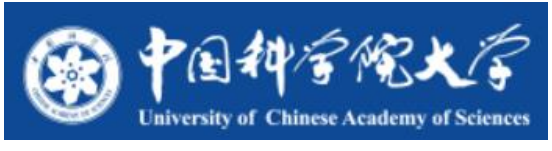
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

虚拟内存管理

郑晨

改编声明

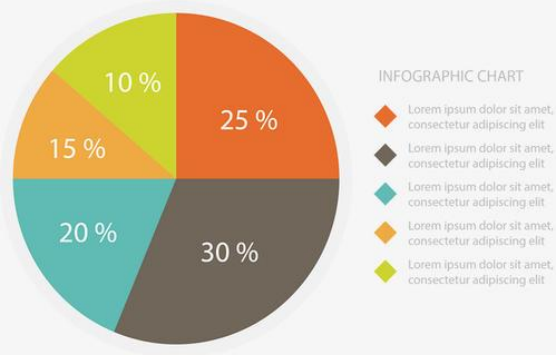
- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



虚拟内存：硬件做了什么？

内存使用

内存的重要性



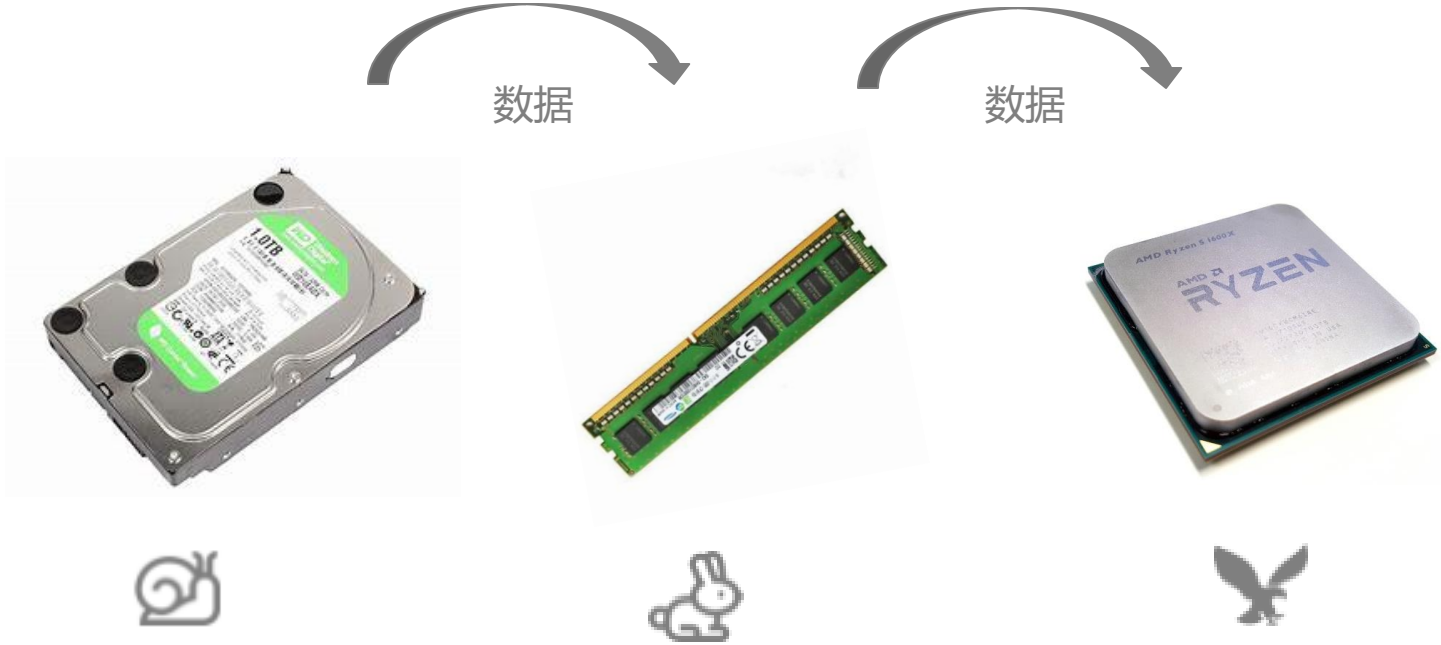
市场份额

三星内存厂着火

国产光威内存

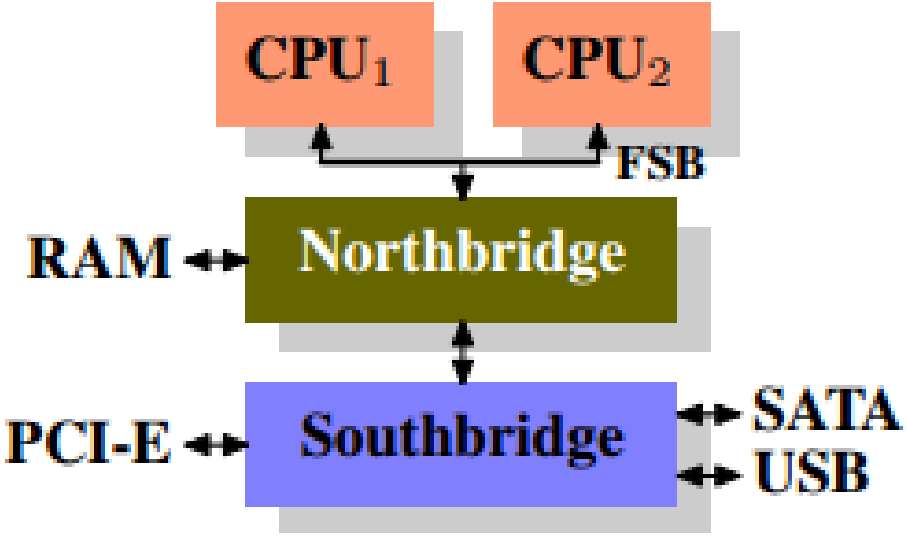
竞品降价

Data flow



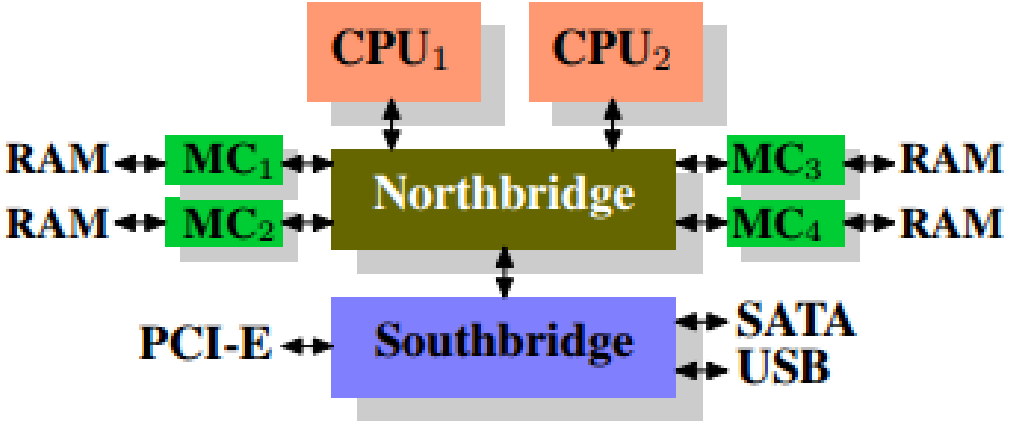
内存

- 内存在计算机体系结构中的位置



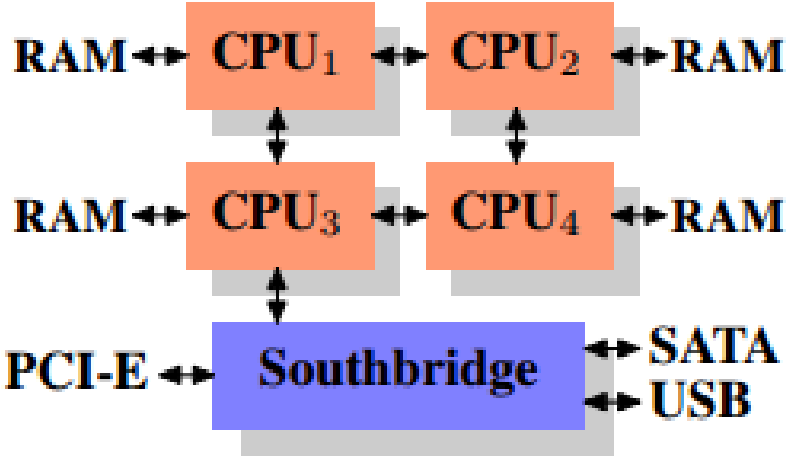
内存

- 内存在计算机体系结构中的位置



内存

- 内存在计算机体系结构中的位置



NUMA（Non-Uniform Memory Architecture）,非一致性内存访问架构

内存

- **内存分类**
  - 可分为SRAM（静态RAM）、DRAM（动态RAM）
  - SRAM比DRAM速度快，那么

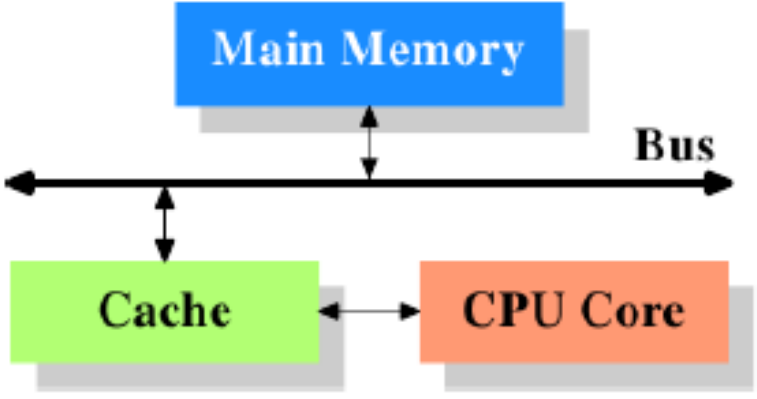
为什么会有两种类型的RAM?
为什么所有的RAM不采用SRAM?

