IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

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

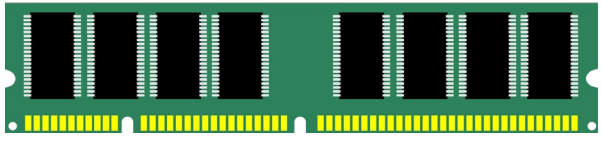
https://ipads.se.sjtu.edu.cn

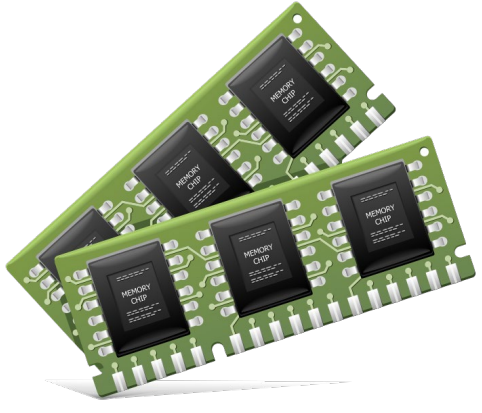
版权声明

- 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

物理内存

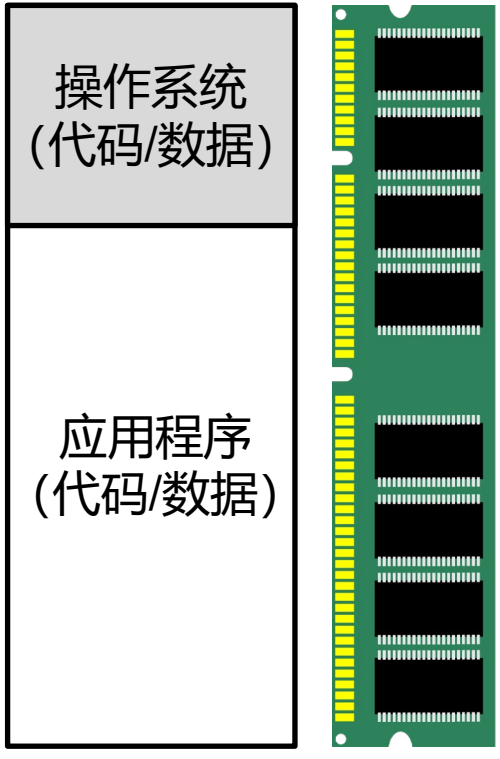
- **常说的"内存条"就是指物理内存**
- **数据从磁盘中加载到物理内存后，才能被CPU访问**
  - 操作系统的代码和数据
  - 应用程序的代码和数据





