也意味着虚拟地址空间大小最大是 $2^{48}$ 字节
- 「47:39」 0级页表索引
- 「38:30」 1级页表索引
- 「29:21」 2级页表索引
- 「20:12」 3级页表索引
- 「11:0」 页内偏移

## 页表基地址寄存器 ( Translation Table Base Register )

- AARCH64 有两个
  - TTBR0\_EL1 & TTBR1\_EL1
  - 根据虚拟地址第63位选择，若为0则选择TTBR\_EL0
  - 通常（以Linux为例）：  
应用程序使用TTBR0\_EL1，  
操作系统使用TTBR1\_EL1
- 对比 x86\_64
  - 只有一个CR3寄存器

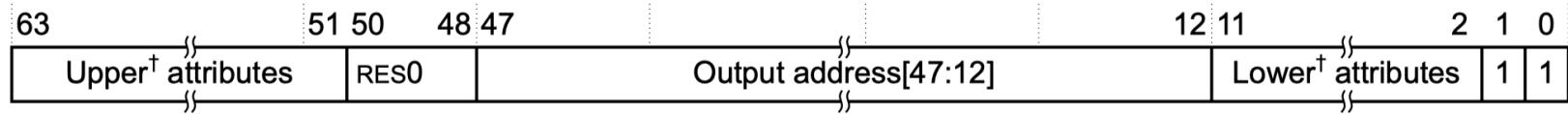
# 页表使能 ( Enabling )

- CPU启动流程
  - 上电后默认进入物理寻址模式
  - 系统软件配置控制寄存器，使能页表，进入虚拟寻址模式
- AARCH64
  - **SCTLR\_EL1** ( System Control Register, EL1 )
  - 第0位 ( M位 ) 置1，即在EL0和EL1权限级使能页表
- 对比x86\_64
  - CR0，第31位 ( PG位 ) 置1，使能页表

# 页表页

- **每级页表有若干离散的页表页**
  - 每个页表页占用一个物理页
- **第 0 级（顶层）页表有且仅有一个页表页**
  - 页表基地址寄存器（TTBR）存储的就是该页的物理地址
- **每个页表页中有 512 个页表项**
  - 每项为 8 个字节， $4096/8$ ，用于存储物理地址和权限

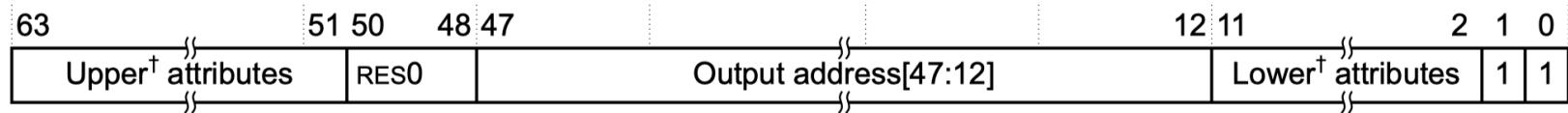
# AARCH64页表项



- **第3级页表页中的页表项**

- 第0位 ( valid位 ) 表示该项是否有效
- 第1位必须是1
- Upper attributes包括：
  - 第54位 ( XN位 ) 为1表示EL0不能执行 ( eXecution Never )
  - 第53位 ( PXN位 ) 为1表示EL1不能执行
  - 第51位 ( DBM位 ) , 类似于x86\_64中的dirty bit

# AARCH64页表项

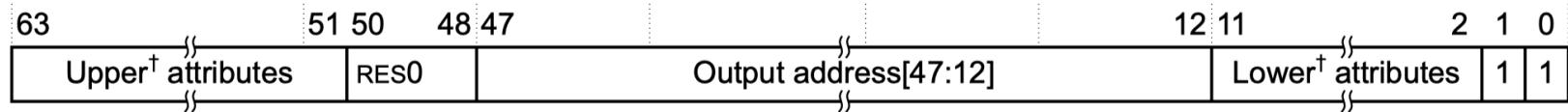


- 第3级页表页中的页表项

- Lower attributes 包括：
  - 第7位-第6位表示读写权限位AP[2:1]

AP[2:1]	Access from higher Exception level	Access from EL0
00	Read/write	None
01	Read/write	Read/write
10	Read-only	None
11	Read-only	Read-only

# AARCH64页表项



- **第3级页表页中的页表项**

- Lower attributes 包括：
  - 第10位 (AF位) 是Access Flag，若设为0则访问时发生异常
    - 可供软件追踪内存访问情况
  - 第9位-第8位是Shareability field (用于核间、核与设备间的共享)
  - 第4位-第2位是AttrIdx[2:0]，表示内存类型
    - Normal (其cacheable属性由TCR\_EL1指定)
    - Device (设为non-cacheable，设备内存，又再细分四种)