因为SRAM比DRAM更贵，使用的代价更高。

SRAM用于CPU cache中。

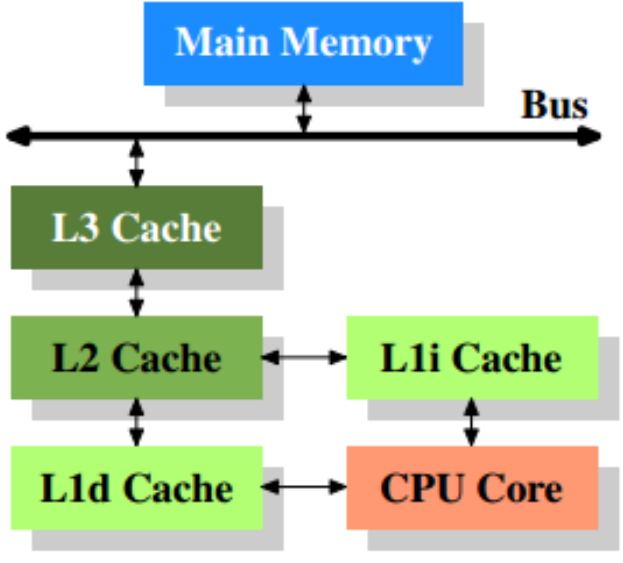
CPU 缓存

- CPU Cache



CPU Cache

- 多级缓存

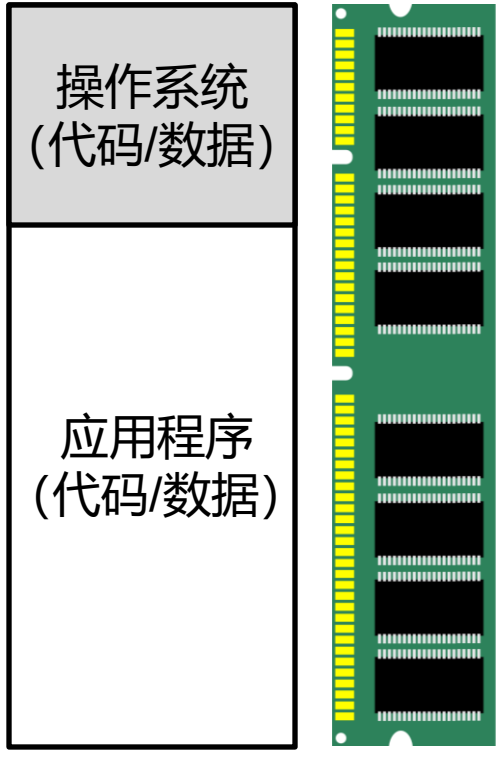


指令cache
数据cache
TLB （Translation Lookaside Buffer）

存取数据的速度比较： L1 > L2 > L3 cache

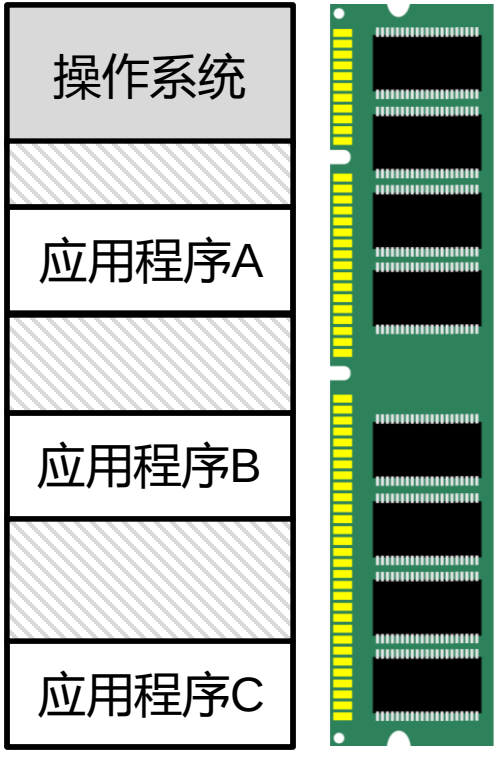
最早期的计算机系统

- **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



多道编程时代

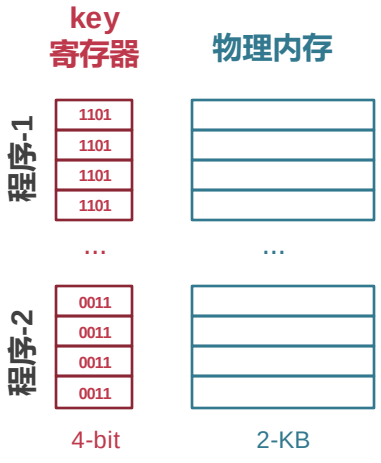
- **多用户多程序**
  - 计算机很昂贵，多人同时使用（远程连接）
- **分时复用CPU资源**
  - 保存恢复寄存器速度很快
- **分时复用物理内存资源**
  - 将全部内存写入磁盘开销太高
- **同时使用、各占一部分物理内存**
  - 没有安全性（隔离性）



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?

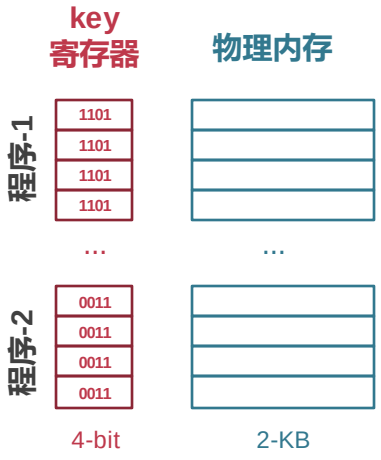
IBM 360的内存隔离：Protection Key

- **Protection key机制（无需虚拟内存也可实现隔离）**
  - 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
  - 当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
    - 不匹配则拒绝内存访问



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
- **可能的解决方法**
  - 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：
    - 每次访存都要为地址增加偏移量，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？



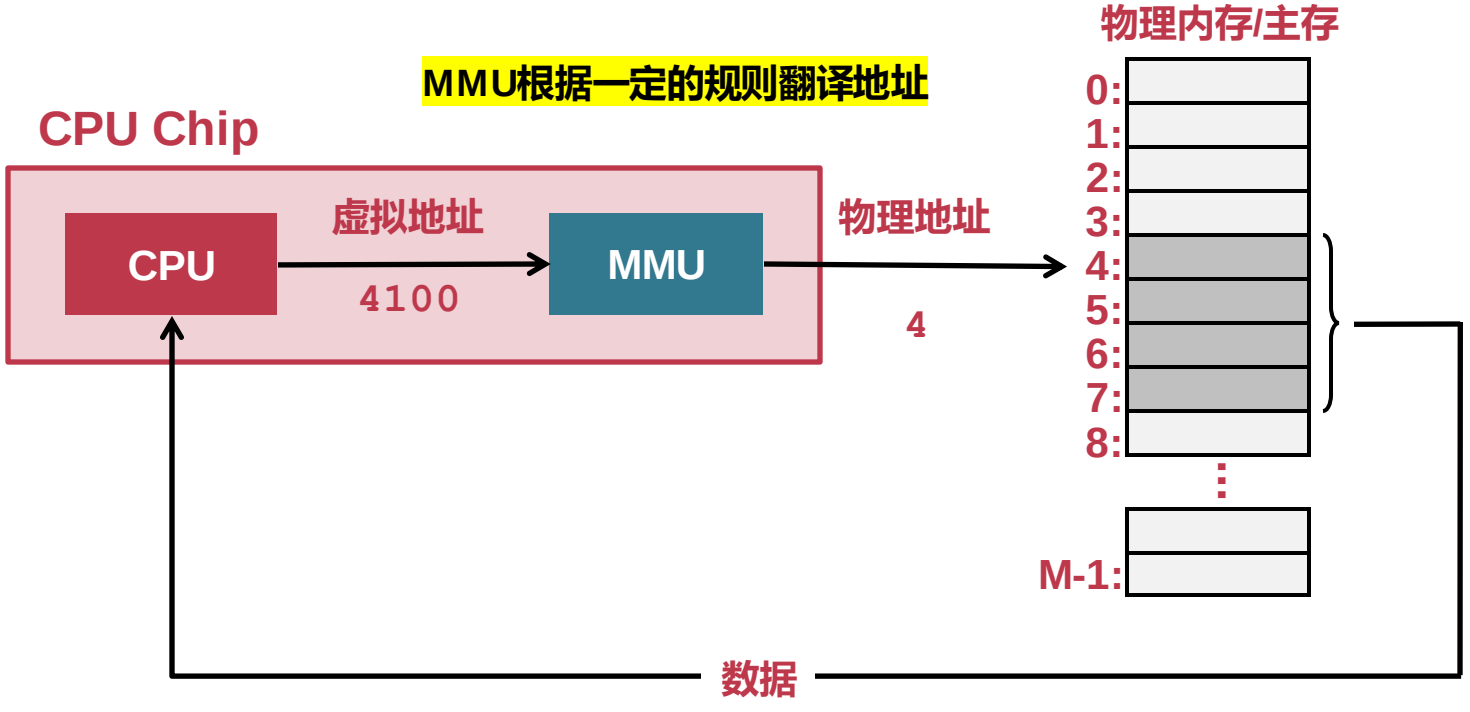
使用物理地址的缺点

- **物理地址对应用是可知的，导致：**
  - 干扰性：一个应用会因其他应用的加载而受到影响
  - 扩展性：一旦物理地址范围确定，则很难使用更大范围的内存
  - 安全性：一个应用可通过自身的内存地址，猜测其他应用的位置
- **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

虚拟地址 VS. 物理地址

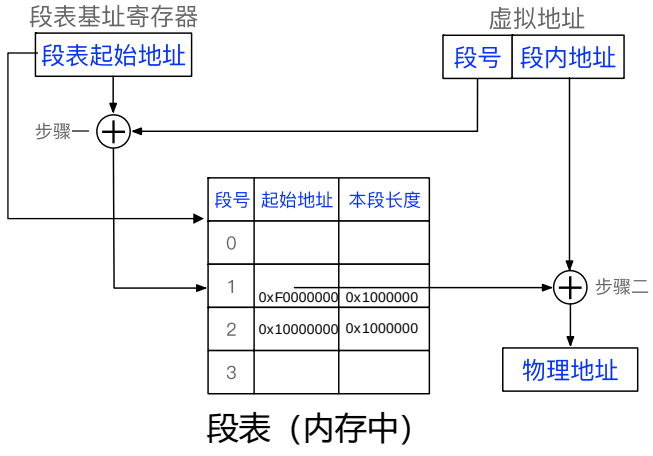
- **虚拟内存抽象下，程序使用虚拟地址访问主存**
  - 虚拟地址会被硬件"自动地"翻译成物理地址
- **每个应用程序拥有独立的虚拟地址空间**
  - 应用程序认为自己独占整个内存
  - 应用程序不再看到物理地址
  - 应用加载时不用再为地址增加一个偏移量
- **同时也可天然地支持隔离：不同的地址空间彼此隔离**