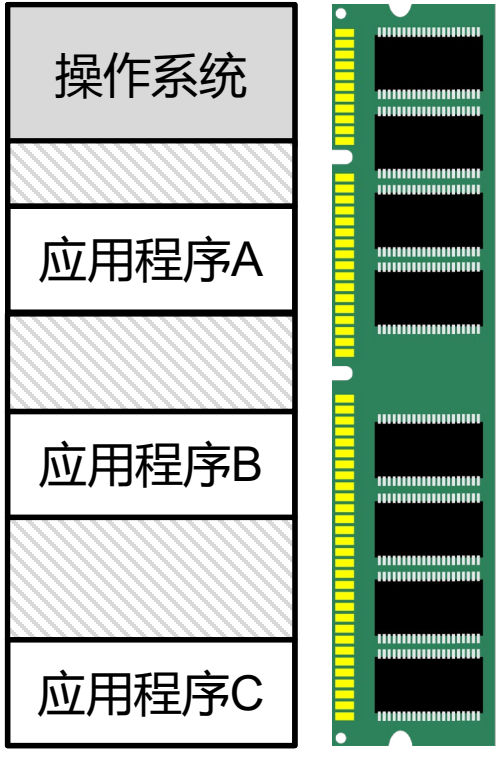
最早期的计算机系统

- **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

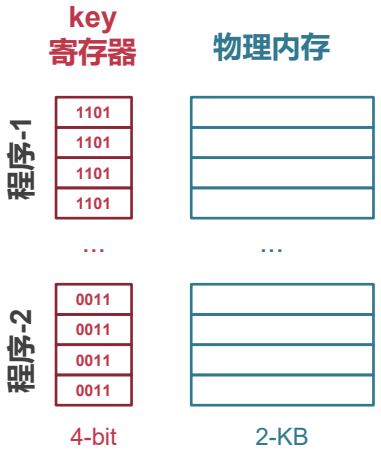
- **多用户多程序**
  - 计算机很昂贵，多人同时使用（远程连接）
- **分时复用CPU资源**
  - 保存恢复寄存器速度很快
- **分时复用物理内存资源**
  - 将全部内存写入磁盘开销太高
- **同时使用、各占一部分物理内存**
  - 没有安全性（隔离性）



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?

