IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

内存地址翻译

上海交通大学

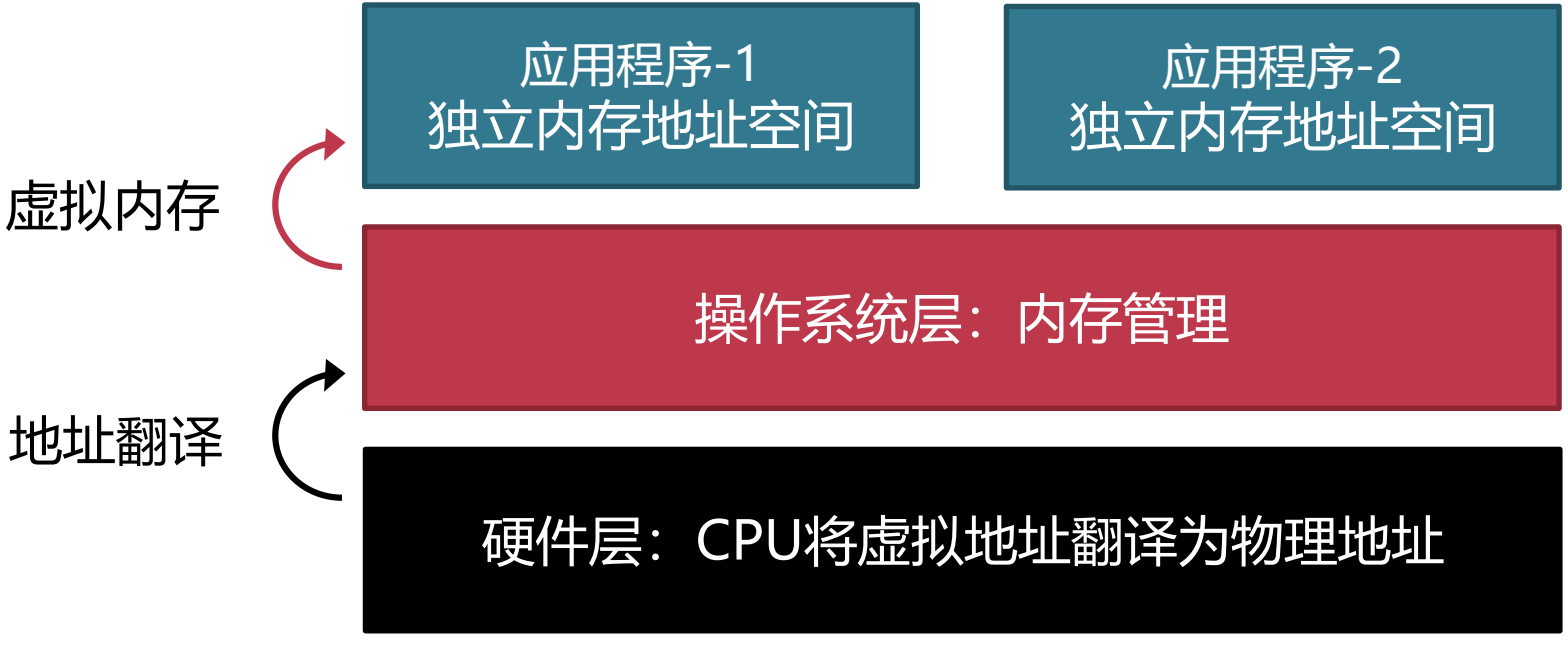
https://www.sjtu.edu.cn

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

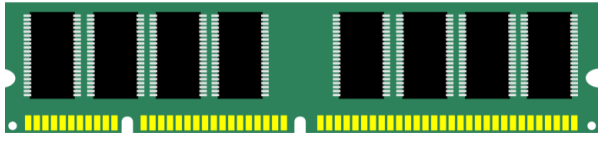
内存管理

内容提纲


应用程序-1
独立内存地址空间
应用程序-2
独立内存地址空间
虚拟内存
操作系统层：内存管理
地址翻译
硬件层：CPU将虚拟地址翻译为物理地址

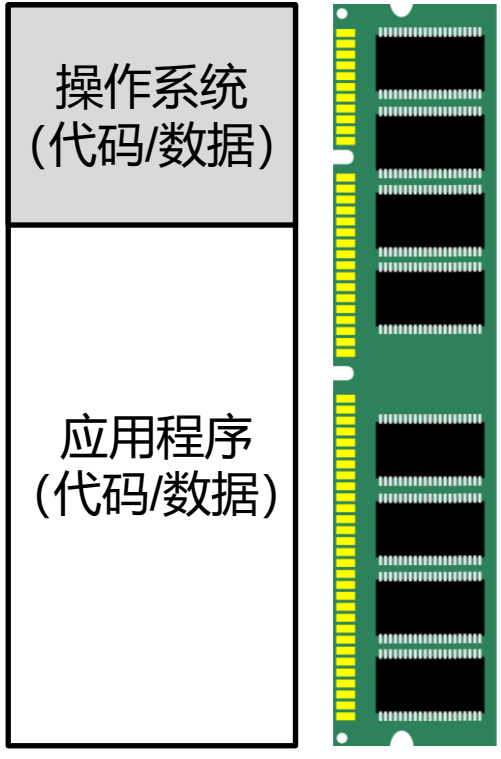
物理内存

- **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



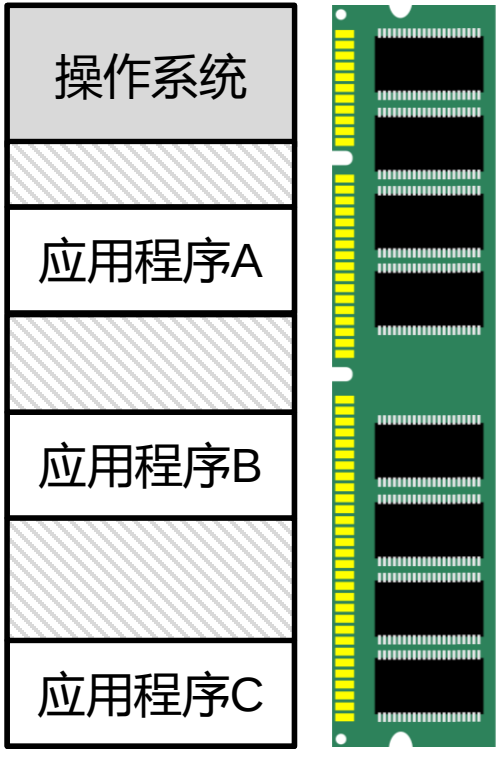
早期系统

- **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**
  - “发生在共享单车前的共享计算机”
- **分时复用CPU资源**
  - 保存恢复寄存器速度很快
- **如何共享物理内存资源?**
  1. 分时复用物理内存资源
     - 将全部内存写入磁盘开销太高
  2. 同时使用、各占一部分物理内存
     - 缺乏安全性/隔离性
  - 如何不被物理内存容量限制



回顾：可执行程序中的内存地址

```c
// main.c中的C代码
void multstore(long, long, long*);

int main() {
   long d;
   multstore(2, 3, &d);
   printf( “2 * 3 --> %d\n”, d);
   return 0;
}
```

```
000000000040046c <multstore>:
 40046c:  a9be7bfd   stp  x29, x30, [sp, #-32]!
 400470:  910003fd   mov  x29, sp
 400474:  f9000bf3   str  x19, [sp, #16]
 400478:  aa0203f3   mov  x19, x2
 40047c:  94000012   bl   7ec <mult2>
 400480:  f9000260   str  x0, [x19]
 400484:  f9400bf3   ldr  x19, [sp, #16]
 400488:  a8c27bfd   ldp  x29, x30, [sp], #32
 40048c:  d65f03c0   ret
```

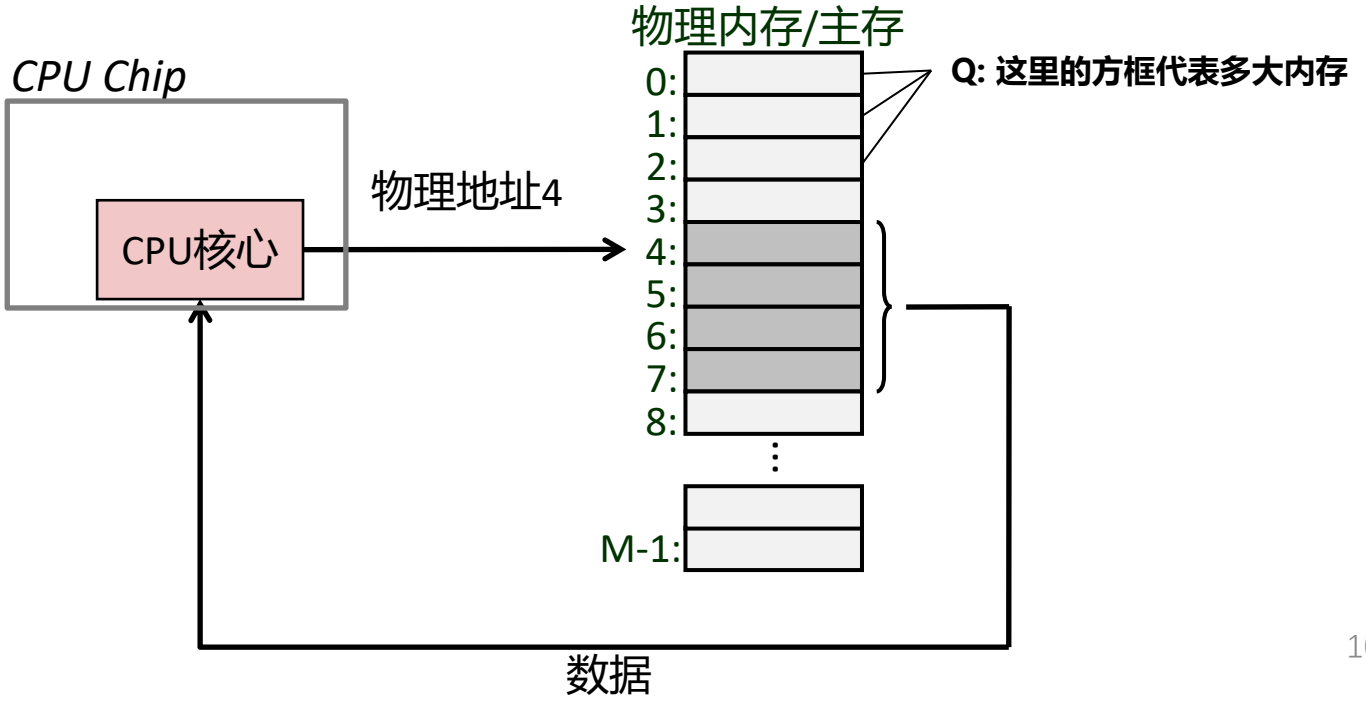
这些地址，在C代码中并没有，是从哪里来的？　编译器生成

虚拟地址空间易于程序开发

- **编译器面临的问题**
  - 如果某个地址的物理内存被别人占用了怎么办？
    - 例如：0x400000
  - 如果物理内存大小不够怎么办？
- **解决：编译器假设应用运行时，会独占所有内存，且内存足够大**
  - 内存大小：2^{32} Byte，或2^{64} Byte，称为“虚拟内存空间”
  - 编译器基于该假设进行内存布局，并生成指令
- **谁来实现这个假设？CPU+操作系统**
  - 思路：地址映射（虚拟地址翻译到物理地址）
  - 由CPU将虚拟地址翻译到物理地址，由操作系统来配置如何翻译

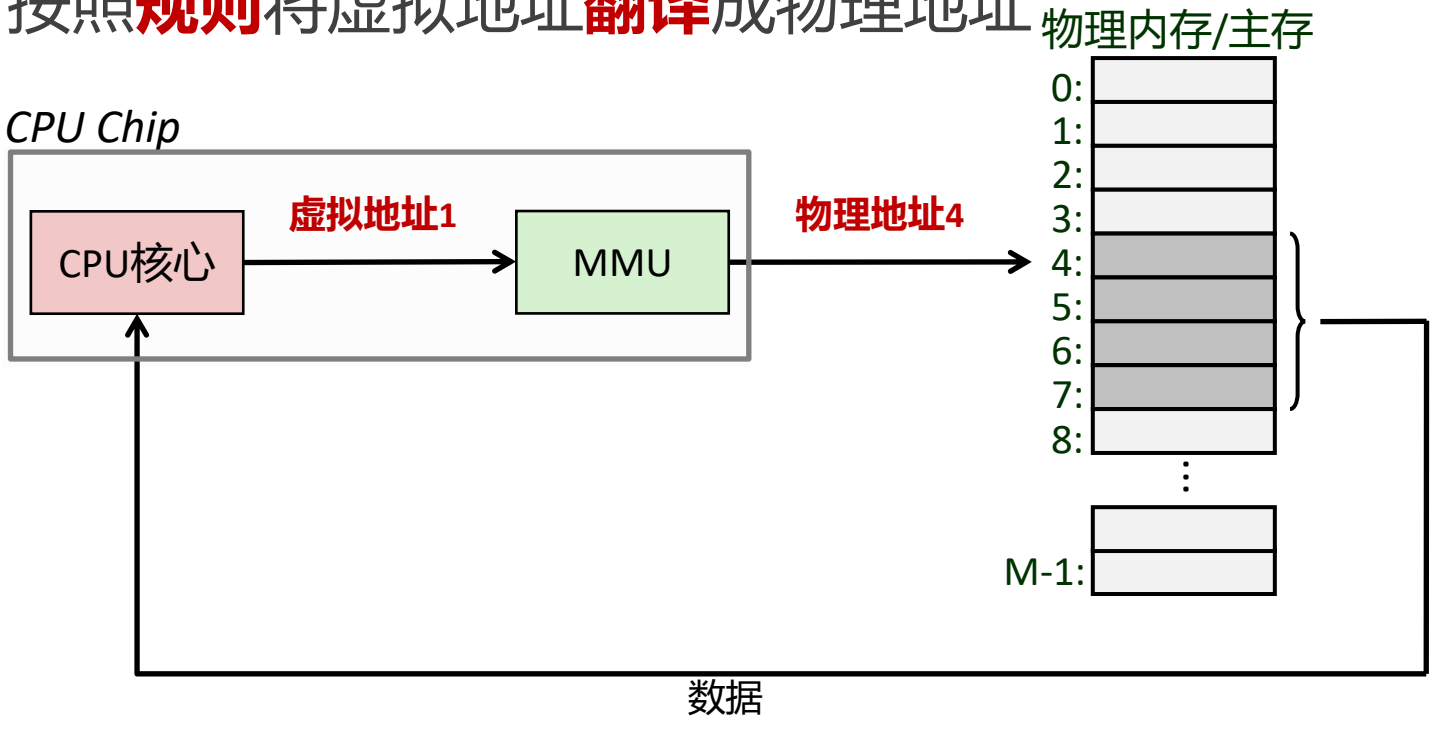
物理地址（physical address, pa）

- **物理内存可以看成由连续字节组成的数组**
  - 每个字节都有唯一的物理地址



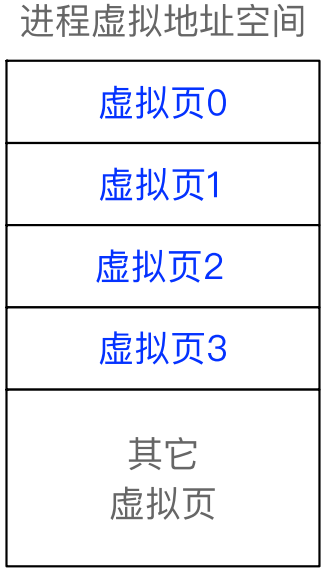
虚拟地址（virtual address, va)

