
中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

# 虚拟内存管理 I

SSE202/204: 操作系统原理

苏玉鑫

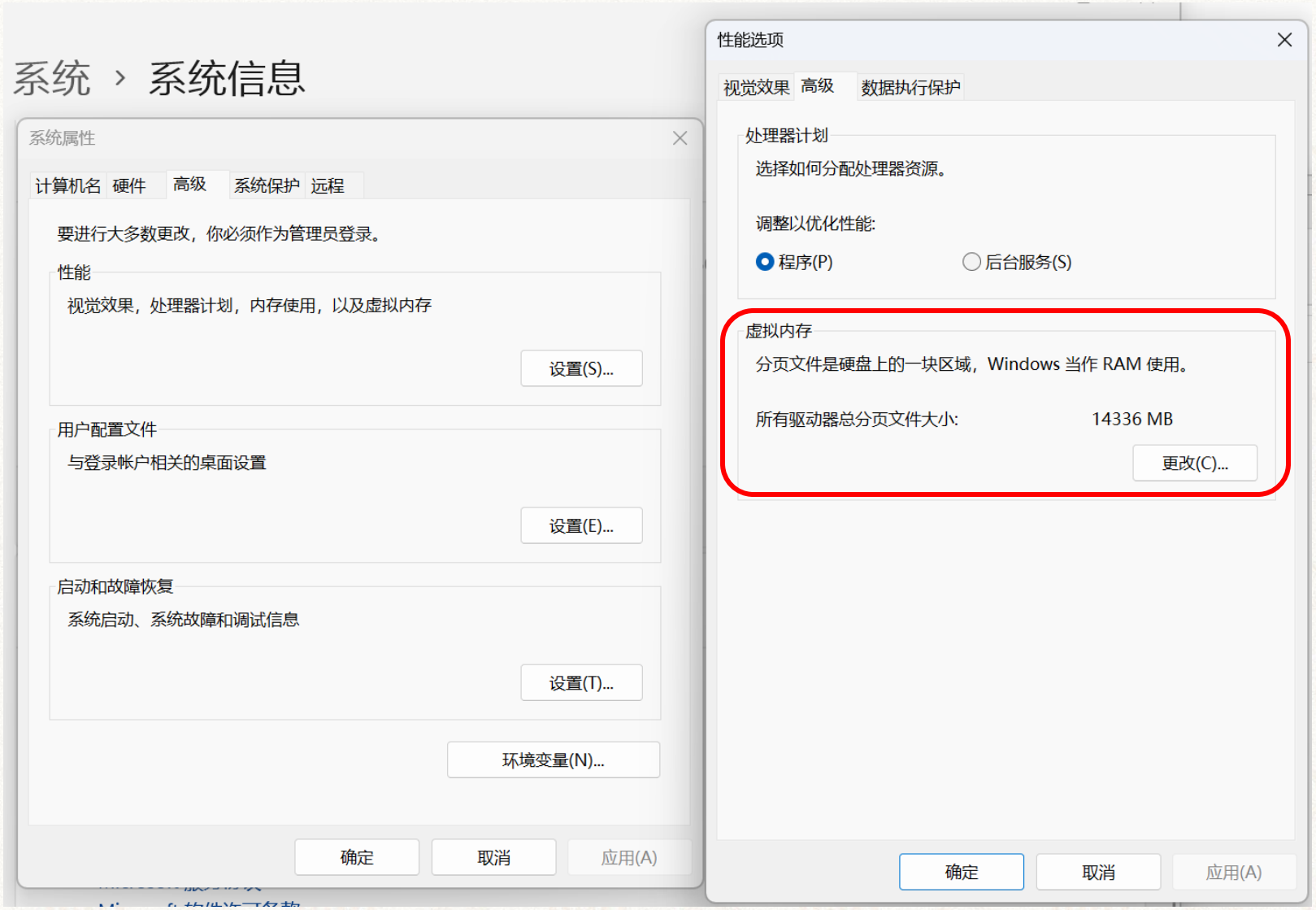
suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

# 版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

# 此虚拟内存非彼“虚拟内存”



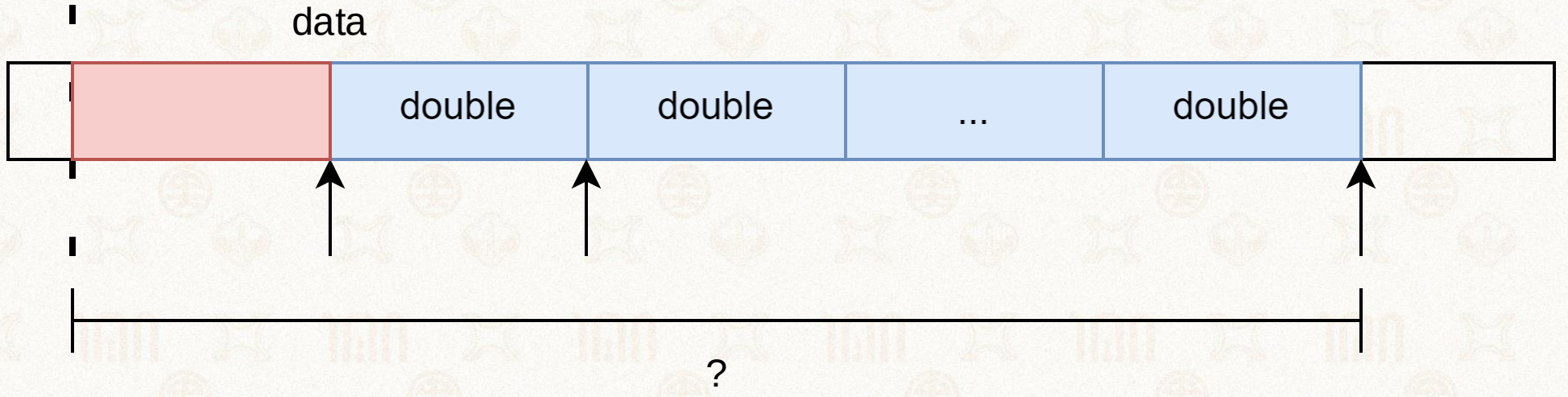
# 这段代码有什么特殊之处?

```c
#include "stdio.h"

int main() {
    double data[10] = {0};

    printf("sizeof(double) = %ld\n", sizeof(double));
    printf("address of data is %p\n", data);
    for(int i = 0; i < 20; i++) {
        printf("data[%d] = %f, addr = %p\n", i, data[i], &data[i]);
    }
    printf("success!");
    return 0;
}
```

- 数组下标越界访问
  - 非常多安全漏洞的源头

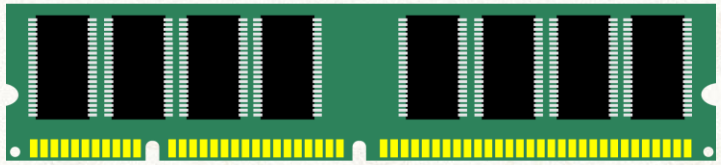


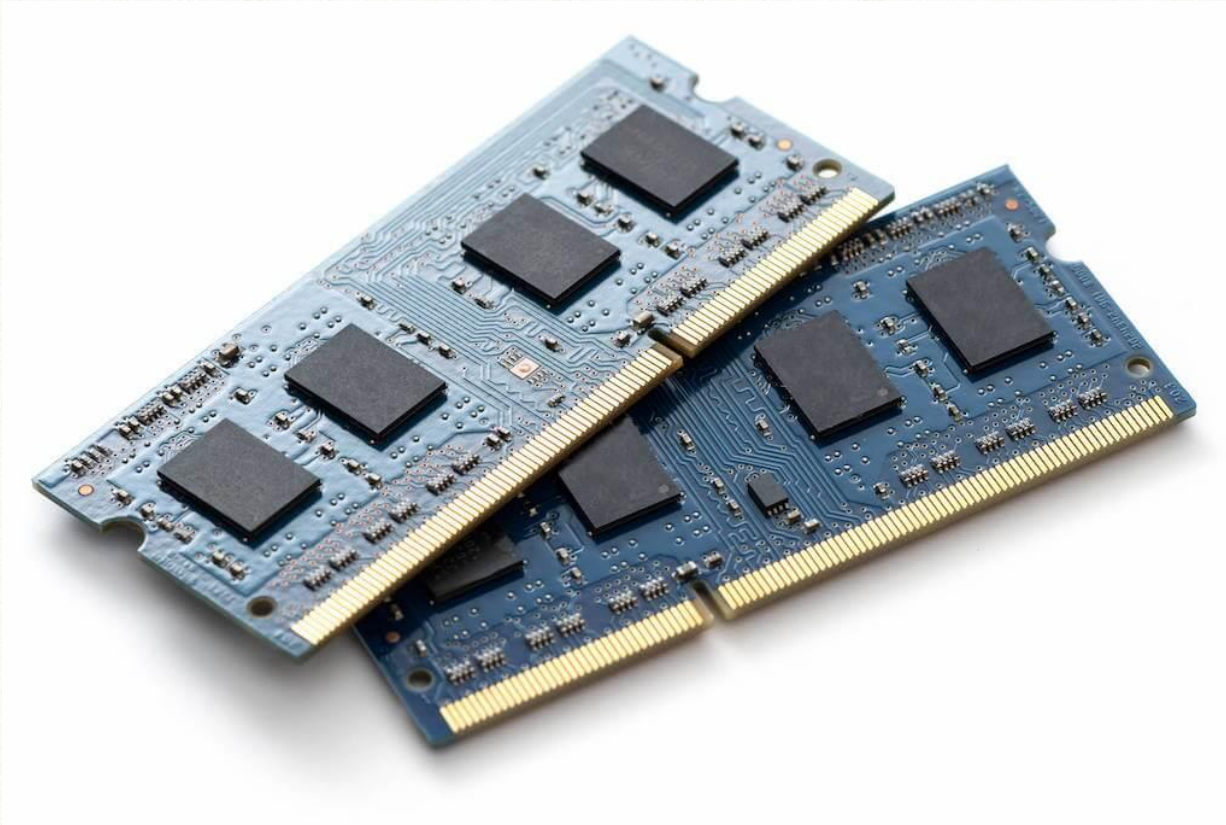
# 大纲

- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# 物理内存

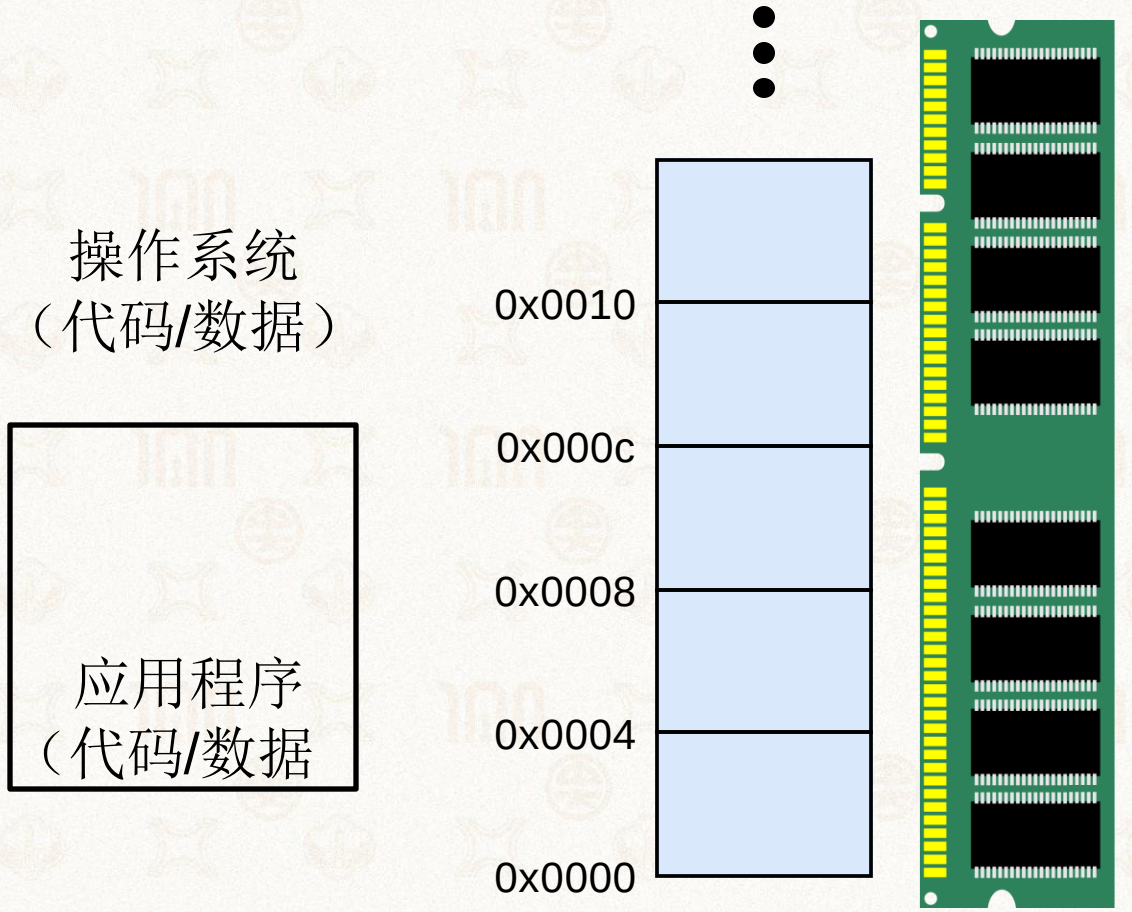
- 常说的"内存条"就是指物理内存
- 数据从磁盘中加载到物理内存后，才能被CPU访问
  - 操作系统的代码和数据
  - 应用程序的代码和数据





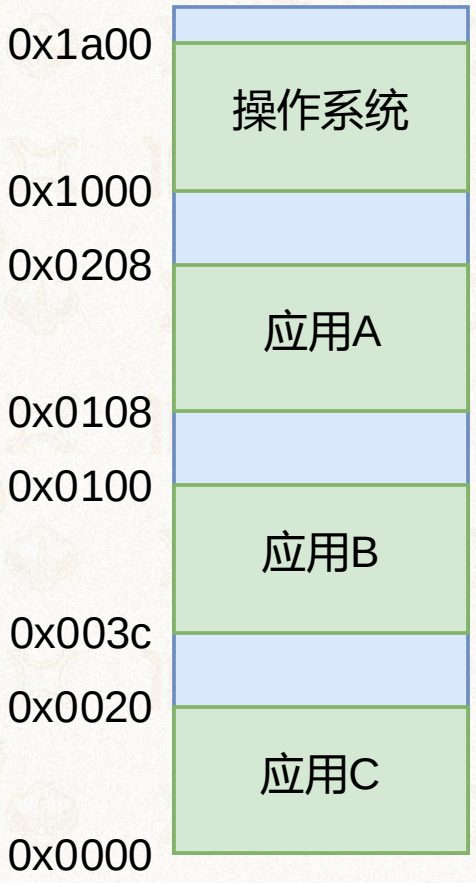
# 最早期的计算机系统

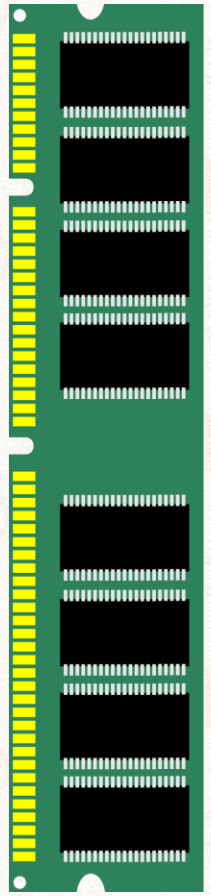
- 硬件
  - 物理内存容量小
- 软件
  - 单个应用程序 + （简单）操作系统
  - 直接面对物理内存编程
  - 各自使用物理内存的一部分



# 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）



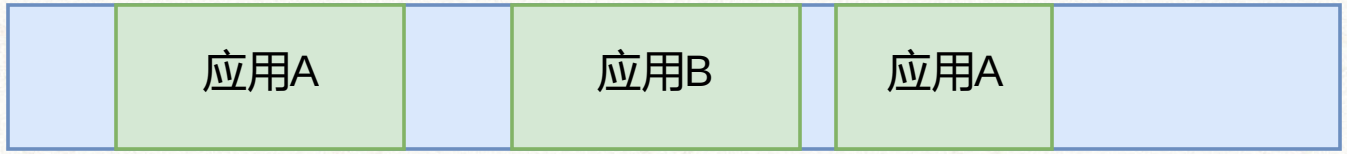



如何让OS与不同的应用程序都高效又安全地使用物理内存资源?