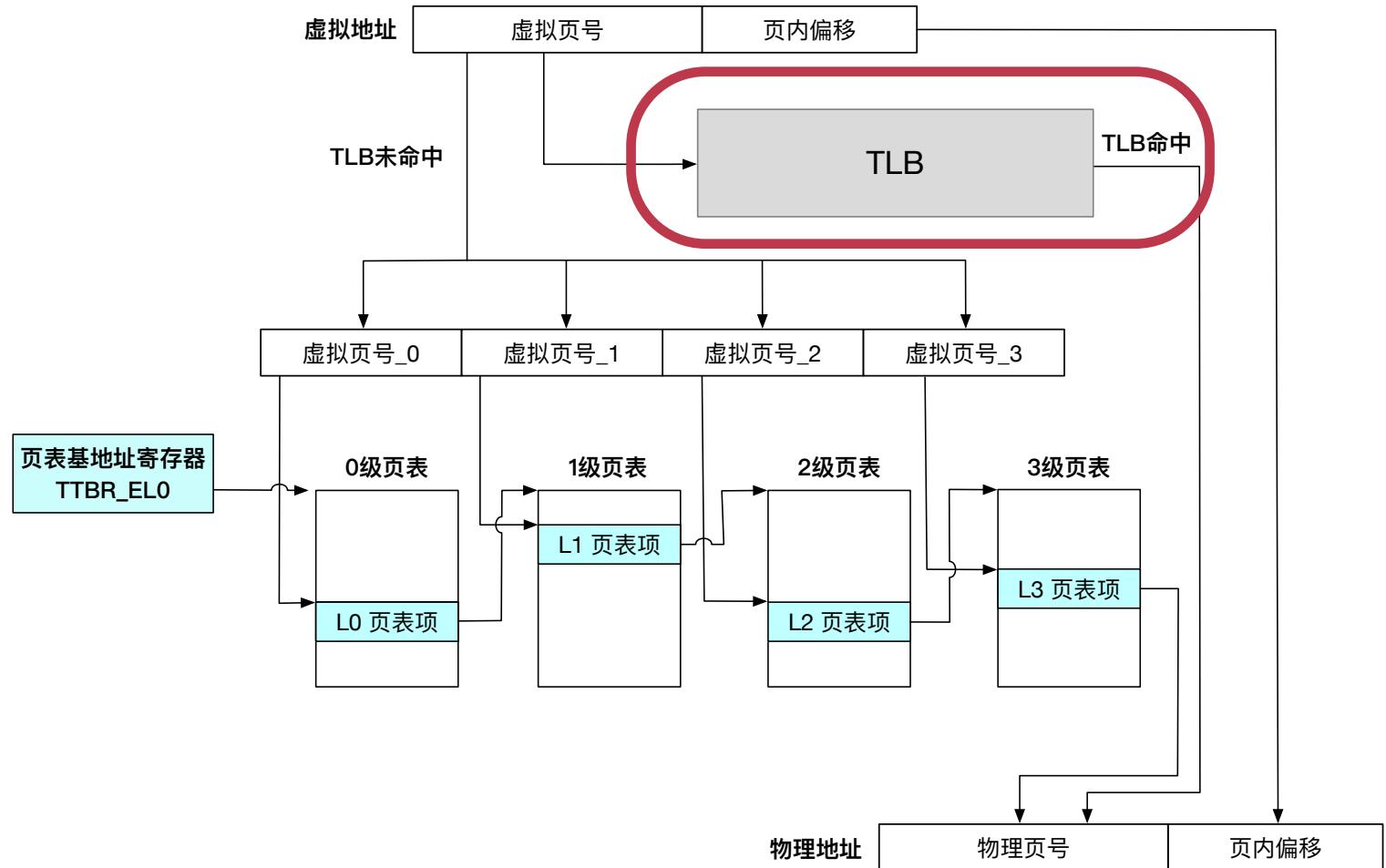
# ARM的Cache Lockdown特性

- Cache lockdown寄存器
  - 可以配置部分Cache不被evict，使数据一直驻留在CPU内部
  - 非统一标准，取决于具体的实现（也可不实现）
- Cache lockdown用途
  - 提高性能：可保证访问部分重要数据永远cache hit
    - 硬件的Cache替换策略不够完美
  - 提高安全性：限制部分数据永远不离开CPU
    - 若数据量大于Cache容量，可加密后离开CPU

# （多级）页表不是完美的

- 多级页表的设计是典型的用时间换空间的设计
  - 能够减小页表所占空间
  - 但是增加了访存次数（逐级查询，级数越多越慢）
- *Tradeoff* 是计算机中经典而永恒的话题
- 如何降低地址翻译的开销？

# TLB：地址翻译的加速器



# 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管理：应该缓存哪些映射？

- 在AArch64和x86\_64中，TLB由硬件管理
  - 硬件的简单替换策略为什么有效？（时空局部性）
- 在一些体系结构（如MIPS）中，TLB由软件进行管理
  - 即“software TLB”
  - TLB未命中时触发异常
  - 软件的优势在于灵活性

# 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**
  - 分别保存在TTBR0\_EL1和TTBR1\_EL1
  - 系统调用过程不用切换
- **x86\_64上只有唯一的基址寄存器 ( CR3 )**
  - 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 VMALLE1IS
  - 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
    - KPTI: Kernel Page Table Isolation
    - 即内核与应用不共享页表，防御Meltdown攻击 <https://meltdownattack.com/>
- **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

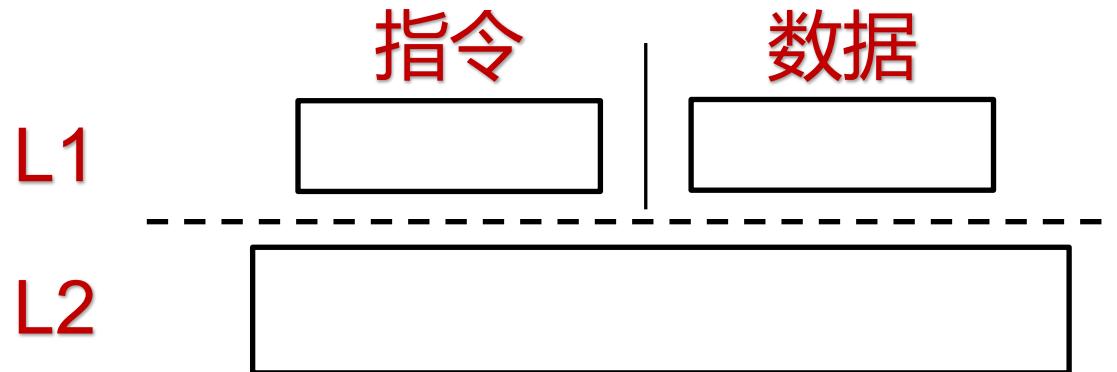
- **使用了ASID之后**
  - 切换页表不再需要刷新TLB
  - 修改页表映射后，仍需刷新TLB
- **在多核场景下**
  - 需要刷新其它核的TLB吗？
  - 如何知道需要刷新哪些核？
  - 怎么刷新其他核