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

方法-1：分段机制



物理内存空间

虚拟内存空间（进程-1）

+0xF0000000

0xF0500000

0xF0400000

0x400000

段号：1

0x10500000

+0x10000000

虚拟内存空间（进程-2）

0x400000

段号：2

0x10400000

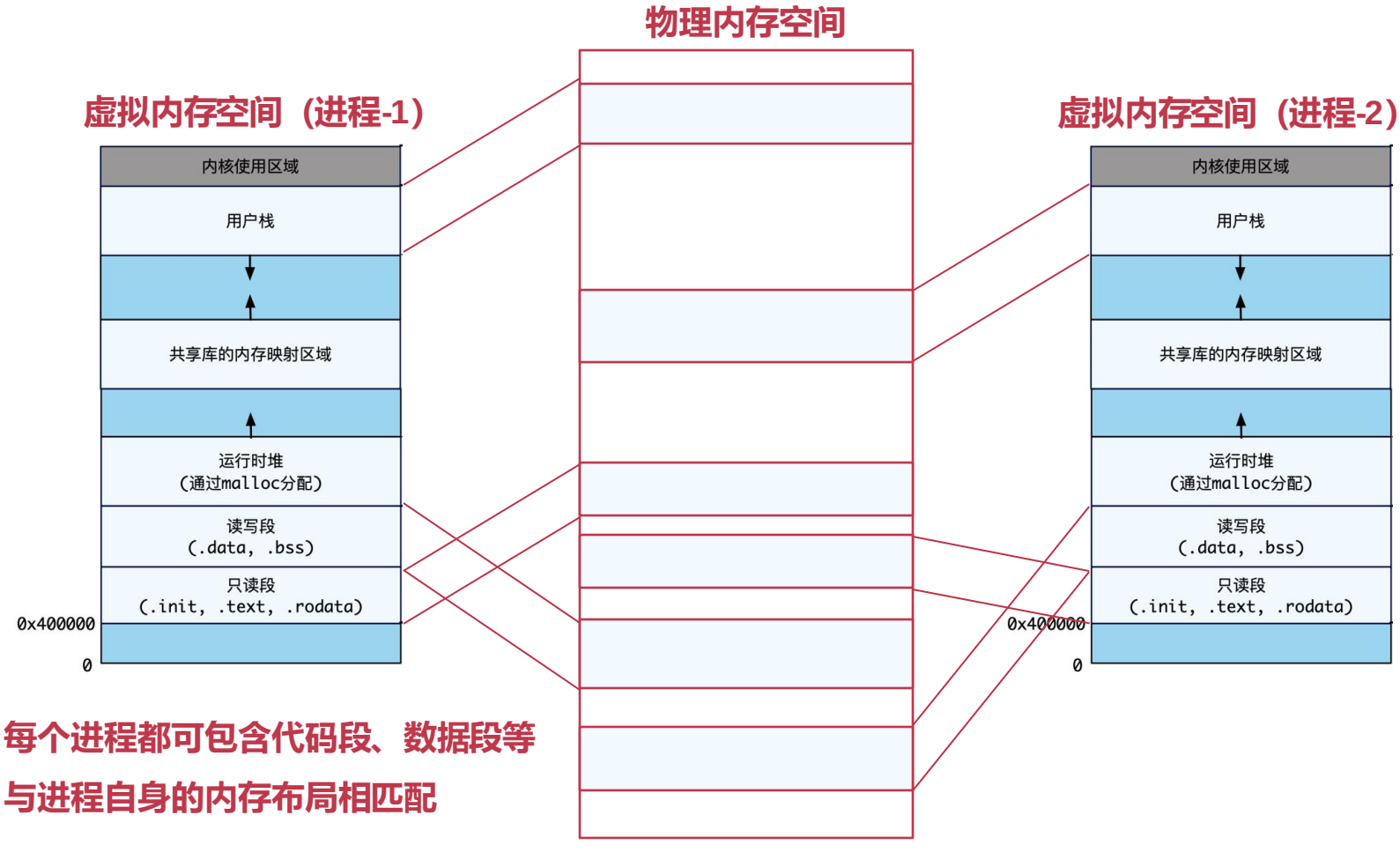
0x0

虚拟地址分为：段号 + 段内地址

- 段号：由操作系统决定
- 段内地址：由编译器生成

- 段表：由操作系统配置
- 翻译：由CPU负责查表并转换

方法-1：分段机制（更细粒度）

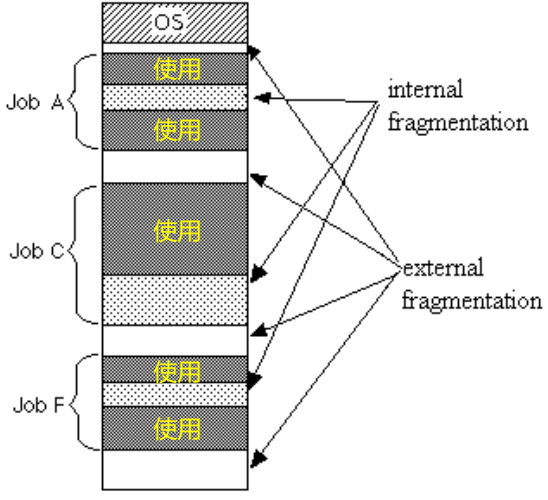


每个进程都可包含代码段、数据段等

与进程自身的内存布局相匹配

分段机制的问题

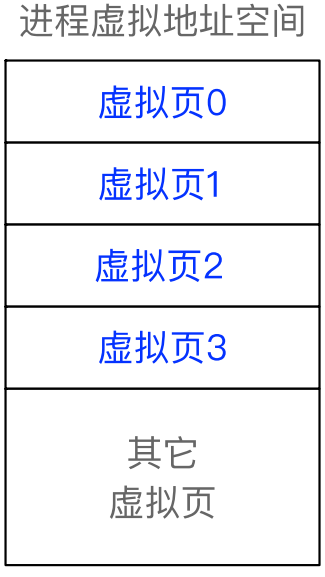
- **对物理内存连续性的要求**
  - 物理内存也必须以段为单位进行分配
- **存在问题：内存利用率**
  - 外部碎片：段与段之间留下碎片空间
  - 内部碎片：段内部预留未使用的碎片空间