- **Memory Management Unit （MMU)**
  - 按照**规则**将虚拟地址**翻译**成物理地址



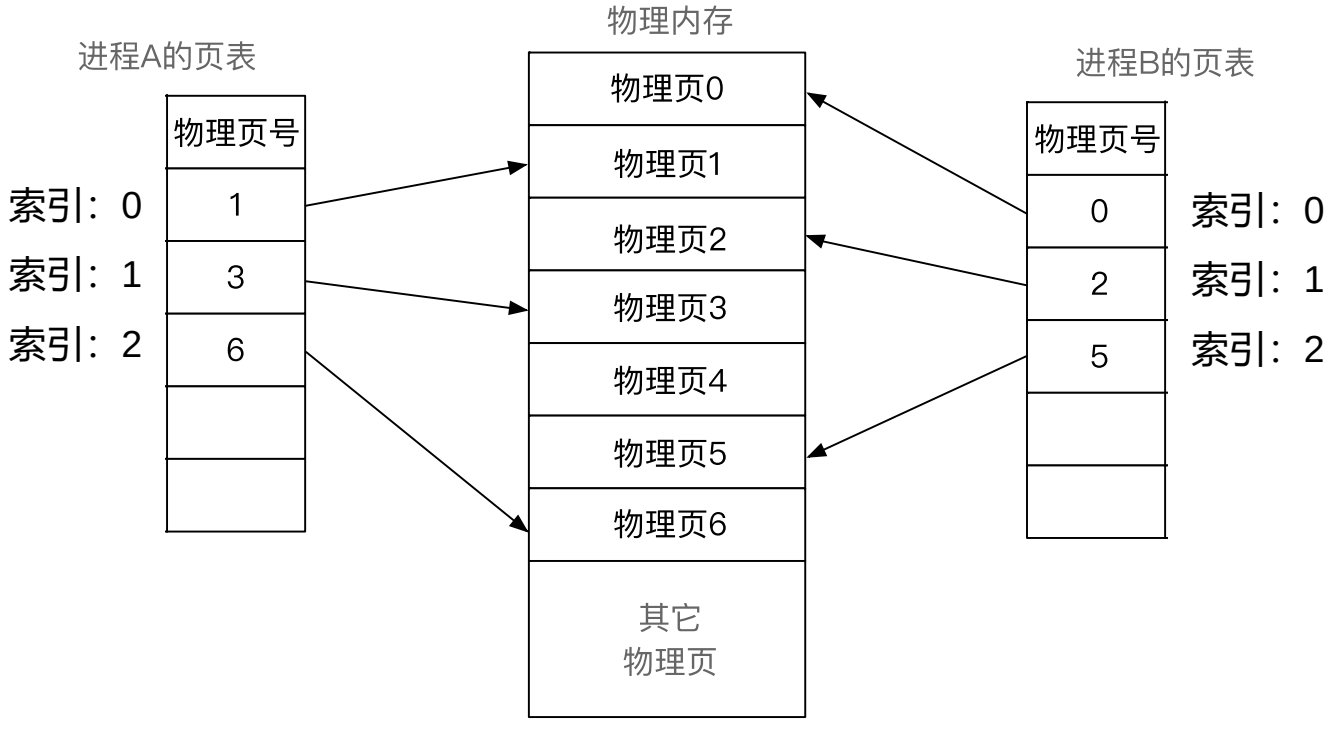
主流翻译规则：分页机制

- **分页机制**
  - 虚拟地址空间划分成连续的、等长的虚拟页
  - 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
  - 虚拟地址分为：虚拟页号 + 页内偏移
- **使用页表记录虚拟页号到物理页号的映射**
  - 页表：Page Table



页表：分页机制的核心数据结构

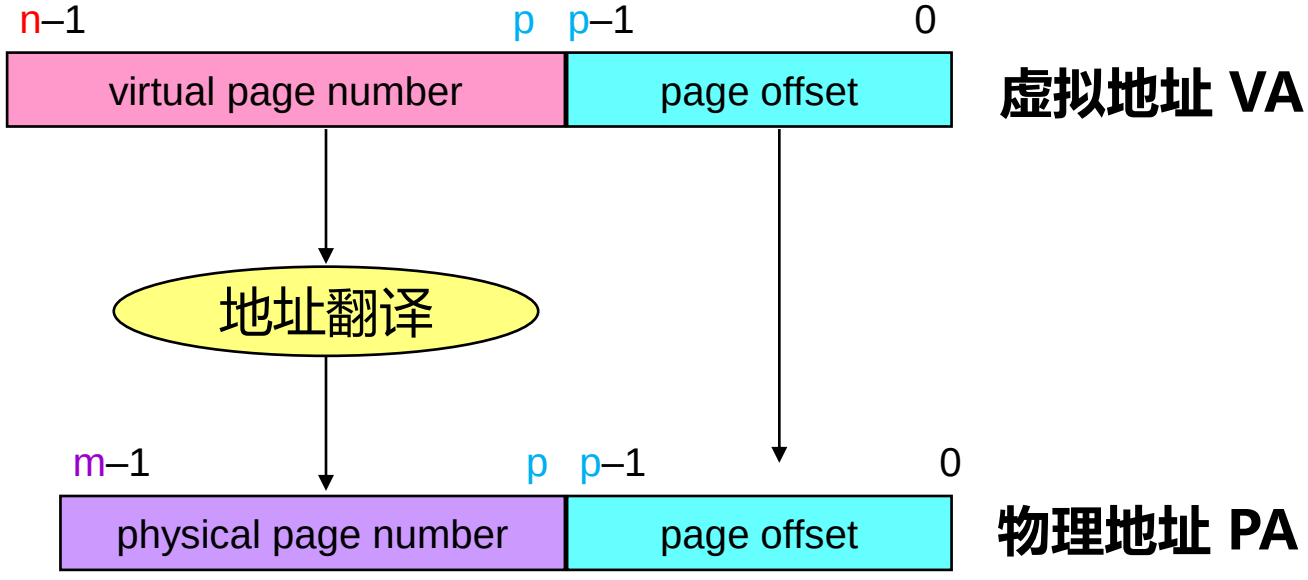
- **页表包含多个页表项，存储物理页的页号（虚拟页号为索引）**



分页机制的特点

- **物理内存离散分配**
  - 任意虚拟页可以映射到任意物理页
  - 大大降低对物理内存连续性的要求
- **主存资源易于管理，利用率更高**
  - 按照固定页大小分配物理内存
  - 能大大降低外部碎片和内部碎片
- **被现代处理器和操作系统广泛采用**

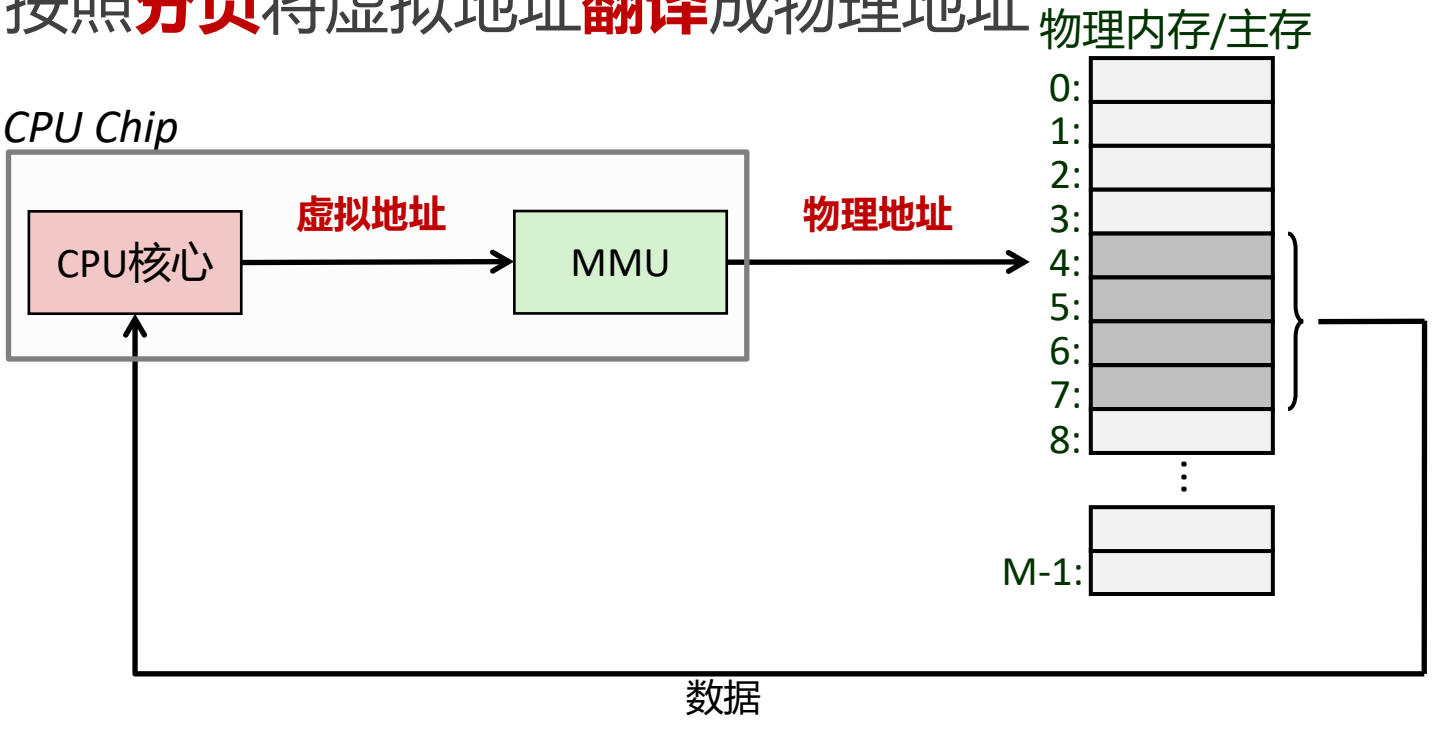
基于分页的地址翻译



请注意，页内偏移（page offset）不会因翻译而改变，由页面大小决定

思考：MMU怎么知道页表位置?

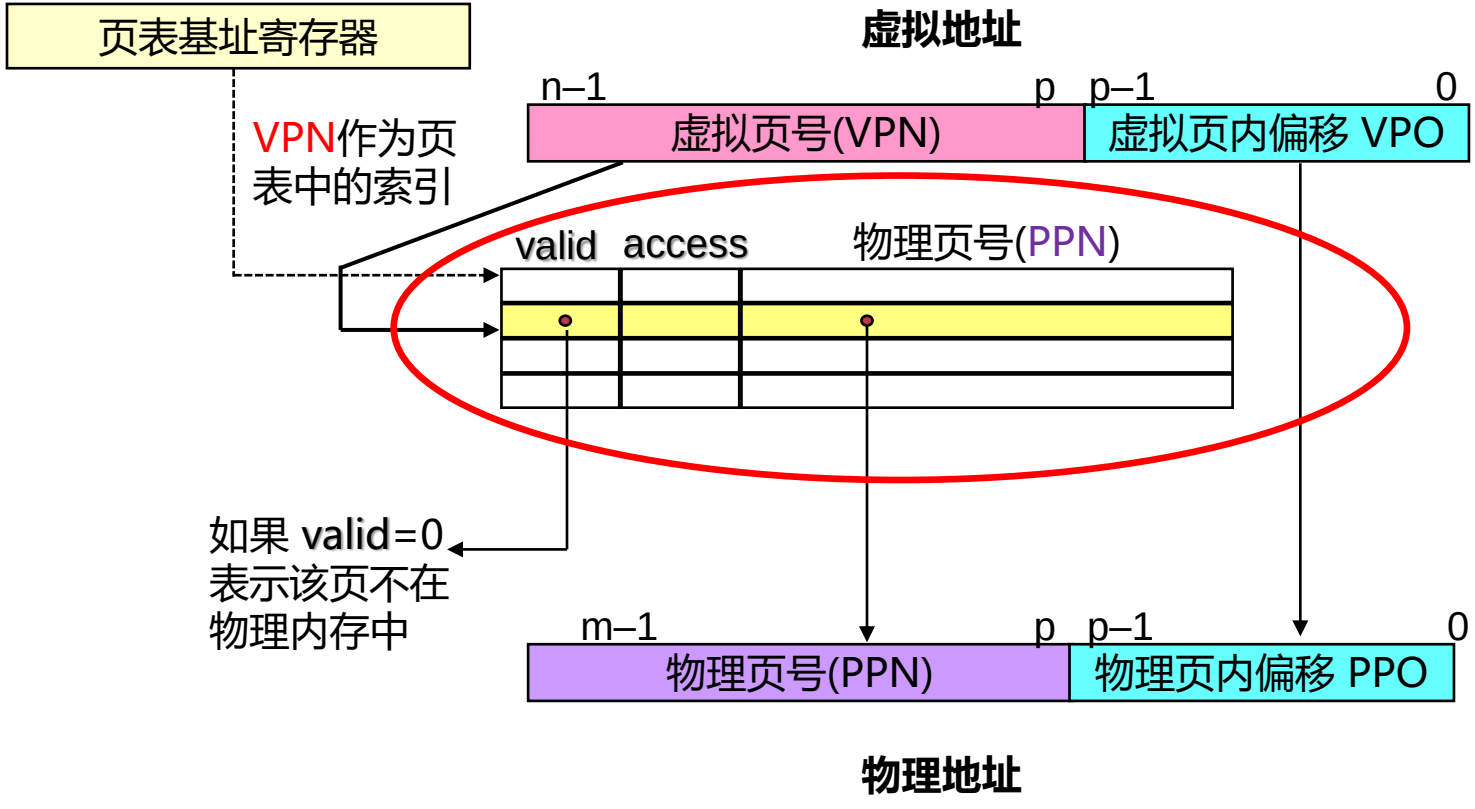
- **Memory Management Unit**
  - 按照**分页**将虚拟地址**翻译**成物理地址



通过页表进行地址翻译

页表：

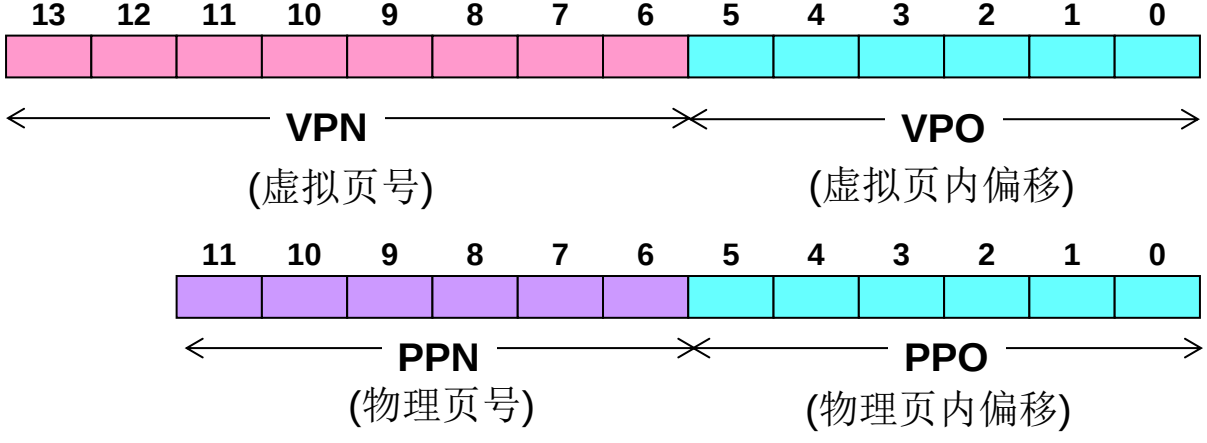
- **存储在物理内存中，由 OS 负责维护；**
- **其起始地址存放在页表基地址寄存器中**



内存地址翻译简单示例

- **假定一种地址格式**
  - 14 位虚拟地址
  - 12 位物理地址
  - 页大小 = 64 字节 (6位)