# TLB与多核

- **需要刷新其它核的TLB吗？**
  - 一个进程可能在多个核上运行
- **如何知道需要刷新哪些核？**
  - 操作系统知道进程调度信息
- **怎么刷新其他核？**
  - AARCH64: 可在local CPU上刷新其它核TLB
    - TLBI ASIDE1IS
  - x86\_64: 发送IPI中断某个核，通知它主动刷新

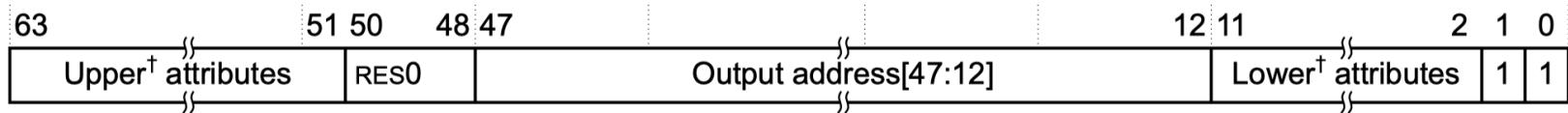
# TLB结构简介

- 回顾：ICS中学习的分级cache结构（L1/L2/L3）
- TLB设计通常也采用分级结构（以AArch64为例）



- 思考：为什么采用分级结构？

# 全局TLB



- **再看（第3级）页表项**
  - Lower attributes的第11位是nG ( not Global ) 位
    - nG == 0: 相应TLB缓存项对所有进程有效
    - nG == 1: 仅对特定进程 ( ASID ) 有效
- **思考：为什么需要nG位/全局TLB？**



## 按需分配与换页

# 物理内存的超售(Over-commit)和按需分配

- **情景1:**
  - 两个应用程序各自需要使用 3GB 的物理内存
  - 整个机器实际上总共只有 4GB 的物理内存
- **情景2:**
  - 一个应用程序申请预先分配足够大的（虚拟）内存
  - 实际上其中大部分的虚拟页最终都不会用到

# 换页机制 ( Swapping )

- **换页的基本思想**
  - 将物理内存里面存不下的内容放到磁盘上
  - 虚拟内存使用不受物理内存大小限制
- **如何实现**
  - 磁盘上划分专门的Swap分区
  - 在处理缺页异常时，触发物理内存页的换入换出

# 缺页异常 ( Page Fault )

- 缺页异常
  - CPU控制流传递
  - 提前注册缺页异常处理函数
- x86\_64
  - 异常号 #PF ( 13 ) , 错误地址在CR2
- AARCH64
  - 触发 ( 通用的 ) 同步异常 ( 8 ) ,
  - 根据ESR信息判断是否缺页 , 错误地址在FAR\_EL1

# 按需分配中的权衡

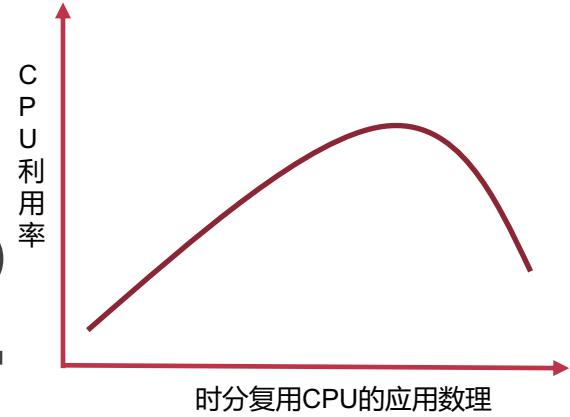
- 优势：节约内存资源
- 劣势：缺页异常导致访问延迟增加
- 如何取得平衡？
  - 应用程序访存具有时空局部性
  - 在缺页异常处理函数中采用预取（Prefetching）机制
  - 即节约内存又能减少缺页异常次数

# 页替换策略

- 常见的替换策略
  - 随机替换、FIFO、LRU/MRU、Clock Algorithm、...
- 替换策略评价标准
  - 缺页发生的概率（参照理想但不能实现的OPT策略）
  - 策略本身的性能开销
    - 如何高效地记录物理页的使用情况？
      - Recap：上节课说到的页表项中Access/Dirty Bits
- Thrashing Problem

# 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，则需要将该集合保持在内存中
- 工作集模型：
  - all-or-nothing模型
  - 进程工作集要不都在内存中，否则全都换出
  - 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）
    - $\text{Age} = \text{当前时间} - \text{上次使用时间}$
    - 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	
2010	1
2000	1
1970	0
1990	0