# 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
  - 一个应用不用关心其他应用占了什么地址，不受其他应用的影响
  - 看不见其他应用的信息，带来更强的隔离能力




  - 回想刚才内存泄漏的例子，为什么要追求隔离能力

# 大纲

- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# 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

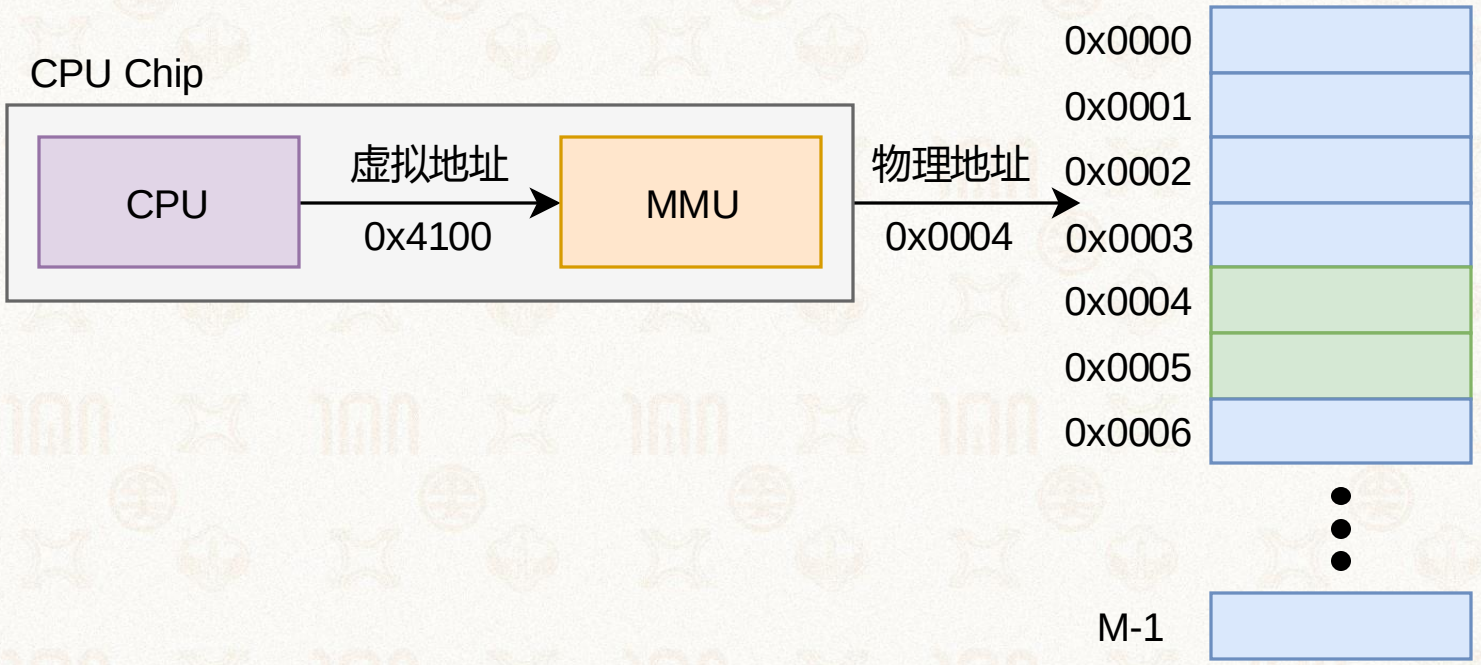
```
hello.o:     file format elf64-x86-64
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .plt.sec      00000010  0000000000001050  0000000000001050  00001050  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .text         00000185  0000000000001060  0000000000001060  00001060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 17 .rodata       00000011  0000000000002000  0000000000002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

# 地址翻译过程

- MMU (Memory Management Unit)根据一定的规则翻译地址

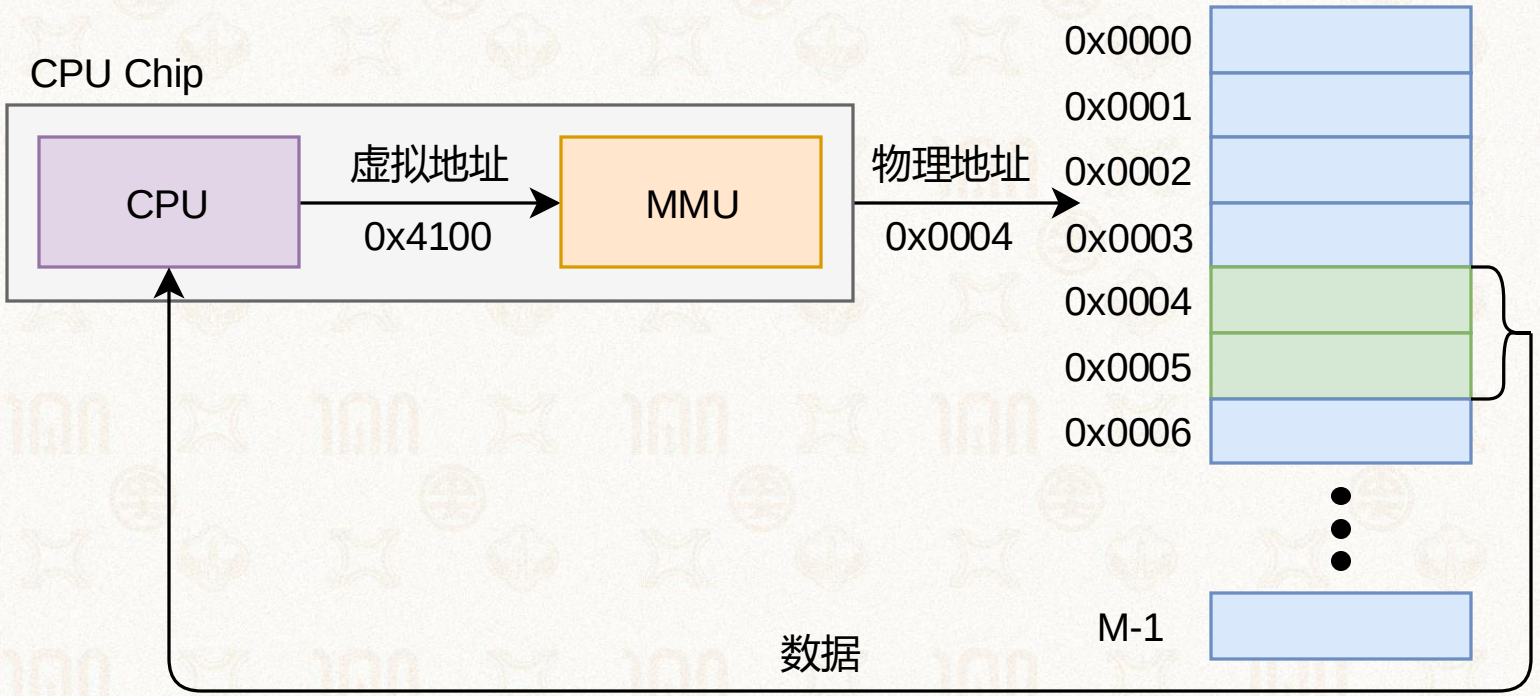
# 地址翻译过程

- MMU (Memory Management Unit)根据一定的规则翻译地址



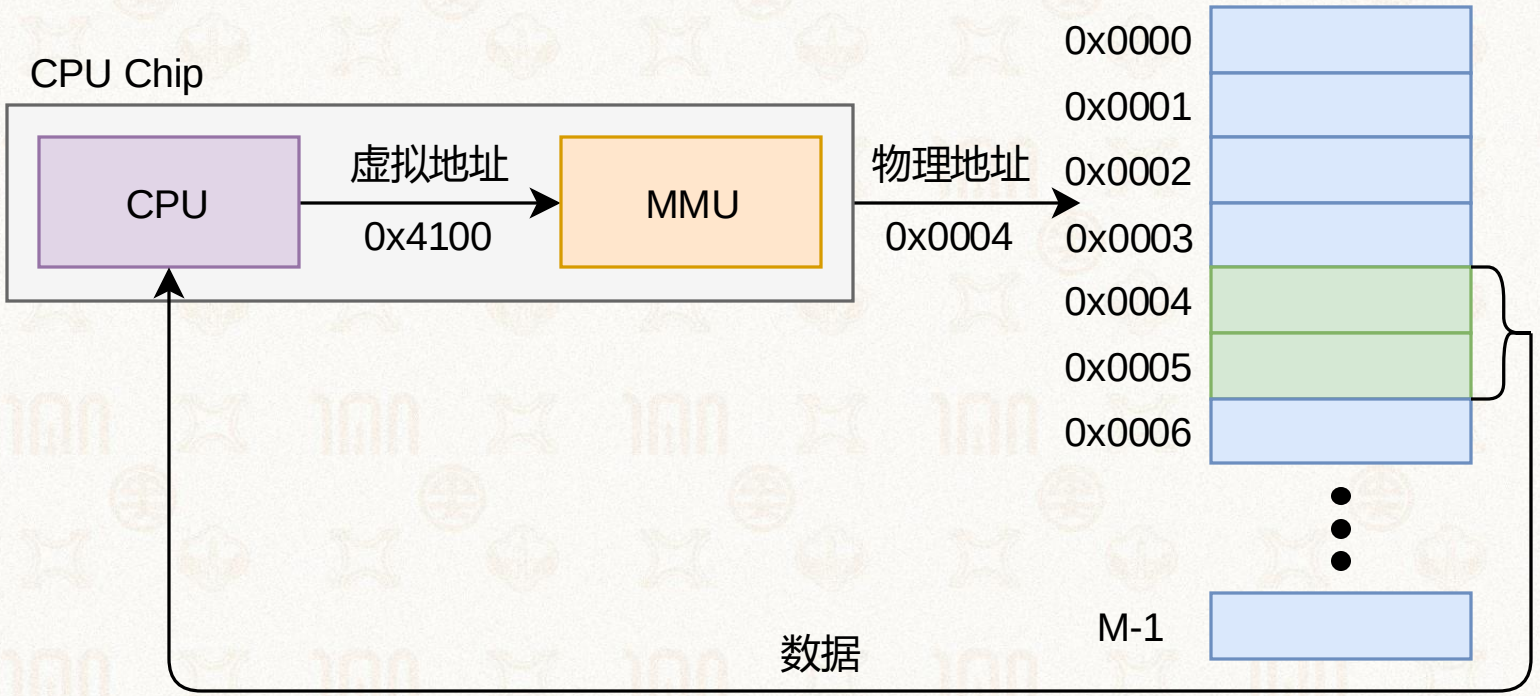