- **分段机制常见于x86平台**
  - 现代操作系统通常不再依赖分段

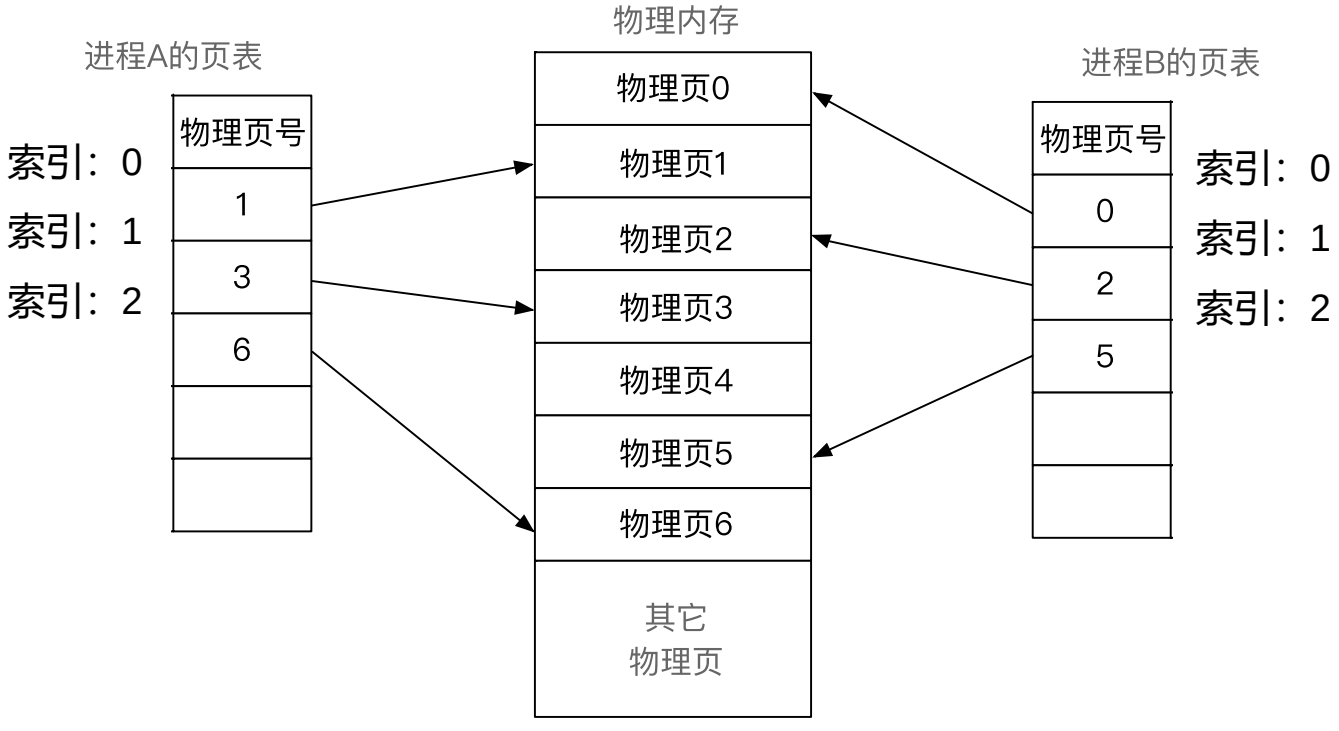
方法-2：分页机制

- **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**

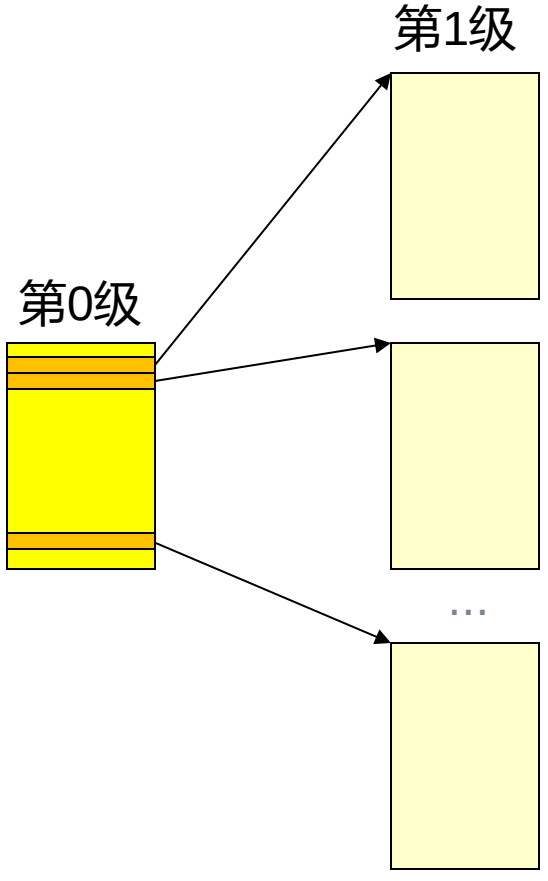
硬件层

页表格式

多级页表

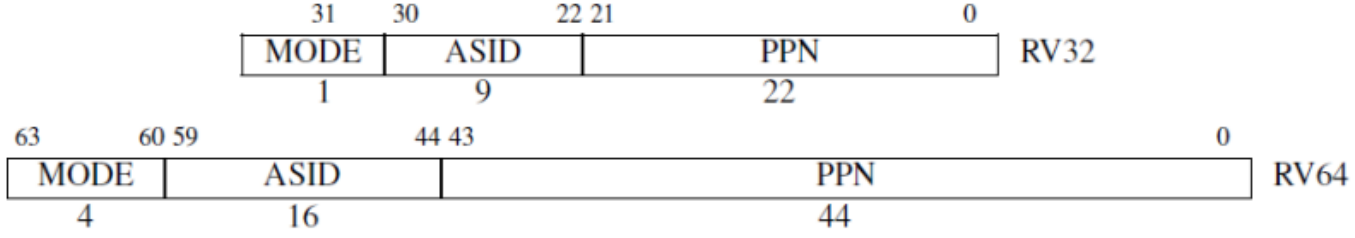
- **多级页表能有效压缩页表的大小**

- **原因：允许页表中出现空洞**
  - 若某级页表中的某条目为空，那么该条目对应的下一级页表便无需存在
  - 应用程序的虚拟地址空间大部分都未分配



内存系统相关寄存器：RISC-V

- satp（Supervisor Address Translation and Protection，监管者地址转换和保护）



图：satp CSR

satp（Supervisor Address Translation and Protection，监管者地址转换和保护）的 S 模式控制状态寄存器控制了分页系统

satp 有三个域。MODE 域可以开启分页并选择页表级数。

ASID（Address Space Identifier，地址空间标识符）域是可选的，它可以用来降低上下文切换的开销。

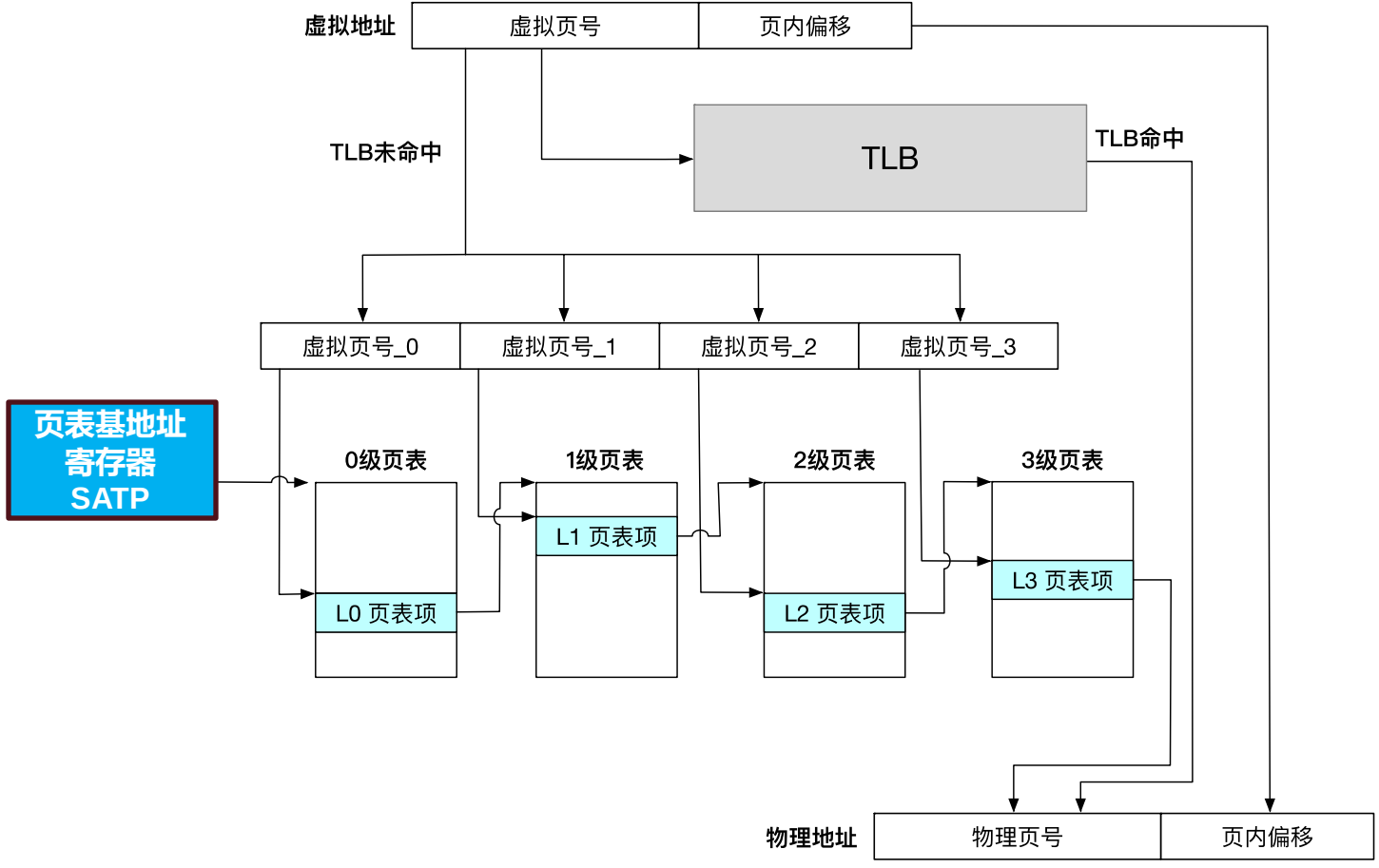
PPN 字段保存了根页表的物理地址，它以 4 KiB 的页面大小为单位。

对比X86-64: CR3寄存器

页表基地址寄存器

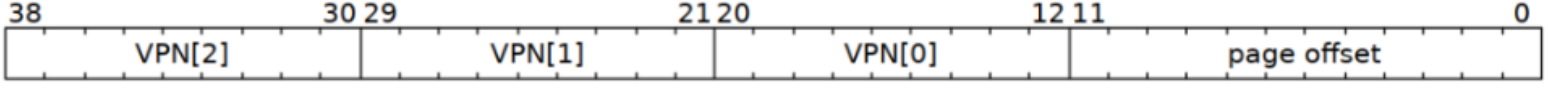
- **AARCH64**
  - 两个寄存器：TTBR0_EL1 & TTBR1_EL1
    - 根据虚拟地址第63位选择
  - 以Linux为例
    - 应用程序（地址首位为0）：使用TTBR0_EL1
    - 操作系统（地址首位为1）：使用TTBR1_EL1
- **X86_64**
  - 一个寄存器：CR3（Control Register 3）

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
SATP
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

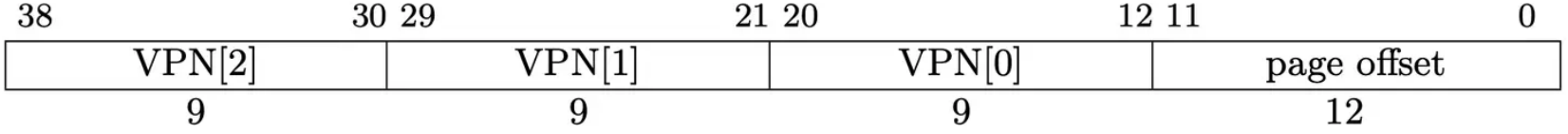
39位虚拟地址解析

- RV64支持多种分页方案，最受欢迎的是SV39



- 「38：30」9-bit：2级页表索引
- 「29：21」9-bit：1级页表索引
- 「20：12」9-bit：0级页表索引
- 「11：0」12-bit：页内偏移

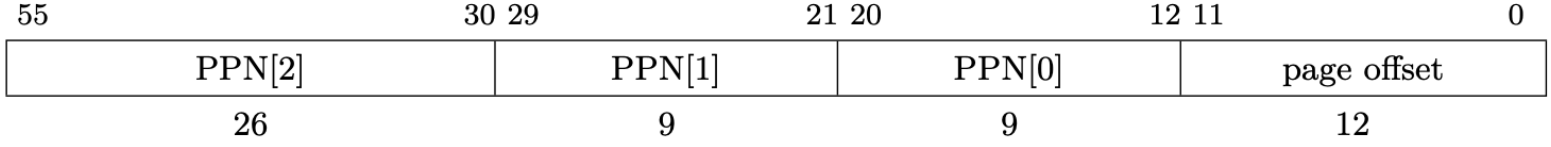
SV39 分页机制



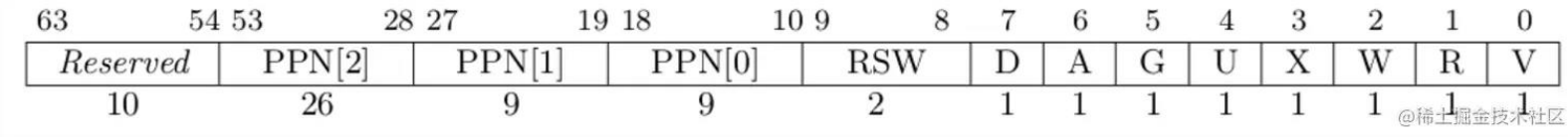
- **以SV39的虚拟内存管理为例**
  - 使用了64位虚拟地址中的**低39位**，而高25位未使用
  - 每一页占用4KB内存，**页内**使用虚拟地址低12位寻址。
  - 虚拟地址的高27位划分为**三级页号**，每一级都有512个可用的页号。
  - 每个页表项占64bit，一个页表占多大内存空间?
  - 三级页表共有多少个页表项?