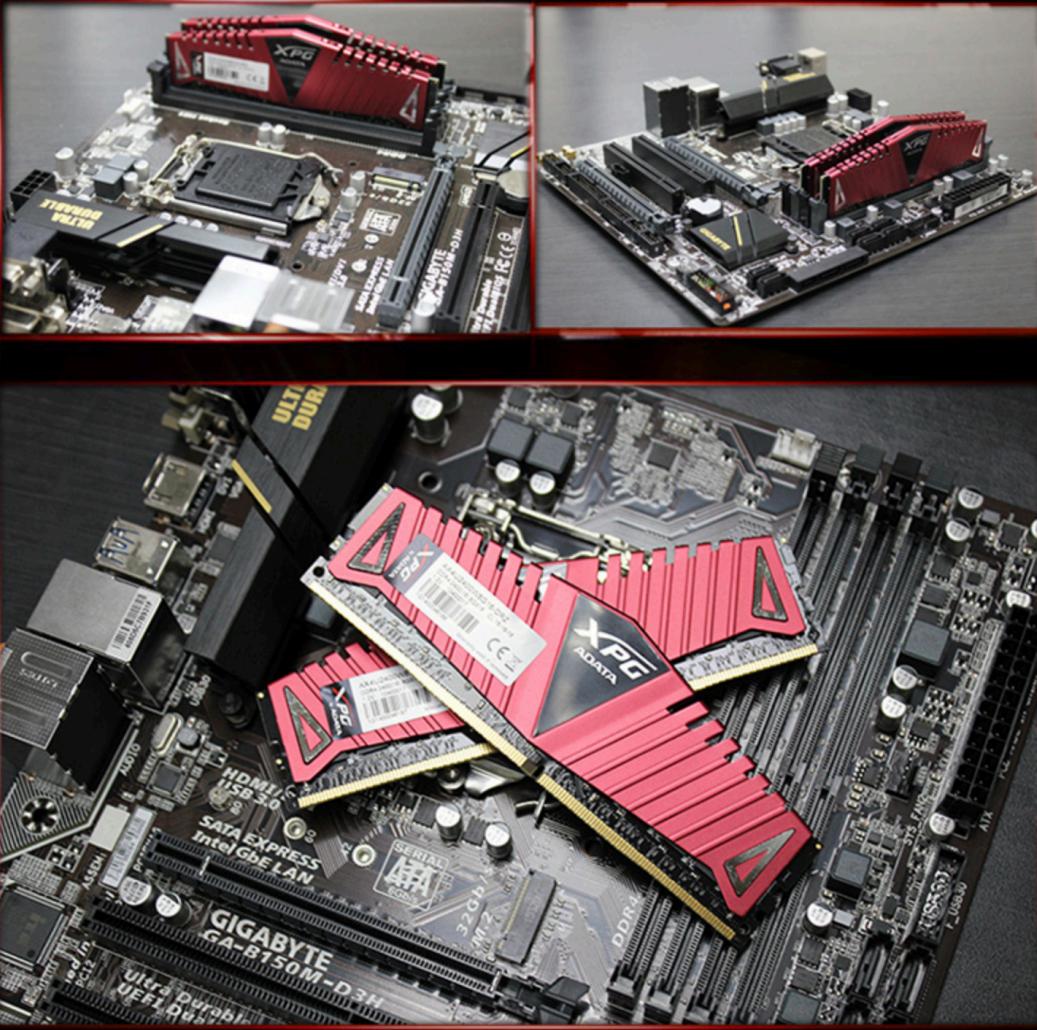
上次使用时间      访问位

# 21世纪第三个十年再看换页

- **今天换页还需要吗？**
  - 物理内存容量增大、价格下降
  - 服务器的内存通常到达上百GB，甚至更大
  - 非易失性内存的出现会在存储架构方面带来新的革命
    - 传统存储层次：register – cache – memory – disk/SSD
    - NVM的出现：取代SSD？取代memory
      - 让memory变成L4 cache？



# 物理内存管理



“内存条”