# 地址翻译过程

- MMU (Memory Management Unit)根据一定的规则翻译地址



# 地址翻译过程

- MMU (Memory Management Unit)根据一定的规则翻译地址
- 翻译规则取决于虚拟内存采用的组织机制
  - 分段机制
  - 分页机制



# 大纲

- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# 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

| 段号 | 起始地址 | 本段长度 |
|---|---|---|
| 0 | 0x0200000 | 16M |
| 1 | 0x0500000 | 2M |
| 2 | 0x0800000 | 512K |
| 3 | 0x2000000 | 256M |

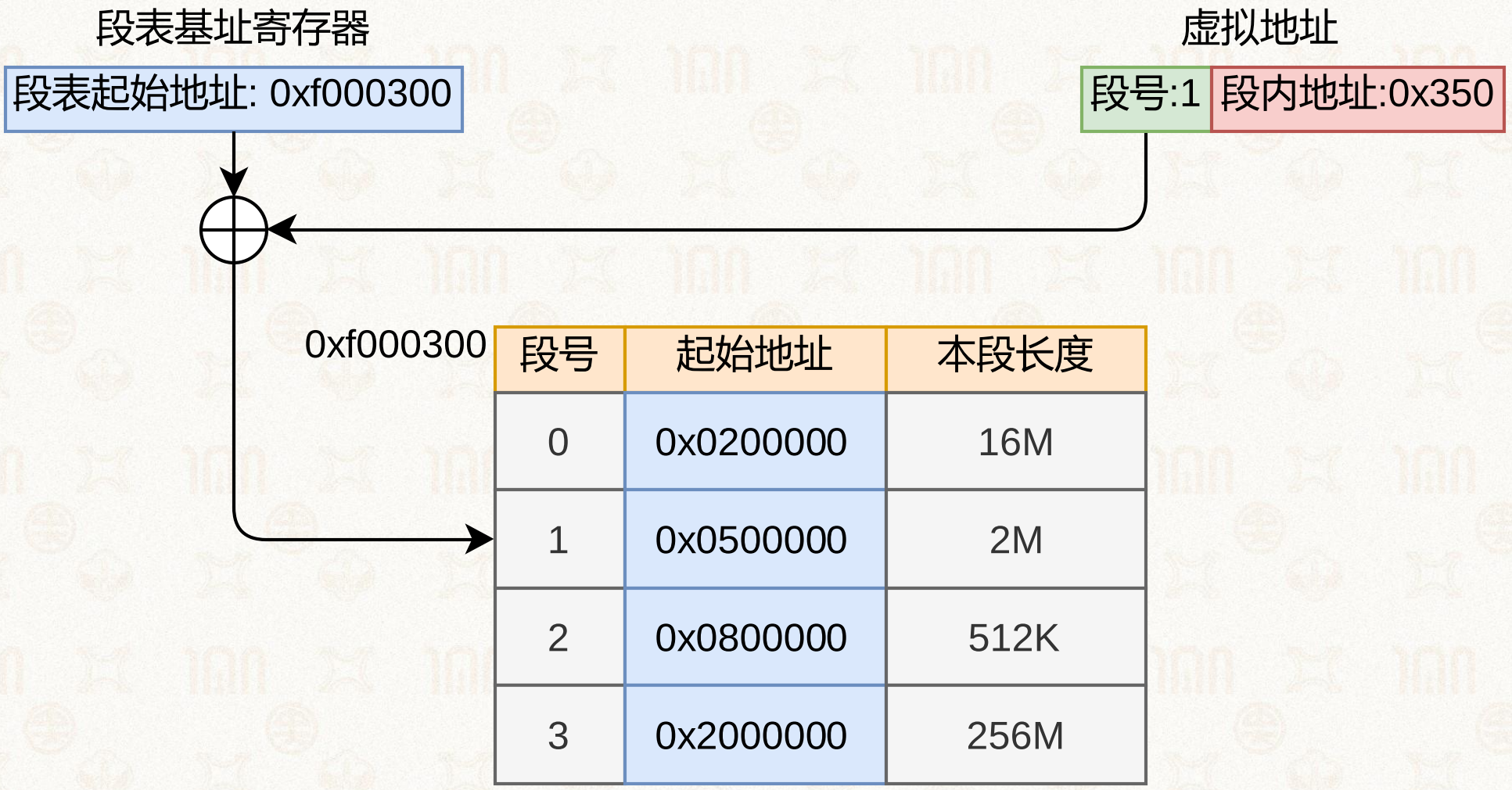
# 分段机制

虚拟地址

| 段号:1 | 段内地址:0x350 |
| --- | --- |

# 分段机制

# 分段机制

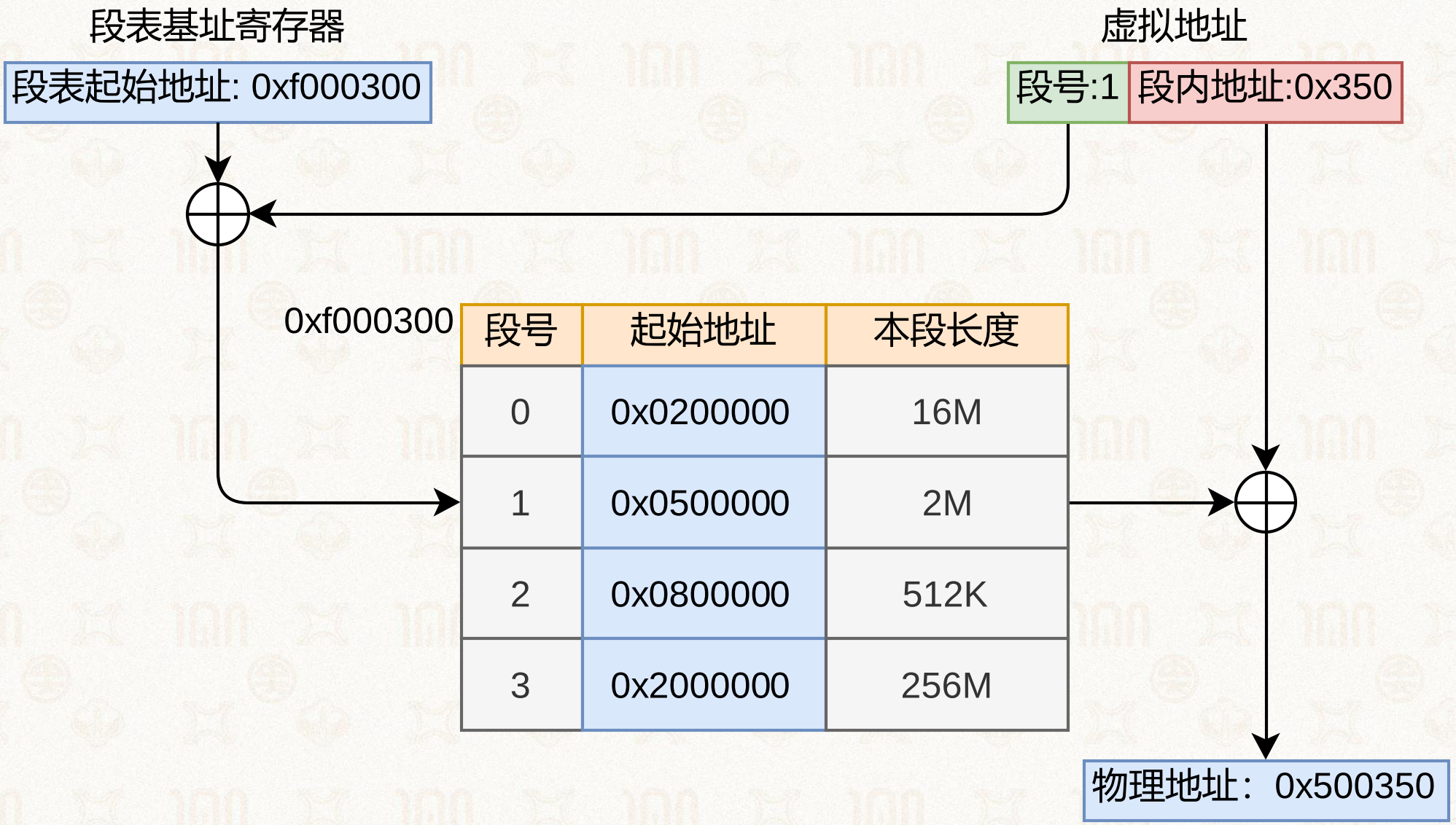



| 段号 | 起始地址 | 本段长度 |
|---|---|---|
| 0 | 0x0200000 | 16M |
| 1 | 0x0500000 | 2M |
| 2 | 0x0800000 | 512K |
| 3 | 0x2000000 | 256M |

# 分段机制

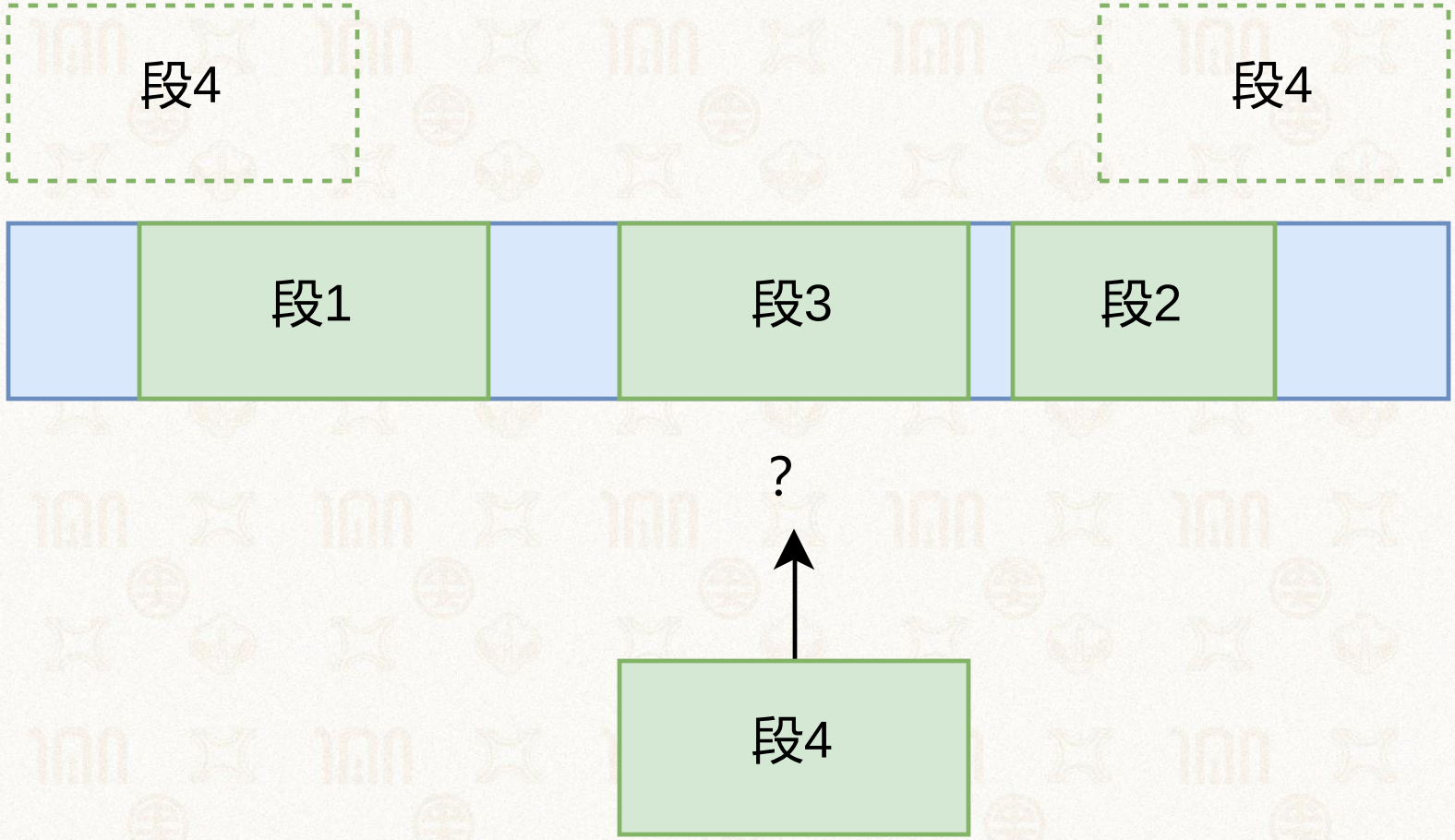

段表基址寄存器
段表起始地址: 0xf000300
虚拟地址
段号:1
段内地址:0x350
0xf000300
段号
起始地址
本段长度
0
0x0200000
16M
1
0x0500000
2M
2
0x0800000
512K
3
0x2000000
256M

# 分段机制


段表基址寄存器
段表起始地址: 0xf000300
虚拟地址
段号:1
段内地址:0x350
0xf000300
段号
起始地址
本段长度
0
0x0200000
16M
