IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

内存管理2

陈海波 / 夏虞斌

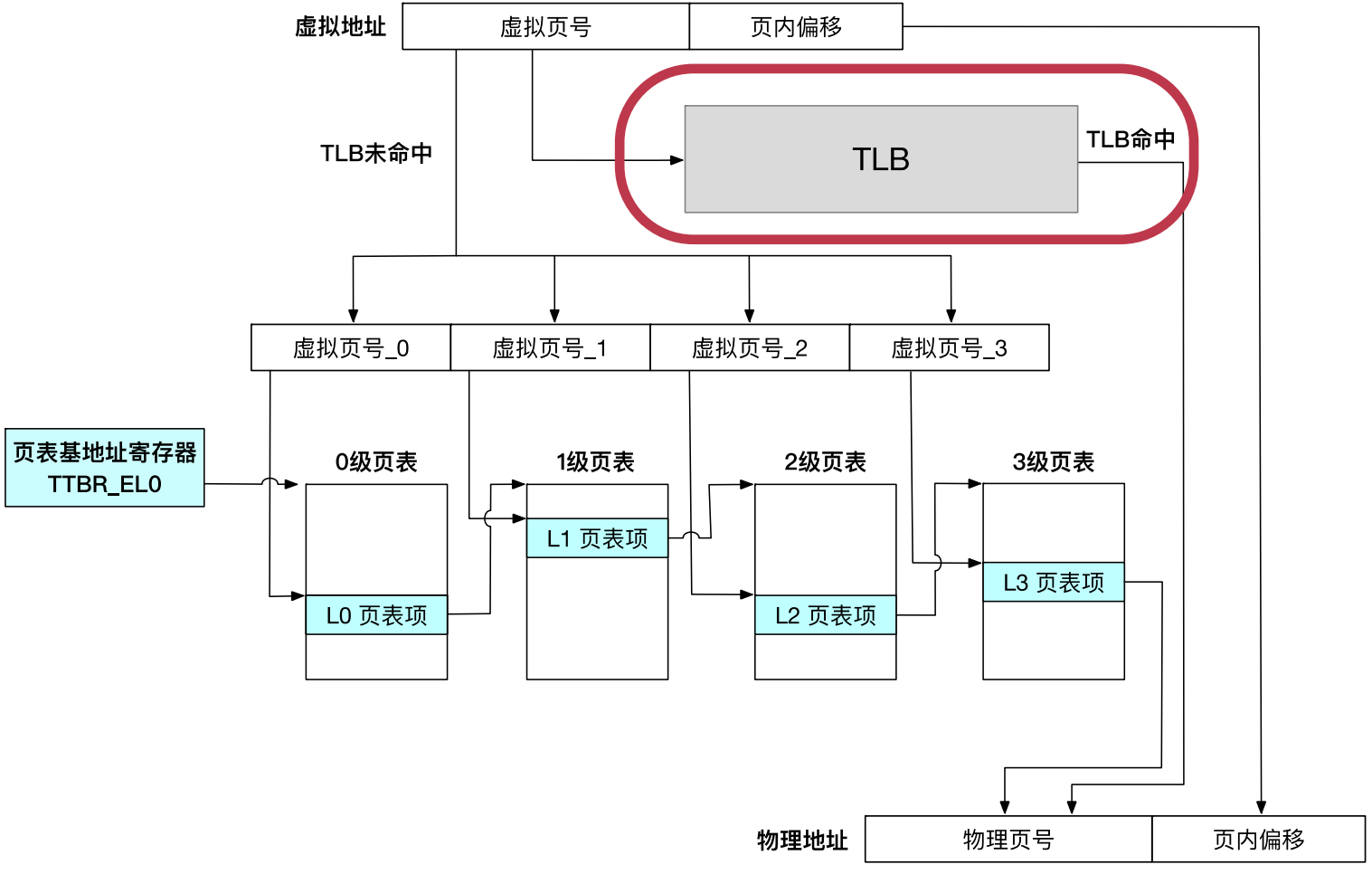
上海交通大学并行与分布式系统研究所

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
  - 资料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

回顾：地址翻译的加速器 -- TLB



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 (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 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**
  - PCID，存储在CR3的低12位
  - 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
    - 为什么?
- **AARCH64：ASID**
  - 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**
  - 切换页表可以不刷新TLB
  - 修改页表映射后，仍需刷新TLB

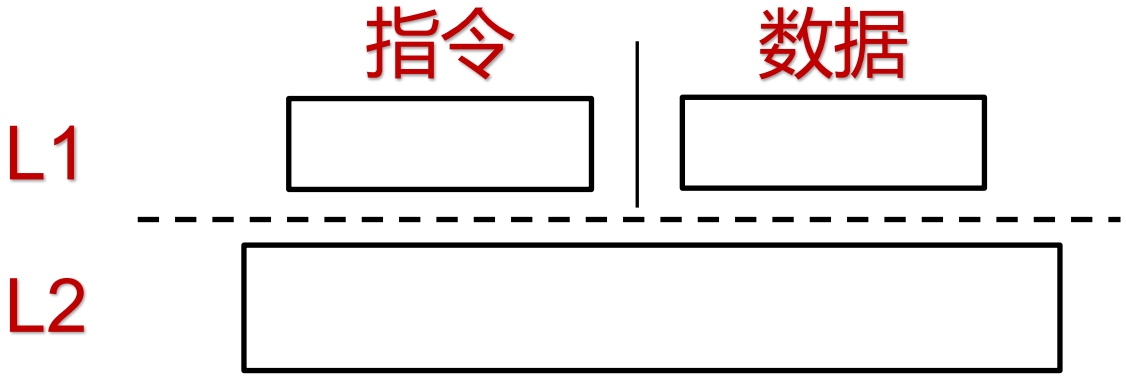
- **在多核场景下**
  - 需要刷新其它核的TLB吗?
  - 如何知道需要刷新哪些核?
  - 怎么刷新其它核?

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
- **怎么刷新其它核?**
  - x86_64: 发送IPI中断某个核，通知它主动刷新
  - AARCH64: 可在local CPU上刷新其它核TLB
    - TLBI ASIDE1IS