# 物理内存结构

Channel

DIMM

Rank

Chip

Rank 1 with 8 chips



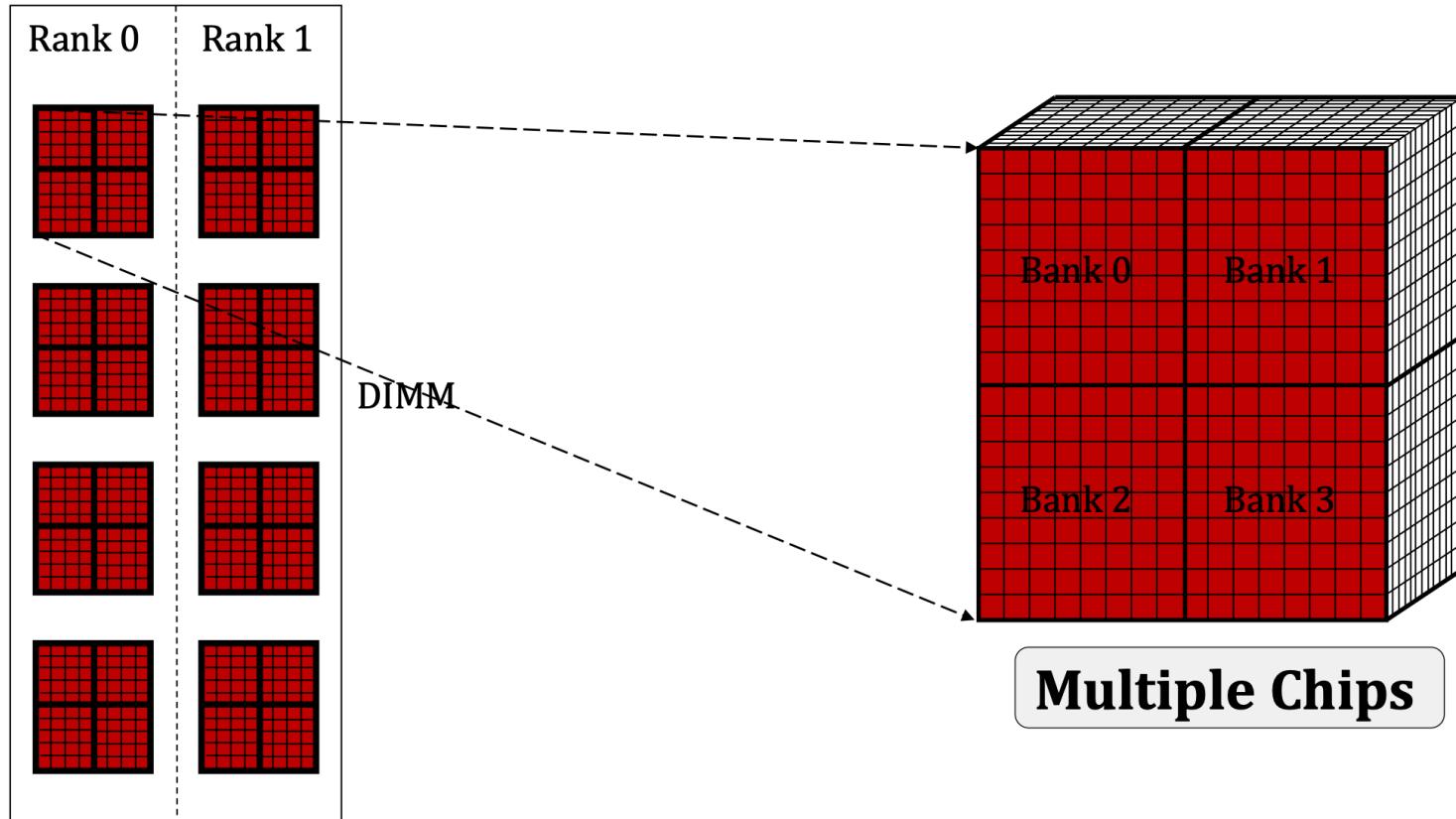
Rank 0 with 8 chips

Bank

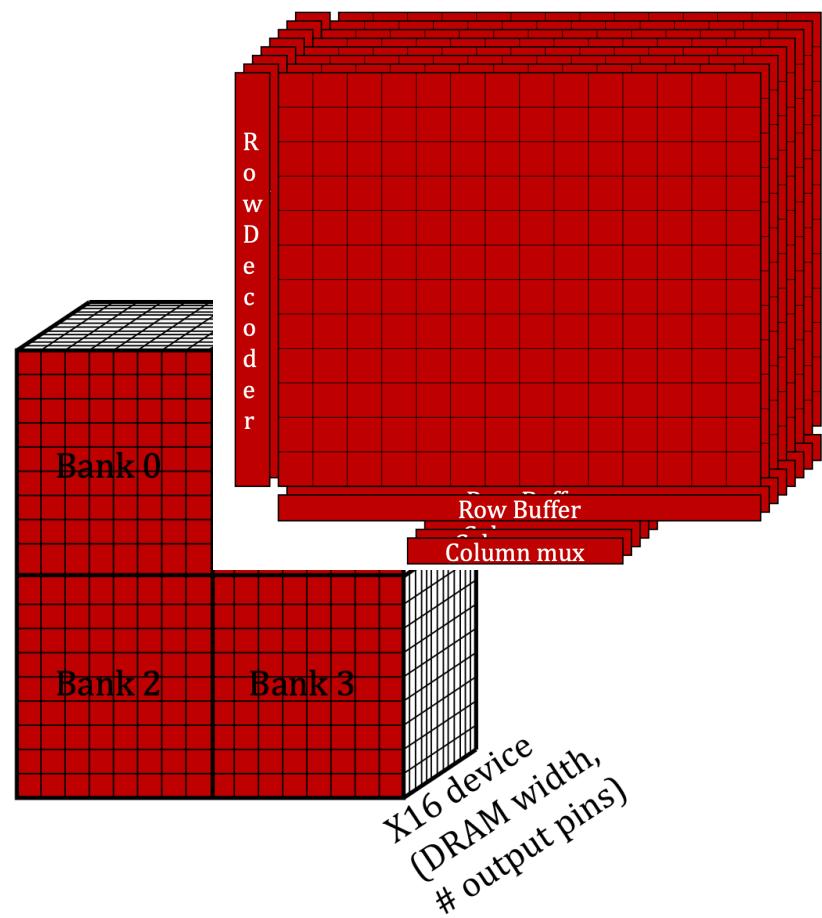
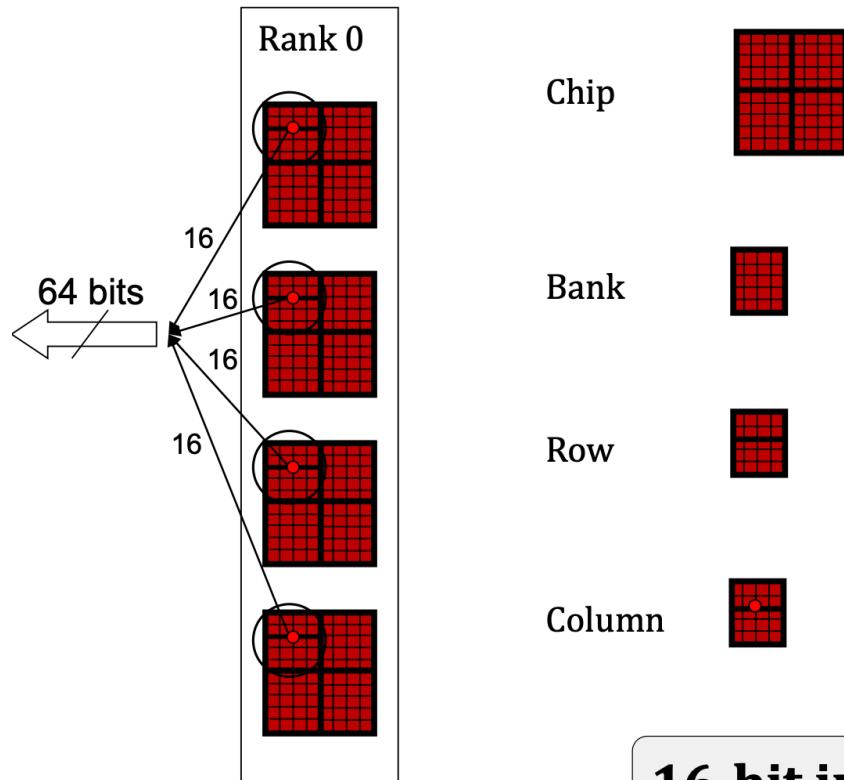
Row