SV39 页表项

- **SV39的物理地址：**



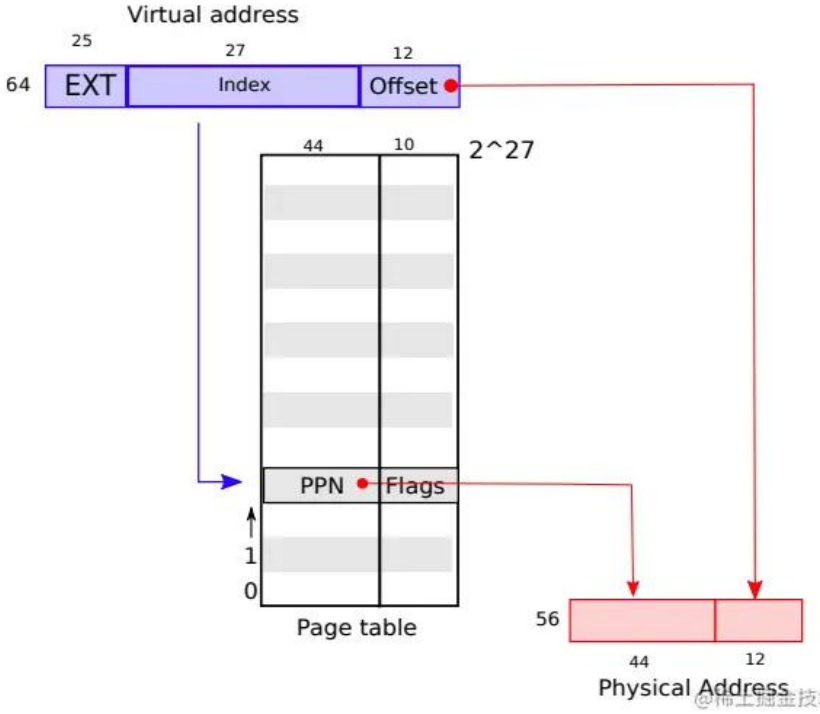
- **SV39的页表项（PTE）：**



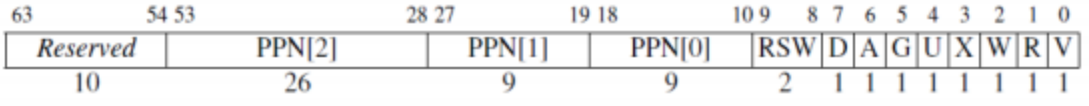
SV39 页表项

63–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

- 每个PTE包含一个44位的**物理页号**（**PPN**，physical page number）和一些标志位
- 分页硬件使用39位虚拟地址中的高27位作为索引，在页表中找到对应的PTE，然后用**PTE**中的44位地址作为高位（物理页号），**虚拟地址**中的低12位作为低位（页内偏移量），构造出一个56位的物理地址。

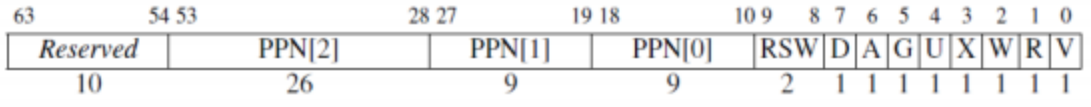


SV39 页表项



- V 位决定了该页表项的其余部分是否有效（V = 1 时有效）。若 V = 0，则任何遍历到此页表项的虚址转换操作都会导致页错误
- R、W 和 X 位分别表示此页是否可以读取、写入和执行。如果这三个位都是 0，那么这个页表项是指向下一级页表的指针，否则它是页表树的一个叶节点

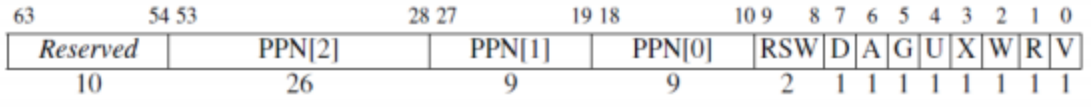
SV39 页表项



- U 位表示该页是否是用户页面。若 U = 0，则 U 模式不能访问此页面，但 S 模式可以；若 U = 1，则 U 模式下能访问这个页面，而 S 模式不能
- G 位表示这个映射是否对所有虚址空间有效，硬件可以用这个信息来提高地址转换的性能，这一位通常只用于属于操作系统的页面

图片来自于ARMv8手册

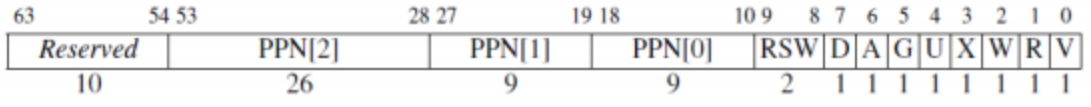
SV39 页表项



- A 位表示自从上次 A 位被清除以来，该页面是否被访问过
- D 位表示自从上次清除 D 位以来页面是否被弄脏（例如被写入）
- RSW 域留给操作系统使用，它会被硬件忽略。

图片来自于ARMv8手册

SV39 页表项



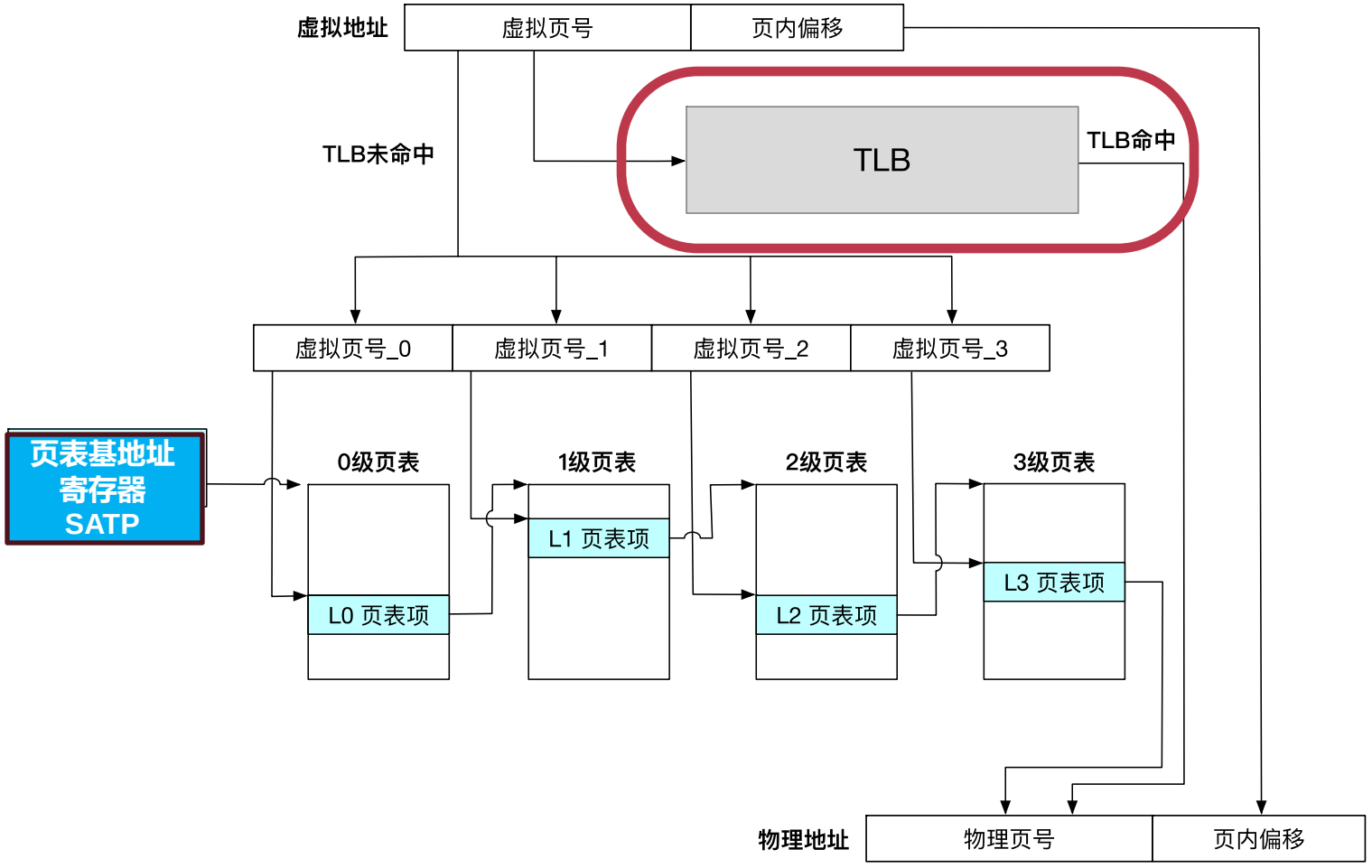
- PPN 域包含物理页号，这是物理地址的一部分
- 若这个页表项是一个叶节点，那么 PPN 是转换后物理地址的一部分
- 否则 PPN 给出下一节页表的地址

图片来自于ARMv8手册

硬件层

TLB：页表的CACHE

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清空：TLB Flush

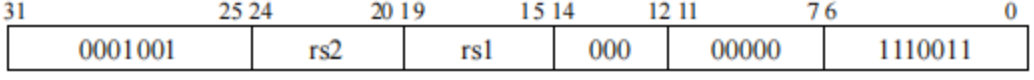
- **TLB 使用虚拟地址索引**
  - 当OS切换页表时需要全部刷掉
- **AARCH64上内核和应用程序使用不同的页表**
  - 分别存在TTBR0_EL1和TTBR1_EL1
  - 系统调用过程不用切换
- **x86_64、RISC-V上只有唯一的基地址寄存器**
  - 内核映射到应用页表的高地址
  - 避免系统调用时TLB清空的开销

- **清空TLB的相关指令**
  - SFENCE.VMA
  - 清空全部
  - 清空指定ASID相关
  - 清空指定虚拟地址

有些地方翻译为"TLB刷新"，可能会引起误导:是"刷掉TLB"（即删掉），而不是"刷新"TLB

TLB清空：TLB Flush

- **TLB 使用虚拟地址索引**
  - 当OS切换页表时需要全部刷掉
- **RV64添加了sfence.vma指令来实现**
  - 该指令会通知处理器，软件可能已经修改了页表，于是处理器可以相应地刷掉转换缓存
  - rs1指示了页表哪个虚址对应的转换被修改了，rs2给出了被修改页表的进程的地址空间标识符（ASID），如果两者都是 x0则刷掉整个转换缓存



有些地方翻译为"TLB刷新"，可能会引起误导:是"刷掉TLB"（即删掉），而不是"刷新"TLB

如何降低TLB清空导致的开销

- **新的硬件特性ASID（RISC-V）： Address Space ID**
  - OS为不同进程分配8位或16位 ASID
    - OS会将ASID填写在SATP的第44-59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SATP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，不再需要刷掉TLB，提高性能
  - 修改页表映射后，仍需刷掉TLB（为什么？）

TLB与多核

- **在多核场景下**
  - OS修改页表后，需要刷掉其它核的TLB吗?
  - OS如何知道需要刷掉哪些核的TLB?
  - OS如何刷掉其它核的TLB?