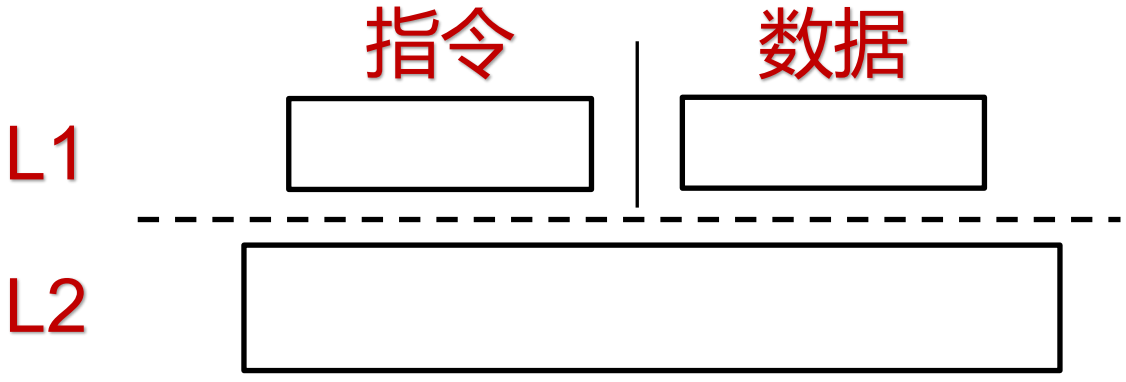
TLB结构简介

- 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）



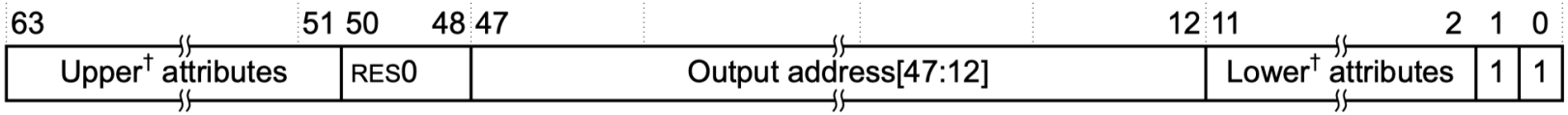
TLB结构简介

- 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?**

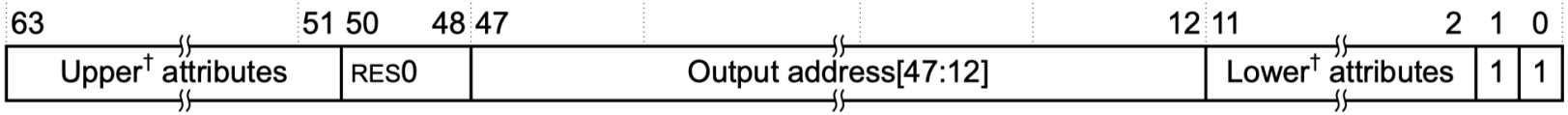
全局TLB



- **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

图片来自于ARMv8手册

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?**

图片来自于ARMv8手册

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/Flash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**
  - 提前注册缺页异常处理函数(Page Fault Handler)
  - CPU控制流传递到缺页异常处理函数
- **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

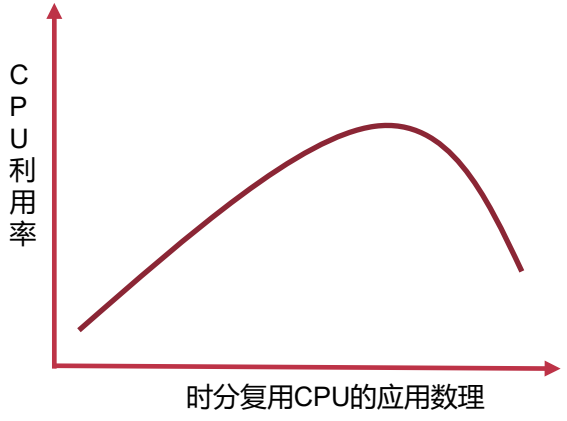
- **常见的替换策略**
  - 随机替换、FIFO、LRU/MRU、Clock Algorithm、…
- **替换策略评价标准**
  - 缺页发生的概率 （参照理想但不能实现的**OPT策略**）
  - 策略本身的性能开销
    - 如何高效地记录物理页的使用情况?
      - Recap：上节课说到的页表项中Access/Dirty Bits

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）
    - 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

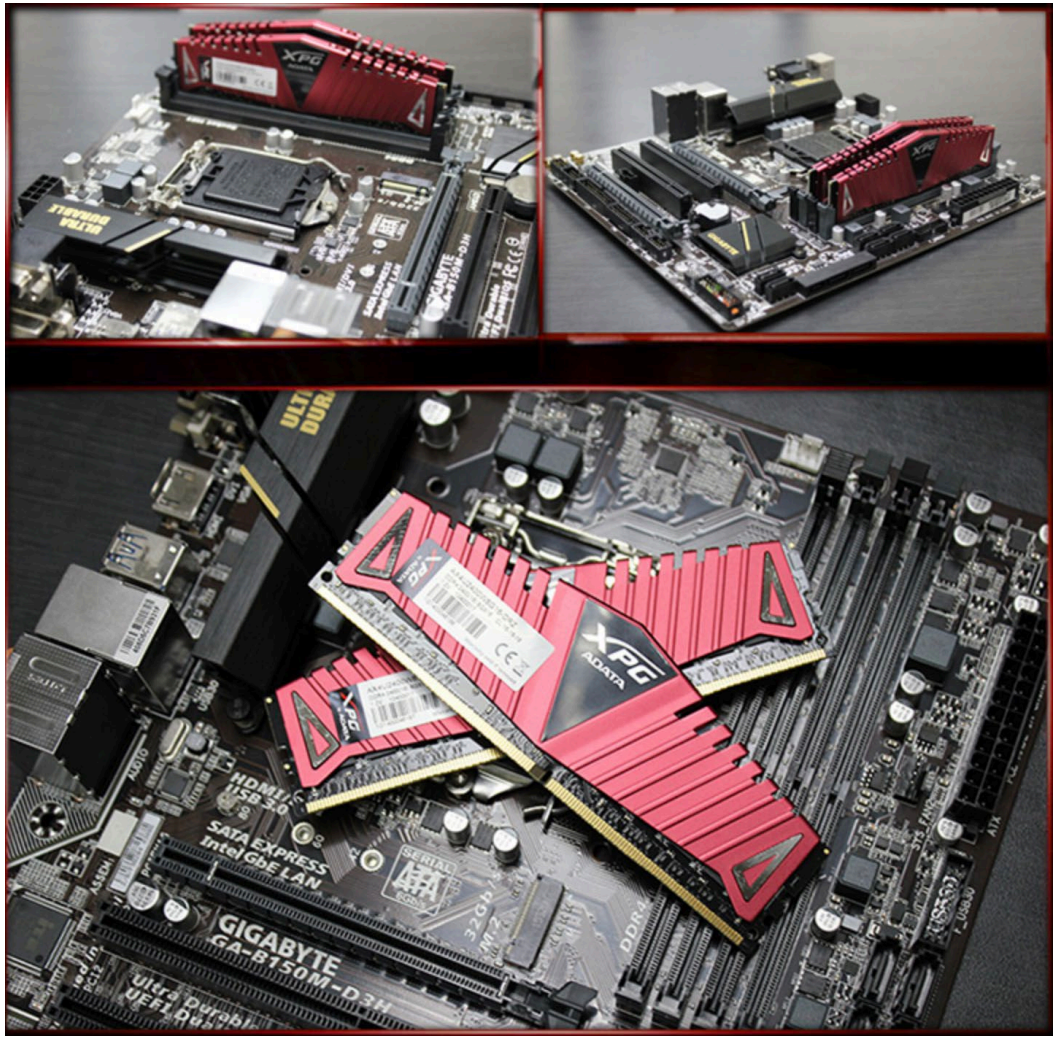
2010	1
2000	1
1970	0
1990	0

上次使用时间　访问位

21世纪第三个十年再看换页

- **思考：今天换页还必要吗？**
  - 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



“内存条”