Column

# 物理内存结构



# 物理内存结构



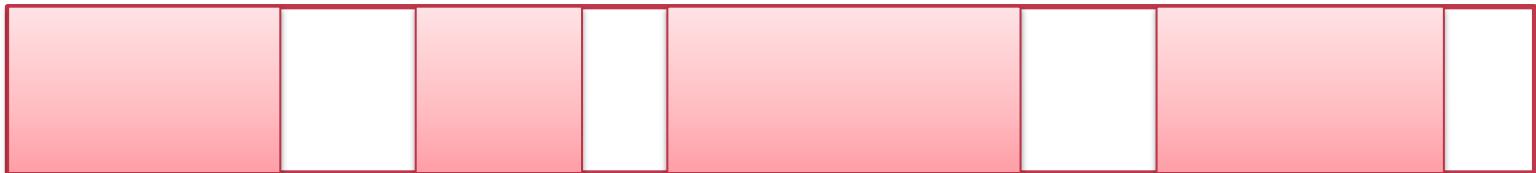
**16-bit interface: 16 bits from each chip in one go**

# 内存控制器：Memory Controller

- 为操作系统提供了易用的物理内存抽象
  - 逐字节可寻址的“大数组”
  - 屏蔽了硬件细节
  - 操作系统的物理内存管理变得简单

# 物理内存管理中的碎片问题

- 外部碎片（空闲的但不连续，无法被使用）



- 内部碎片（分配大小大于实际需要）

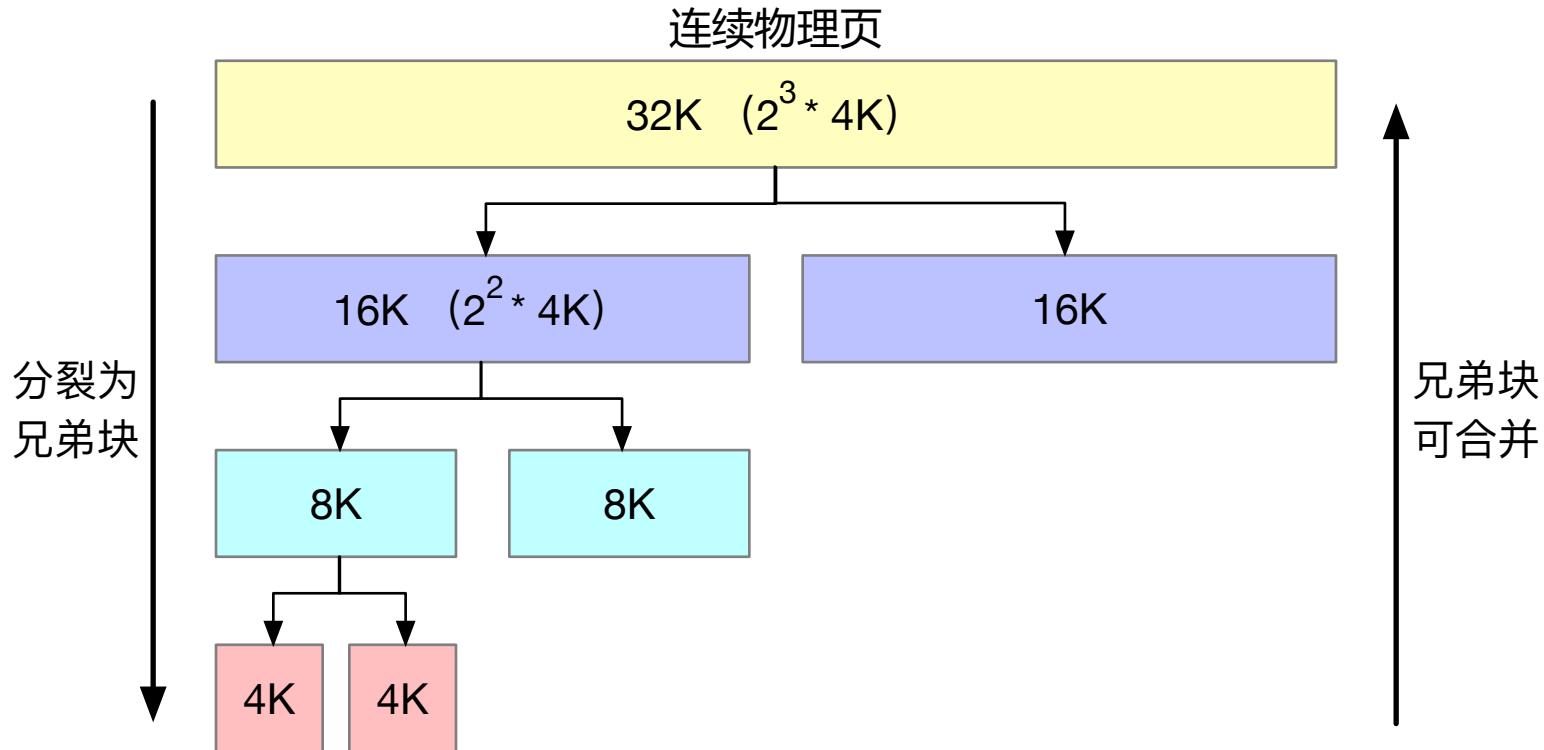


# 物理内存管理的评价指标

- **内存资源利用率**
  - 外部碎片和内部碎片
- **分配速度**
  - 复杂的算法可以更好地解决碎片问题
  - 但是内存分配操作的性能同样重要
- **Tradeoff?**

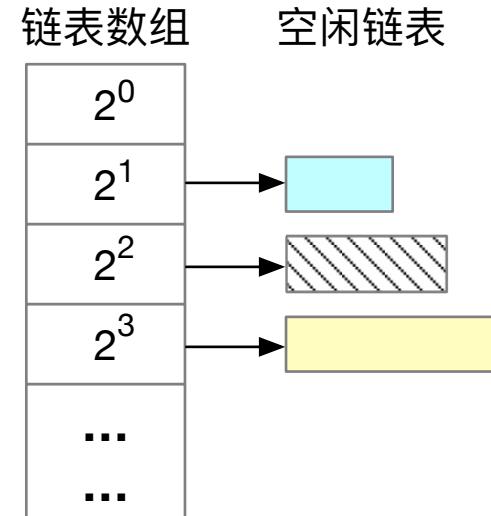