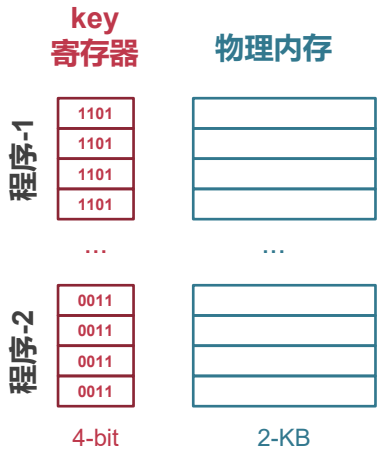
IBM 360的内存隔离：Protection Key

- **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？**
  - 不用关心其他进程，不受其他进程的影响
  - 看不见其他进程的信息，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**

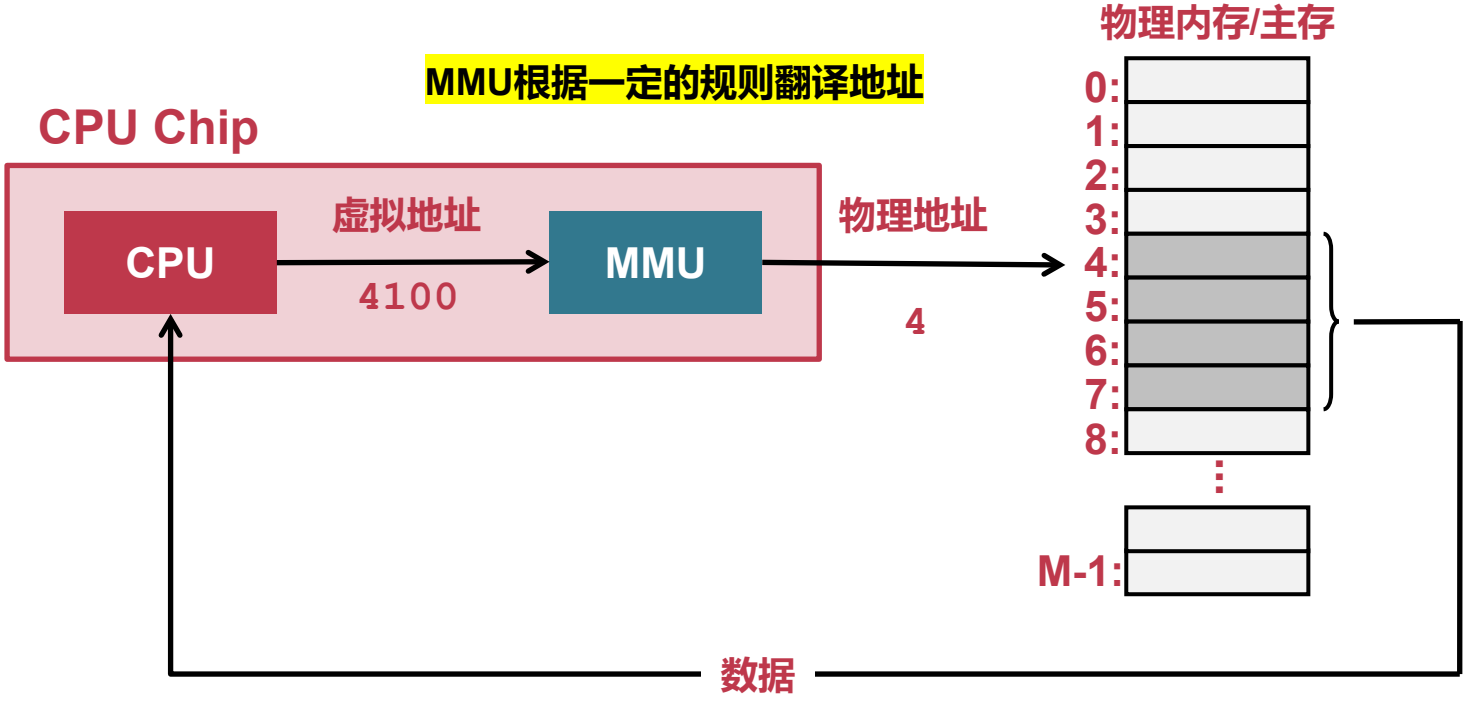
- **以虚拟内存抽象为核心的内存管理**
  - **CPU**：支持虚拟内存功能，新增了虚拟地址空间
  - **操作系统**：配置并使能虚拟内存机制
  - **所有软件**（包括OS）：均使用虚拟地址，无法直接访问物理地址