图片来自于京东商城网站

物理内存结构

Channel

DIMM

Rank

Chip

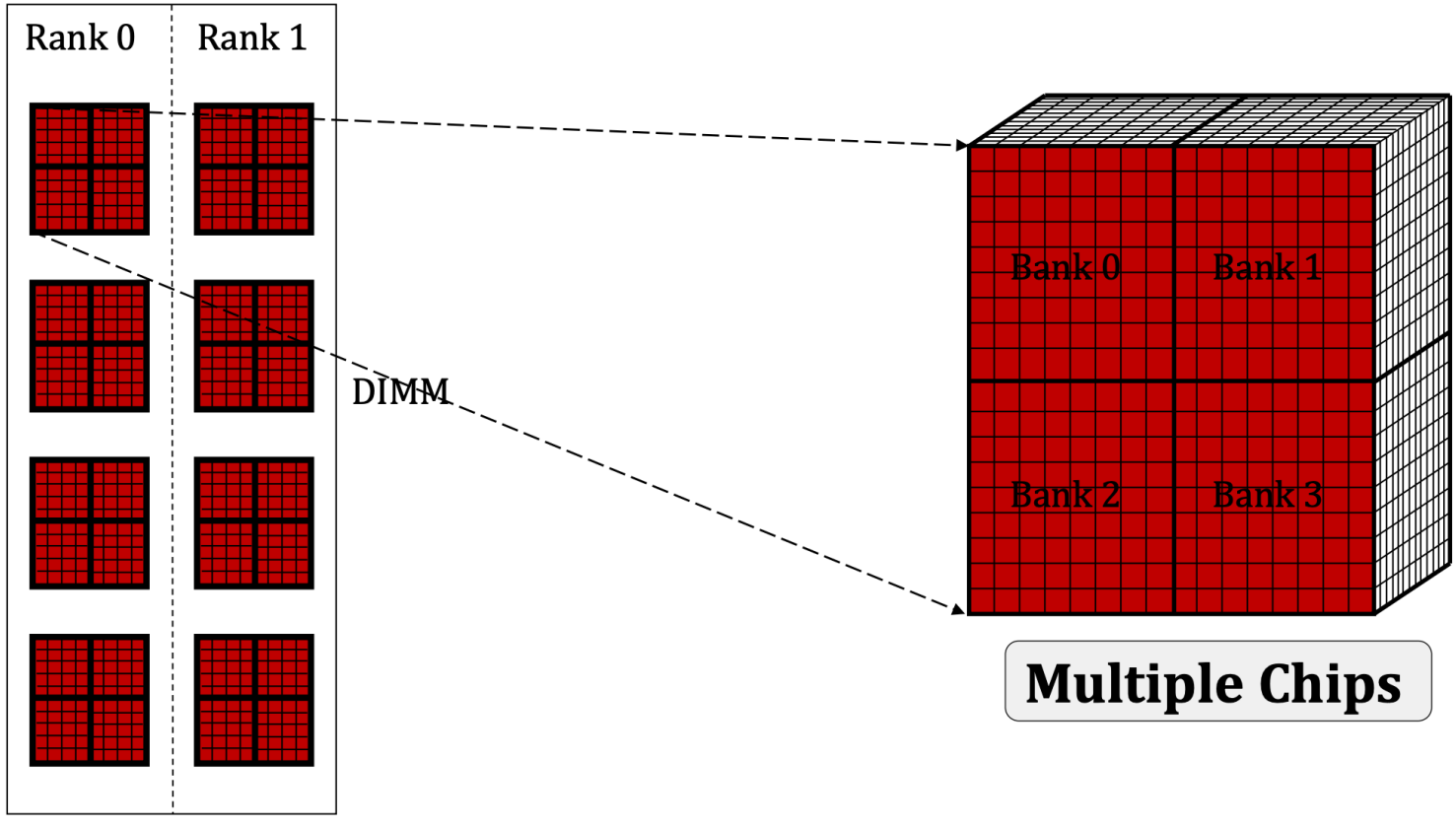
Bank

Row

Column

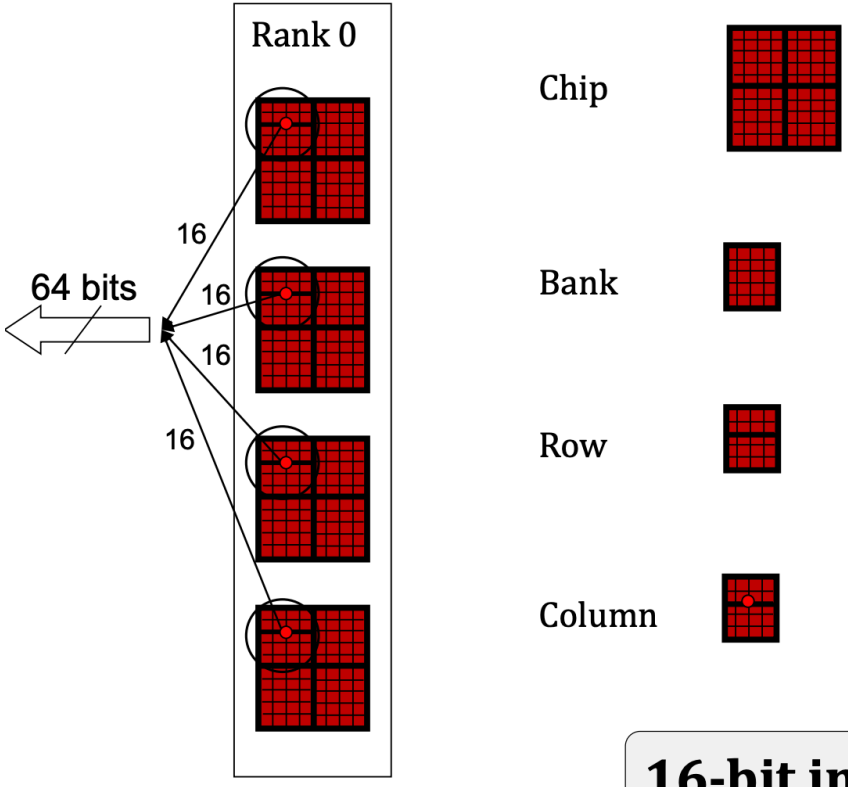
本页来自于https://www.cse.iitk.ac.in/users/biswap/CS698Y.html