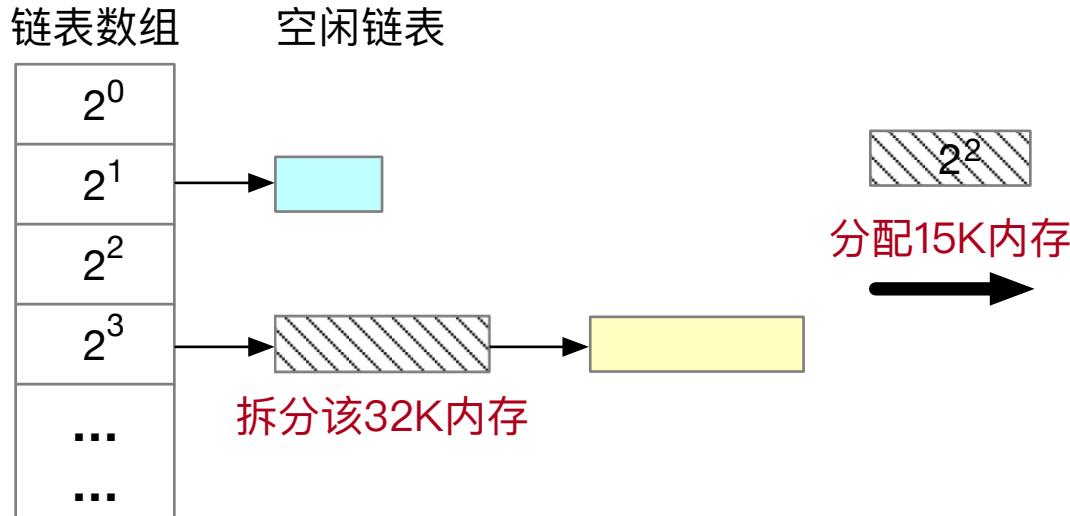
# 物理内存管理之buddy system

- 伙伴系统（能避免外部碎片吗？）



# 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

# 建立在伙伴系统之上的分配器

- **SLAB分配器家族 (Linux)**
  - SLAB分配器
  - SLUB分配器
  - SLOB分配器
- **伙伴系统分配的最小单位是一个物理页 ( 4K )**
  - 操作系统里面的结构体大小常位几十、几百字节
  - 避免内部碎片

# SLAB分配器

- 目标：快速分配小内存对象
- SLAB分配器历史
  - 上世纪 90 年代，Jeff Bonwick在Solaris 2.4中首创SLAB
  - 07年左右，Christoph Lameter在Linux中提出SLUB
    - SLAB的设计过于复杂
    - 在Linux-2.6.23及之后的版本中，成为默认分配器
  - 发展过程中，针对内存稀缺场景又提出了SLOB