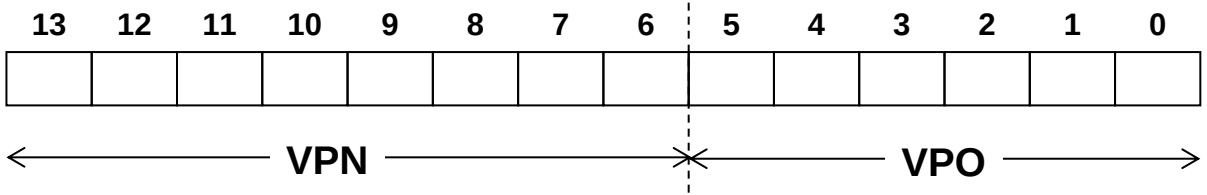
	PPN	Valid
00	28	1
01	-	0
02	33	1
03	02	1
04	-	0
05	16	1
06	-	0
07	-	0



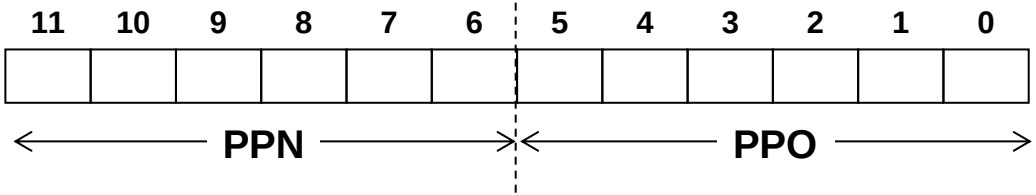
内存地址翻译简单示例

	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

Virtual Address: 0x03D4



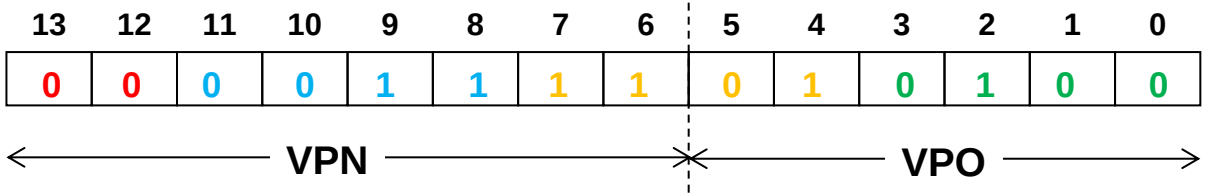
VPN: _____ VPO: _____ Page Fault? _____



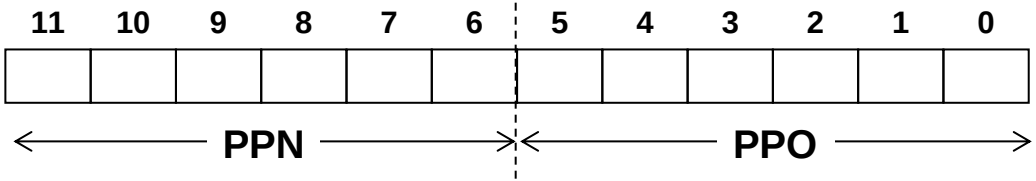
PPN: _____ PPO: _____ PA: _____

内存地址翻译简单示例

Virtual Address: 0x03D4



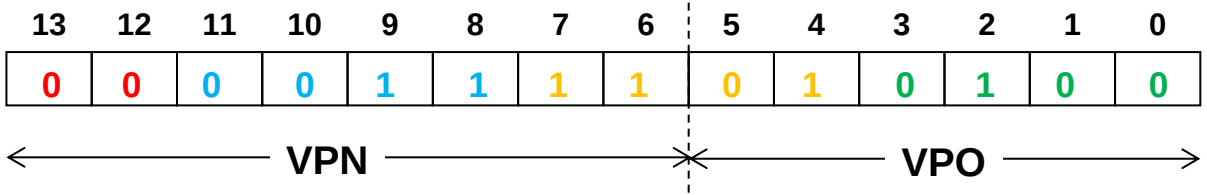
VPN: **0x0F** VPO: **0x14** Page Fault?



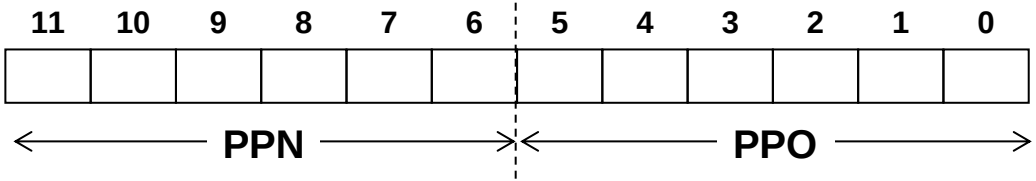
PPN: _____ PPO: _____ PA: _____

内存地址翻译简单示例

Virtual Address: 0x03D4



VPN: **0x0F** VPO: **0x14** Page Fault?



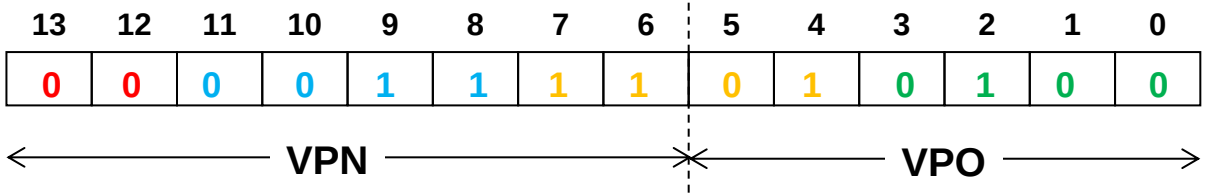
PPN: ____ PPO: **0x14** PA: ____

VPO == PPO

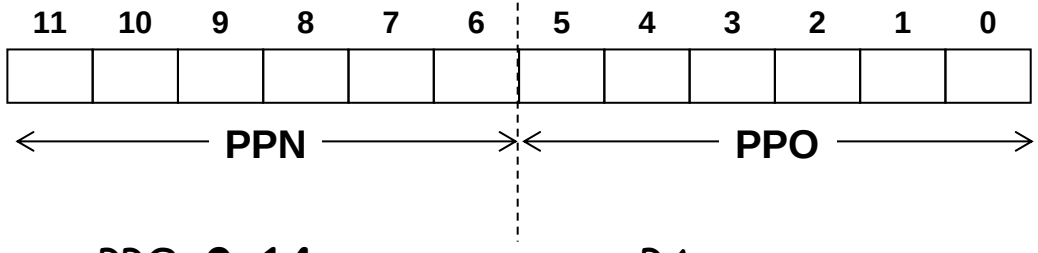
内存地址翻译简单示例

	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

Virtual Address: 0x03D4



VPN: 0x0F VPO: 0x14 Page Fault?

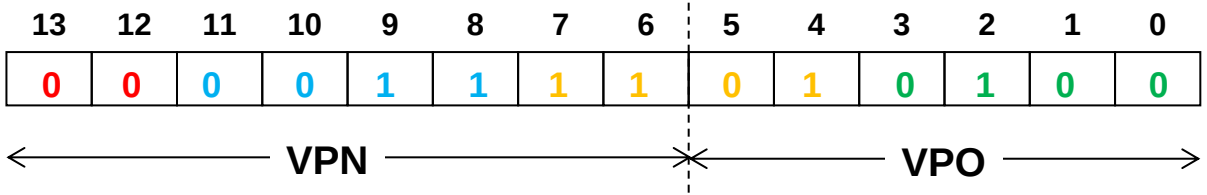


PPN: ____ PPO: 0x14 PA: ____

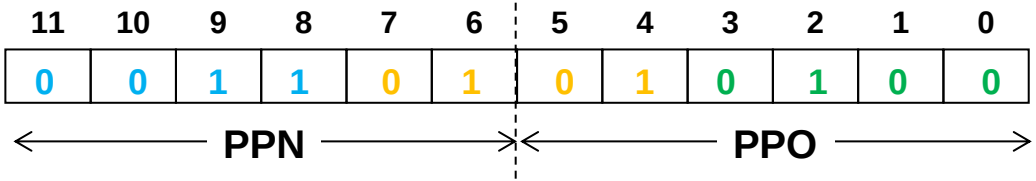
内存地址翻译简单示例

	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	-	0
0C	-	0
0D	2D	1
0E	11	1
0F	0D	1

Virtual Address: 0x03D4



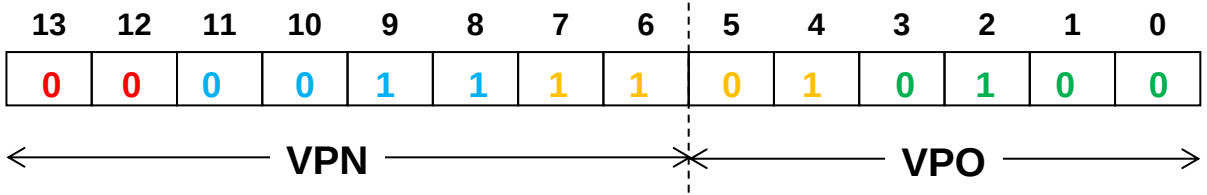
VPN: 0x0F VPO: 0x14 Page Fault? No



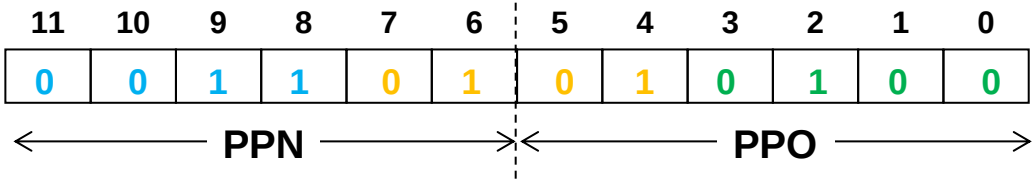
PPN: 0x0D VPO: 0x14 PA: ____

内存地址翻译简单示例

Virtual Address: 0x03D4



VPN: 0x0F VPO: 0x14 Page Fault? No

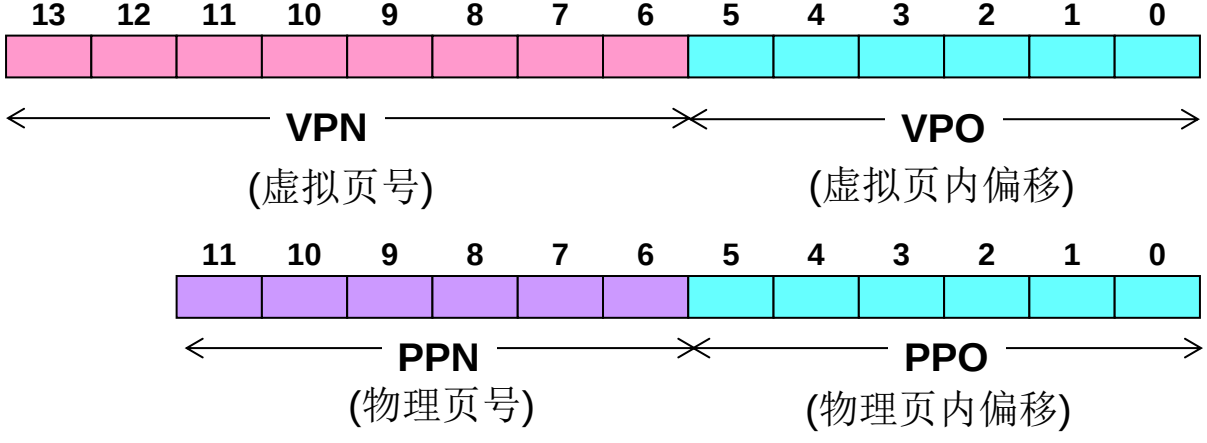


PPN: 0x0D VPO: 0x14 PA: 0x354

小练习：请翻译虚拟地址0

- **假定一种地址格式**
  - 14 位虚拟地址
  - 12 位物理地址
  - 页大小 = 64 字节 (6位)

	PPN	Valid
00	28	1
01	-	0
02	33	1
03	02	1
04	-	0
05	16	1
06	-	0
07	-	0



PA：0xa00

多级页表

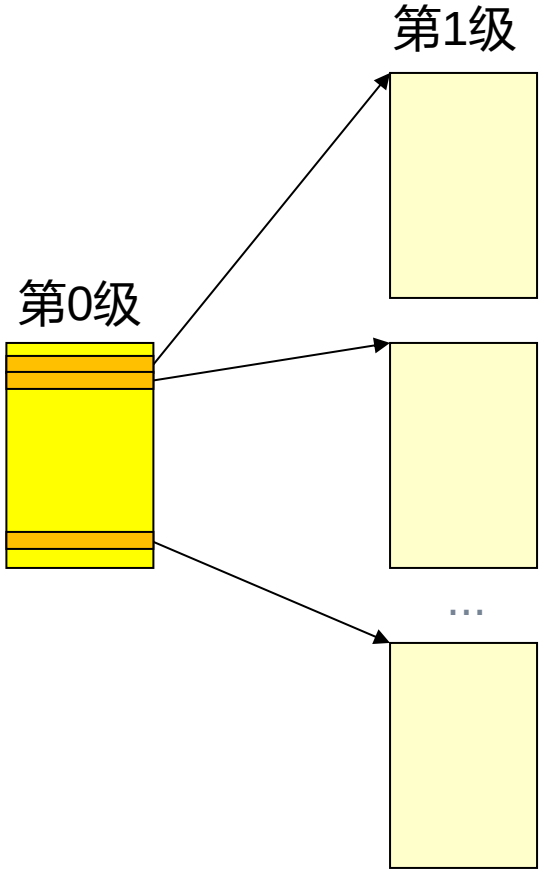
单级页表的问题

- **如果使用上述单级页表结构，那么页表本身有多大呢?**
  - 假设32位的地址空间，页大小为4K，页表项为4字节
  - 假设64位的地址空间，页大小为4K，页表项为8字节
- **对于32位地址空间**
  - 2^32 / 4K * 4 = 4MB
- **对于64位地址空间**
  - 2^64 / 4K * 8 = 33,554,432 GB（完全不现实！）

多级页表

- **多级页表能有效压缩页表的大小**

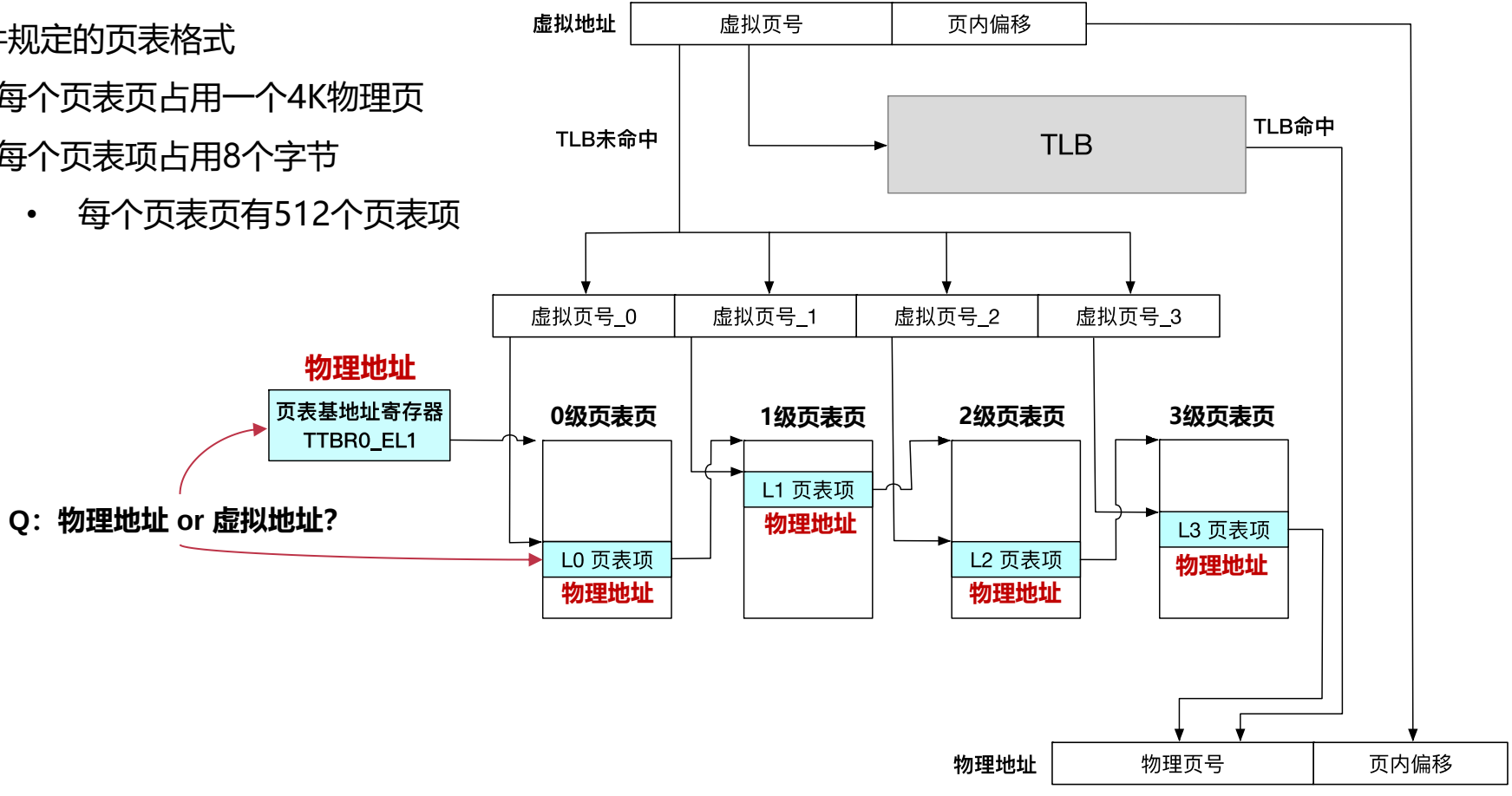
- **原因：允许页表中出现空洞**
  - 若某个页表页中的某个页表项为空，
    那么其对应的下一级页表页便无需存在
  - 应用程序的虚拟地址空间大部分都未分配