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

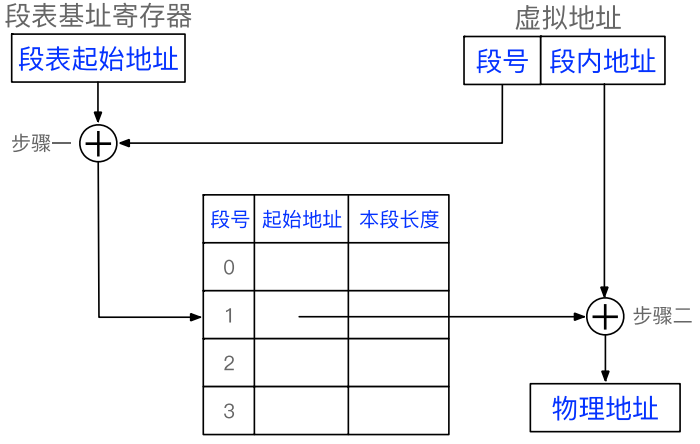
地址翻译过程



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

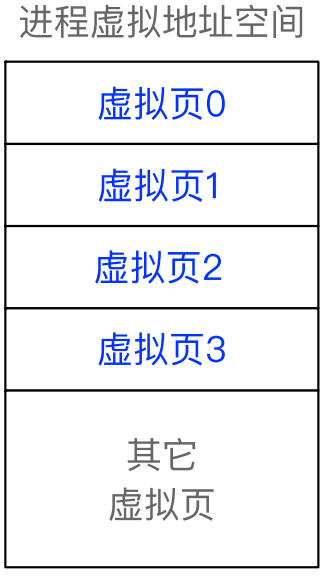
分段机制

- **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



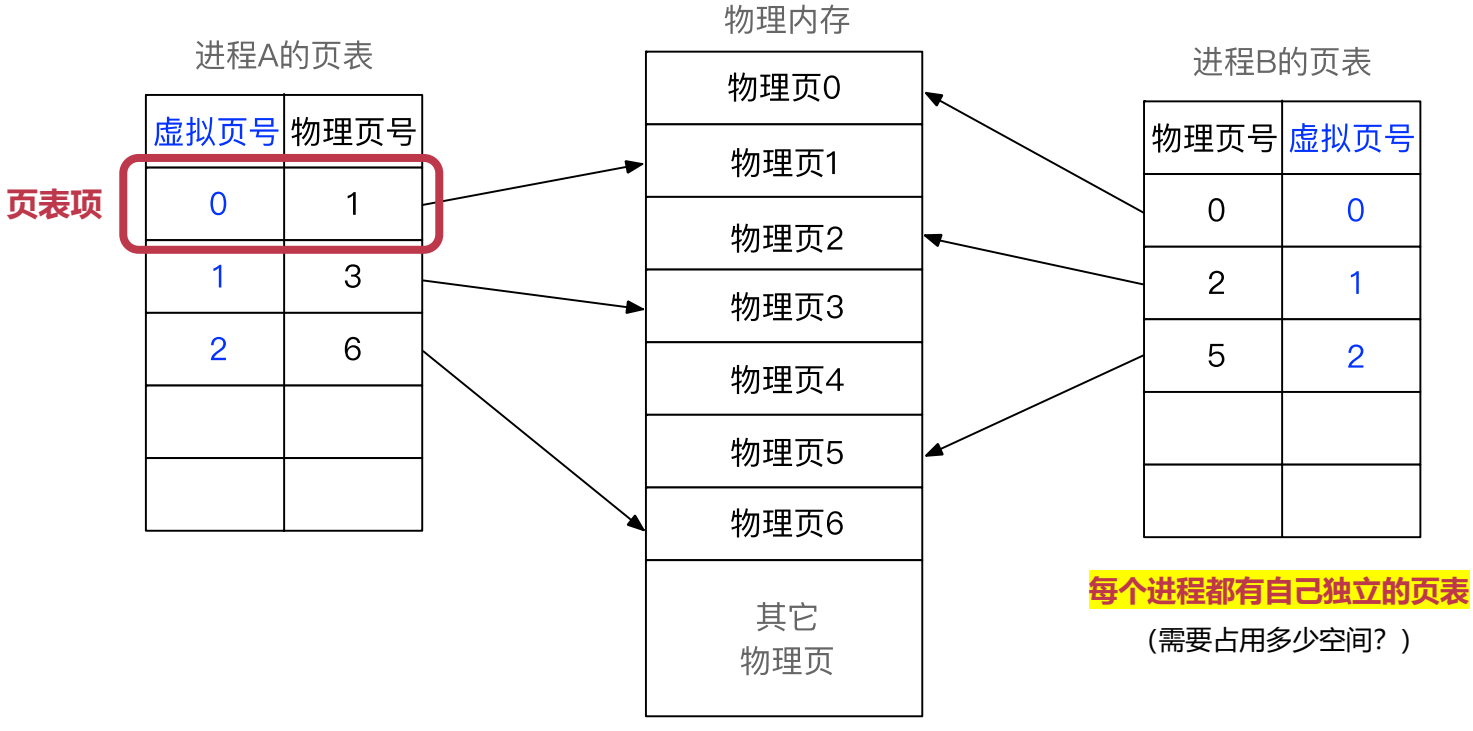
分页机制

- **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**



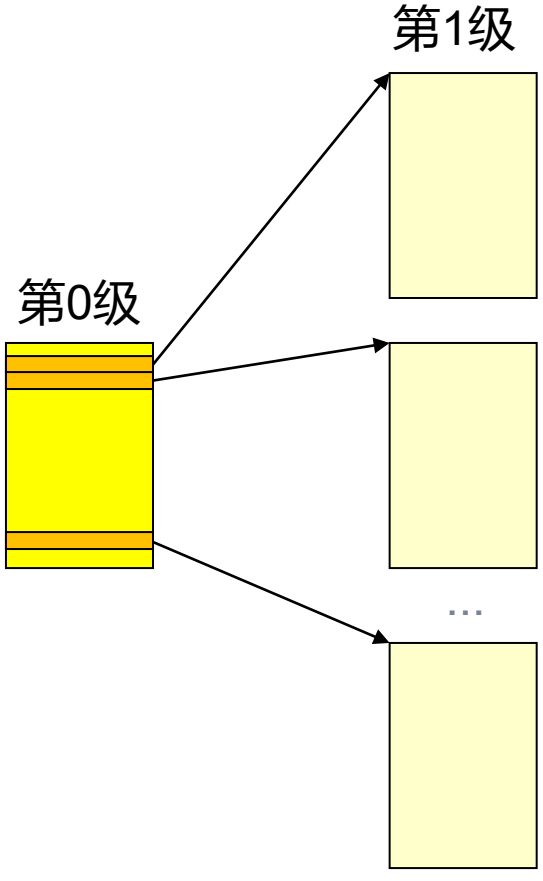