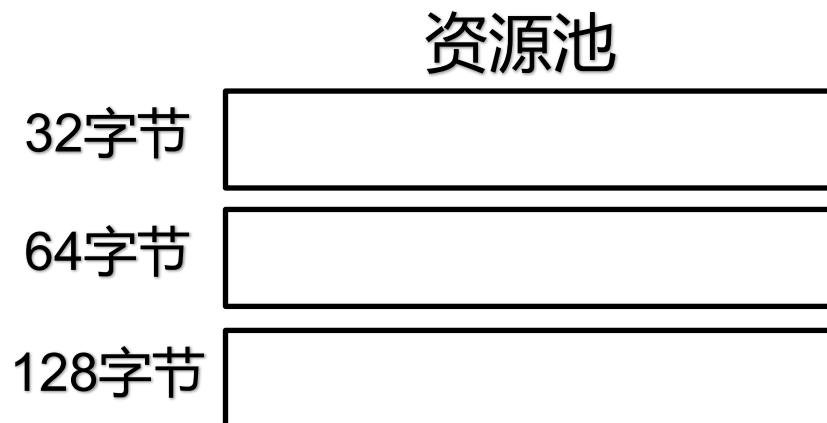
# SLUB

- 观察
  - 操作系统频繁分配的对象大小相对比较固定
- 基本思想
  - 从伙伴系统获得大块内存
  - 进一步细分成固定大小的小块内存进行管理
  - 块大小通常是  $2^n$  个字节（一般来说， $3 \leq n < 12$ ）
    - 可以额外增加特殊大小如198字节从而减小内部碎片

# SLUB设计

- **只分配固定大小块**

- 对于每个固定块大小，SLUB 分配器都会使用独立的内存资源池进行分配
- 采用**best fit**定位资源池



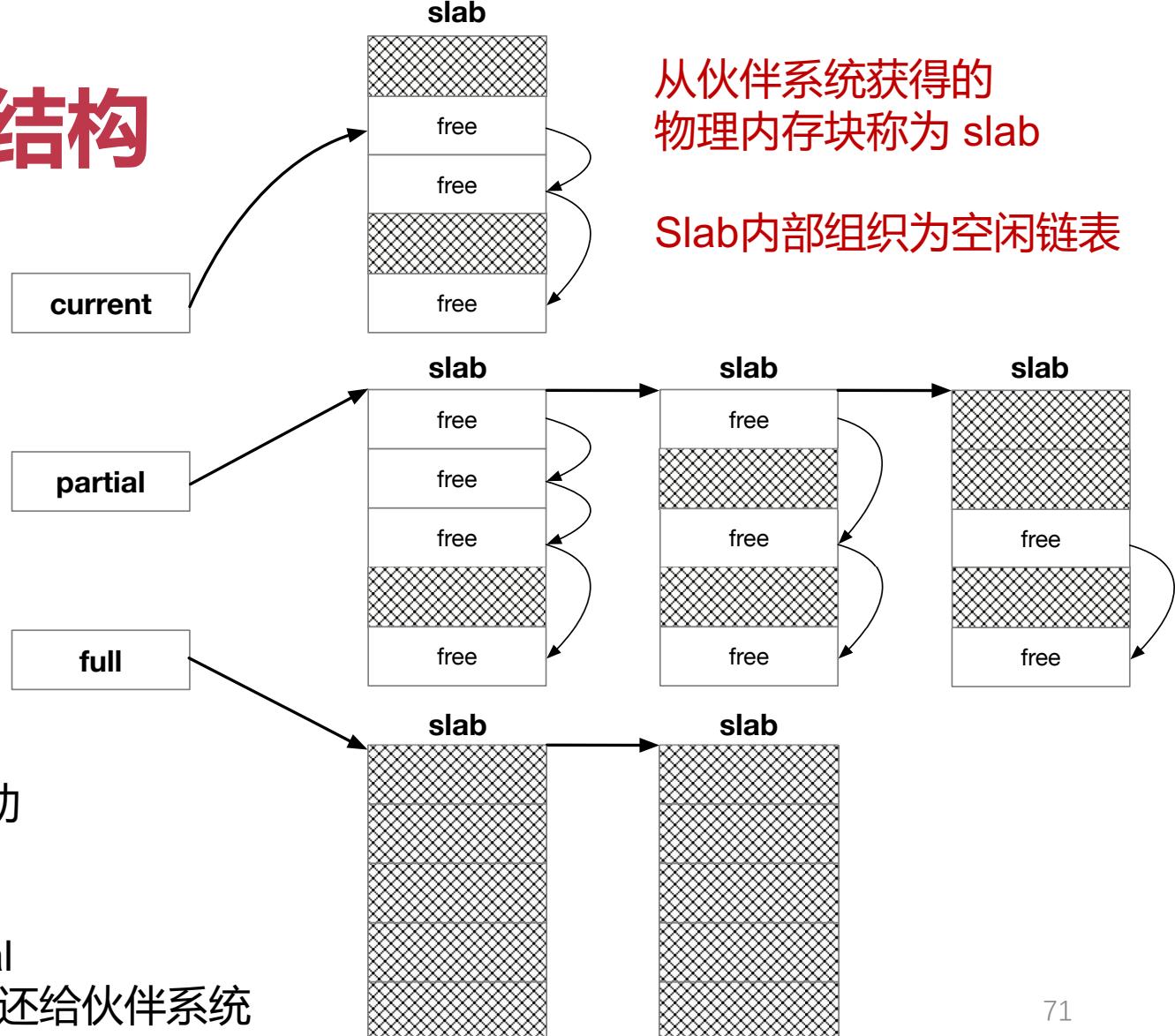
# SLUB数据结构

从伙伴系统获得的物理内存块称为 slab

Slab内部组织为空闲链表

## 三个指针

- current仅指向一个 slab
- partial指向未满slab链表
- full指向全满slab链表



## 分配使用current slab

- 若满发生两个移动

## 释放到对应的slab

- 移动 full 到 partial
- 若partial全free则还给伙伴系统