TLB与多核

- **OS修改页表后，需要清空其它核的TLB吗?**
  - 需要，因为一个进程可能在多个核上运行
- **OS如何知道需要清空哪些核的TLB?**
  - 操作系统知道进程调度信息
- **OS如何清空其它核的TLB?**
  - x86_64: 发送IPI中断某个核，通知它主动清空
  - AARCH64: 可在local CPU上清空其它核TLB
    - 调用的ARM指令：TLBI ASIDE1IS
  - RISC-V：内存管理栅栏指令 (fence instruction) SFENCE.VMA
    - 指令参数：rs1 针对页表，指示了页表哪个虚址对应的转换被修改了；rs2 给出了被修改页表的进程的地址空间标识符 (ASID)。如果两者都是 x0，便会刷新整个 TLB。
    - SBI call + IPI实现remote sfence

软件层

虚拟内存：段和VMA

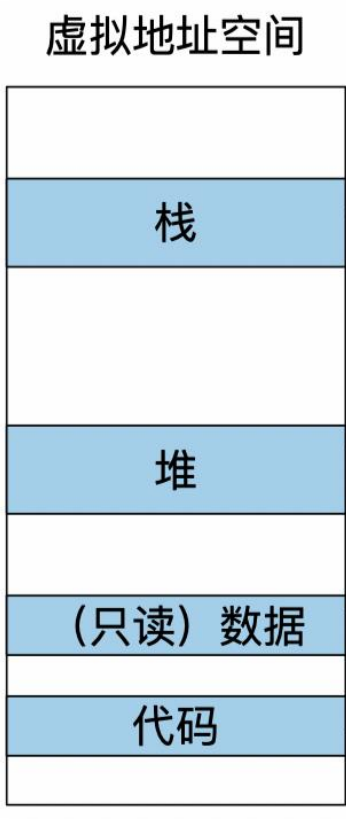
Segmentation fault

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char *p = NULL;
6    printf("%s\n", p);
7    return 0;
8  }
9
10 The output after the execution is like :
11 Segmentation fault (core dumped)
```

如果改变第5行会怎么样？比如：
char *p = 0x40000;
char *p = 0x400000000;

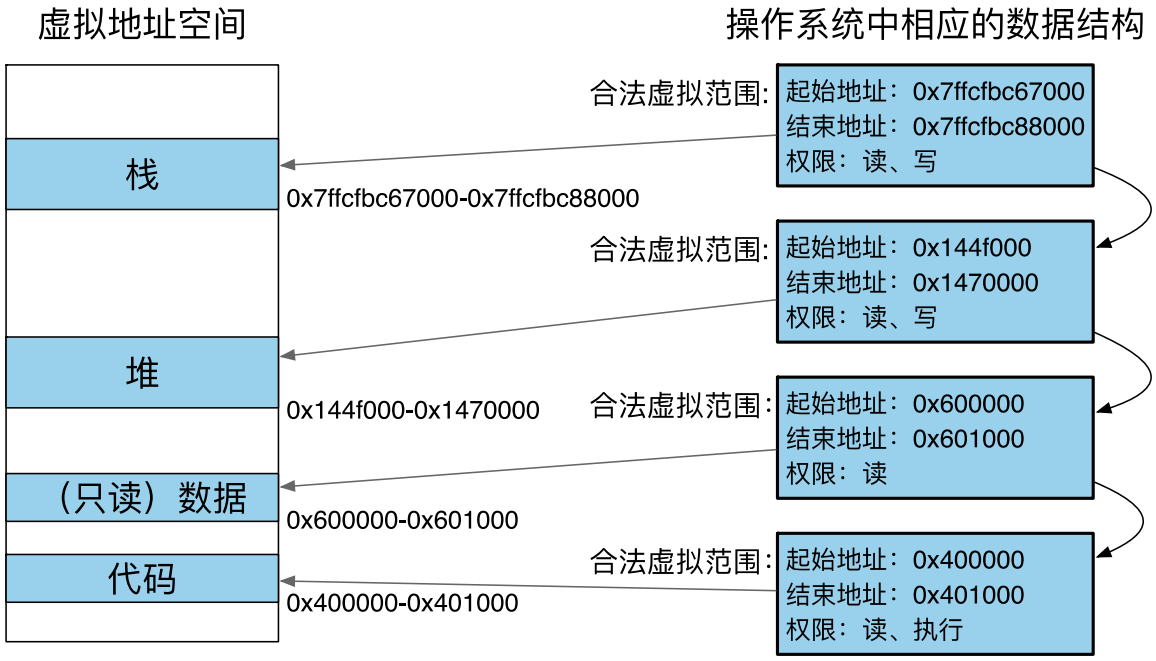
应用是否有权访问整个虚拟地址空间?

- **OS采用段来管理虚拟地址**
  - 段内连续，段与段之间非连续
  - 合法虚拟地址段：代码、数据、堆、栈
  - 非法虚拟地址段：未映射