页表：分页机制的核心数据结构

- **页表包含多个页表项，存储虚拟页到物理页的映射**

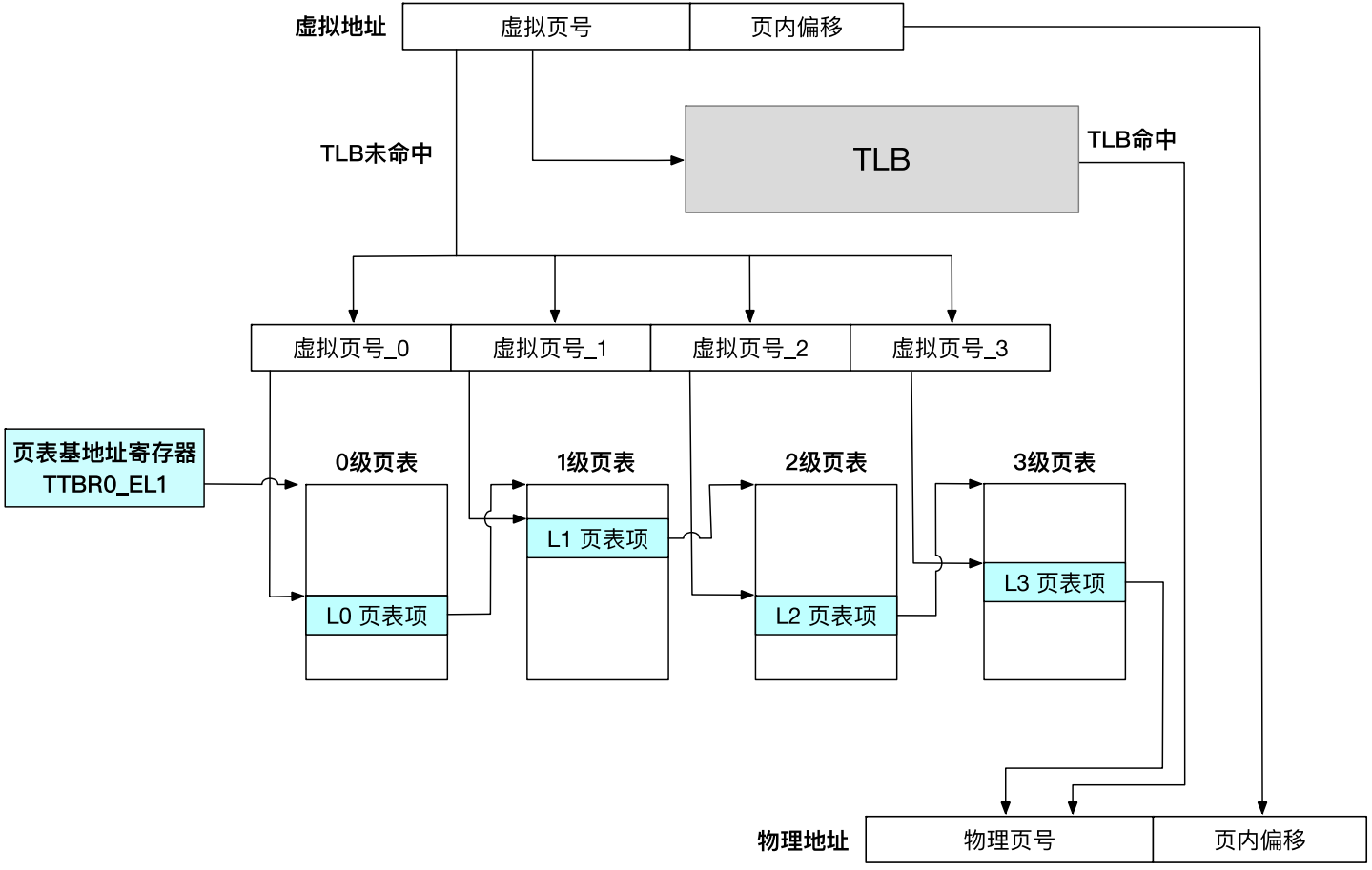


单级页表的问题

- **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

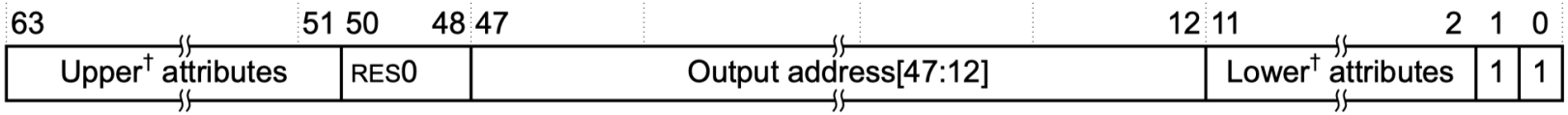
页表使能（Enabling）

- **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
- **x86_64**
  - CR4，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

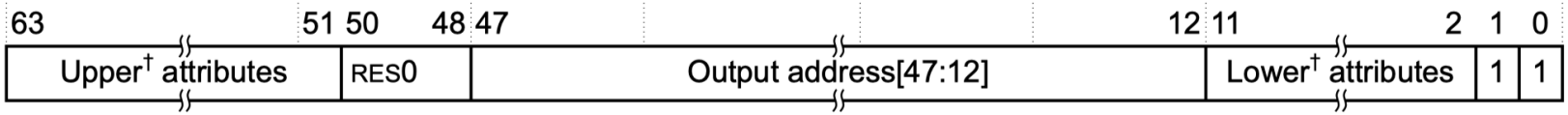
AARCH64页表项



- **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

图片来自于ARMv8手册

AARCH64页表项

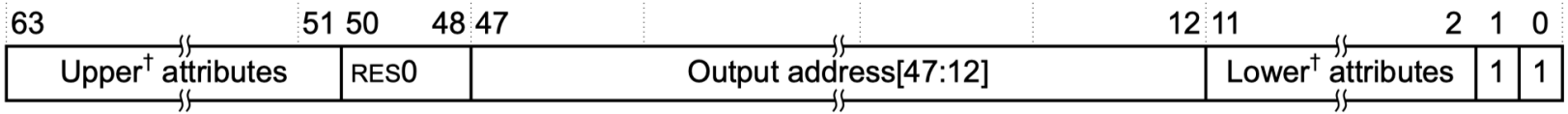


- **第3级页表页中的页表项**
  - Lower attributes 包括：
    - 第7位-第6位表示读写权限位AP[2:1]

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

图片来自于ARMv8手册

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AttrIndx[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）