1
0x0500000
2M
2
0x0800000
512K
3
0x2000000
256M
物理地址：0x500350

# 分段机制

- 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

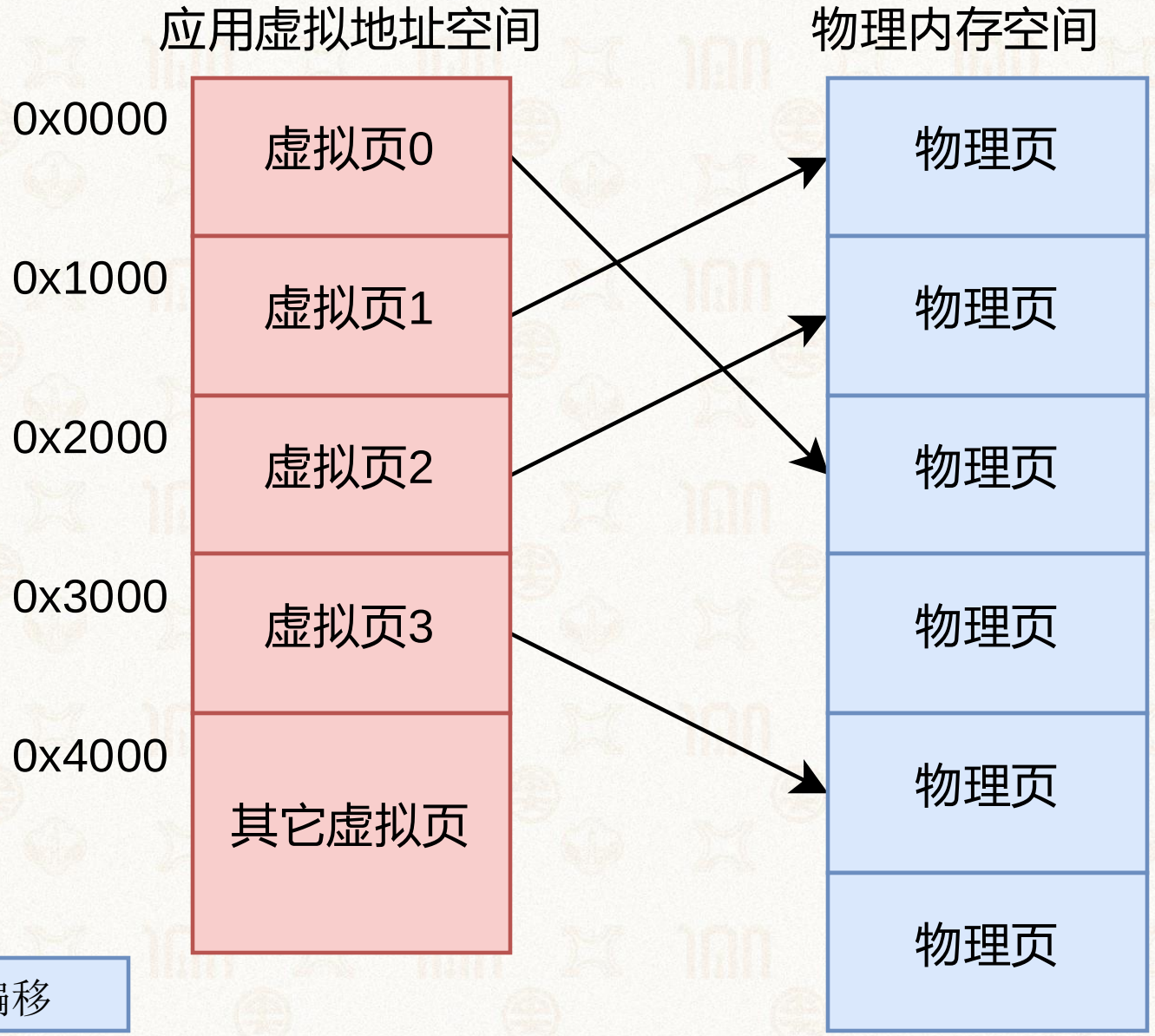


# 大纲

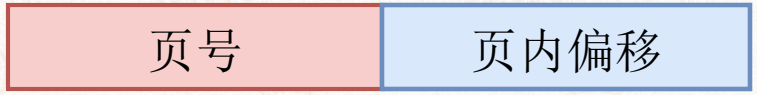
- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# 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

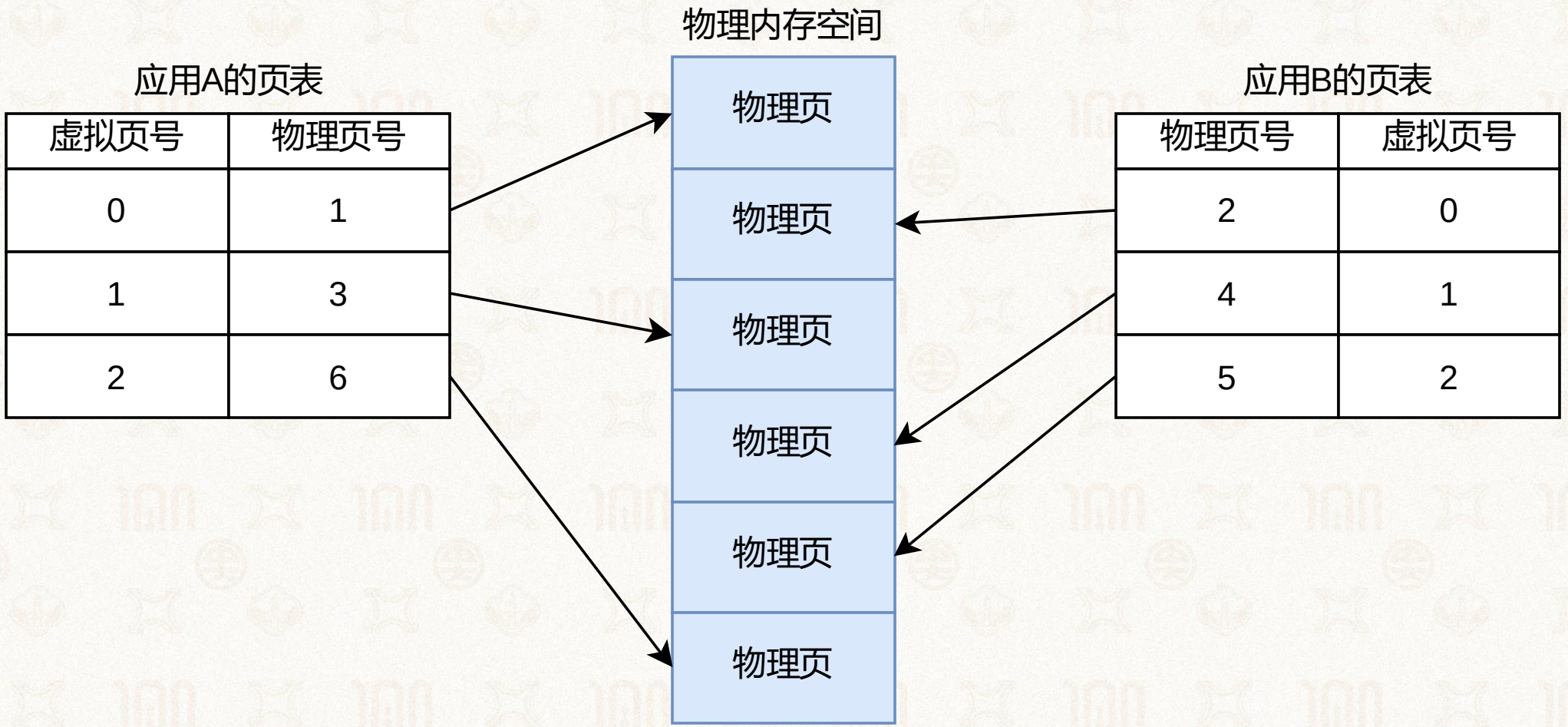



虚拟地址：



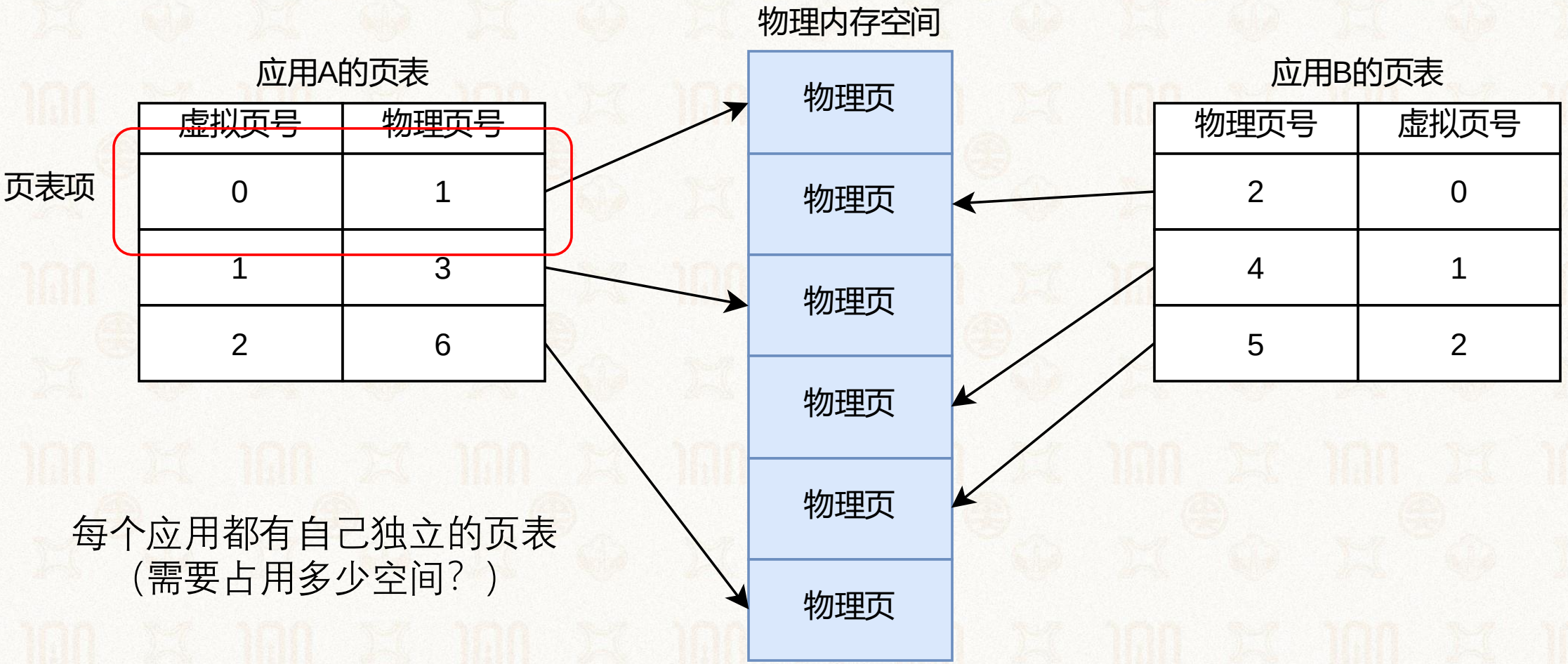
# 页表：分页机制的核心数据结构

- 页表包含多个页表项，存储虚拟页到物理页的映射



# 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

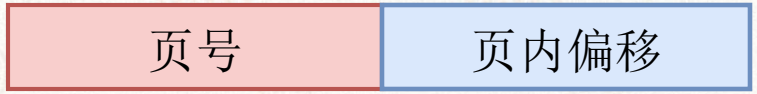


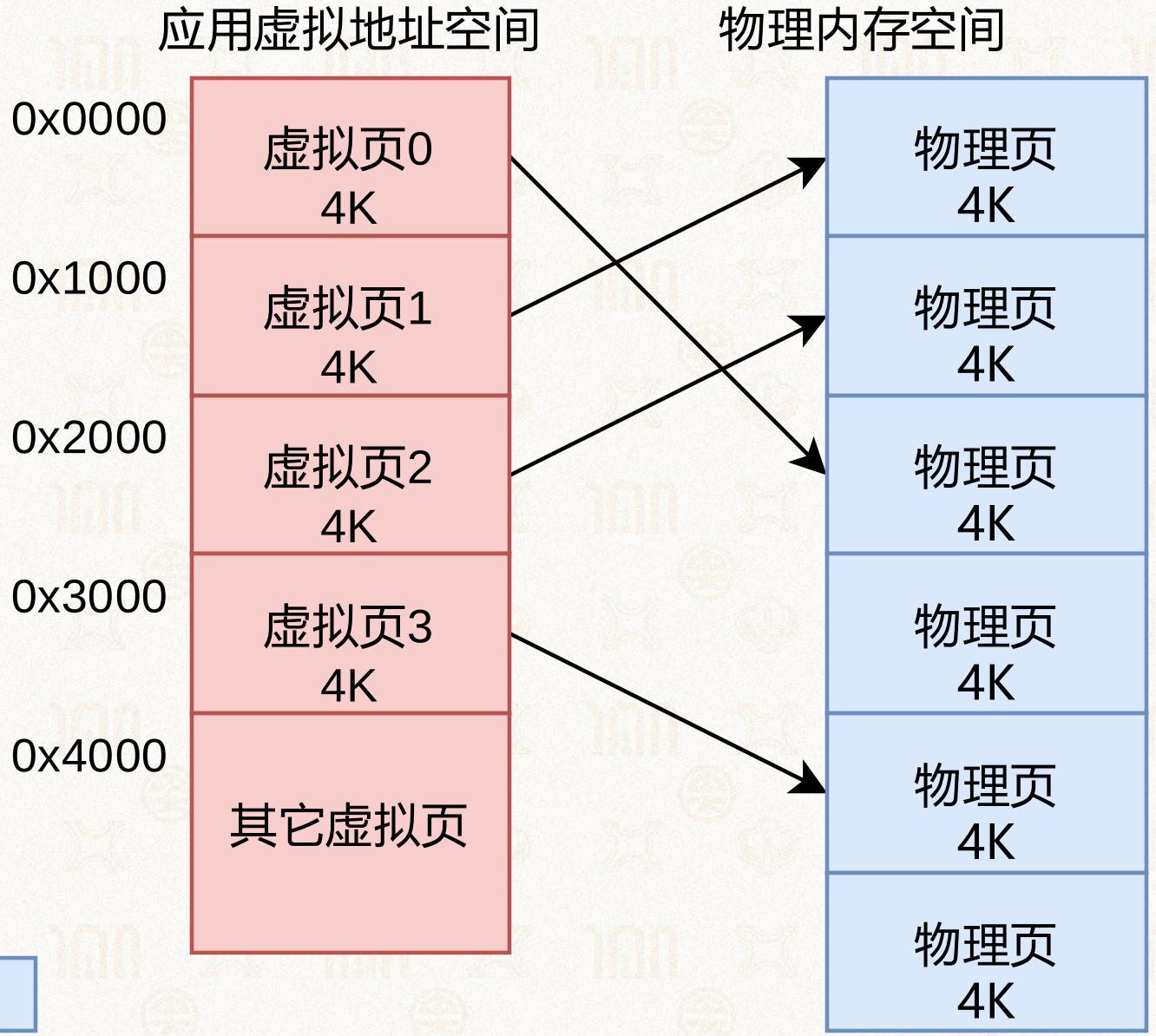
# 单级页表的问题

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

- 继续看之前的代码，用内存泄漏算一下Linux的页有多大

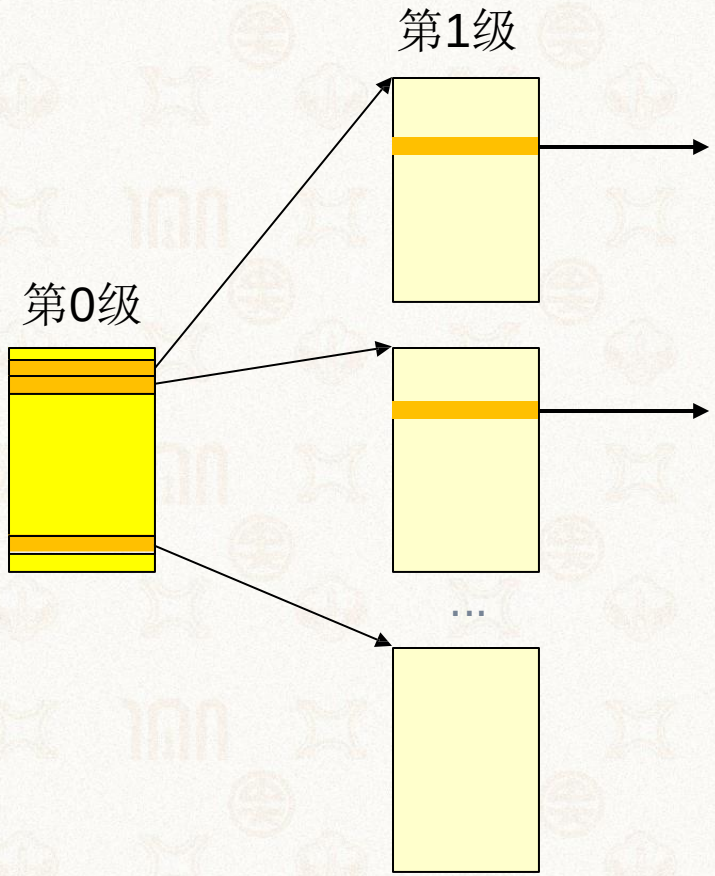
虚拟地址：



# 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4字节，
    - 页表大小：$2^{32}$ / 4K * 4 = 4MB
  - 64位地址空间，页4K，页表项8字节，
    - 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"