AARCH64体系结构下4级页表

硬件规定的页表格式

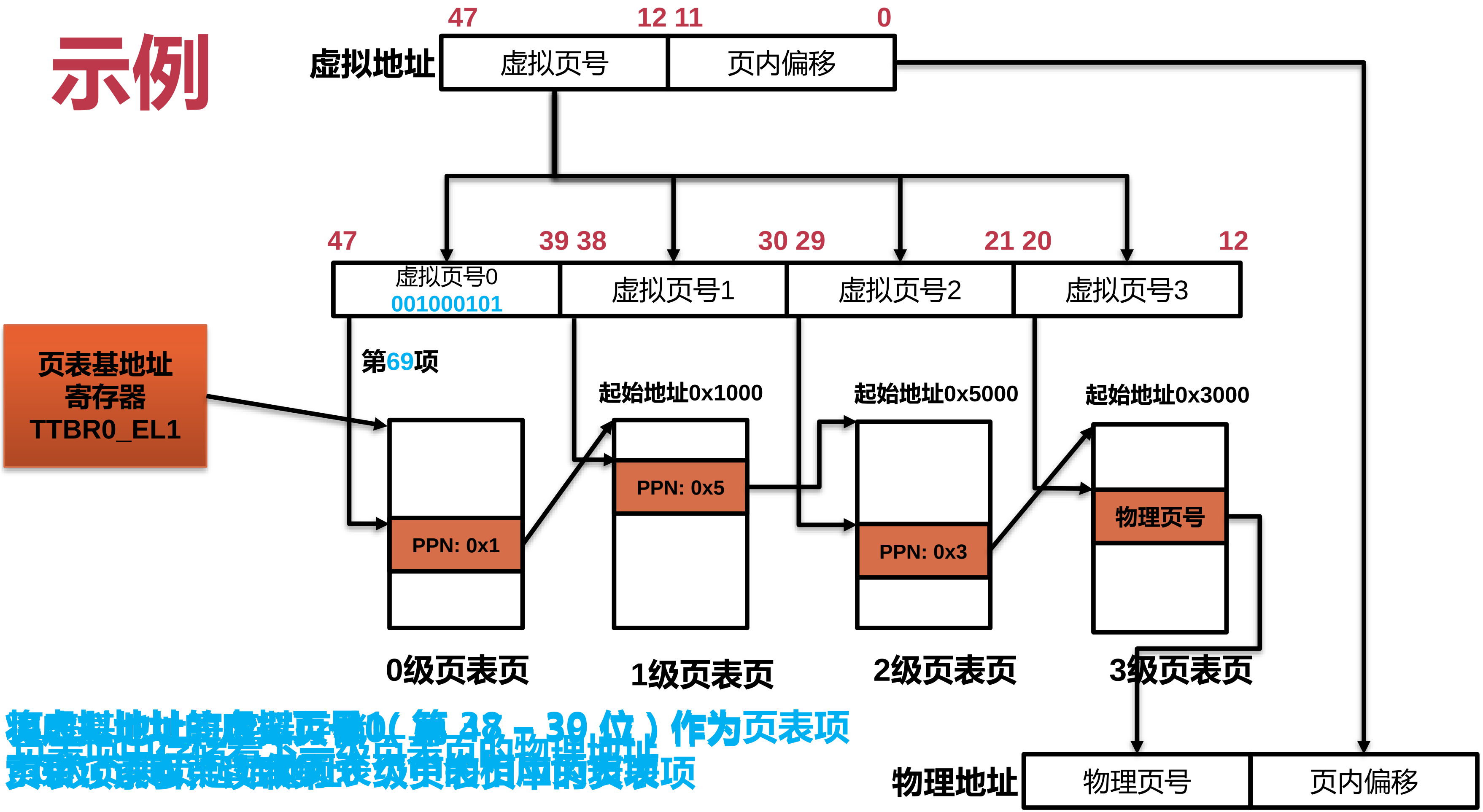
- 每个页表页占用一个4K物理页
- 每个页表项占用8个字节
  - 每个页表页有512个页表项



64位虚拟地址解析

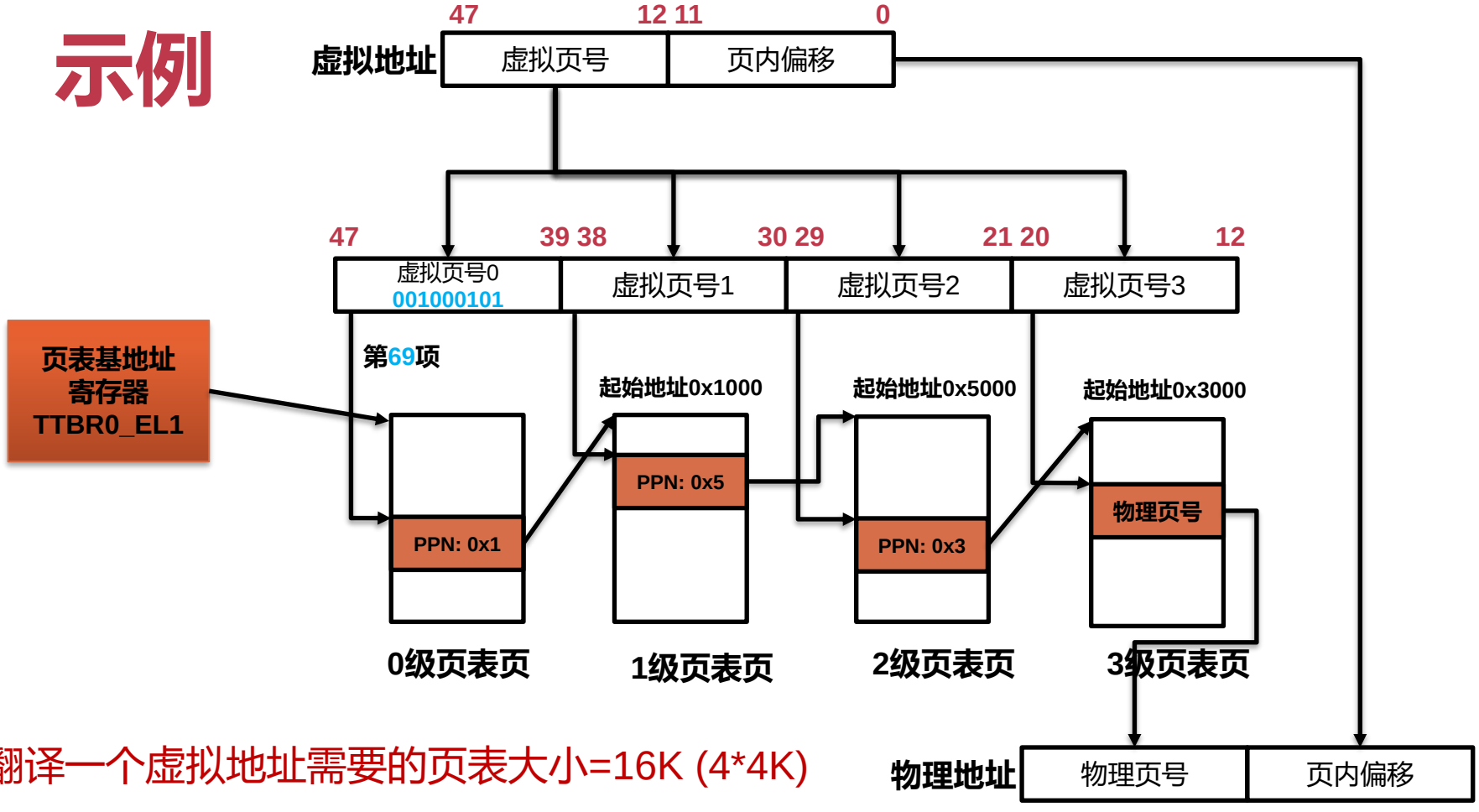
- **「63：48」16-bit**
  - 必须全是0或者全是1（软件用法：一般应用程序地址是0，内核地址是1）
  - 也意味着虚拟地址空间大小最大是2^{48} Byte，即256TB
- **「47：39」9-bit：**0级页表索引
- **「38：30」9-bit：**1级页表索引
- **「29：21」9-bit：**2级页表索引
- **「20：12」9-bit：**3级页表索引
- **「11：0」12-bit：**页内偏移

示例



虚拟地址
47
12 11
0
虚拟页号
页内偏移
47
39 38
30 29
21 20
12
虚拟页号0
001000101
虚拟页号1
虚拟页号2
虚拟页号3
页表基地址
寄存器
TTBR0_EL1
第69项
起始地址0x1000
起始地址0x5000
起始地址0x3000
PPN: 0x1
PPN: 0x5
PPN: 0x3
物理页号
0级页表页
1级页表页
2级页表页
3级页表页
物理地址
物理页号
页内偏移

示例



翻译一个虚拟地址需要的页表大小=16K (4*4K)

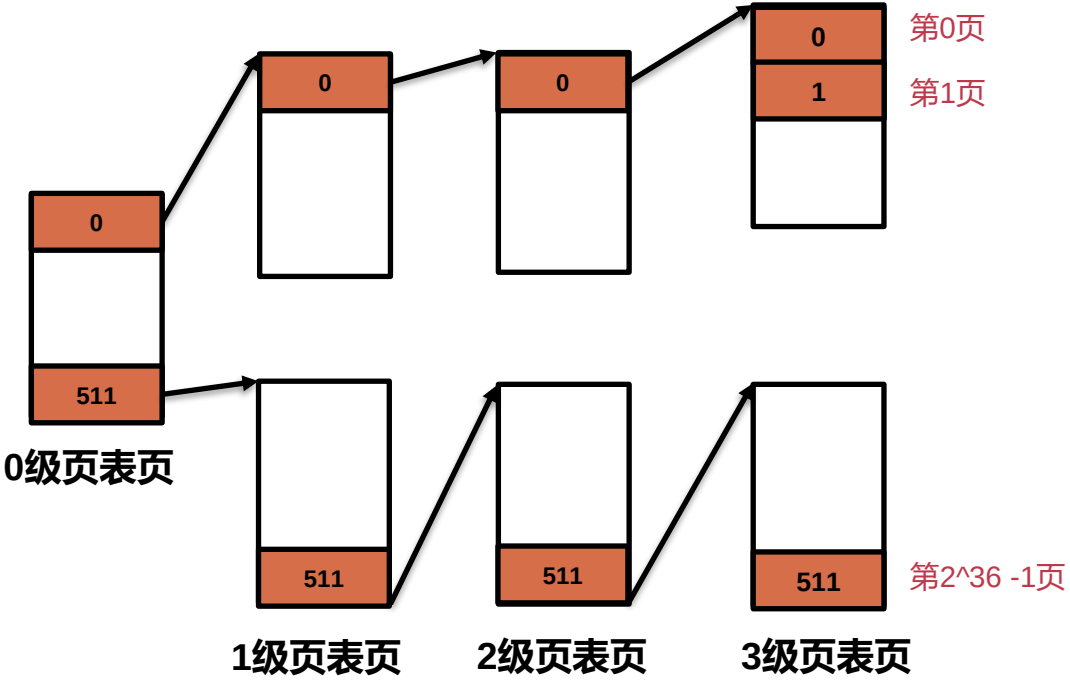
尽管其余511项没有用，但是单个页表页一旦分配就是占用4K

问：若页表共占用16K，能够翻译的虚拟地址范围是？ 2M

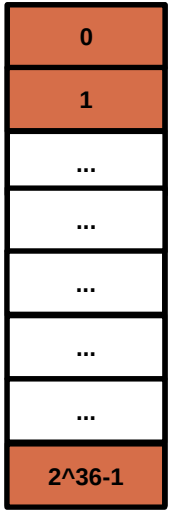
多级页表节省内存示例

假设进程只用到0，1，2^36 – 1三个虚拟页

- 单级页表, 共需2^36 个页表项 ， 共占用2^39B
- 4级页表，仅需要一个0级页表，2个一级页表，2个二级页表，2个三级页表，共七个页表，每个页表页占用 (4KB)，共占用28KB



进程1的页表



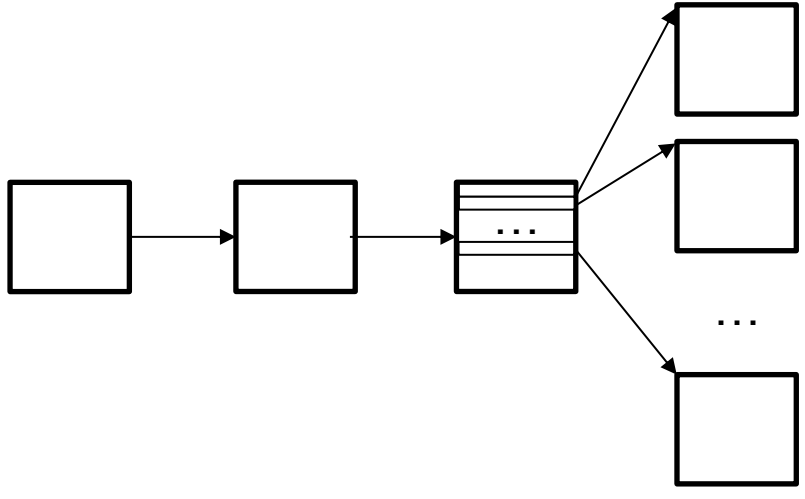
小练习

1. 多级页表是否一定比单级页表占用内存少?

2. 虚拟地址0xFFF8080604FFF对应的各级页表索引?

第三级索引：（VA & (0x1ff << 12）) >> 12

3. 若在页表中填写虚拟地址 0-16M 的映射，则页表至少需要占用多少空间?