    - 一旦访问，则触发segfault
- **思考：为什么要用段来管理?**



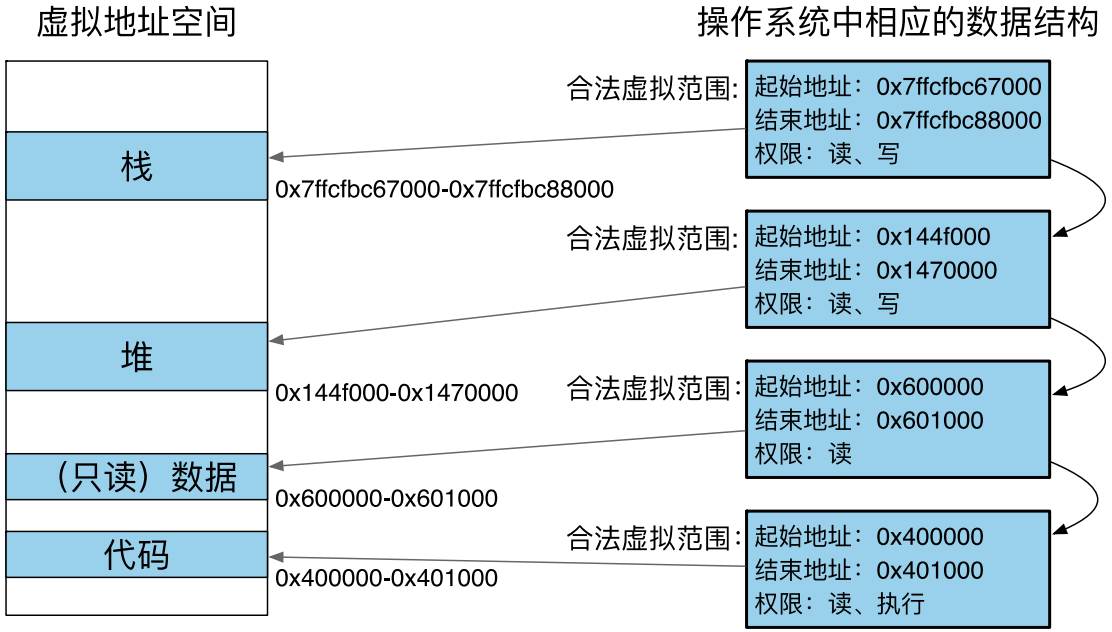
合法虚拟地址信息的记录方式

- **记录应用程序已分配的虚拟内存区域**
  - 在Linux中对应 vm_area_struct（VMA）结构体



VMA是如何添加的

- **途径-1: OS在创建应用程序时分配**
  - 数据（对应ELF段）
  - 代码（对应ELF段）
  - 栈（初始无内容）



VMA是如何添加的

- **途径2: 应用程序主动向OS发出请求**
  - brk()（扩大、缩小堆区域）
    - 可选策略： OS也可以在创建应用时分配初始的堆VMA
  - mmap()
    - 申请空的虚拟内存区域
    - 申请映射文件数据的虚拟内存区域
- **用户态的malloc会改变VMA**
  - 通常是调用brk，在堆中分配新的内存
  - 部分实现也可以调用mmap，由应用管理多个VMA

mmap：分配一段虚拟内存区域

- **通常用于把一个文件（或一部分）映射到内存**
  - void *mmap(void *addr,
    size_t length,
    int prot,
    int flags,
    int fd,
    off_t offset)

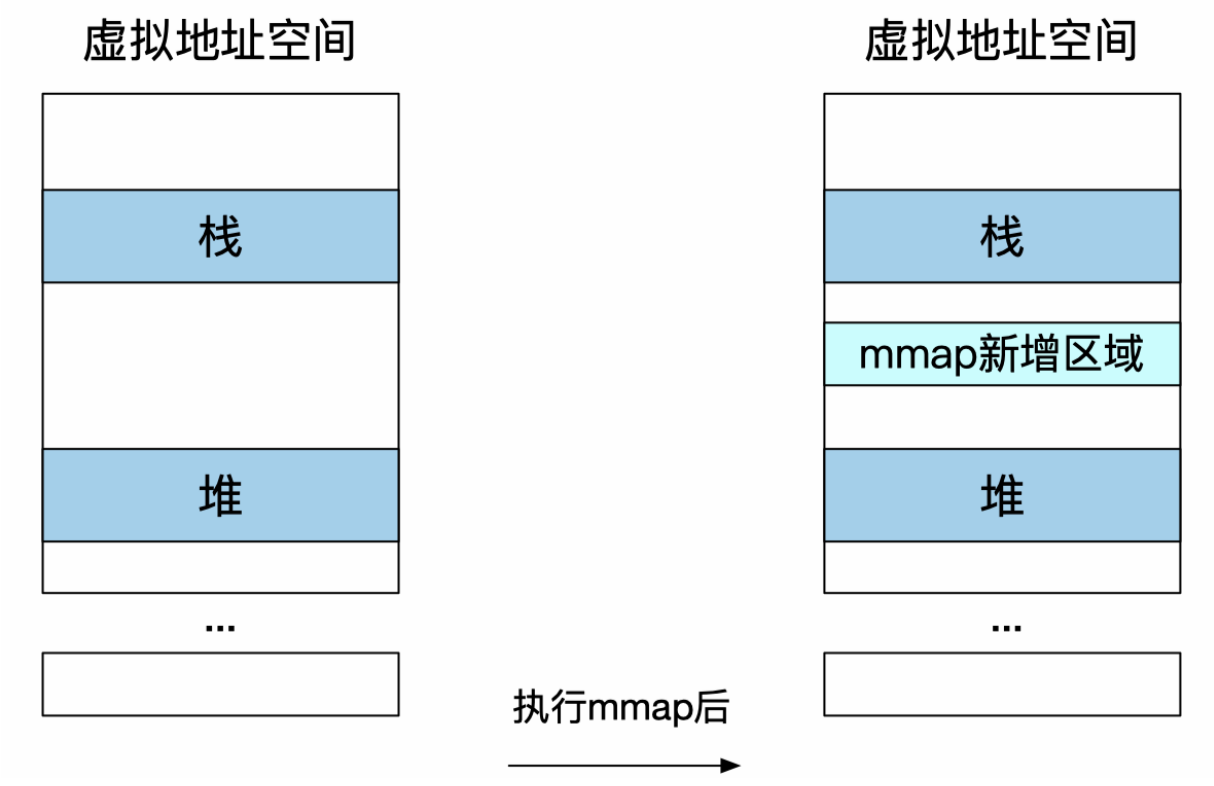
```
// 起始地址
// 长度
// 权限，例如PROT_READ
// 映射的标志，例如MAP_PRIVATE
// -1 或者是有效fd
// 偏移，例如从文件的哪里开始映射
```

- **也可以不映射任何文件，仅仅新建虚拟内存区域（匿名映射）**
  - 注意：匿名映射并非POSIX标准，但主流OS都会支持

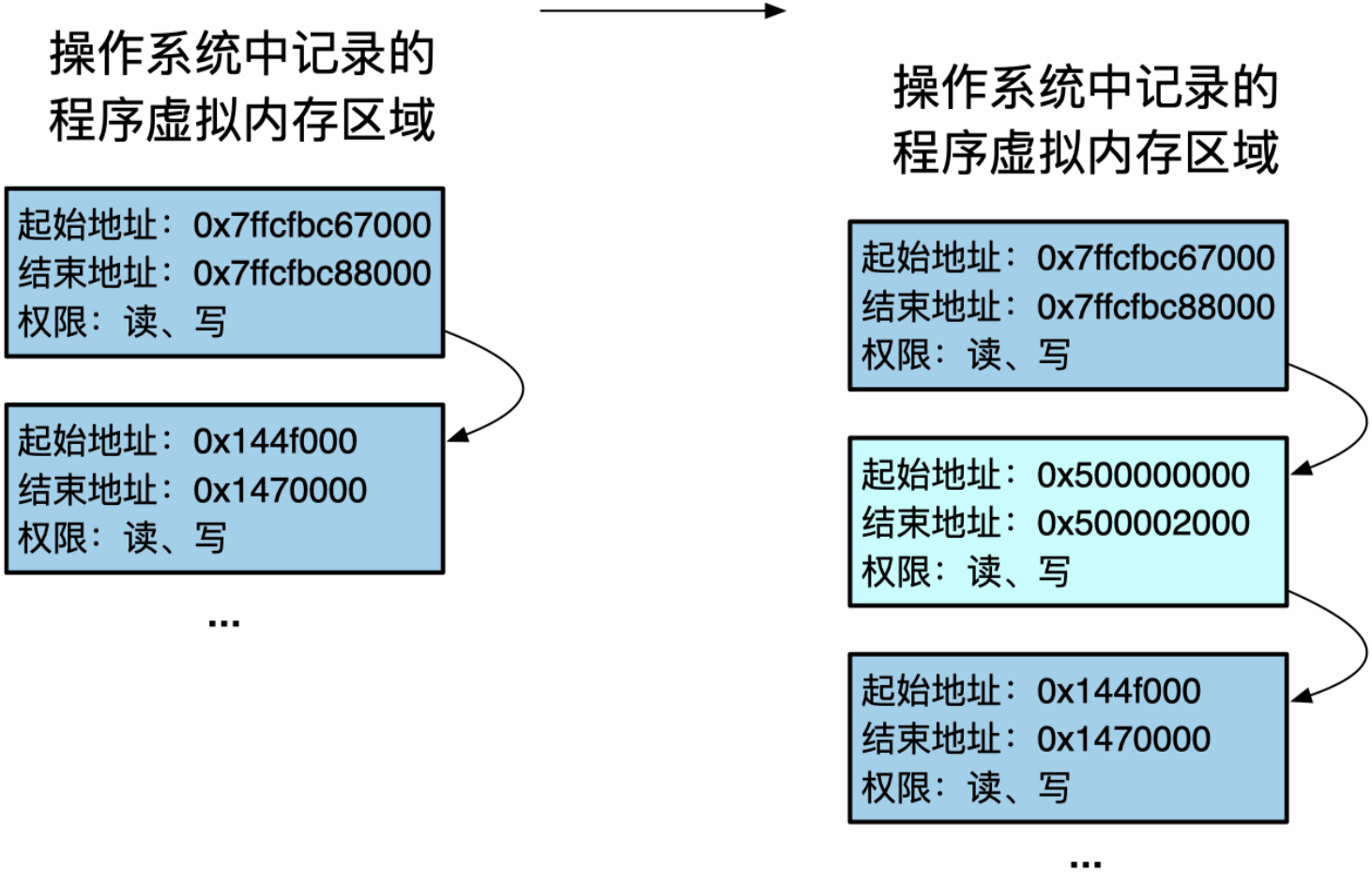
mmap匿名映射