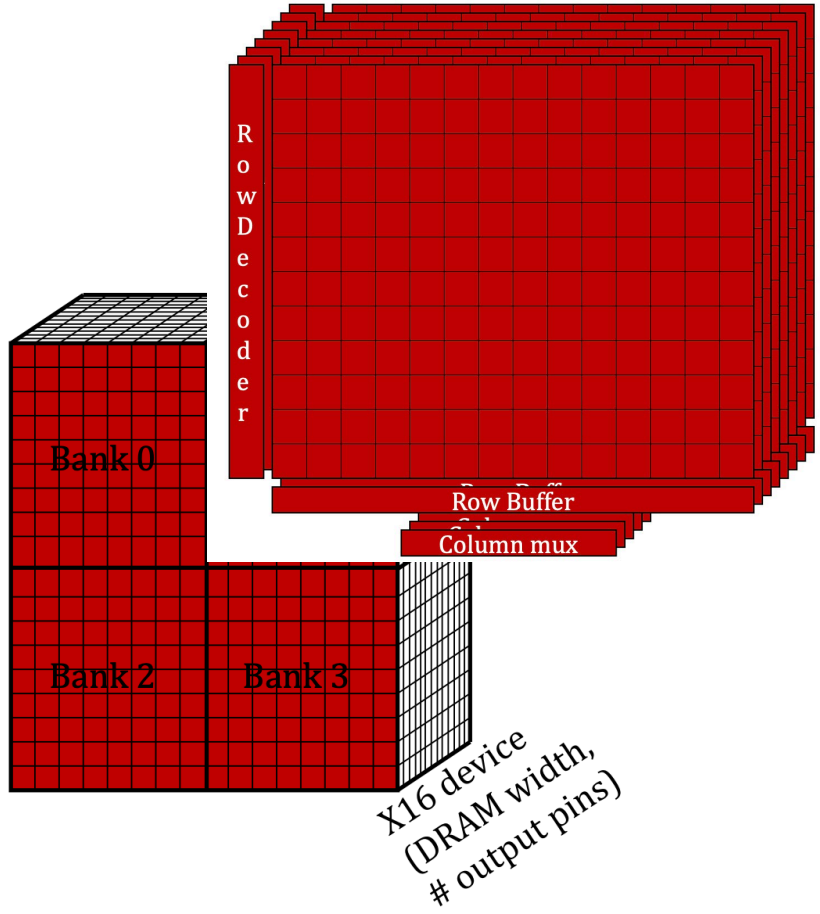
物理内存结构



本页来自于https://www.cse.iitk.ac.in/users/biswap/CS698Y.html

物理内存结构





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

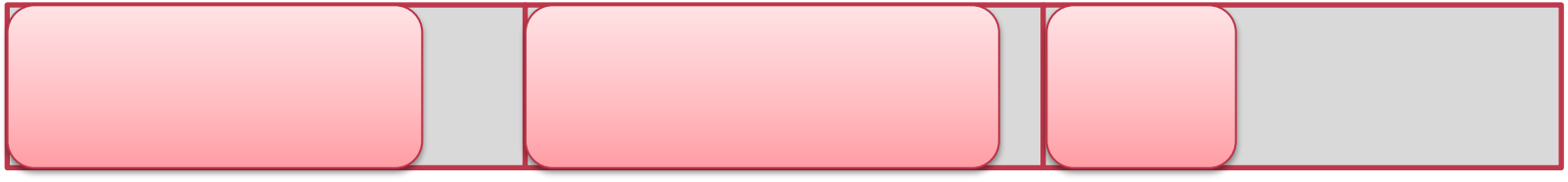
Memory Controller

- **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）

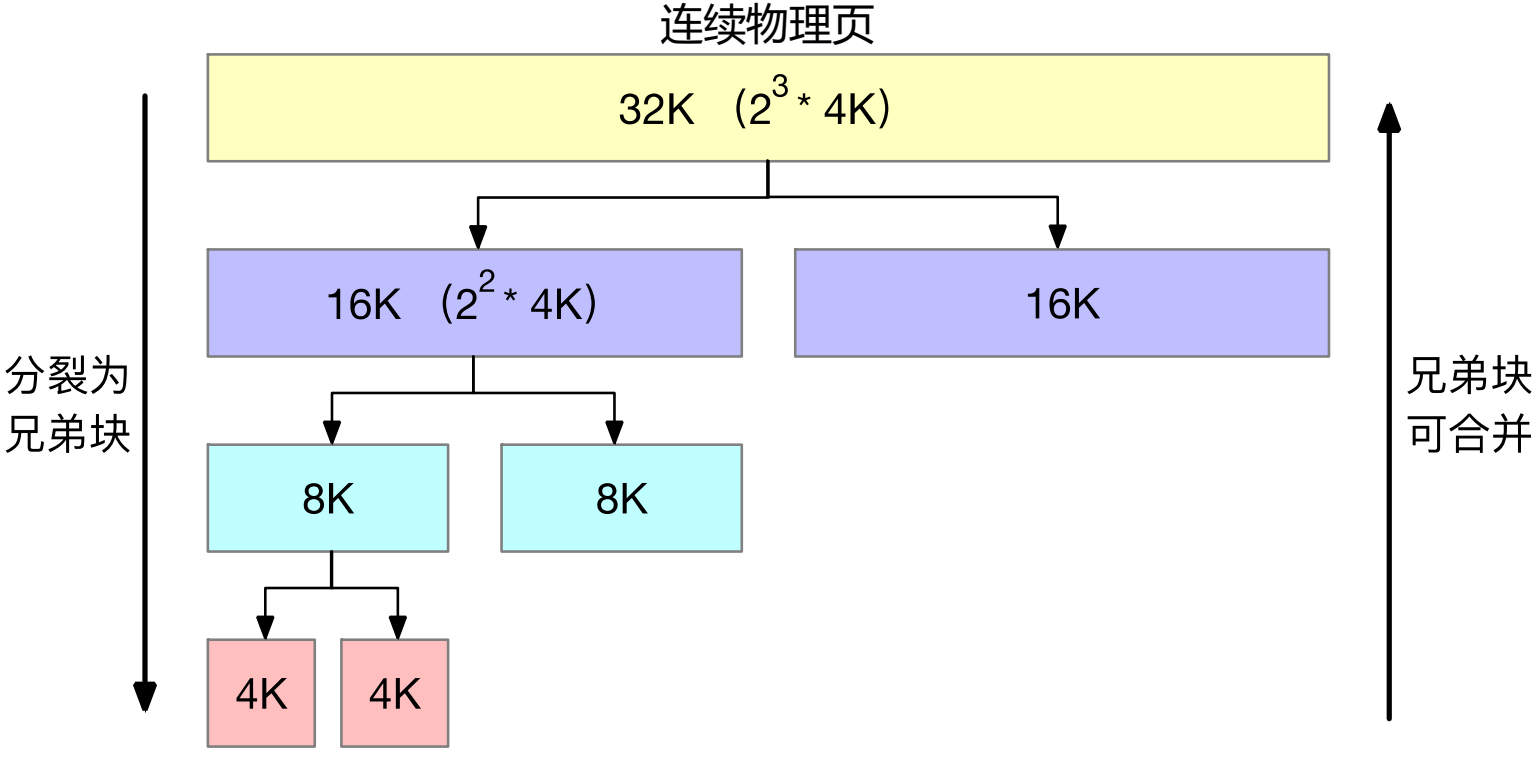


物理内存管理的评价指标

- **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?**