页表基地址寄存器

- **AARCH64**
  - 两个页表基地址寄存器：TTBR0_EL1 & TTBR1_EL1
    - MMU根据虚拟地址第63位选择使用哪一个
  - 以Linux为例
    - 应用程序（地址首位为0）：使用TTBR0_EL1
    - 操作系统（地址首位为1）：使用TTBR1_EL1
- **X86_64**
  - 一个寄存器：CR3（Control Register 3）

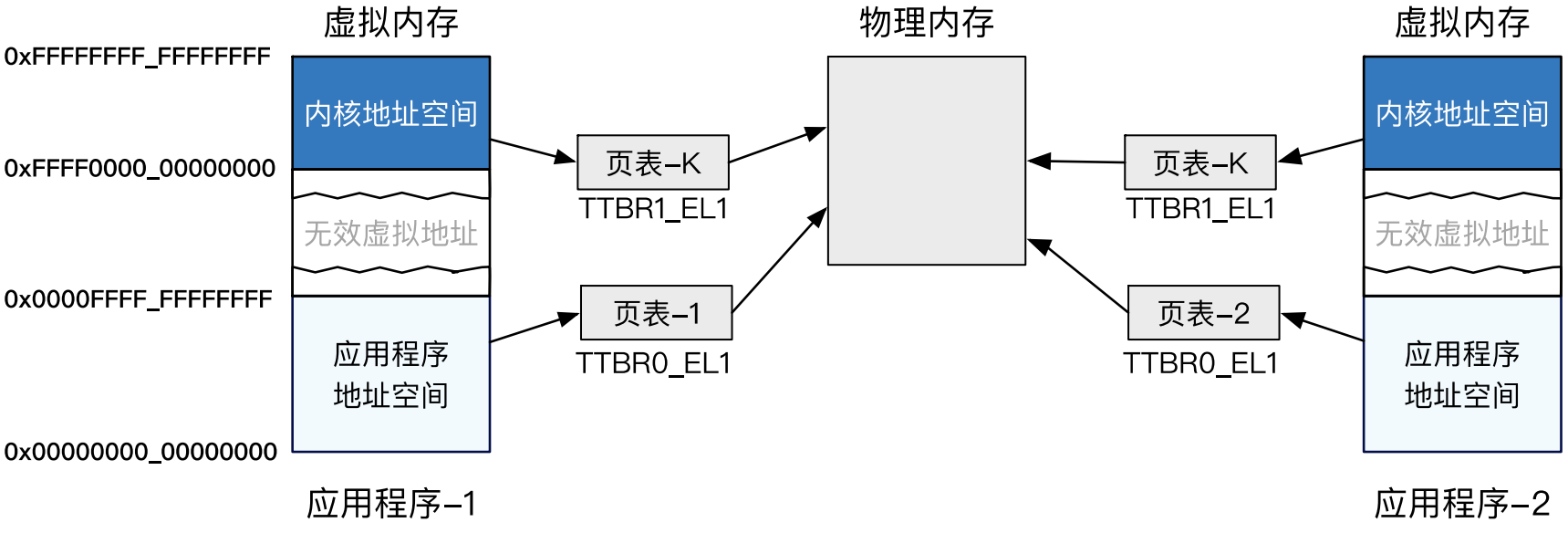
页表使能

- **机器上电会先进入物理寻址模式**
  - 系统软件需配置控制寄存器使能页表从而进入虚拟寻址模式
- **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，即使能页表

问：操作系统在运行过程中使用虚拟地址还是物理地址?

AArch64中的两个页表基地址寄存器

- TTBR0_EL1和TTBR1_EL1分别翻译部分虚拟地址



问：操作系统页表和应用进程页表的数量是多少？

页表项

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（64位），用于存储物理地址和权限

页表项中的属性位

有效的3级页表项：第1位必须是1

页描述符：指向4K页

3级页表项

- V（Valid）：当访问时V=0，则触发缺页异常
- UXN（Unprivileged eXecute Never）：用户态不可执行（例如栈）
- PXN（Privileged eXecute Never）：内核态不可执行（防止内核执行用户代码）
- AF（Access Flag）：当访问时，MMU标记该位
- AP（Access Permissions）
- DBM （Dirty Bit）： 51位，ARMv8.1-TTHM特性支持

AP	用户态 EL0	内核态 EL1
00	不可访问	可读可写
01	可读可写	可读可写
10	不可访问	只读
11	只读	只读

页表项中的属性位

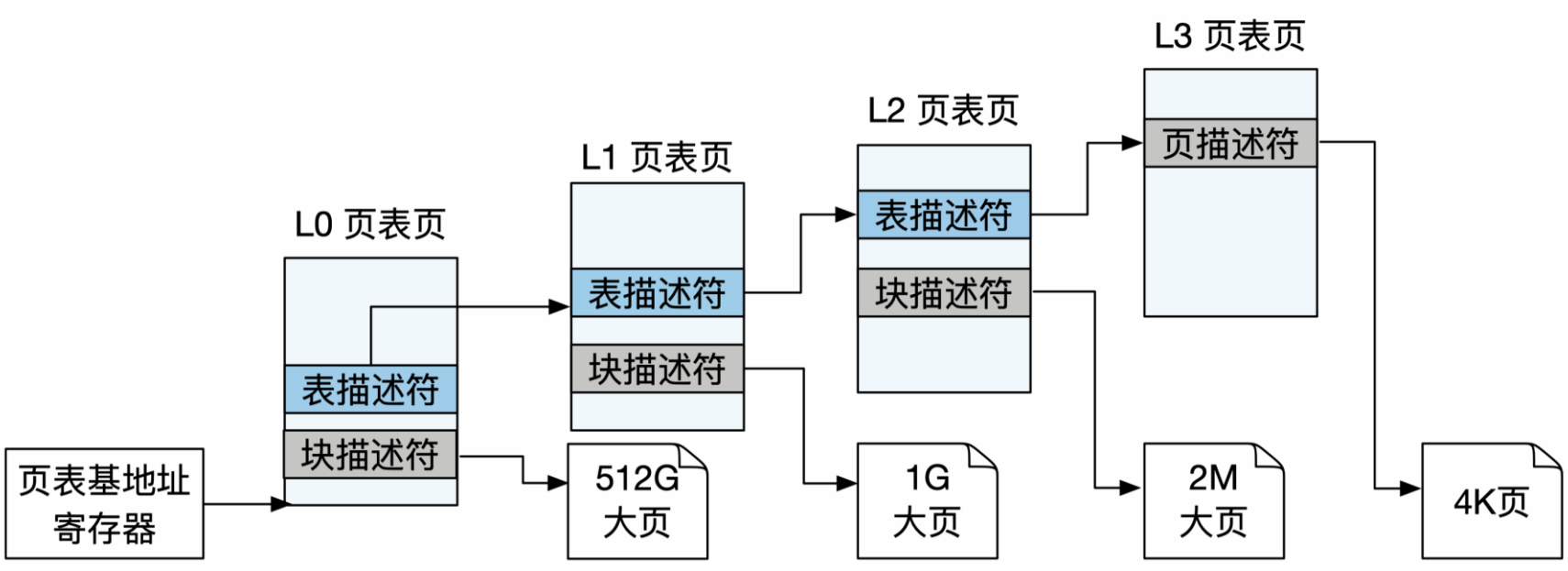
页描述符：指向4K页

3级页表项

表描述符：指向下一级页表

0级、1级、2级页表项

页表项与大页


L3 页表页
L2 页表页
L1 页表页
L0 页表页
页描述符
表描述符
块描述符
表描述符
块描述符
表描述符
块描述符
页表基地址
寄存器
512G
大页
1G
大页
2M
大页
4K页