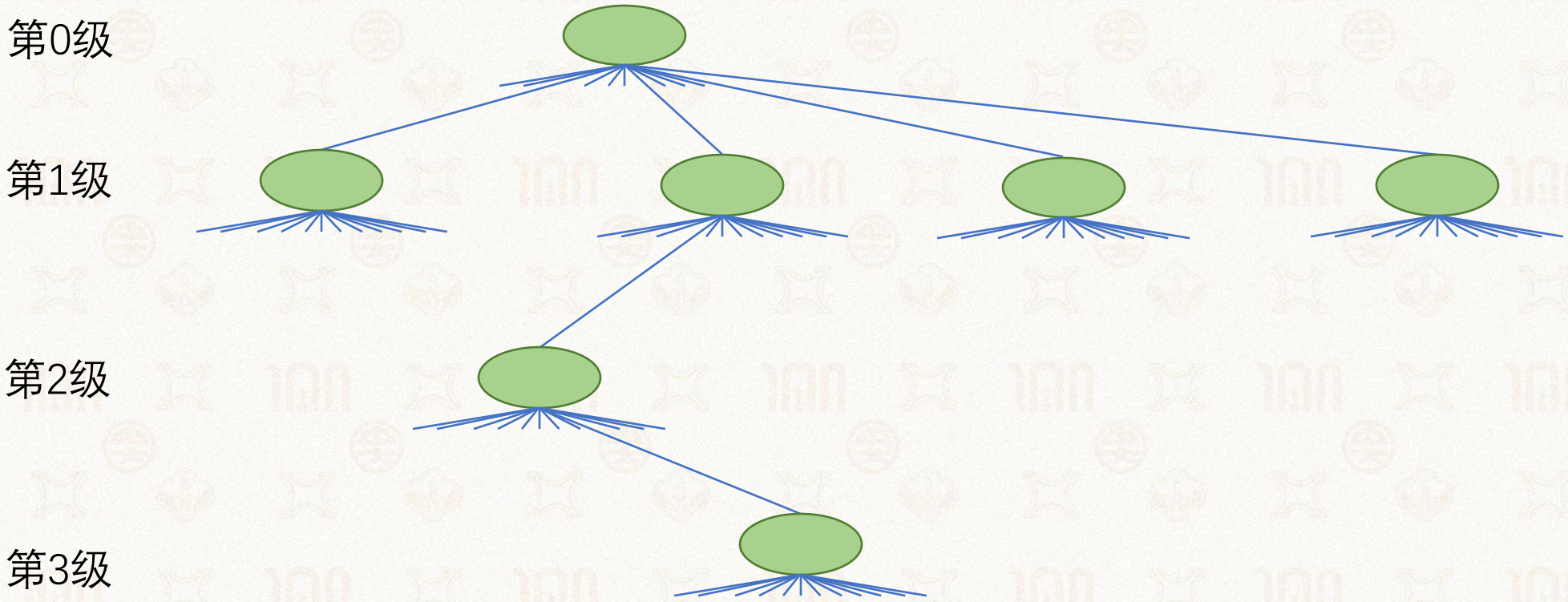


# 大纲

- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# 多级页表 vs. 字典树(R-way tries)

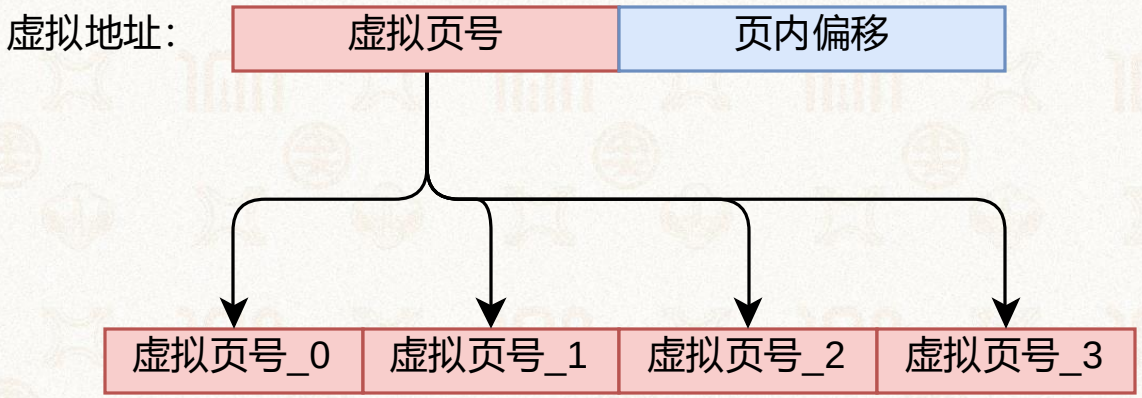
- 四级字典树可以保存所有不超过4个字符长度的单词
- 四级页表可以保存xx长度不超过4的页号



# AArch64的4级页表


- 假设一个页的大小为4K字节，那么页内偏移需要用12位二进制表示
- 虚拟地址有效寻址空间只有48位

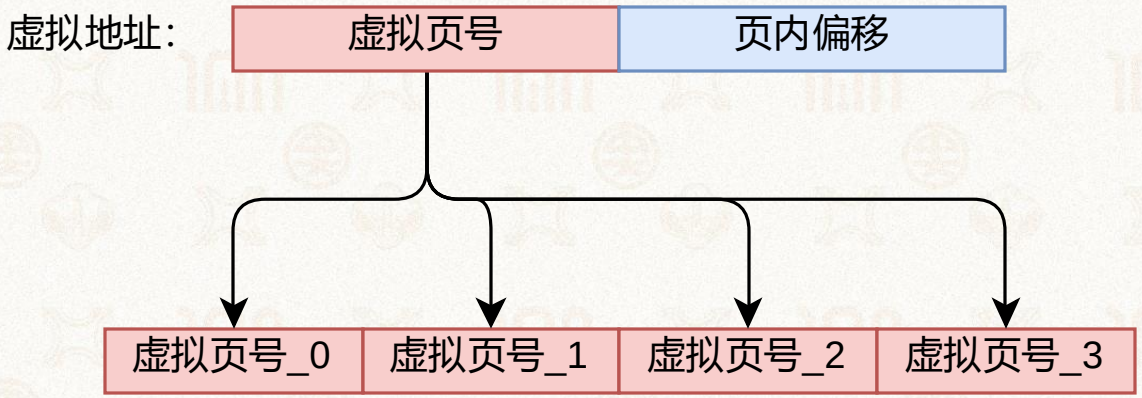
# AArch64的4级页表

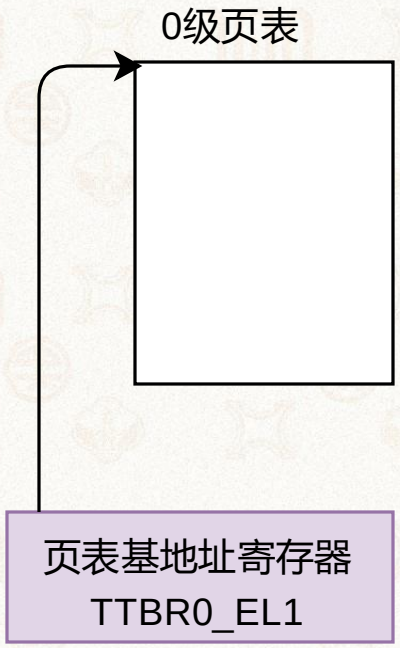



- 虚拟地址48位，减去页内偏移12位，则：

- 虚拟页号只有36位
  - 平均分成4份，每份9位，每份可以指向512个页表项

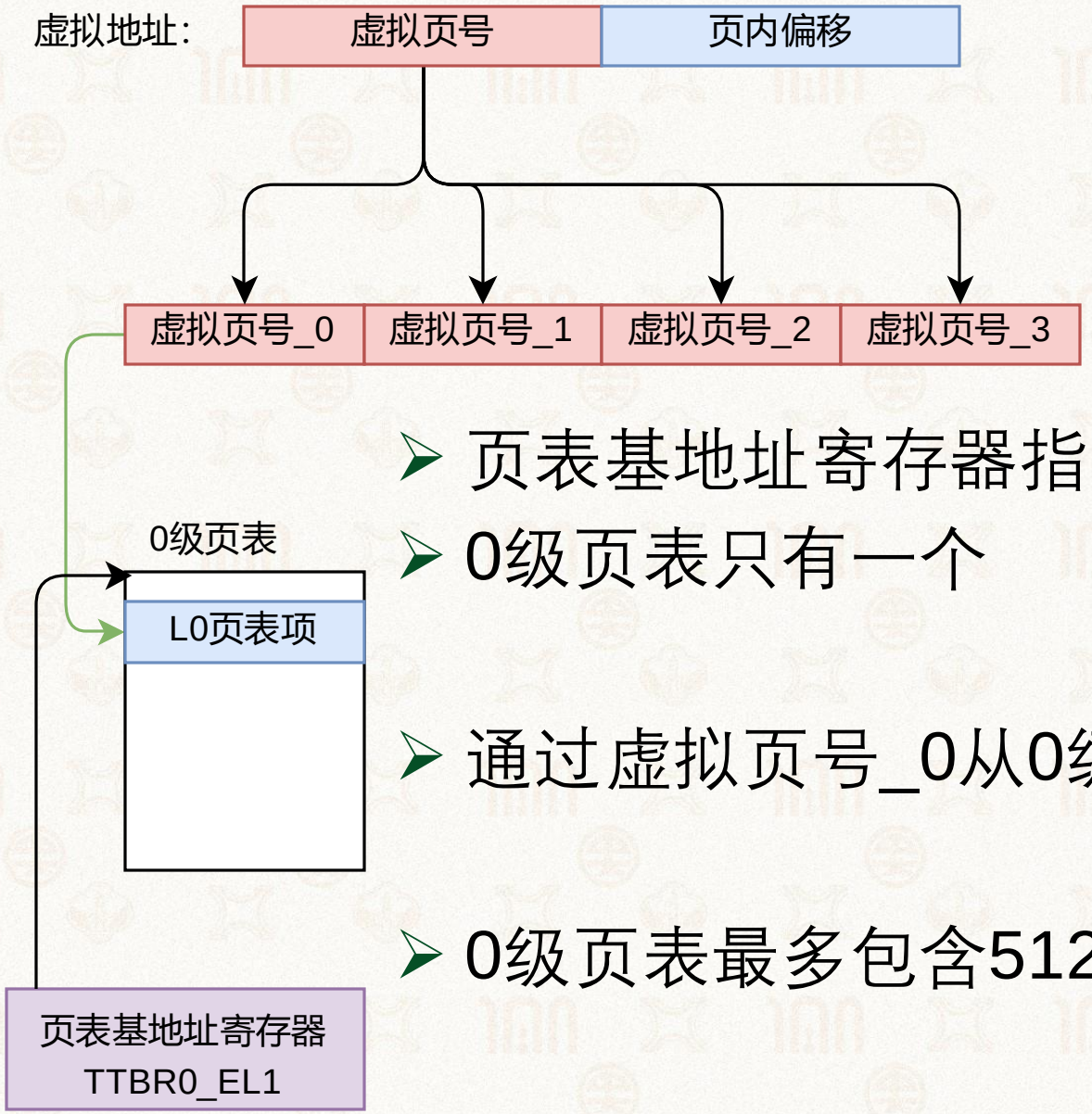
# AArch64的4级页表



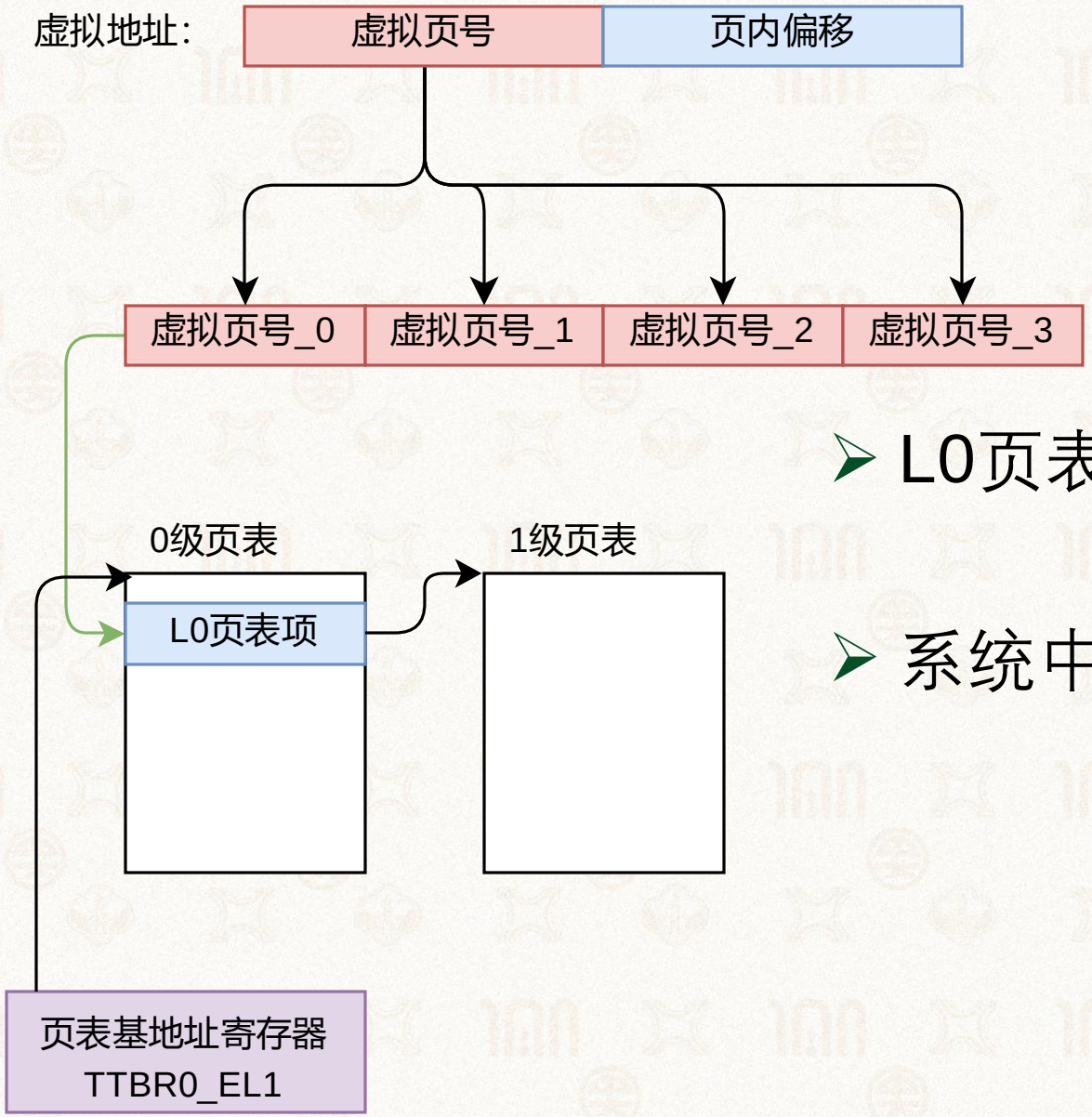

- 页表基地址寄存器(Translation Table Base Register)
  - TTBR0_EL1 & TTBR1_EL1
  - 根据虚拟地址第63位选择， 若为0则选择TTBR0_EL1
  - 通常（以Linux为例）
    - 应用程序使用TTBR0_EL1
    - 操作系统使用TTBR1_EL1

# AArch64的4级页表



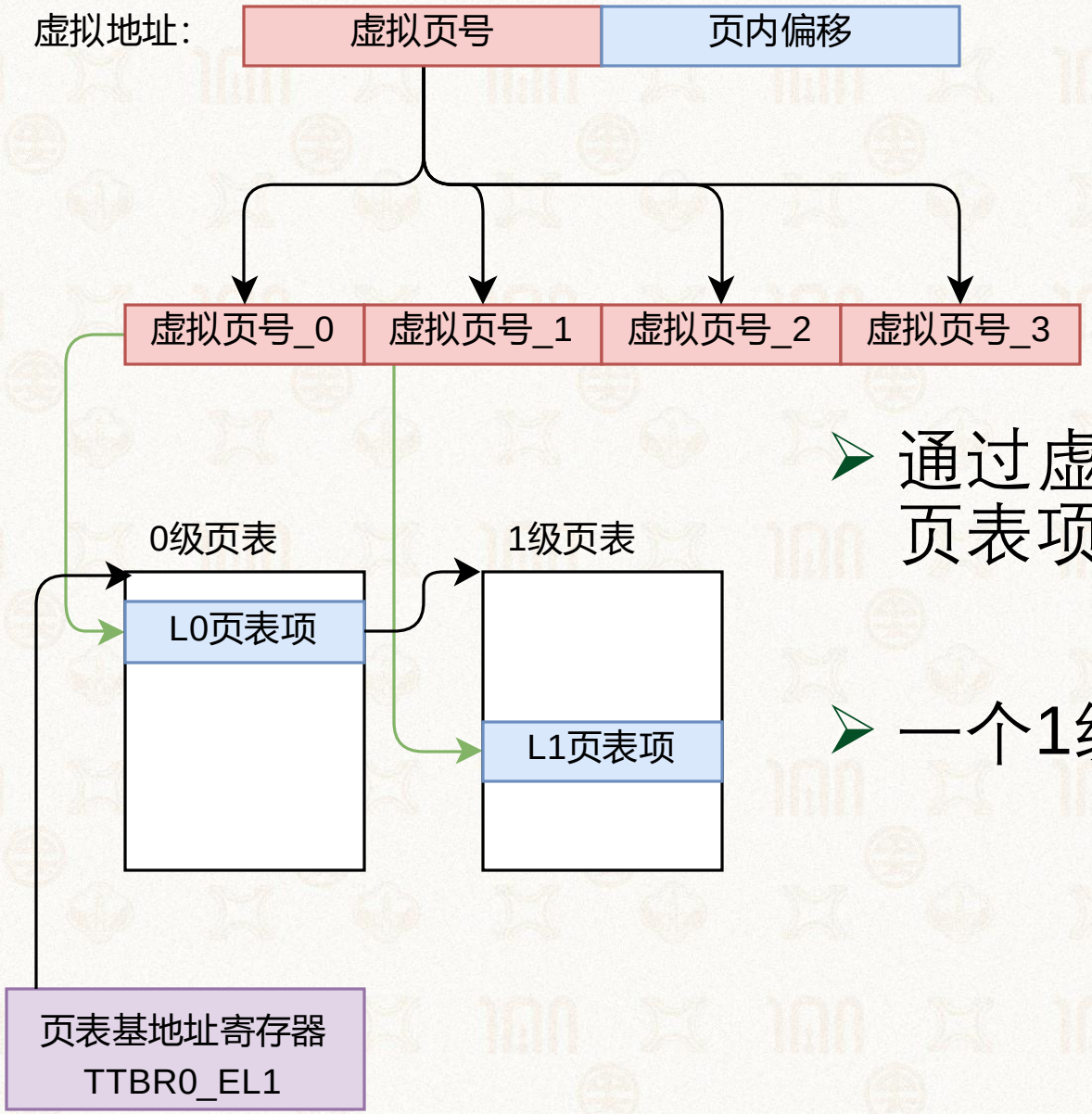

- 页表基地址寄存器指向0级页表的基地址
- 0级页表只有一个
- 通过虚拟页号_0从0级页表中查找L0页表项
- 0级页表最多包含512个L0页表项

# AArch64的4级页表



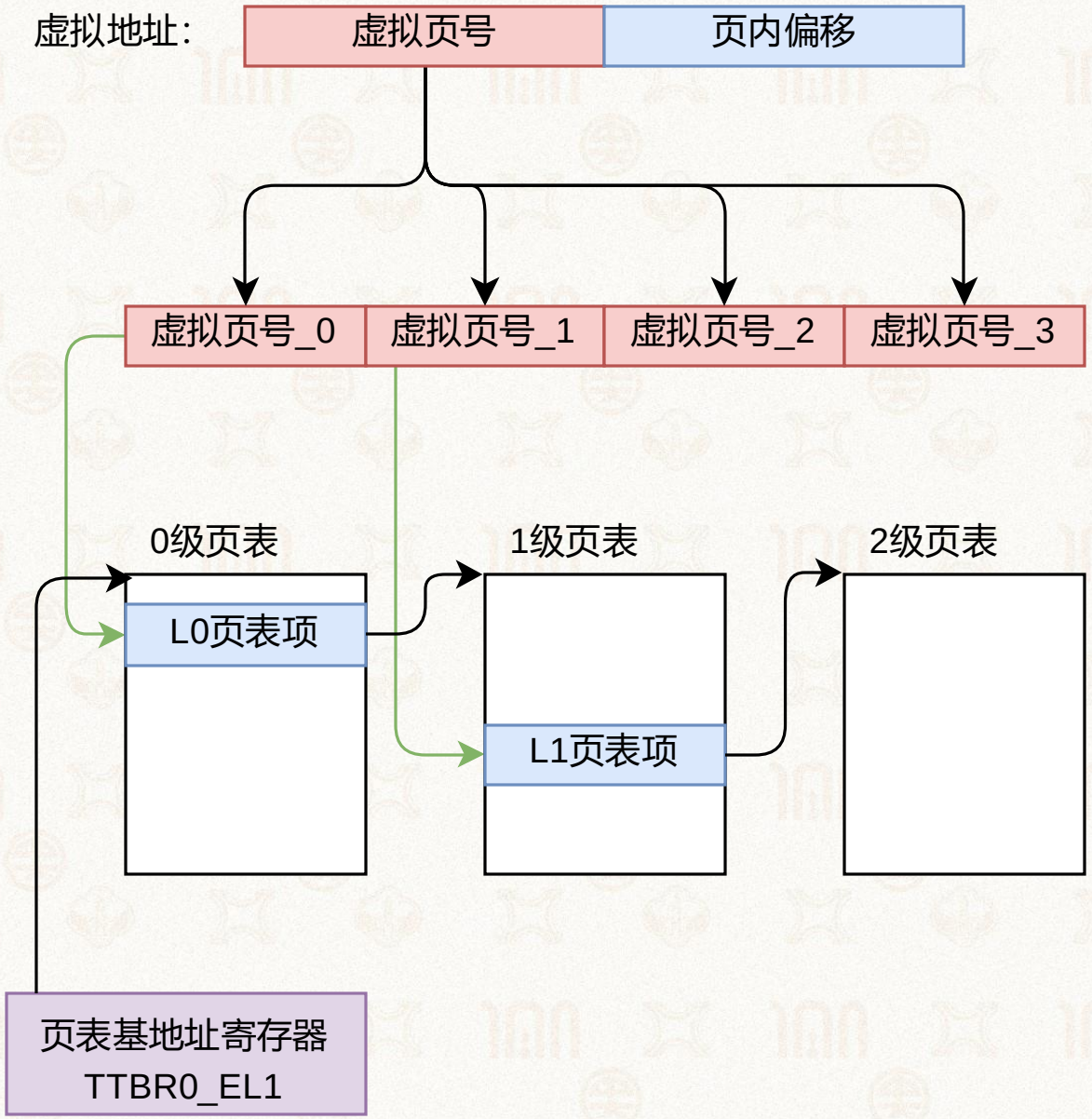

- L0页表项存有对应1级页表的基地址
- 系统中最多可能有512个1级页表