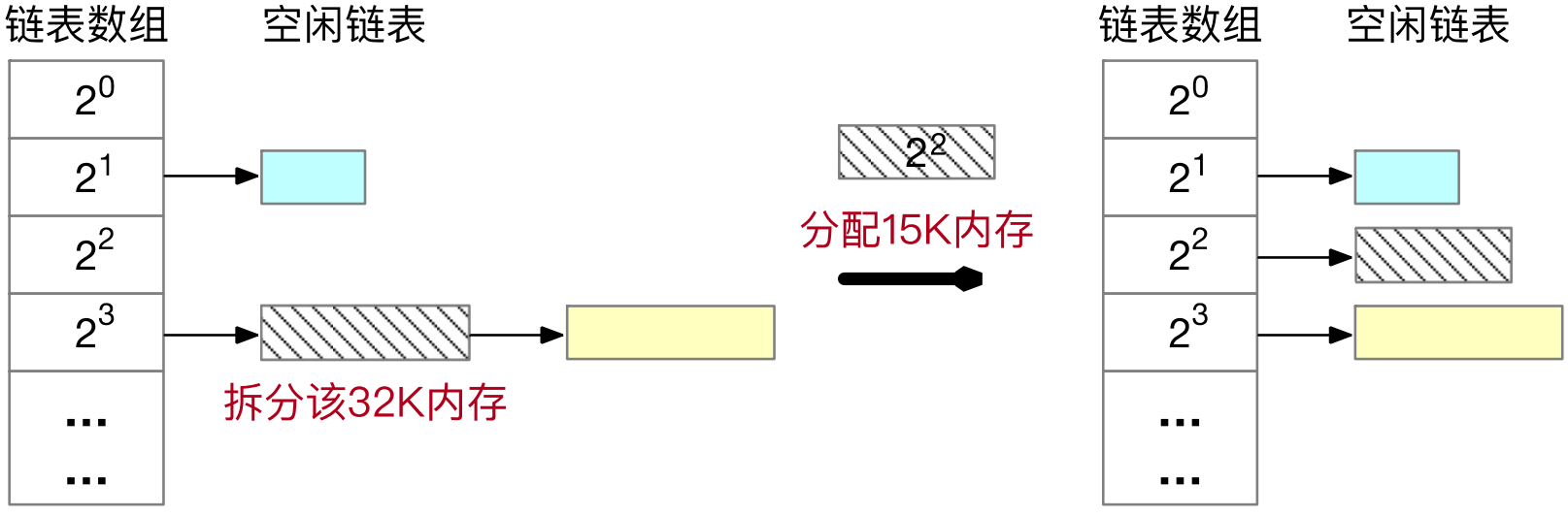
物理内存管理之buddy system

- 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
    - Linux-2.6.23之后成为默认分配器
  - 发展过程中，针对内存稀缺场景的SLOB也被提出

SLUB

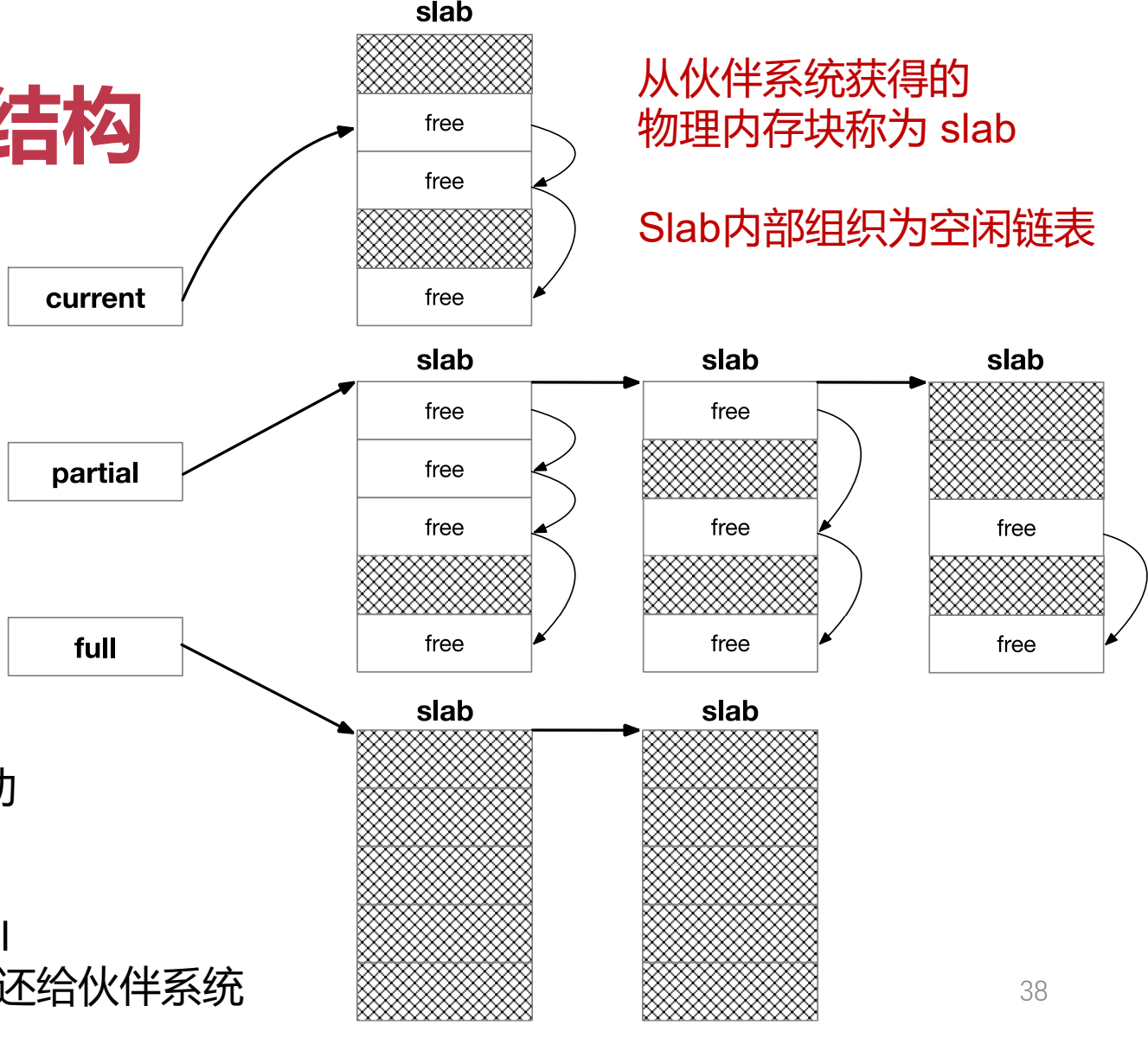
- **观察**
  - 操作系统频繁分配的对象大小相对比较固定
- **基本思想**
  - 从伙伴系统获得大块内存
  - 进一步细分成固定大小的小块内存进行管理
  - 块大小通常是 2^n 个字节（一般来说，$3 \leqslant n < 12$）
    - 可以额外增加特殊大小如198字节从而减小内部碎片