使用大页时的页表项

块描述符：指向大页

表描述符：指向下一级页表

0级、1级、2级页表项

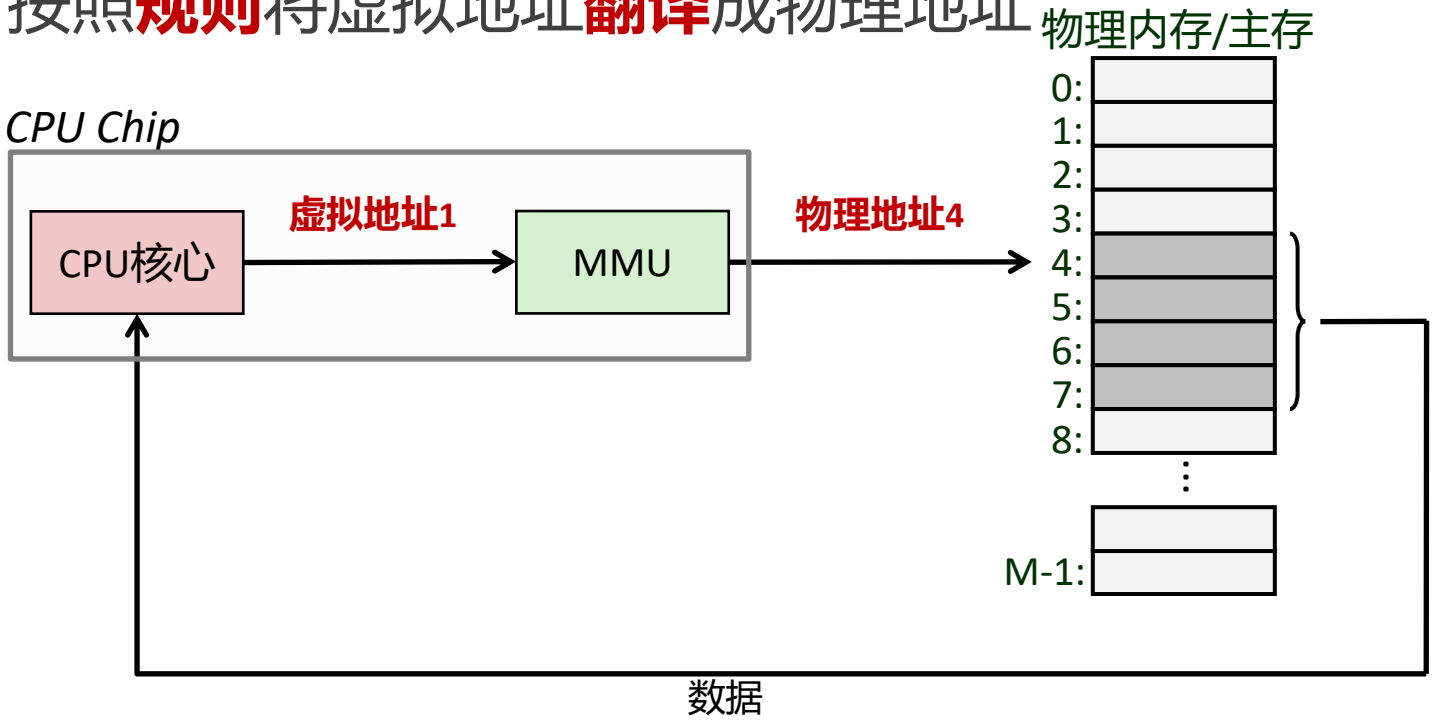
有效的0-2级页表项

- 第1位为0表示PFN指向大页
- 第1位为1表示PFN指向下一级页表

回顾复习

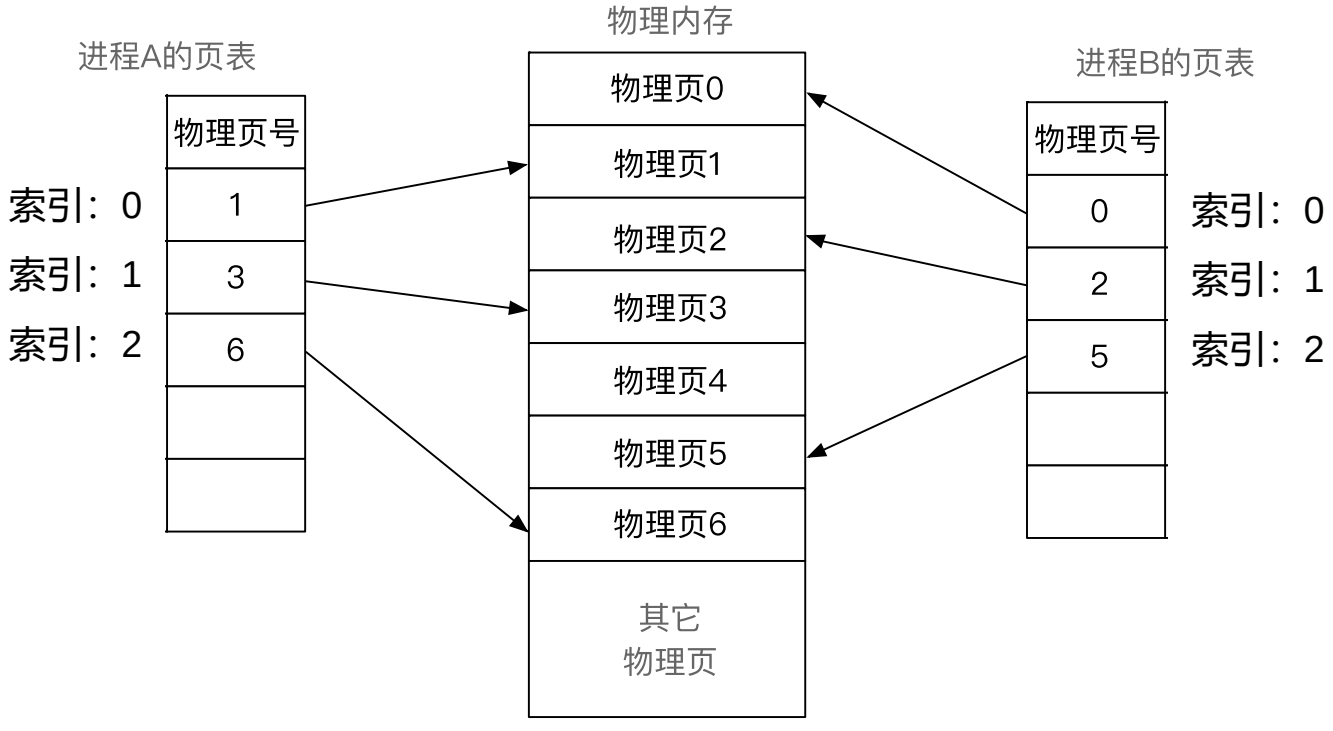
虚拟地址与物理地址

- **Memory Management Unit （MMU）**
  - 按照**规则**将虚拟地址**翻译**成物理地址



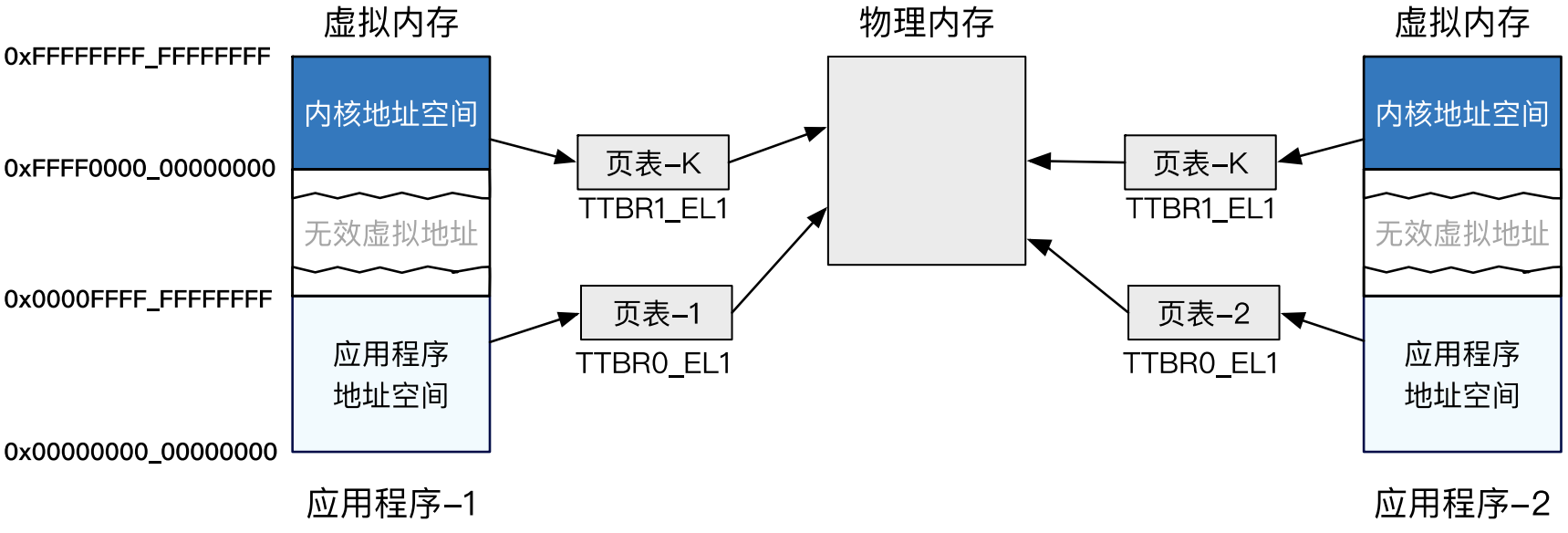
页表：分页机制的核心数据结构

- **页表包含多个页表项，存储物理页的页号（虚拟页号为索引）**



AArch64中的两个页表基地址寄存器

- TTBR0_EL1和TTBR1_EL1分别翻译部分虚拟地址

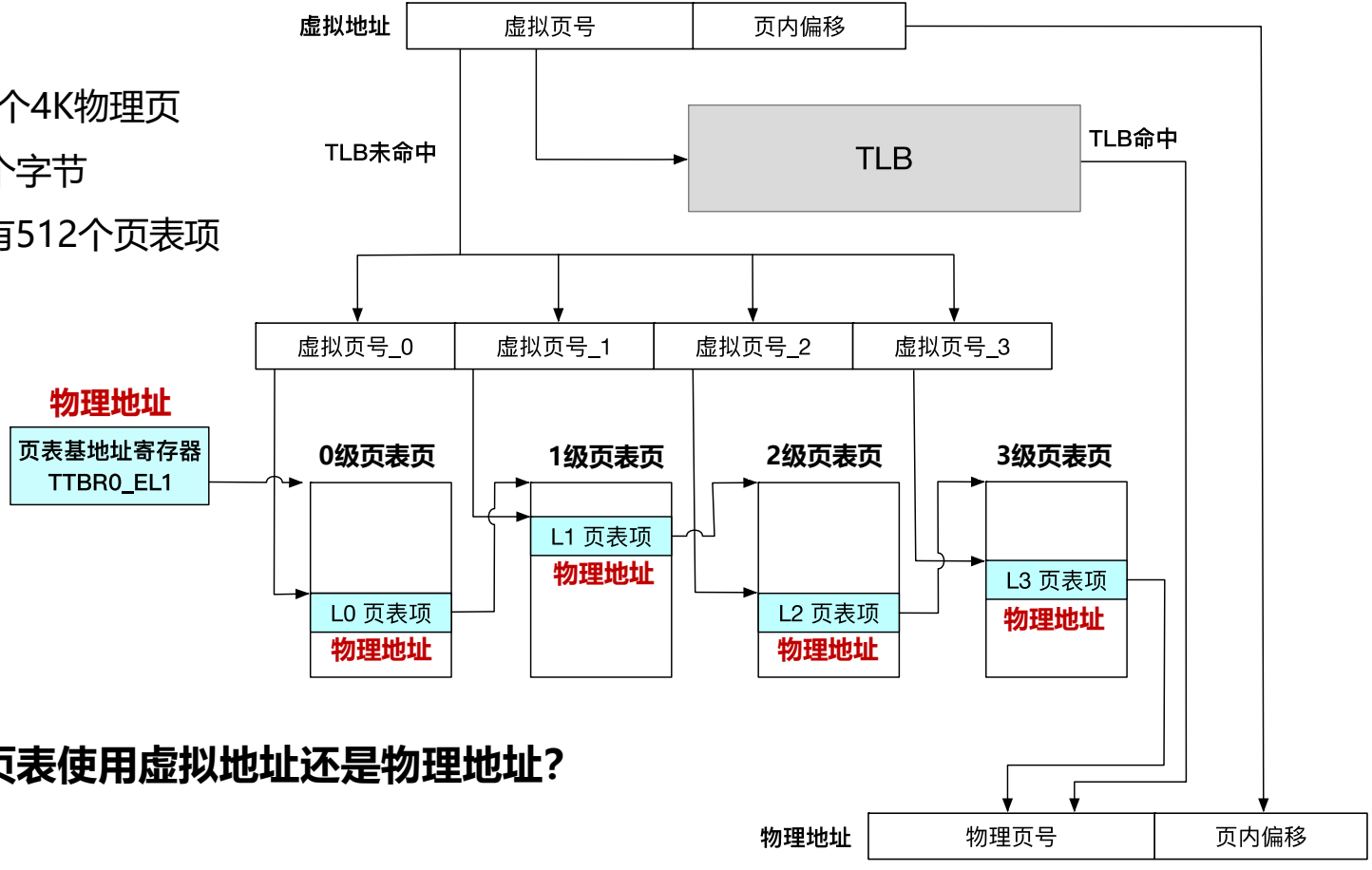


问：操作系统页表和应用进程页表的数量是多少？

AARCH64体系结构下4级页表

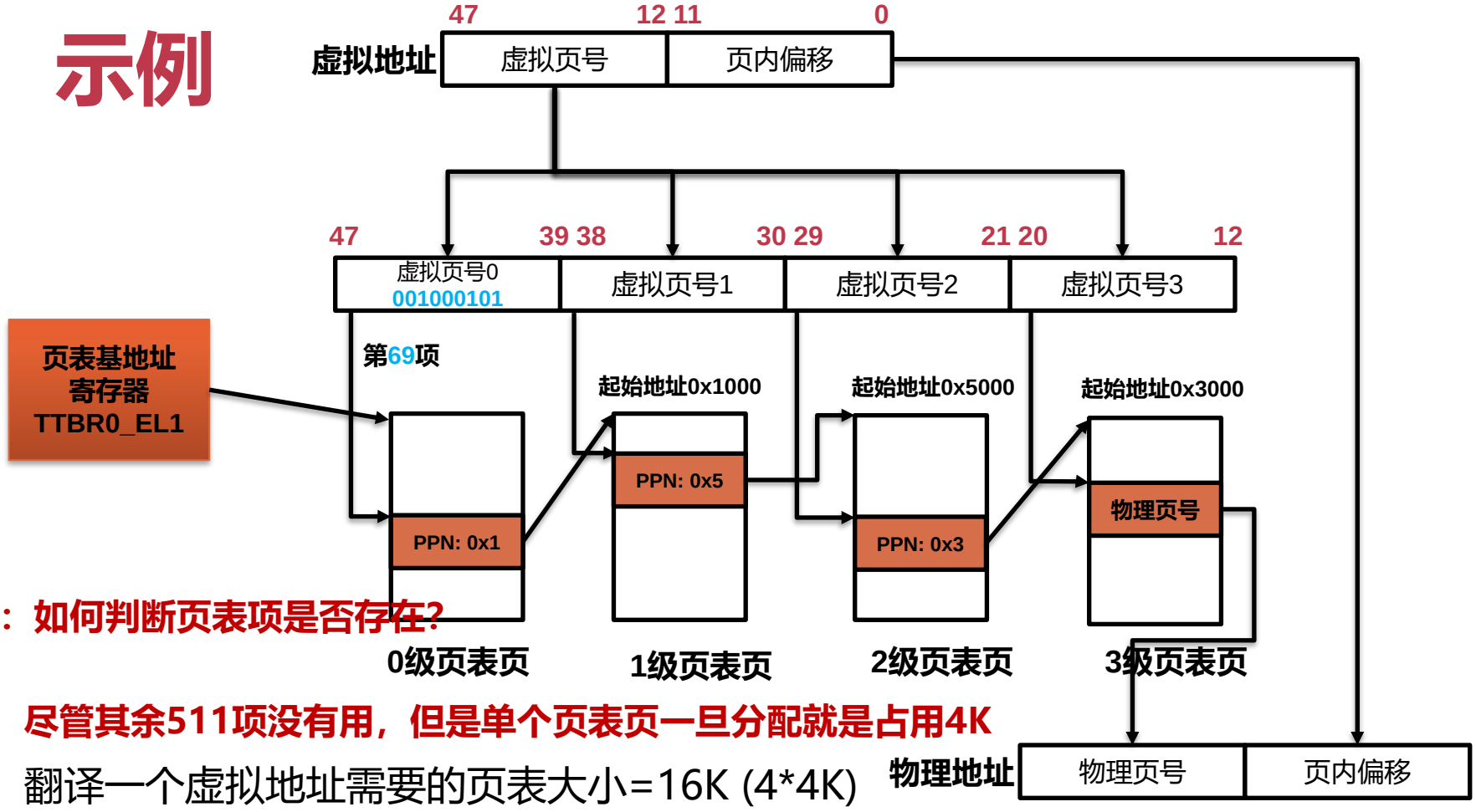
硬件规定的页表格式

- 每个页表页占用一个4K物理页
- 每个页表项占用8个字节
  - 每个页表页有512个页表项



Q: 操作系统填写页表使用虚拟地址还是物理地址?

示例



Q：如何判断页表项是否存在?

尽管其余511项没有用，但是单个页表页一旦分配就是占用4K

翻译一个虚拟地址需要的页表大小=16K (4*4K)

问：若页表共占用16K，能够翻译的虚拟地址范围是？ 2M

页表项中的属性位

页描述符：指向4K页

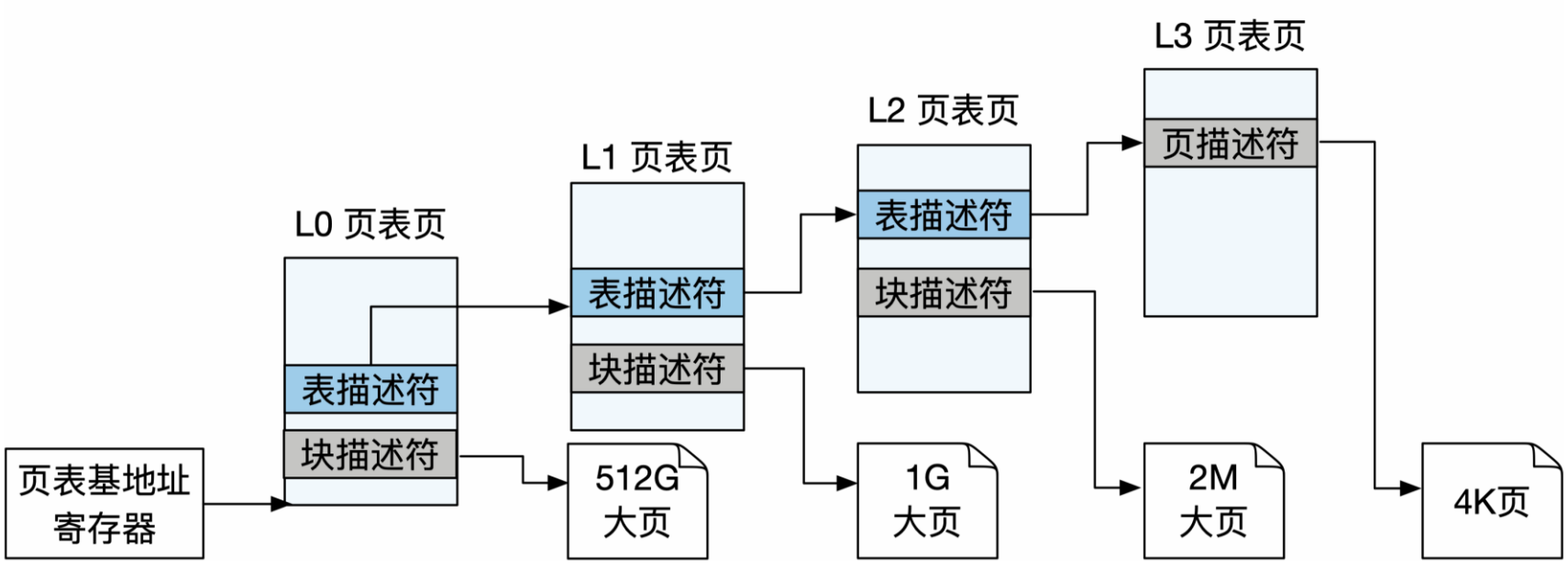
3级页表项

表描述符：指向下一级页表

0级、1级、2级页表项

块描述符：指向大页

页表项与大页


L3 页表页
L2 页表页
L1 页表页
L0 页表页
页描述符
表描述符
块描述符
表描述符
块描述符
表描述符
块描述符
页表基地址
寄存器
512G
大页
1G
大页
2M
大页
4K页

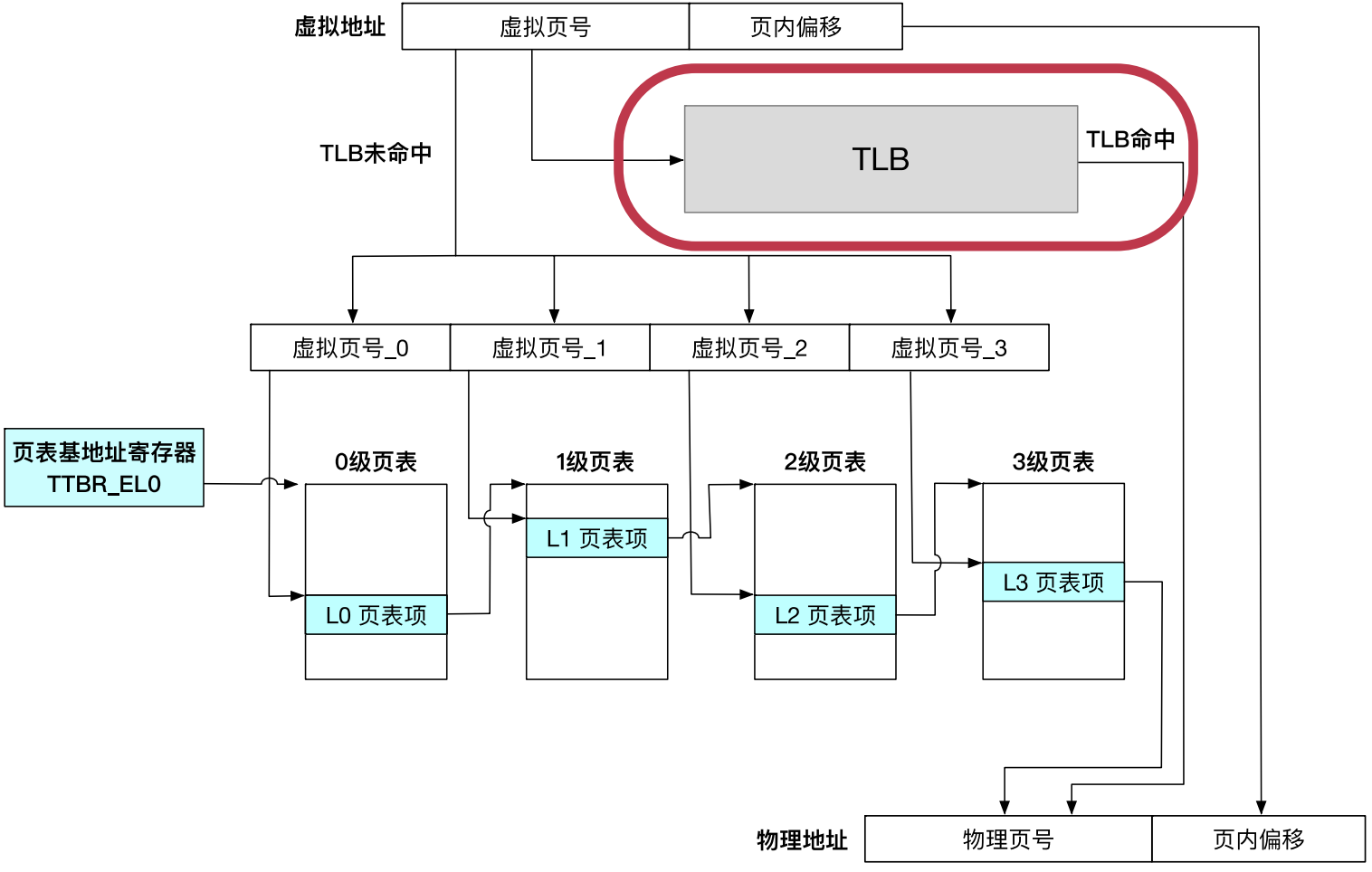
加速地址翻译

TLB：缓存页表项

多级页表不是完美的

- **Tradeoff是计算机中经典而永恒的话题**
- **多级页表的设计是典型的用时间换空间的设计**
  - 优点：压缩页表大小
  - 缺点：增加了访存次数（逐级查询）
    - 1次内存访问，变成了5次内存访问
- **如何降低地址翻译的开销?**