# AArch64的4级页表



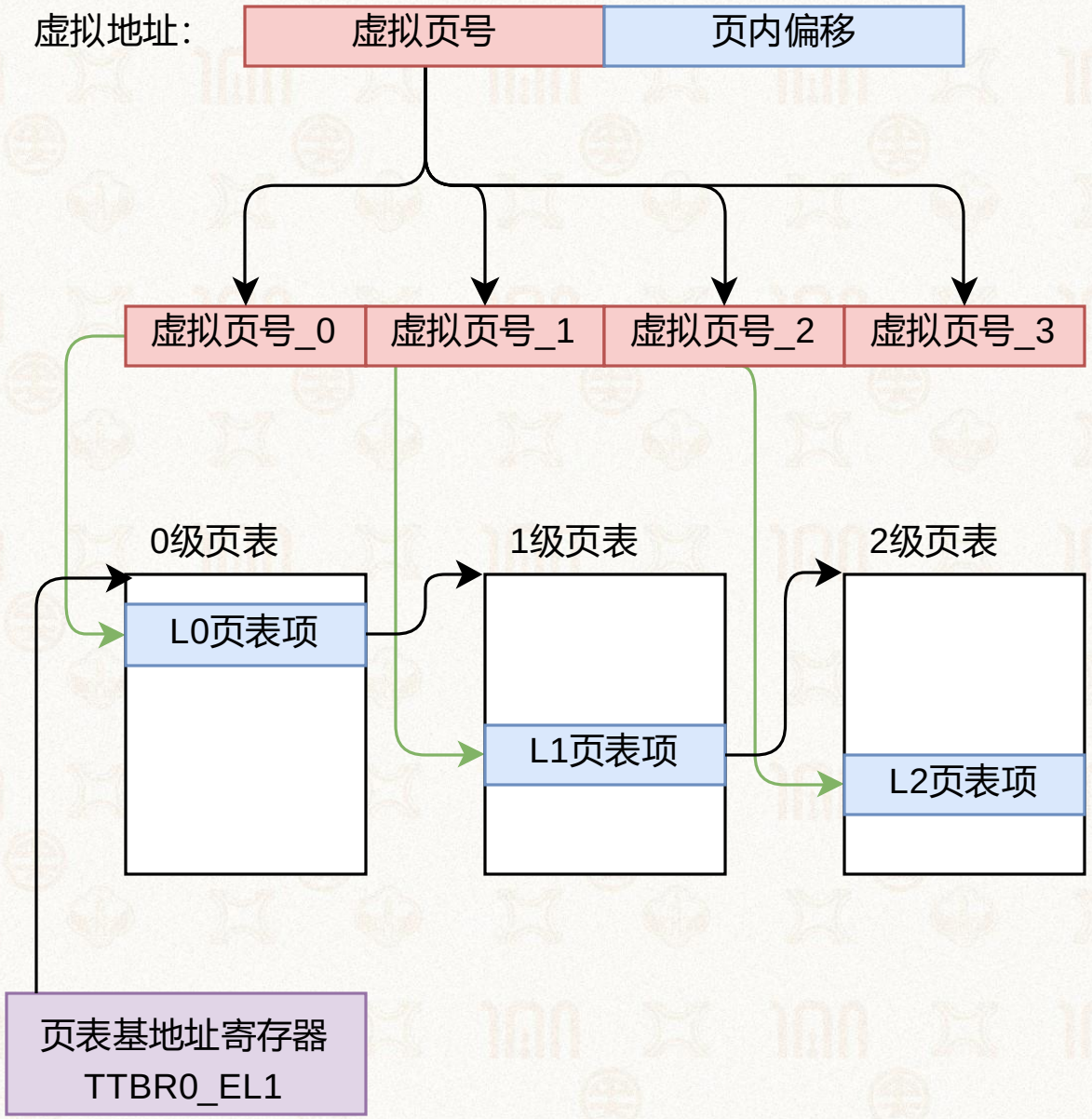

- 通过虚拟页号_1从1级页表中查找L1页表项
- 一个1级页表最多包含512个L1页表项

# AArch64的4级页表



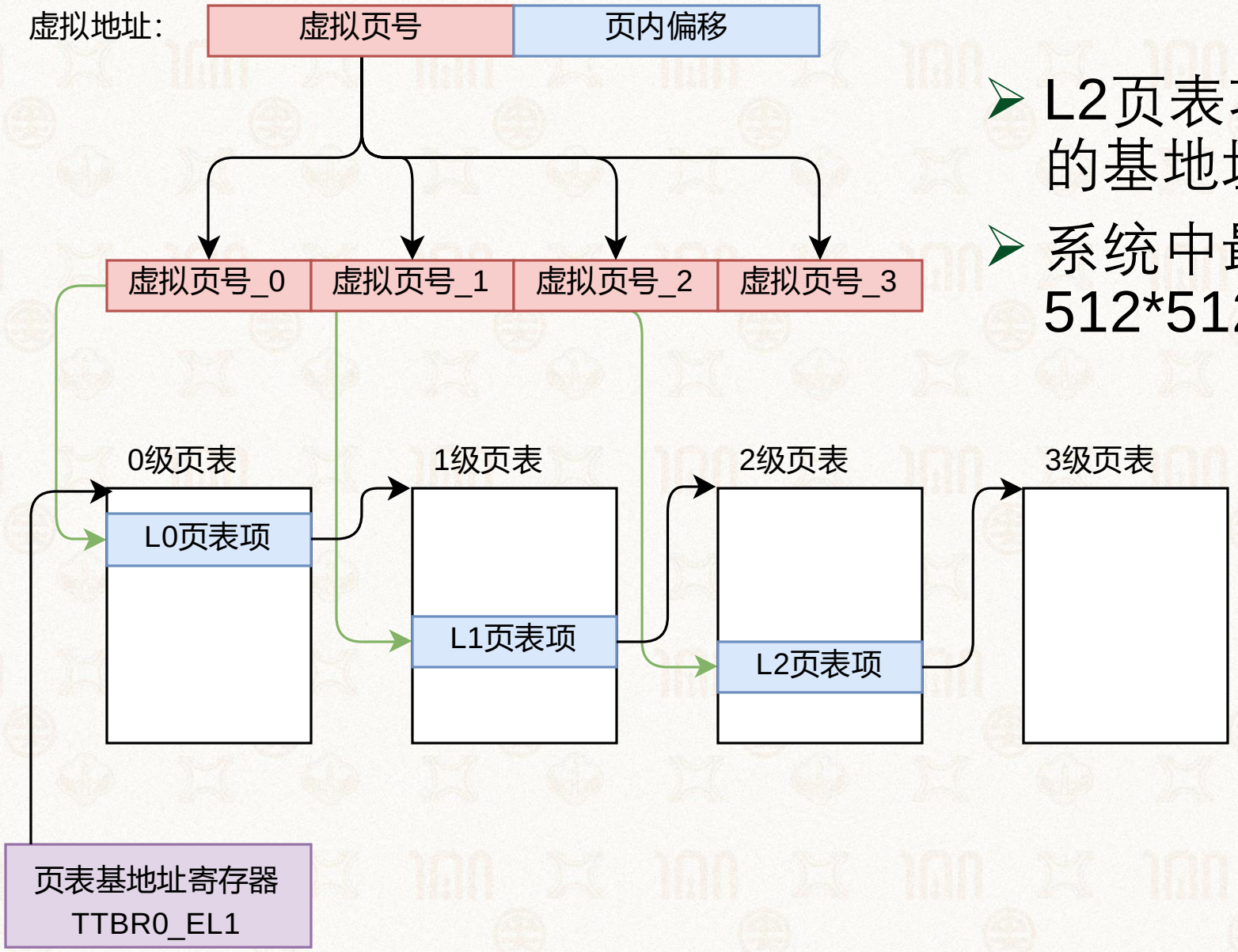

- L1页表项存有对应2级页表的基地址
- 系统中最多可能有512*512个2级页表

# AArch64的4级页表



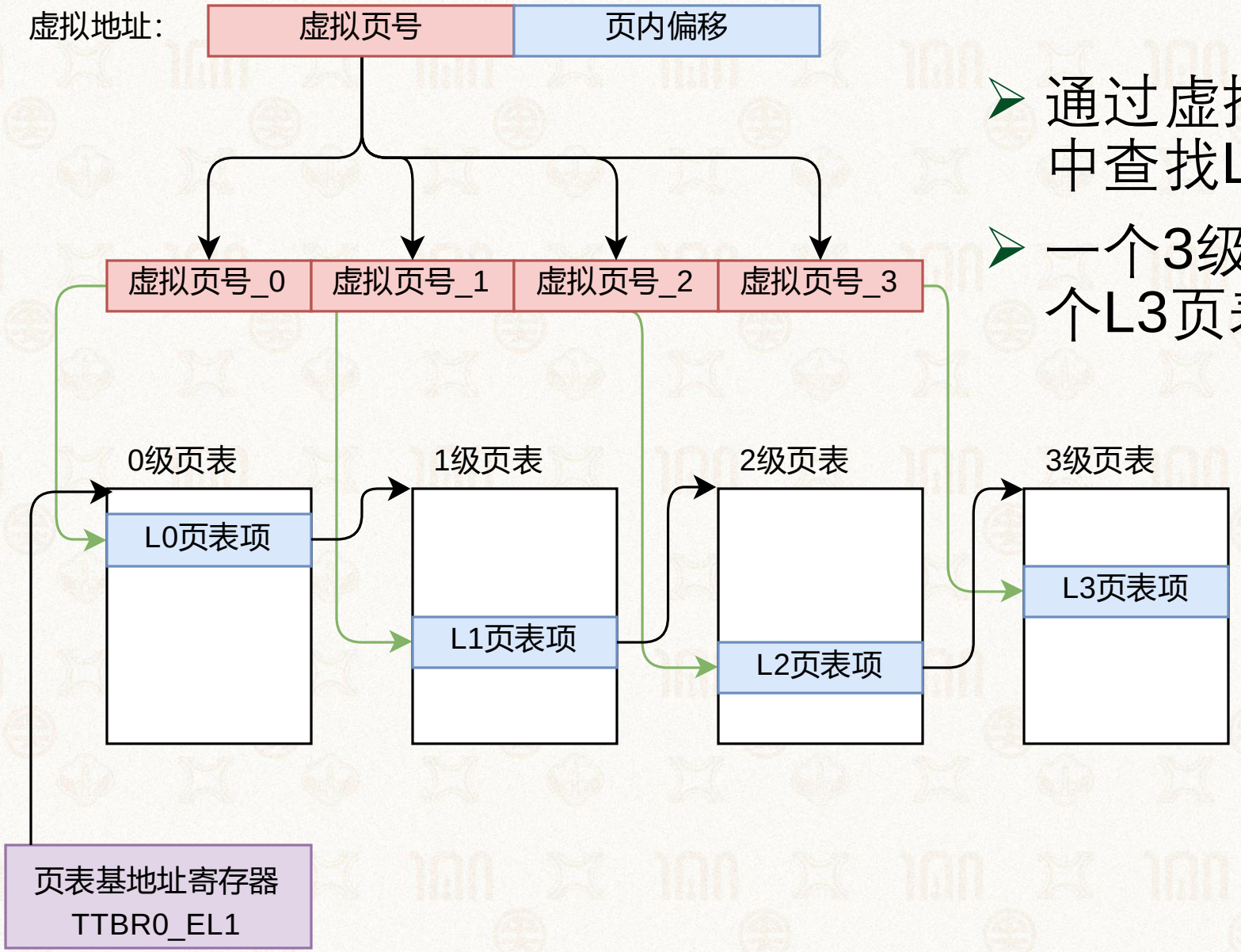

- 通过虚拟页号_2从2级页表中查找L2页表项
- 一个2级页表最多包含512个L2页表项

# AArch64的4级页表



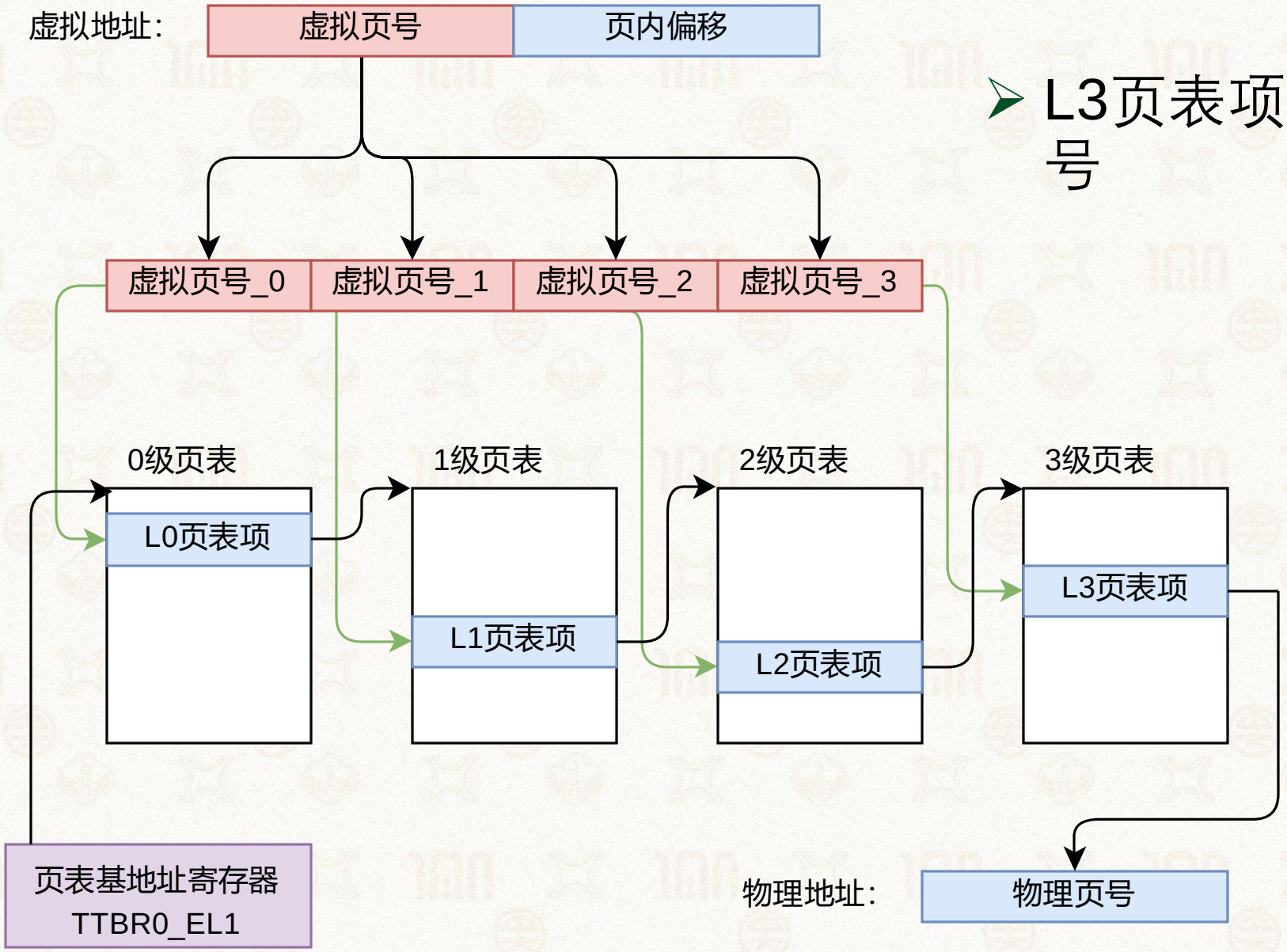

- L2页表项存有对应3级页表的基地址
- 系统中最多可能有512*512*512个3级页表

# AArch64的4级页表



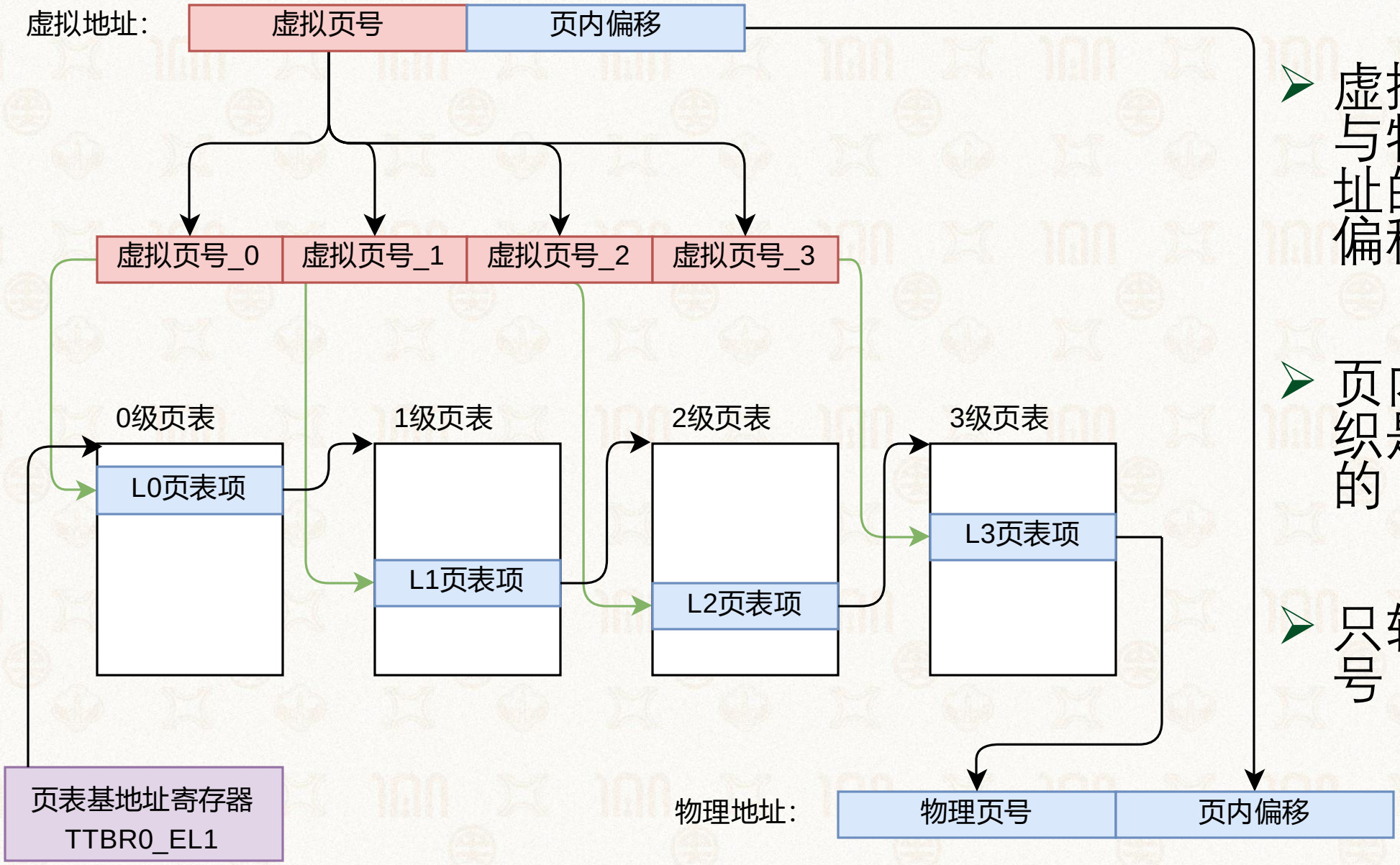

- 通过虚拟页号_3从3级页表中查找L3页表项
- 一个3级页表最多包含512个L3页表项

# AArch64的4级页表



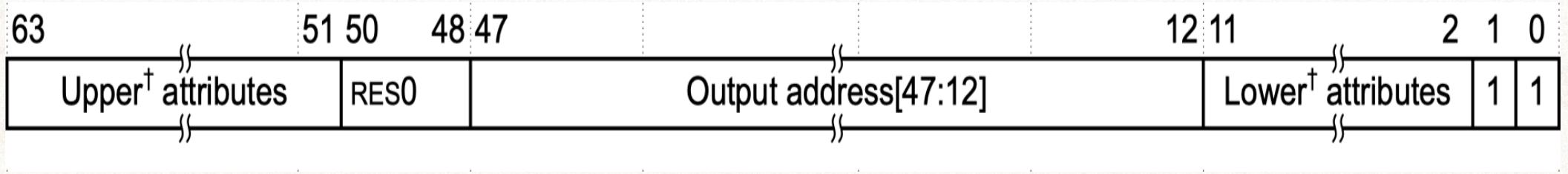

- L3页表项存有对应物理页号

# AArch64的4级页表

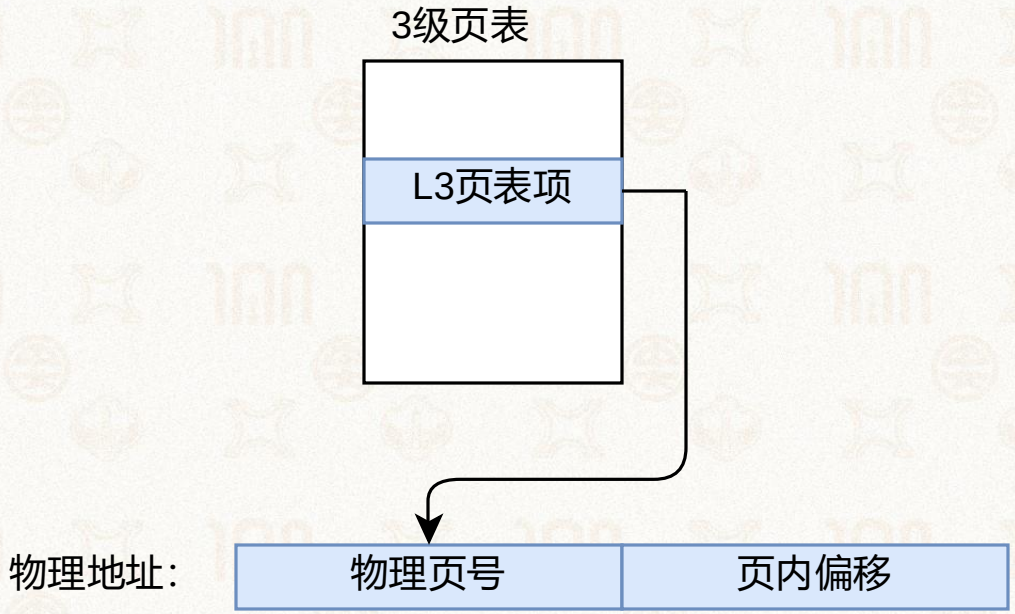



- 虚拟地址与物理地址的页内偏移相同
- 页内的组织是不变的
- 只转换页号

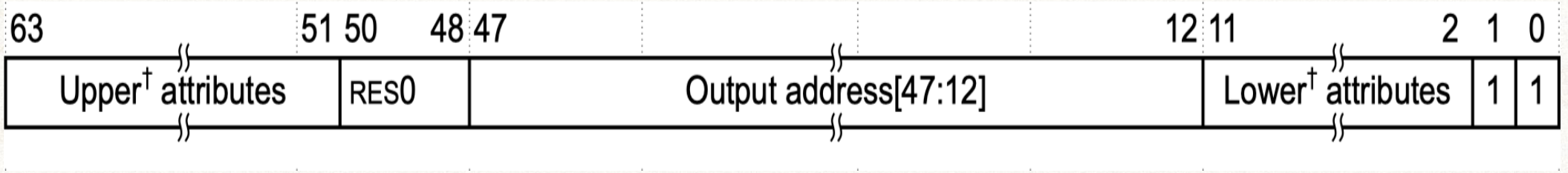
# AArch64页表项




- 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

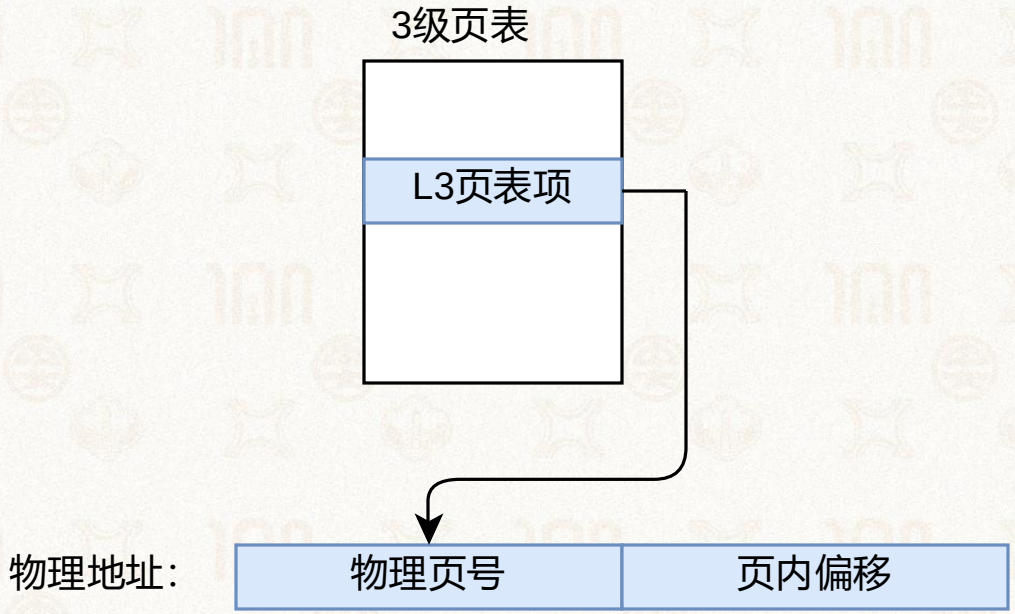


# AArch64页表项

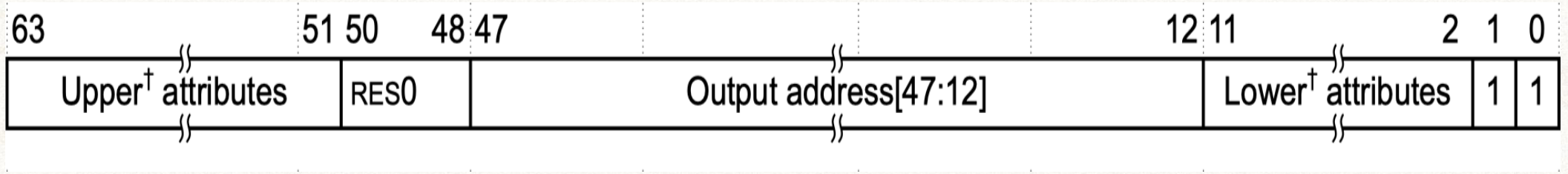



- 第3级页表页中的页表项
  - Lower attributes 包括:
    - 第7位-第6位表示读写权限位 AP[2:1]

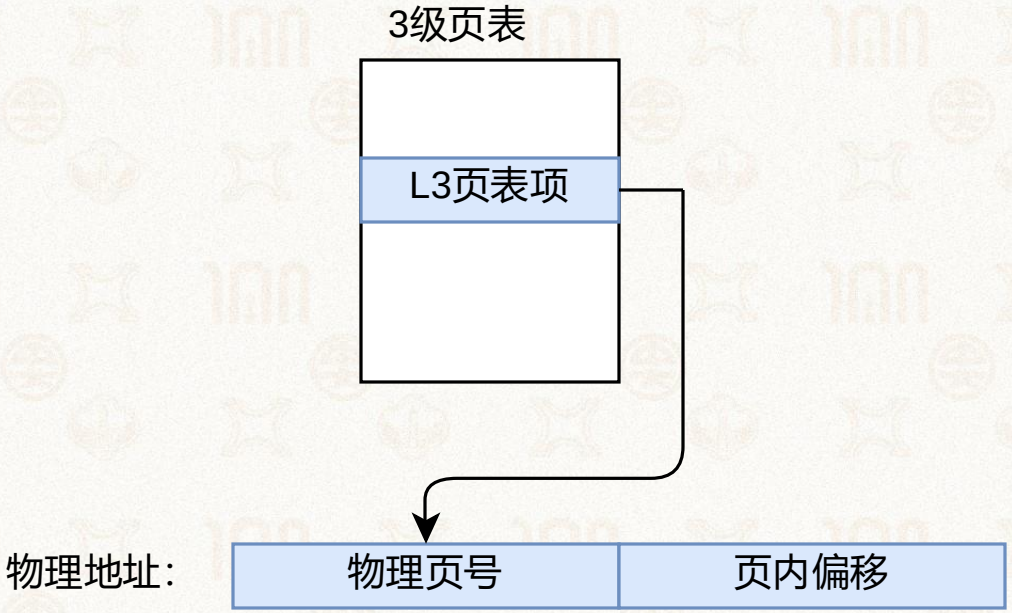
| AP[2:1] | Access from higher Exception level | Access from EL0 |
|---|---|---|
| 00 | Read/write | None |
| 01 | Read/write | Read/write |
| 10 | Read-only | None |
| 11 | Read-only | Read-only |



# 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）



# 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

# （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）

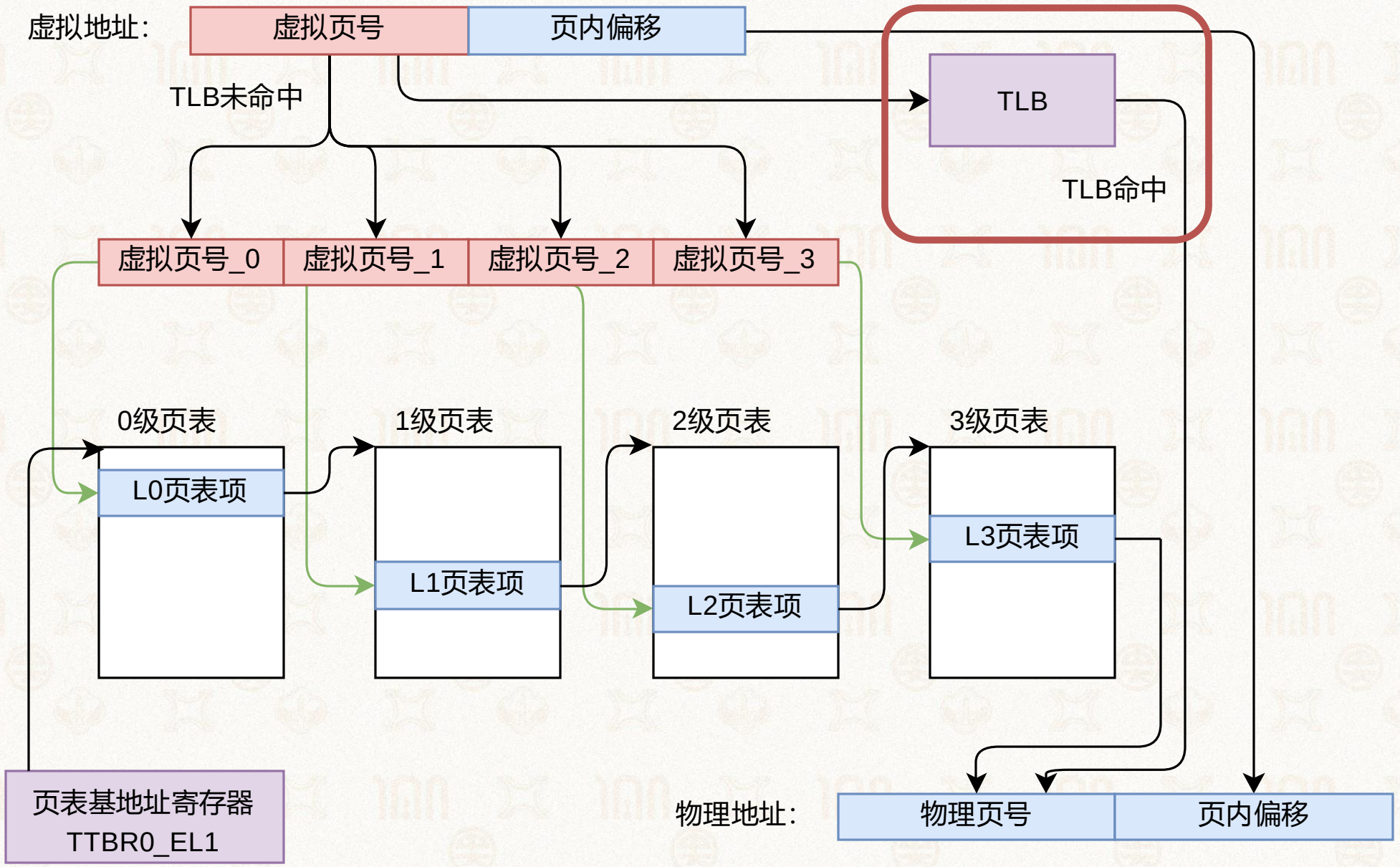
- Tradeoff 是计算机中经典而永恒的话题

- 如何降低地址翻译的开销？

# 大纲

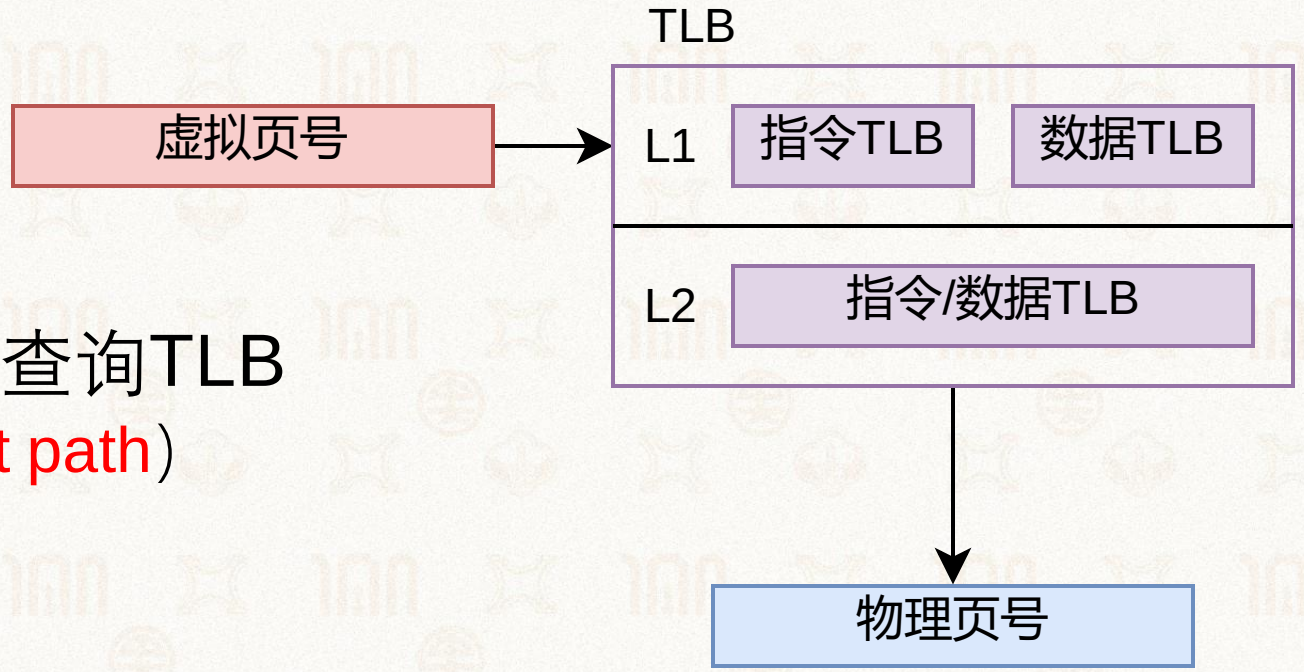
- 物理内存
- 虚拟内存
  - 分段
  - 分页、页表
- 分页机制
- TLB缓存

# TLB：地址翻译的加速器



# TLB: Translation Lookaside Buffer 转址旁路缓存

- 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
  - 就是一个哈希表




- 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

- 按照缓存结构，TLB设计通常也采用分级结构

# TLB管理：应该缓存哪些映射?

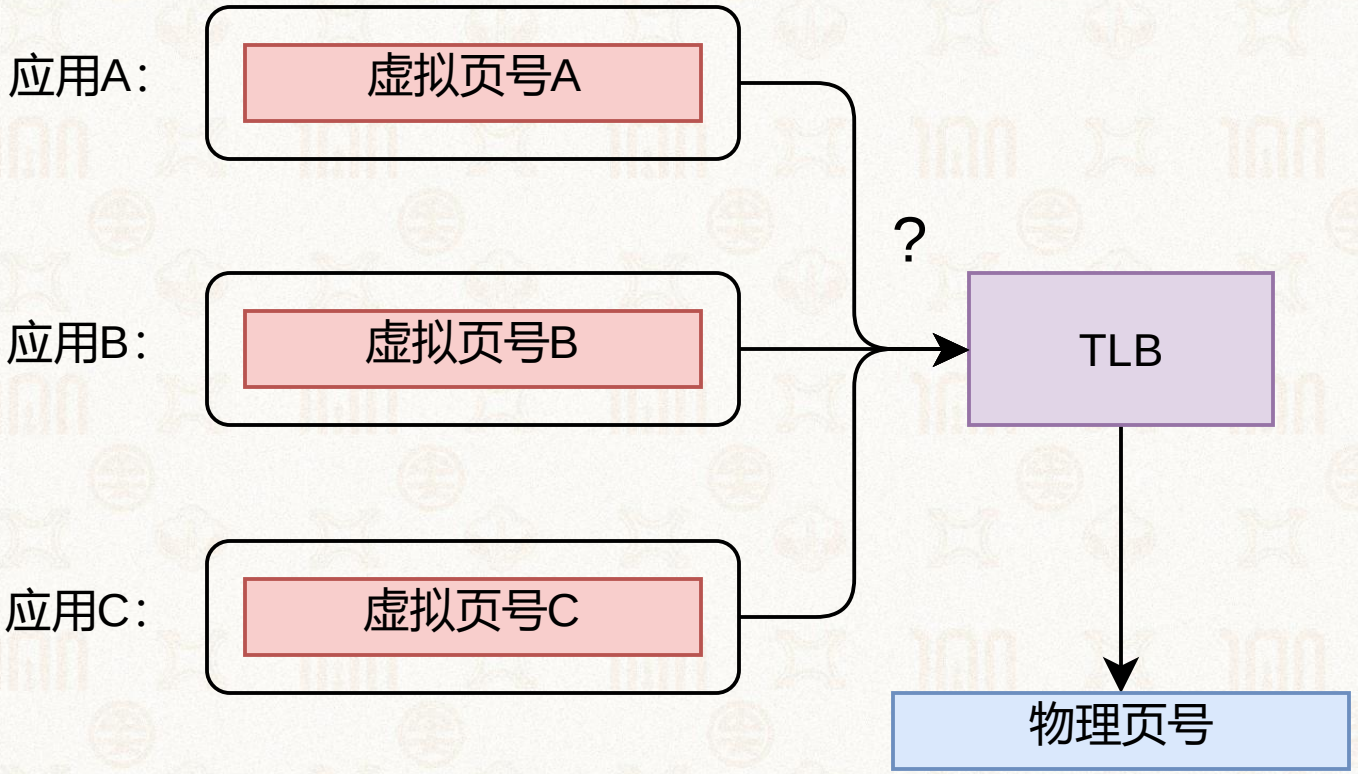
- 在AArch64和x86_64中，TLB由硬件管理
  - 硬件的简单替换策略为什么有效？（时空局部性）