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

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



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

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

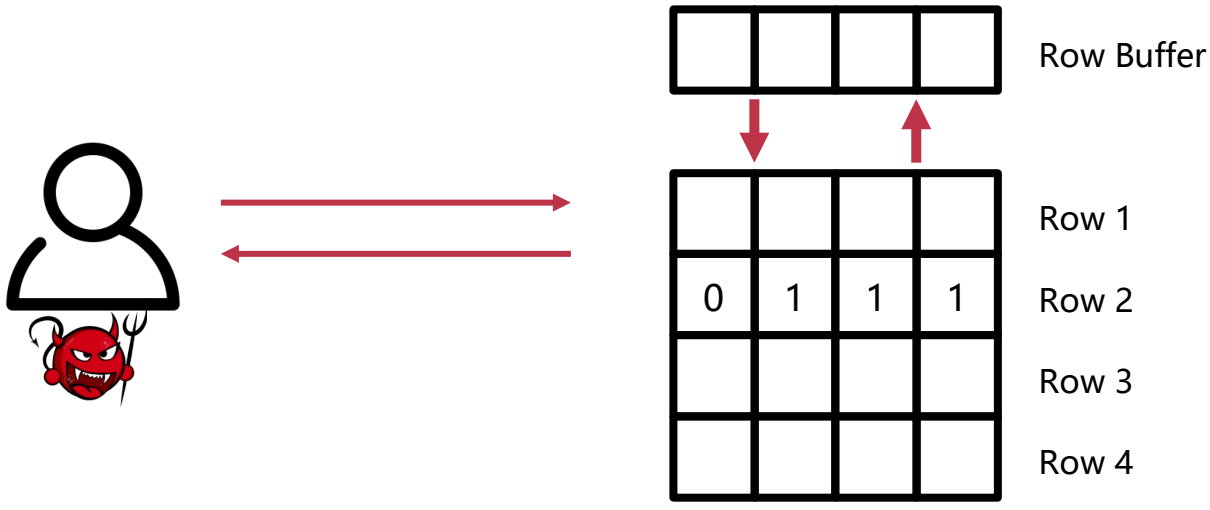
物理内存管理的其它问题

- **安全问题**
  - Rowhammer
  - Cache Side Channel

- **性能问题**
  - 性能隔离、QoS等

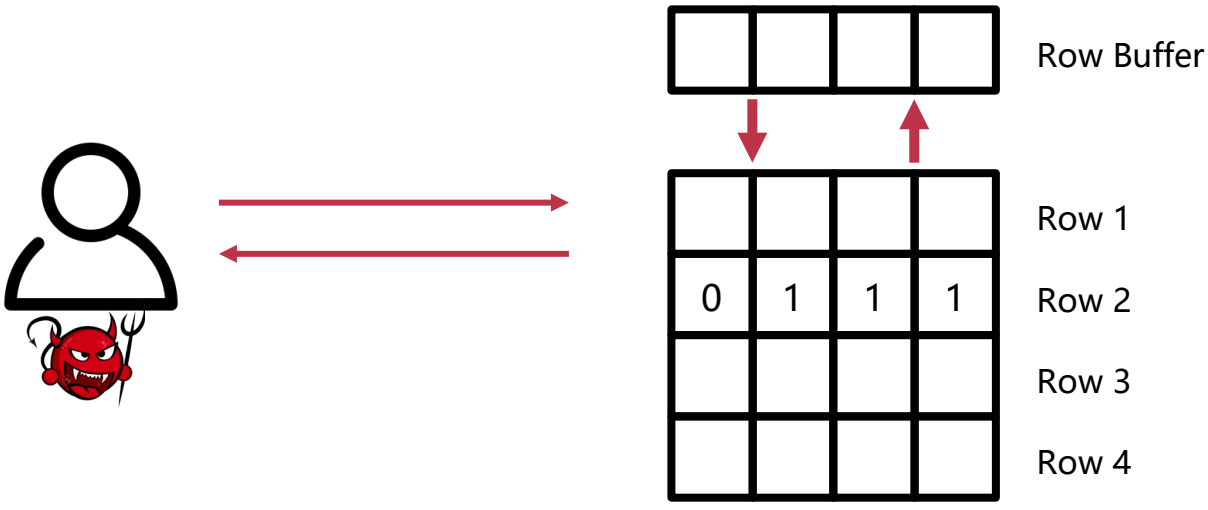
Rowhammer攻击

Recap：尽管Memory controller屏蔽了物理内存细节，
但是真实访问依然会用到Row等物理结构



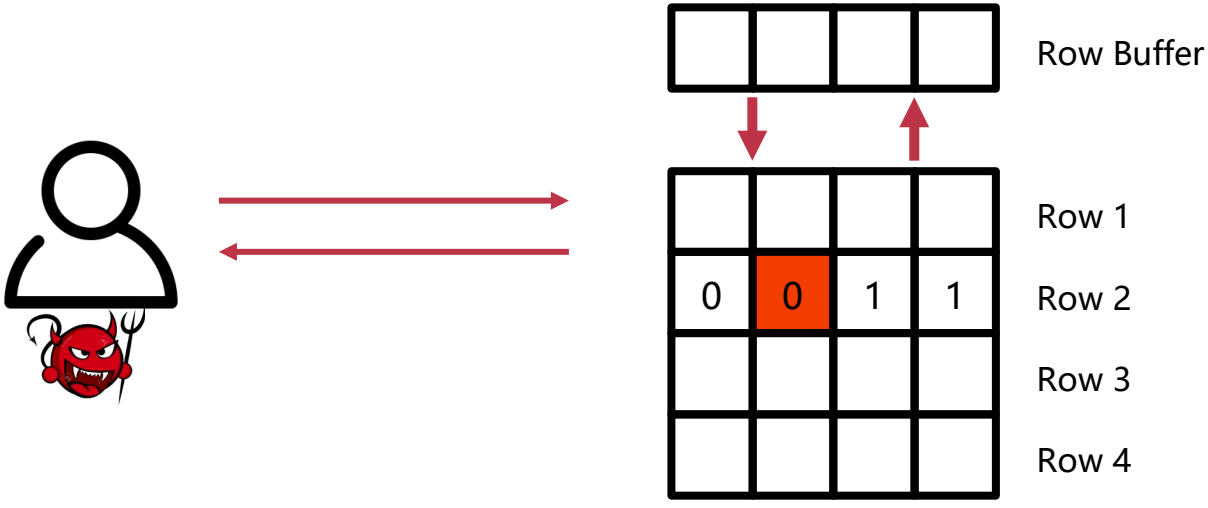
Rowhammer攻击

攻击者利用物理内存缺陷，极频繁访问某一行，其相邻行某些位会发生翻转



Rowhammer攻击

巧妙地利用位翻转，可以实施包括提权在内的多种攻击



安全防御

- 为抵御Rowhammer攻击，实际上操作系统需要知道部分硬件细节，从而能够在物理内存分配时主动加入一些Guard Page