TLB：地址翻译的加速器

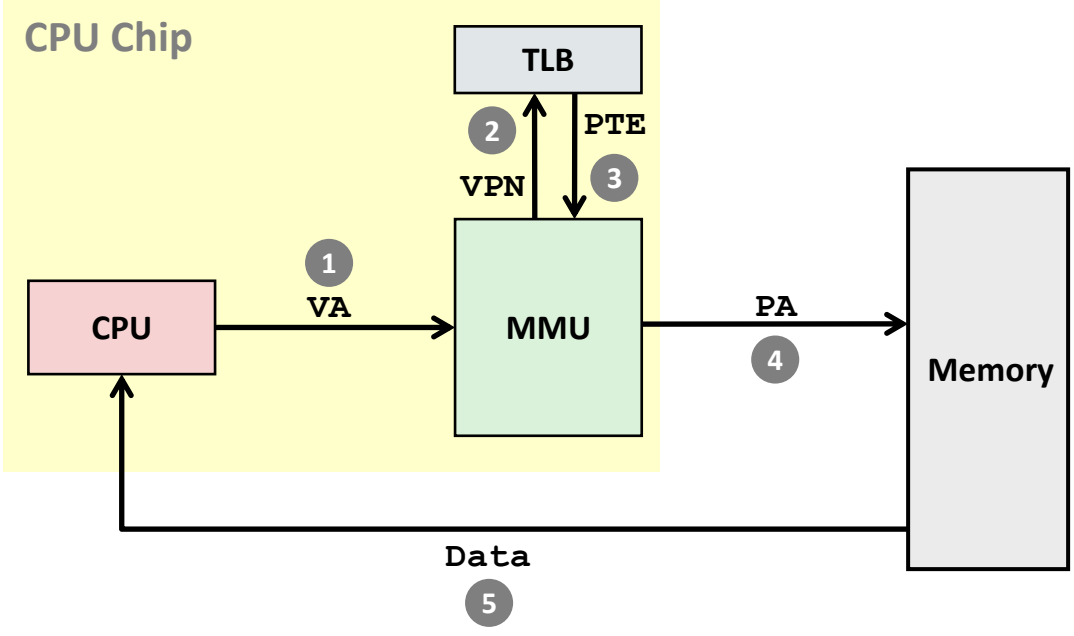


TLB：地址翻译的加速器

- **TLB 位于CPU内部，是页表的缓存**
  - Translation Lookaside Buffer
  - 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

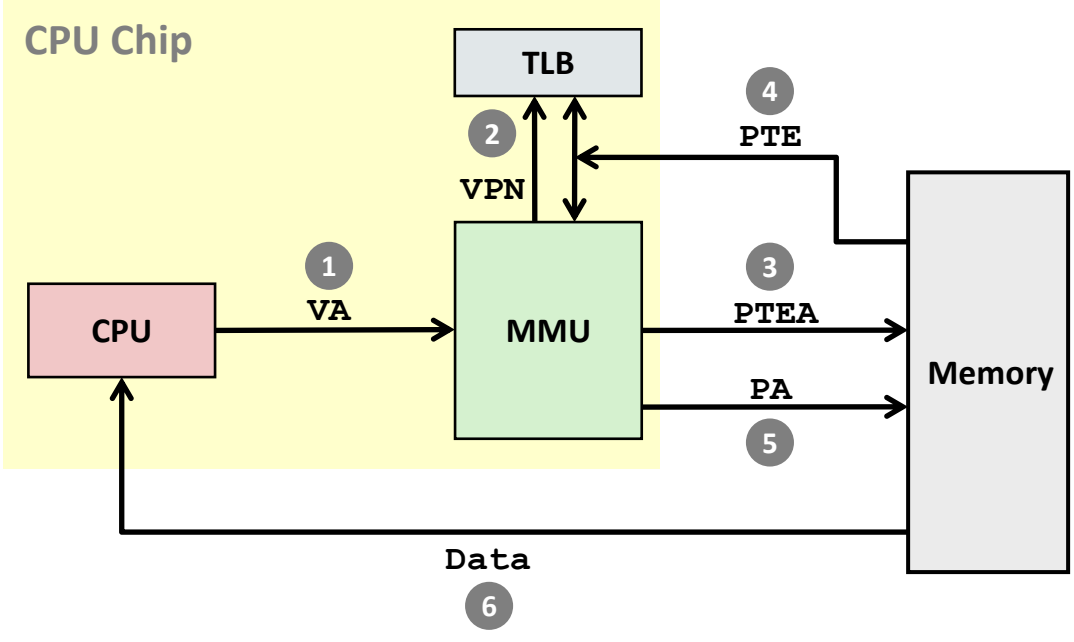
TLB 命中

- TLB 命中可以减少查询页表带来的访存次数



TLB 不命中

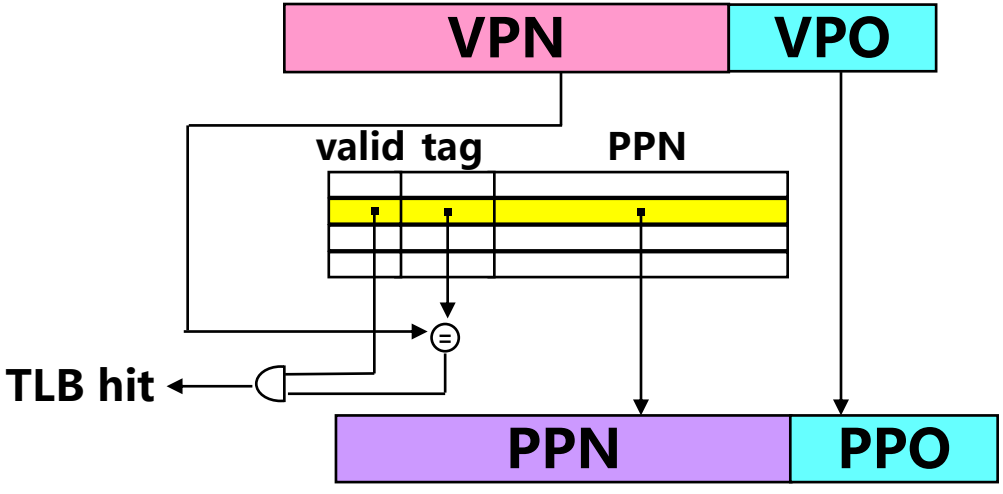
- **TLB 不命中会导致额外的访存（访问页表+访问数据）**
  - 幸运的是，TLB 不命中的情况比较少。为什么？



地址翻译

- **虚拟地址（VA）构成**
  - VPO（Virtual page offset），虚拟页偏移
  - VPN（Virtual page number），虚拟页号
    - TLBI（TLB index）
    - TLBT（TLB tag）
- **物理地址（PA）构成**
  - PPO（Physical page offset），物理页偏移
  - PPN（Physical page number），物理页号

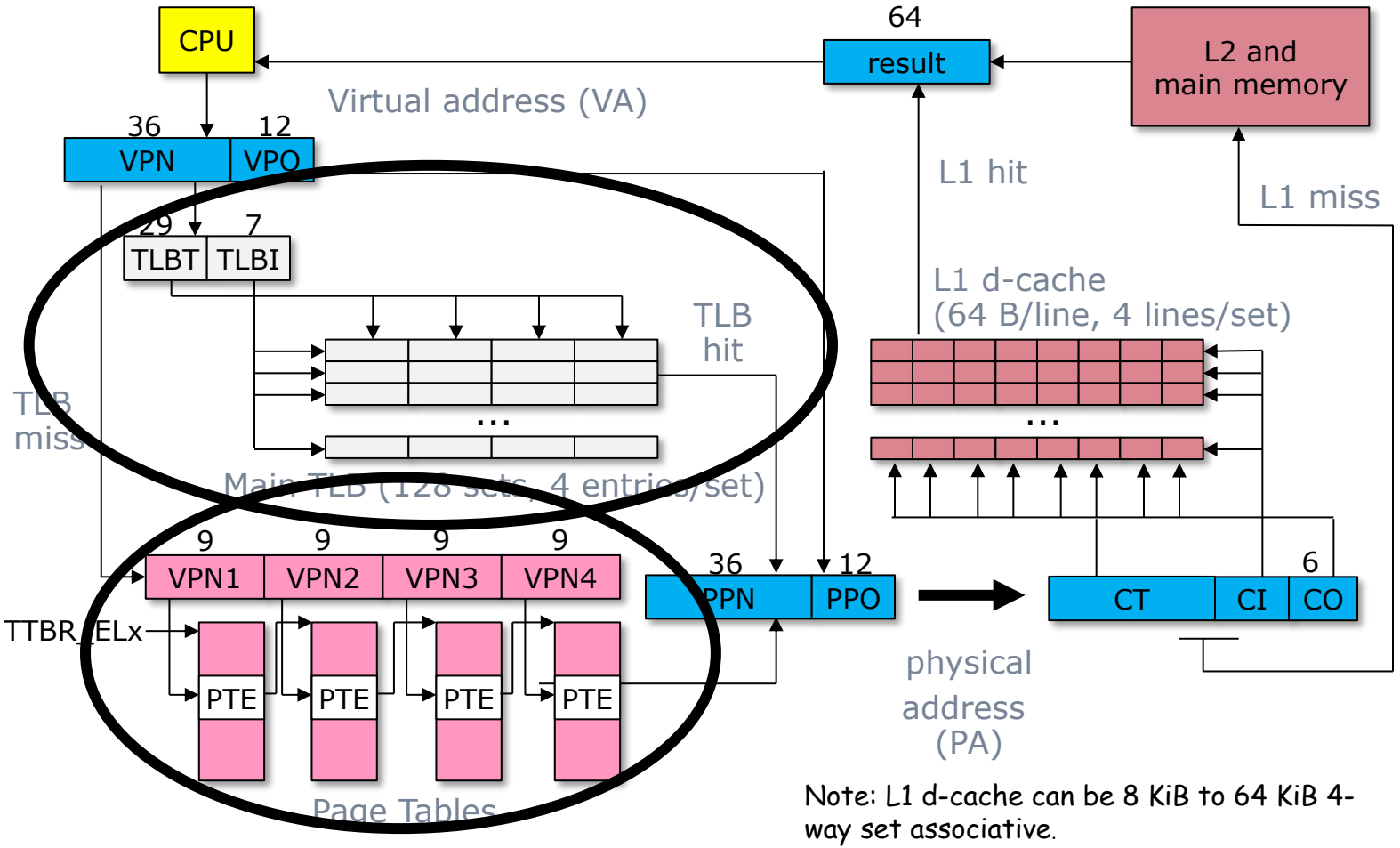
通过 TLB 加速地址翻译



虚拟地址

物理地址

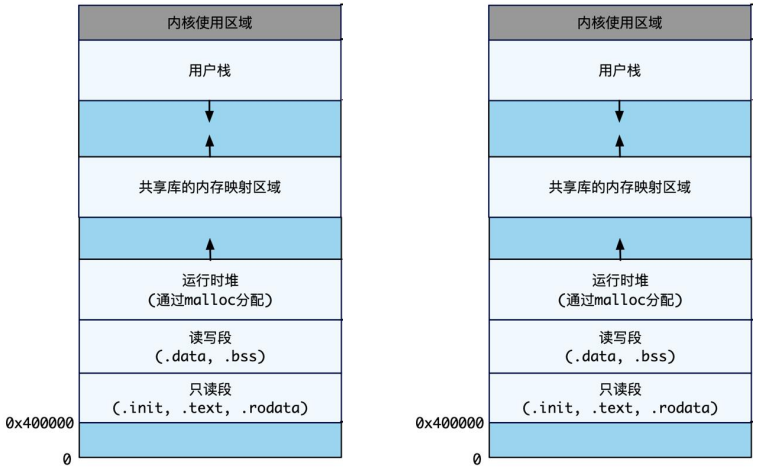
Cortex-A53 CPU实例



Note: L1 d-cache can be 8 KiB to 64 KiB 4-way set associative.

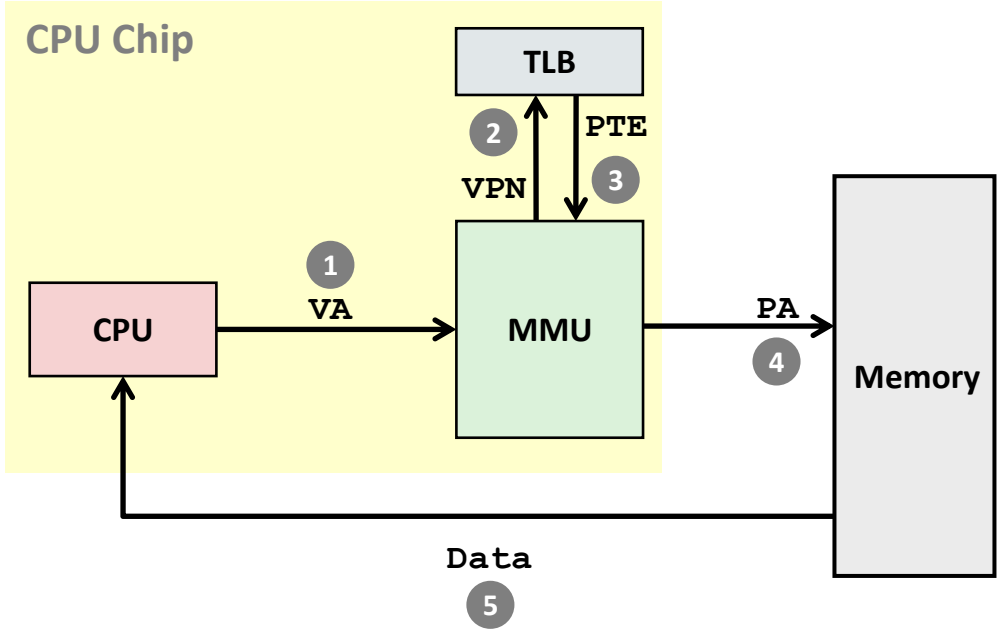
思考：引入TLB会引起翻译错误吗？

虚拟内存空间（进程-1）　虚拟内存空间（进程-2）



应用程序会使用相同的虚拟地址

TLB通过虚拟地址索引
命中后不查询页表



TLB刷新（TLB Flush）

- **TLB 使用虚拟地址索引**
  - 当OS切换进程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
  - 使用的虚拟地址范围不同
  - 系统调用过程不用切换

- **刷新TLB的相关指令**
  - 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刷新导致的开销

- **硬件特性ASID（AARCH64）： Address Space ID**
  - OS为不同进程分配8位或16位 ASID
    - ASID的位数由TCR_EL1的第36位（AS位）决定
    - OS负责将ASID填写在TTBR0_EL1的高8位或高16位
  - TLB的每一项也会缓存ASID
    - 地址翻译时，硬件会将TLB项的ASID与TTBR0_EL1的ASID对比
    - 若不匹配，则TLB miss
- **使用了ASID之后**
  - 切换页表（即切换进程）后，不再需要刷新TLB，提高性能
  - 修改页表映射后，仍需刷新TLB（为什么？）