```
#include<iostream>
using namespace std;
int main() {
    double b[10];
    for(int i = 0; i < 10; i++) {
      cout << "b[" << i << "] = " << b[i] << endl;
    }
    return 0;
}
```

- 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

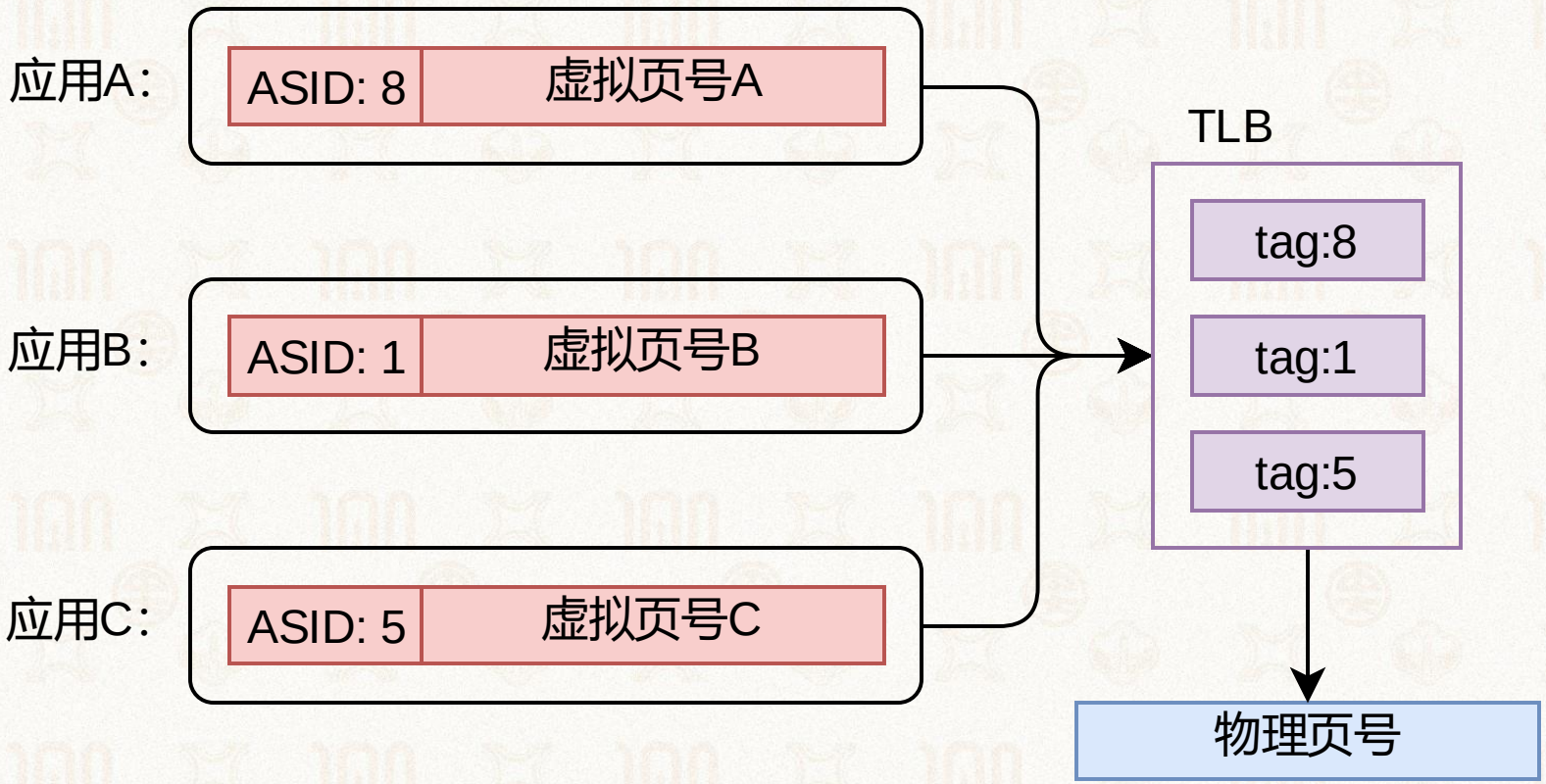


# 如何降低TLB刷新的开销

- 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16位长的 ASID，将ASID填写在TTBR0_EL1的高16位
  - ASID位数由TCR_EL1的第36位（AS位）决定

# 如何降低TLB刷新的开销

- ASID有16位，所以一般操作系统最多支持65536个应用同时运行



# Linux内核中的TLB管理

```
static inline void flush_tlb_mm(struct mm_struct *mm)
{
    unsigned long asid;

    dsb(ishst);
    asid = __TLBI_VADDR(0, ASID(mm));
    __tlbi(aside1is, asid);
    __tlbi_user(aside1is, asid);
    dsb(ish);
}
```

- 指定用户地址空间的所有TLB表项失效
- TLBI
  - Aarch64架构里使TLB失效的汇编指令
- dsb
  - 是指数据同步屏障，涉及多核一致性

https://elixir.bootlin.com/linux/v6.7/source/arch/arm64/include/asm/tlbflush.h

# 谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course