- 为抵御cache Side Channel攻击，操作系统需要知道同样cache映射细节

性能考虑

- **cache miss的开销很高**
  - 物理内存分配同样需要考虑到降低cache冲突几率
  - 典型机制：cache coloring

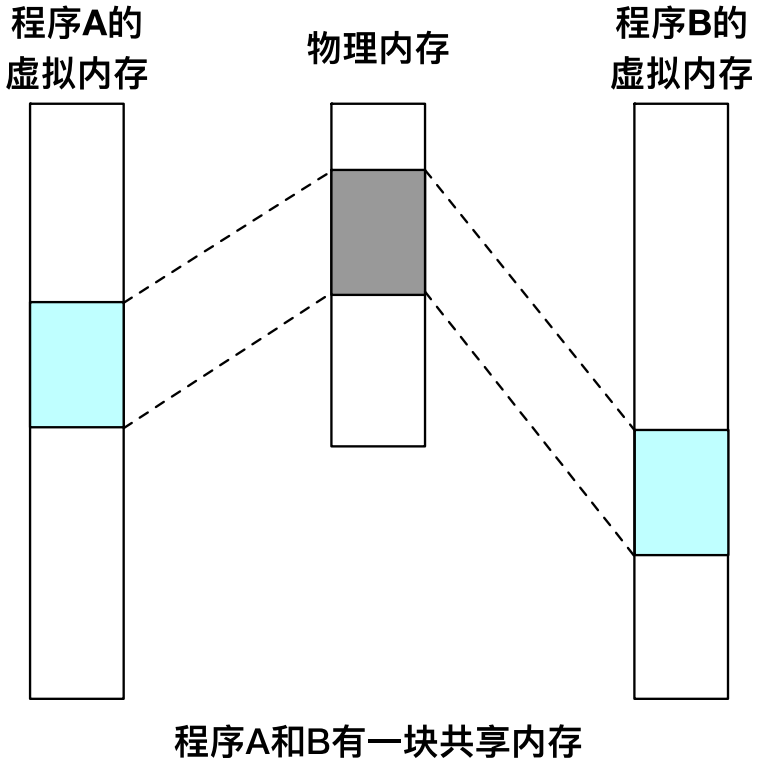
- **保证性能隔离是QoS的必要前提**
  - Intel CAT
  - AARCH64 MPAM

- ☑ 虚拟内存抽象
- ☑ 物理内存分配

操作系统内存管理的功能

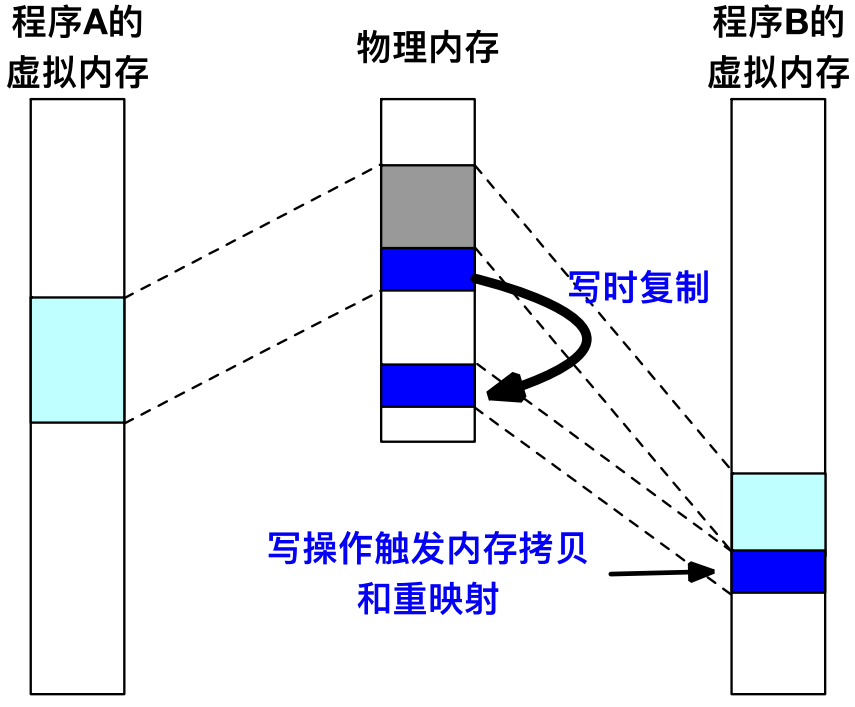
共享内存

- **基本功能**
  - 节约内存，如共享库
  - 进程通信，传递数据



写时拷贝（copy-on-write）

- **实现**
  - 修改页表项权限
  - 在缺页时拷贝、恢复
- **典型场景fork**
  - 节约物理内存
  - 性能加速



以写时拷贝的方式共享内存

内存去重

- **memory deduplication**
  - 基于写时拷贝机制
  - 在内存中扫描发现具有相同内容的物理页面
  - 执行去重
  - 操作系统发起，对用户态透明
- **典型案例：Linux KSM**
  - kernel same-page merging