TLB与多核

- **OS修改页表后，需要刷新其它核的TLB吗?**
  - 需要，因为一个进程可能在多个核上运行
- **OS如何知道需要刷新哪些核的TLB?**
  - 操作系统知道进程调度信息
- **OS如何刷新其他核的TLB?**
  - 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
    - 调用的ARM指令：TLBI ASIDE1IS

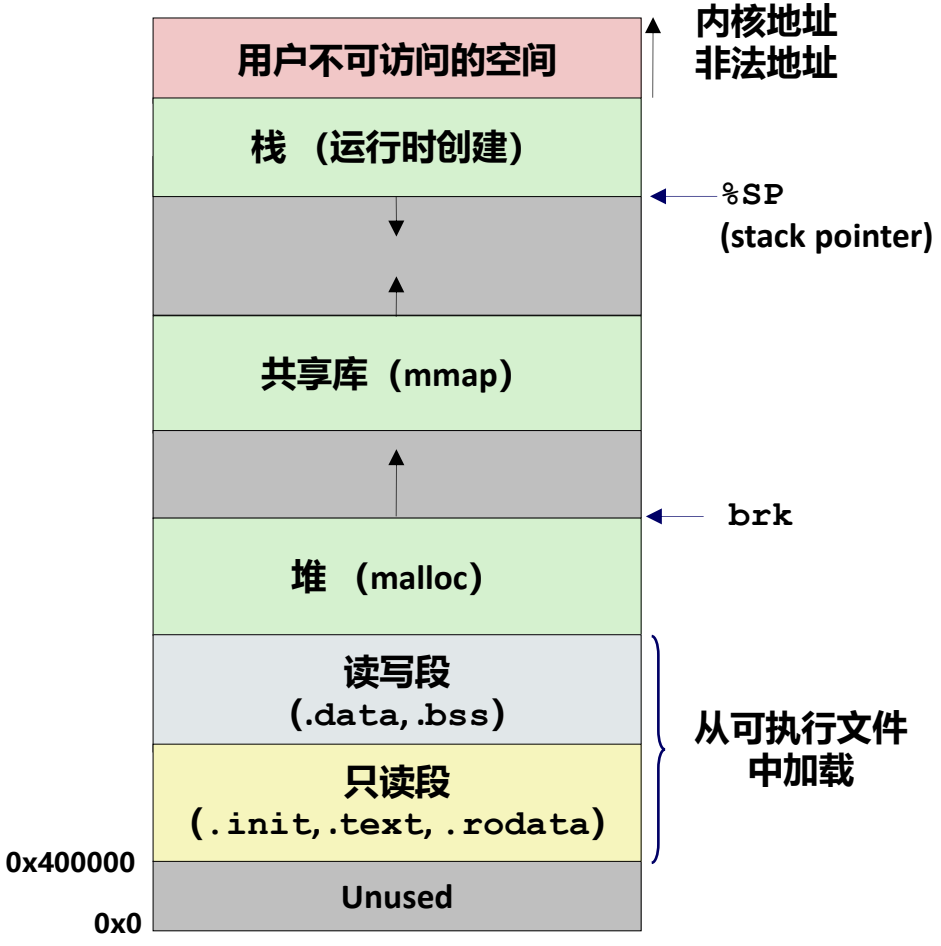
小结：虚拟内存机制的优势

虚拟内存机制的优势

- **高效使用物理内存**
  - 使用 DRAM 作为虚拟地址空间的缓存
- **简化内存管理**
  - 每个进程看到的是统一的线性地址空间
- **更强的隔离与更细的权限控制**
  - 一个进程不能访问属于其他进程的内存
  - 用户程序不能够访问特权更高的内核信息
  - 不同内存页的读、写、执行权限可以不同

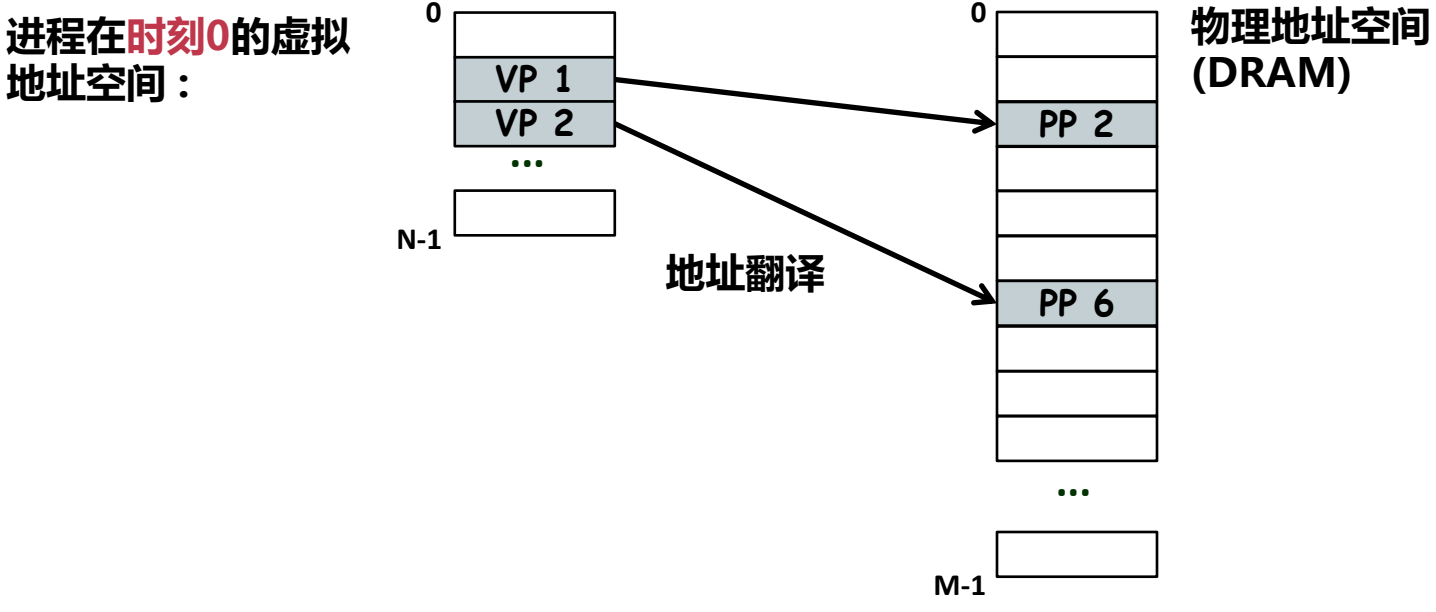
每个进程拥有独立的虚拟地址空间

- **不同进程互不干扰**
  - 仿佛独占所有内存
- **绝大部分地址段均可用**
  - 除了顶部的内核地址区域



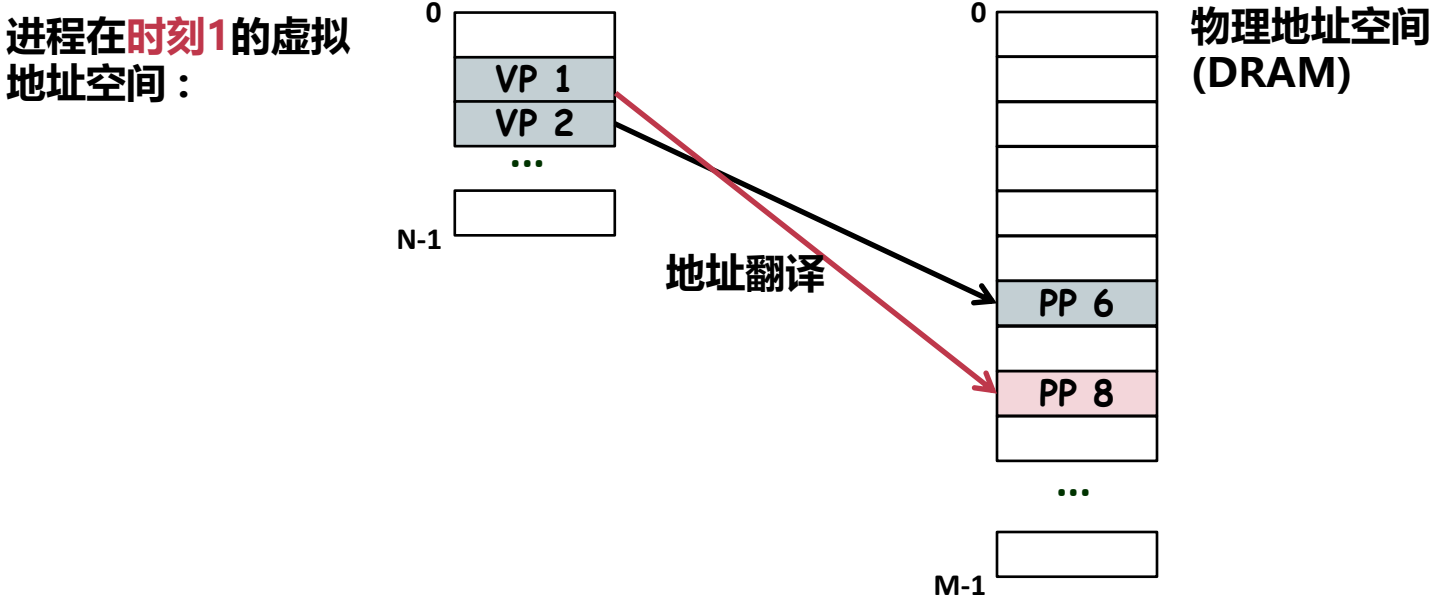
灵活的虚拟内存-物理内存映射

- 每个虚拟页都可以被映射到任意物理页
- 一个虚拟页可以在不同的时刻存储在不同的物理页中



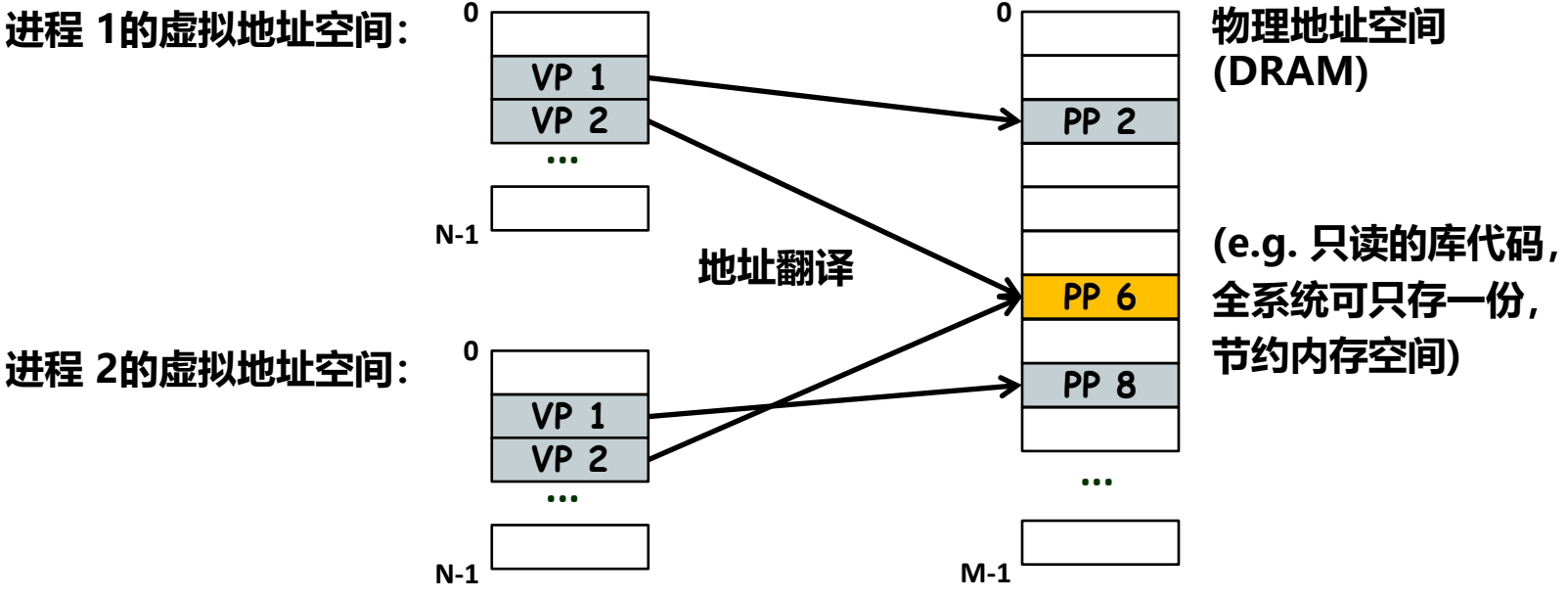
灵活的虚拟内存-物理内存映射

- 每个虚拟页都可以被映射到任意物理页
- 一个虚拟页可以在不同的时刻存储在不同的物理页中



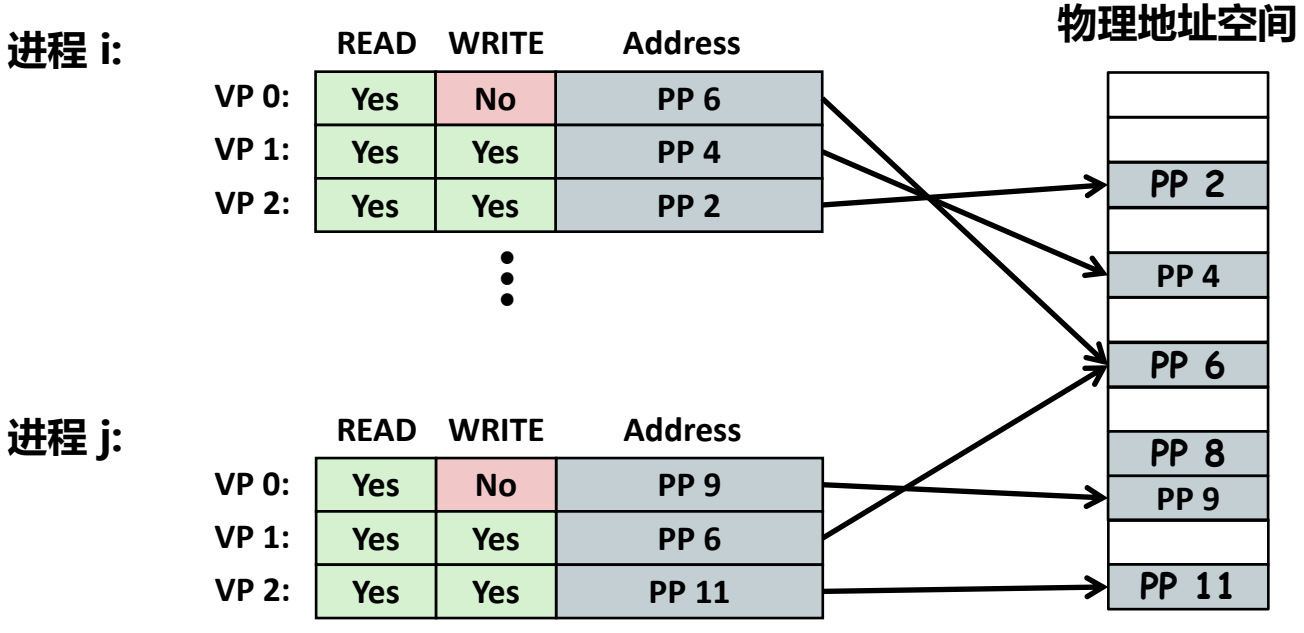
可在不同进程之间共享内存

- 不同虚拟地址空间的虚拟内存页可映射到相同的物理页



基于虚拟内存实现内存保护

- **不同的进程对相同的物理页拥有不同的权限**
  - 通过页表项中的权限位来控制



总结：地址翻译的硬件基础

MMU（硬件）、页表（内存中的数据结构，硬件规定结构）、TLB（MMU内部硬件）

问题

- **启动时，页表机制开启的一刻，发生了什么?**
  - 开启前使用物理地址，开启后使用虚拟地址