图片来自于ARMv8手册

ARM的Cache Lockdown特性

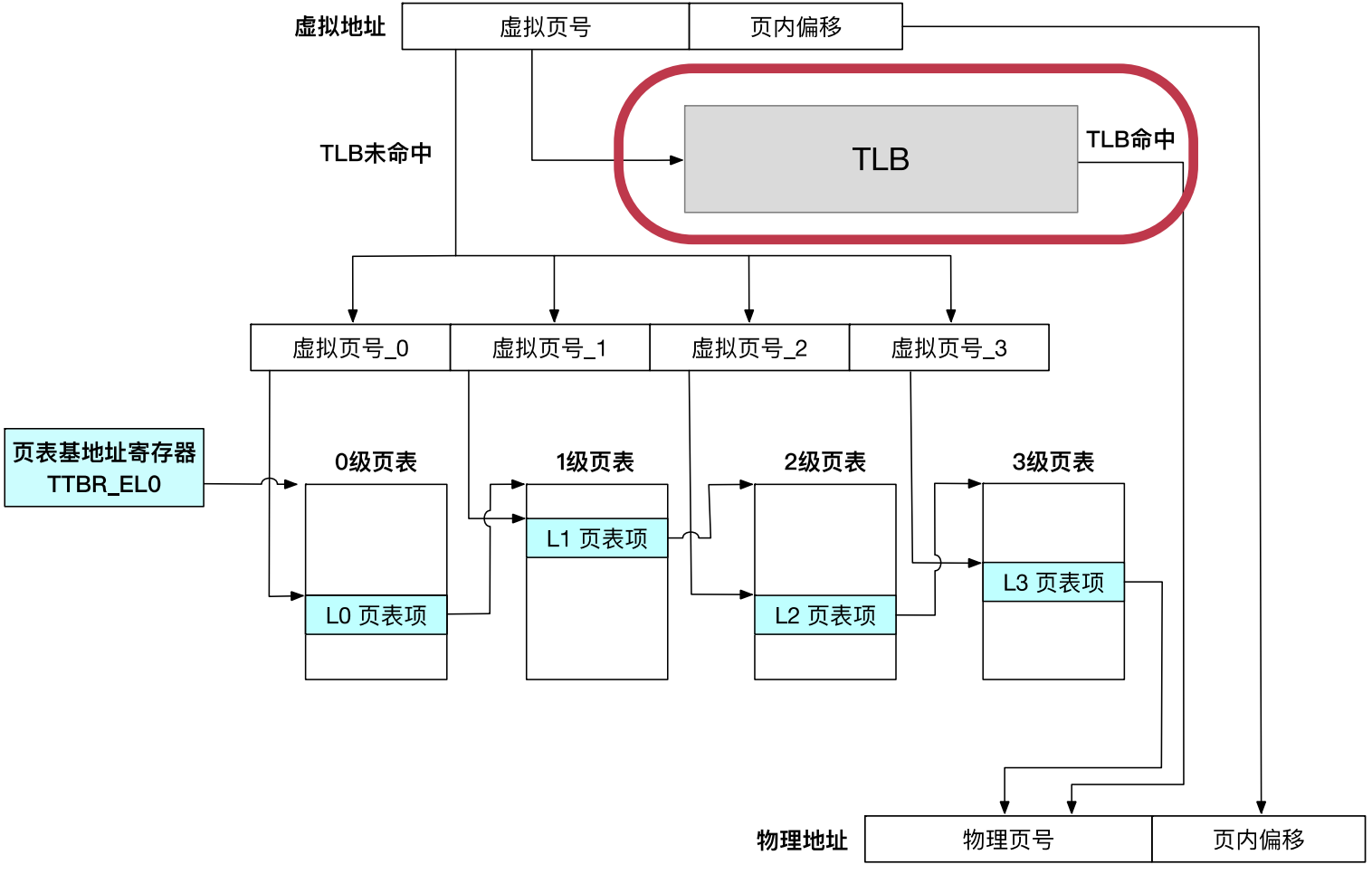
- **Cache lockdown寄存器**
  - 可以配置部分Cache不被evict，使数据一直驻留在CPU内部
  - 非统一标准，取决于具体的实现（也可不实现）
- **Cache lockdown用途**
  - 提高性能：可保证访问部分重要数据永远cache hit
    - 硬件的Cache替换策略不够完美
  - 提高安全性：限制部分数据永远不离开CPU
    - 若数据量大于Cache容量，可加密后离开CPU

（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**
  - 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

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: 可进行全局TLB刷新

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

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?
- **替换策略的问题**
  - Belady's anomaly
  - Thrashing Problem

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?