内存去重潜在安全隐患

- **导致新的side channel**
  - 访问被合并的页会导致访问延迟明显
- **潜在攻击**
  - 攻击者可以确认目标进程中含有构造数据

- **思考：如何平衡这个tradeoff?**

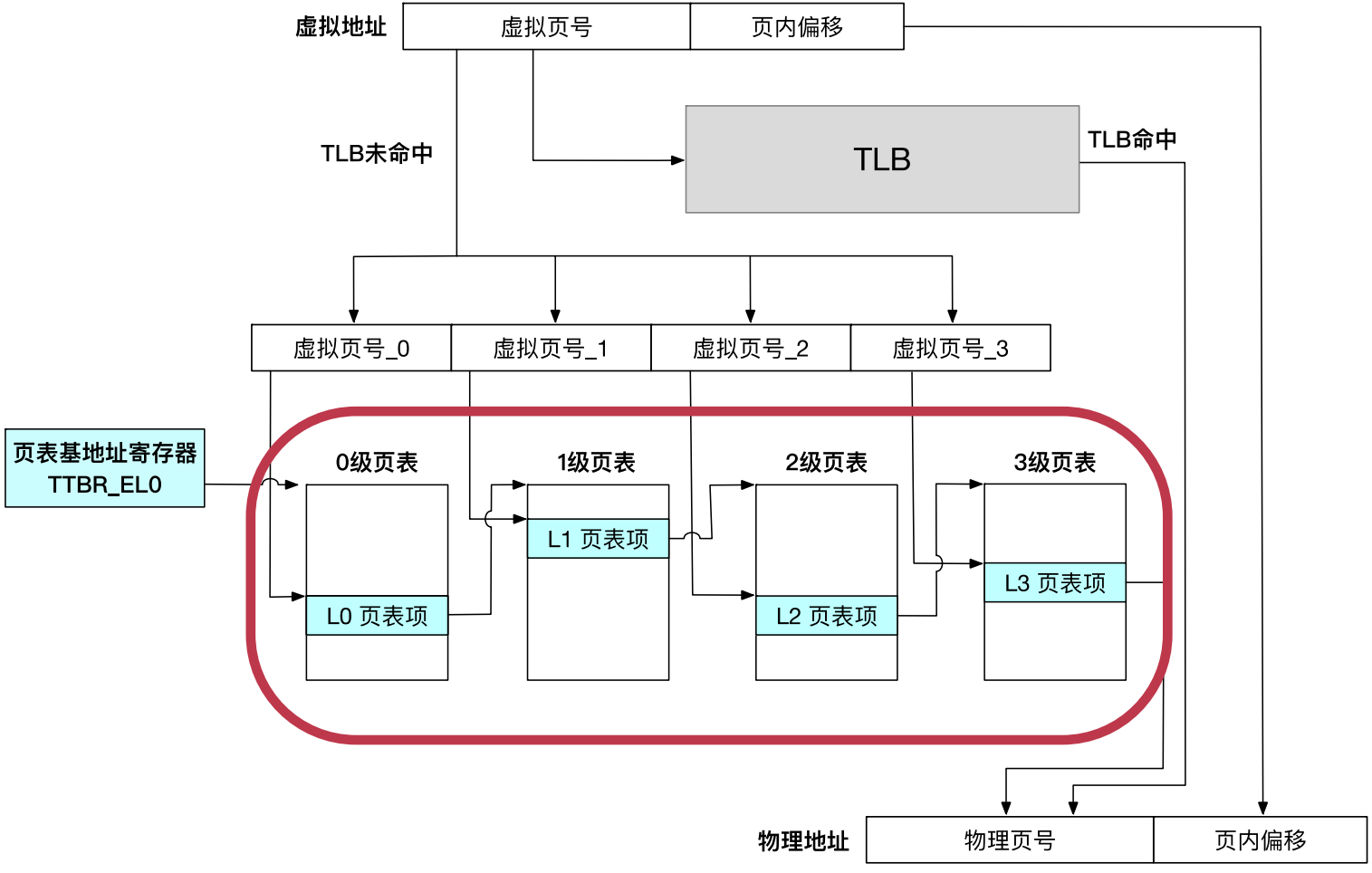
内存压缩

- **基本思想**
  - 当内存资源不充足的时候， 选择将一些“最近不太会使用”的内存页进行数据压缩，从而释放出空闲内存

内存压缩案例

- **Windows 10**
  - 压缩后的数据仍然存放在内存中
  - 当访问被压缩的数据时，操作系统将其解压即可
  - 思考：对比交换内存页到磁盘?
- **Linux**
  - zswap：换页过程中磁盘的缓存
  - 将准备换出的数据压缩并先写入 zswap 区域 （内存）
  - 好处：减少甚至避免磁盘I/O；增加设备寿命

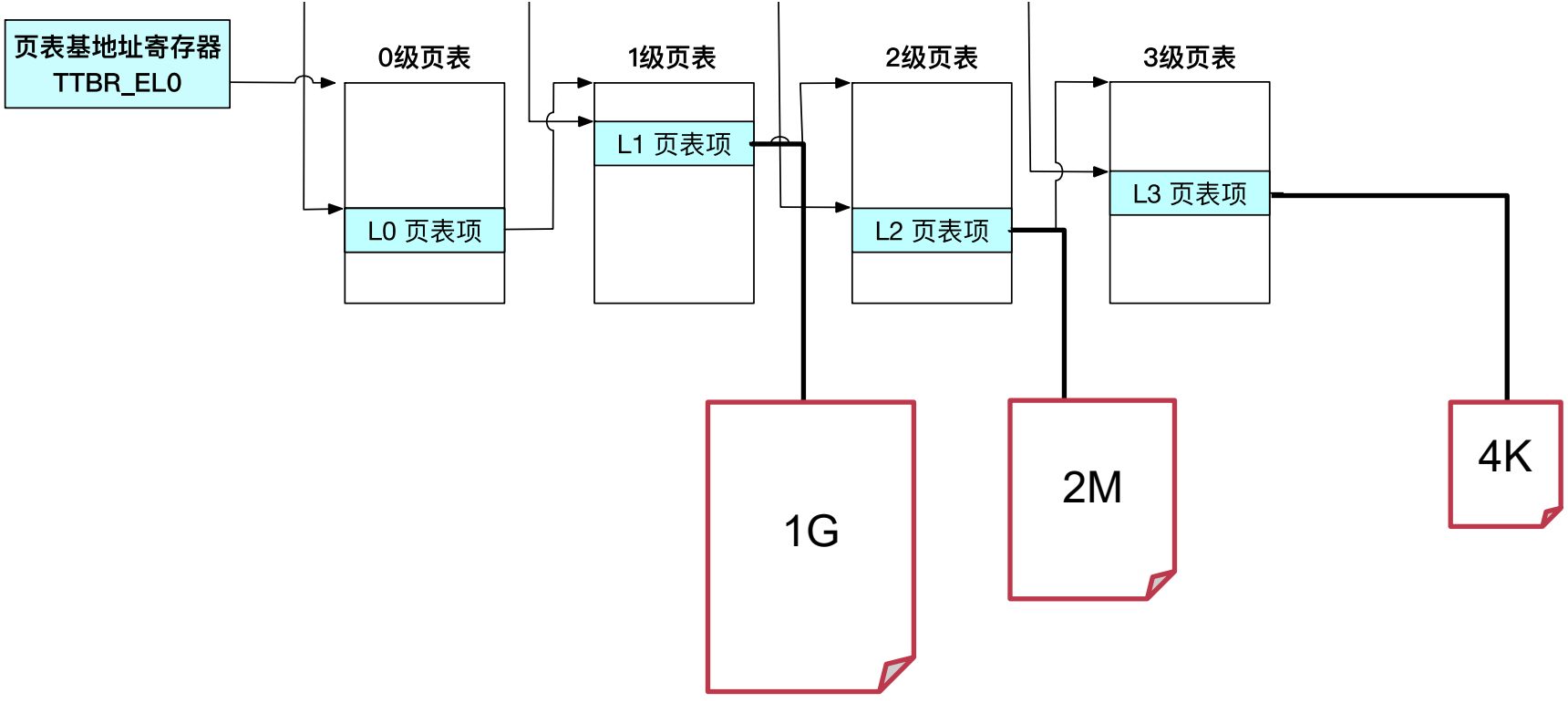
大页：再次回顾4级页表


虚拟地址
虚拟页号
页内偏移
TLB未命中
TLB
TLB命中
虚拟页号_0
虚拟页号_1
虚拟页号_2
虚拟页号_3
页表基地址寄存器
TTBR_EL0
0级页表
1级页表
2级页表
3级页表
L0 页表项
L1 页表项
L2 页表项
L3 页表项
物理地址
物理页号
页内偏移

大页

- **在4级页表中，某些页表项只保留两级或三级页表**
- **L2页表项的第1位**
  - 标识着该页表项中存储的物理地址（页号）是指向 L3 页表页（该位是 1）还是指向一个 2M 的物理页（该位是 0）
- **L1页表项的第1位**
  - 类似地，可以指向一个 1G 的物理页

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

AARCH64支持多种最小页面大小

- **x86_64：4K**
- **AARCH64**
  - TCR_EL1可以配置3种：4K、16K、64K
  - 4K + 大页：2M/1G
  - 16K + 大页：32M（思考为什么是32M？）
    - 只有L2页表项支持大页
  - 64K + 大页：512M
    - 只有L2页表项支持大页 （ARMv8.2之前）

思考

- 什么页/什么情况适合使用大页?

- 安卓关于大页使用的讨论

Linux上有趣的内存管理API

- **mmap**
  - void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)

- **轶事：IPADS学生凭借mmap夺得世界冠军**
  - 文件操作，相比于read/write的优势
  - 避免特权级切换

Linux上有趣的内存管理API

- **madvise**
  - int madvise(void *addr, size_t length, int advice)

- **使用**
  - 用户态的语义信息告诉内核，便于优化
  - 例如：将madvise和mmap搭配起来使用，在使用数据前告诉内核这一段数据需要使用，从而减少缺页异常

思考题

- **在物理内存足够大的今天，虚拟内存是否还有存在的必要？如果不使用虚拟内存抽象，恢复到只用物理内存寻址，会带来哪些改变？**

- **如果不依靠 MMU，是否有可以替换虚拟内存的方法？**
  - 基于高级语言实现多个同一个地址空间内运行实例的隔离
  - 基于编译器插桩实现多个运行实例的隔离
    - 参考 Software Fault Isolation

下节课

- 进程