```c
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>

// void *mmap(void *addr, size_t length, int prot, int
   ↪  flags, int fd, off_t offset);

int main()
{
  char *buf;

  buf = mmap((void *)0x500000000, 0x2000, PROT_READ |
   ↪  PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
  printf("mmap returns %p\n", buf);

  strcpy(buf, "Hello mmap");
  printf("%s\n", buf);

  return 0;
}

The output after the execution is like :
mmap returns 0x500000000
Hello mmap
```

执行mmap后，VMA的变化



执行mmap后，VMA的变化

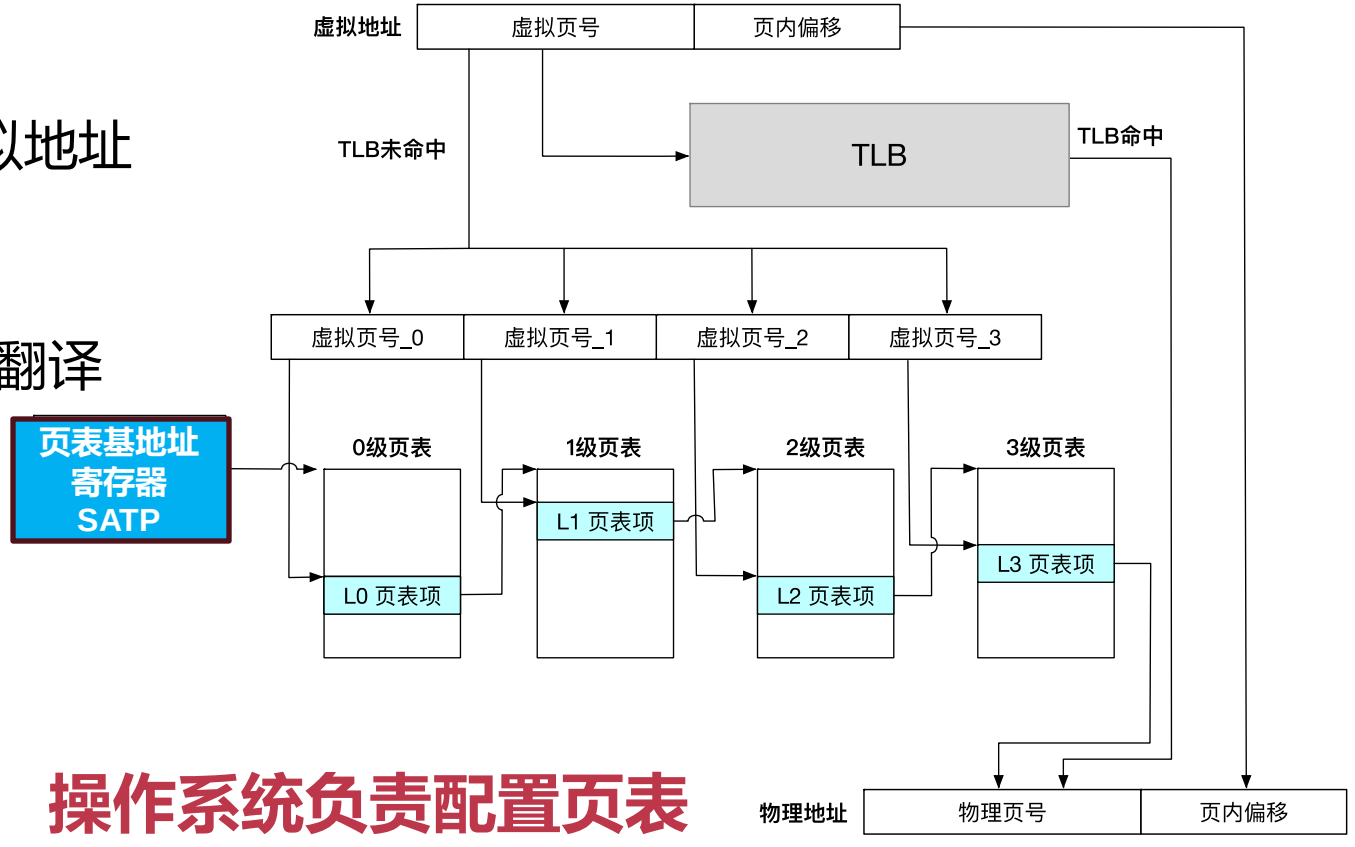


回顾：应用程序仅使用虚拟地址

应用程序使用的虚拟地址

硬件根据**页表**自动翻译

物理地址



操作系统负责配置页表

问：VMA和页表是否冗余?

- **OS通过VMA记录应用程序能够访问的虚拟地址**
  - 未映射的区域没有对应的VMA结构
- **OS通过配置页表控制应用程序能够访问的虚拟地址**
  - 未分配的虚拟地址没有对应的页表

- **那么，VMA是否冗余?**

操作系统何时为应用程序填写页表

- **两种方式**
  - 立即映射：每个虚拟页都对应了一个物理内存页
  - 延迟映射：有些虚拟页不对应任何物理内存页
    - 对应的数据在磁盘上
    - 没有对应的数据（初始化为0）

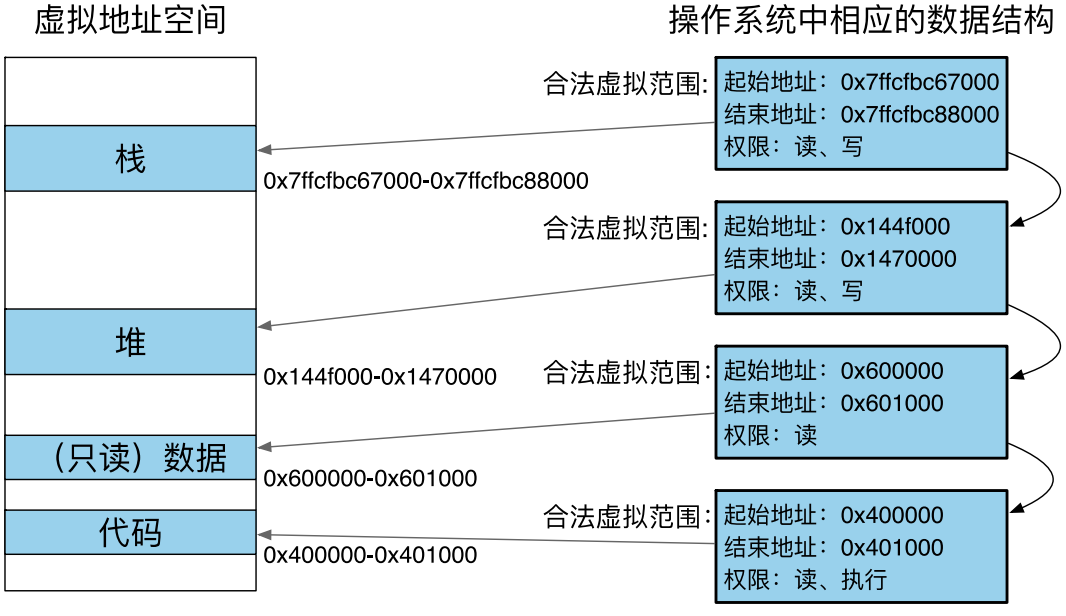
延迟映射/按需调页（DEMAND PAGING）

缺页异常（Page Fault）

- **CPU的控制流转移**
  - CPU陷入内核，找到并运行相应的异常处理函数（handler）
  - OS提前注册缺页异常处理函数
- **x86_64**
  - 异常号 #PF （13），错误地址存放在CR2寄存器
- **AARCH64**
  - 触发（通用的）同步异常（8）
  - 根据ESR信息判断是否缺页，错误地址存放在FAR_EL1
- **RISC-V**
  - 通过cause寄存器的值来识别page fault
  - 12: page fault caused by an instruction fetch
  - 13: page fault caused by a read
  - 15: page fault caused by a write
  - 错误地址存放在STVAL，SEPC指向哪?

如何判断缺页异常的合法性?

- OS记录应用程序已分配的虚拟内存端（VMA）
  - 不落在VMA区域，则为非法



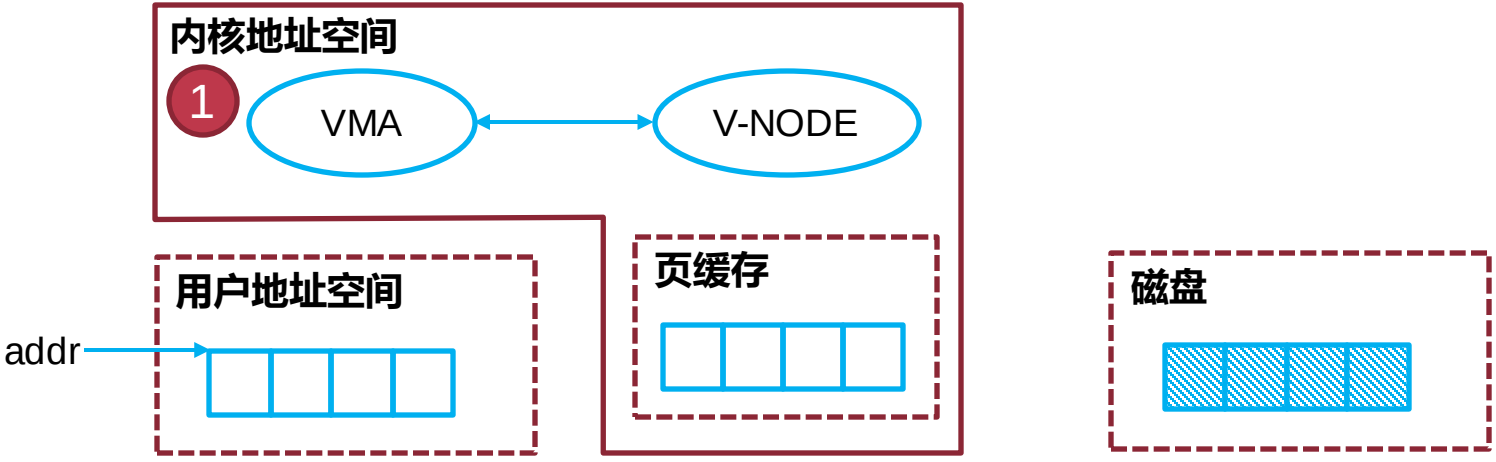
按需分配考虑的权衡

- **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（Locality）
  - 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

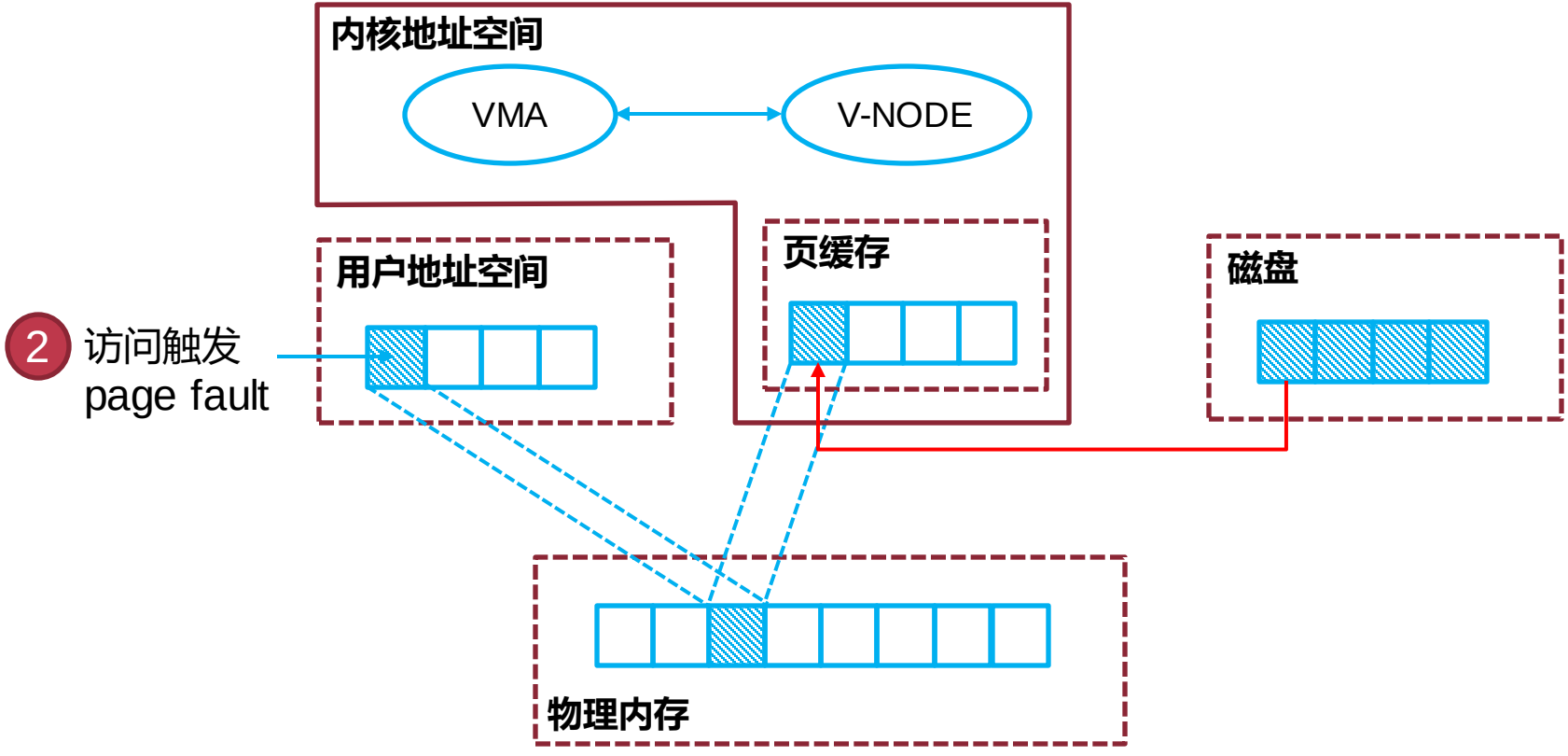
思考

- **导致缺页异常的三种可能**
  1. 访问非法虚拟地址
  2. 按需分配（尚未分配真正的物理页）
  3. 内存页数据被换出到磁盘上
  - **问：后两种都是合法的缺页异常，如何区分?**

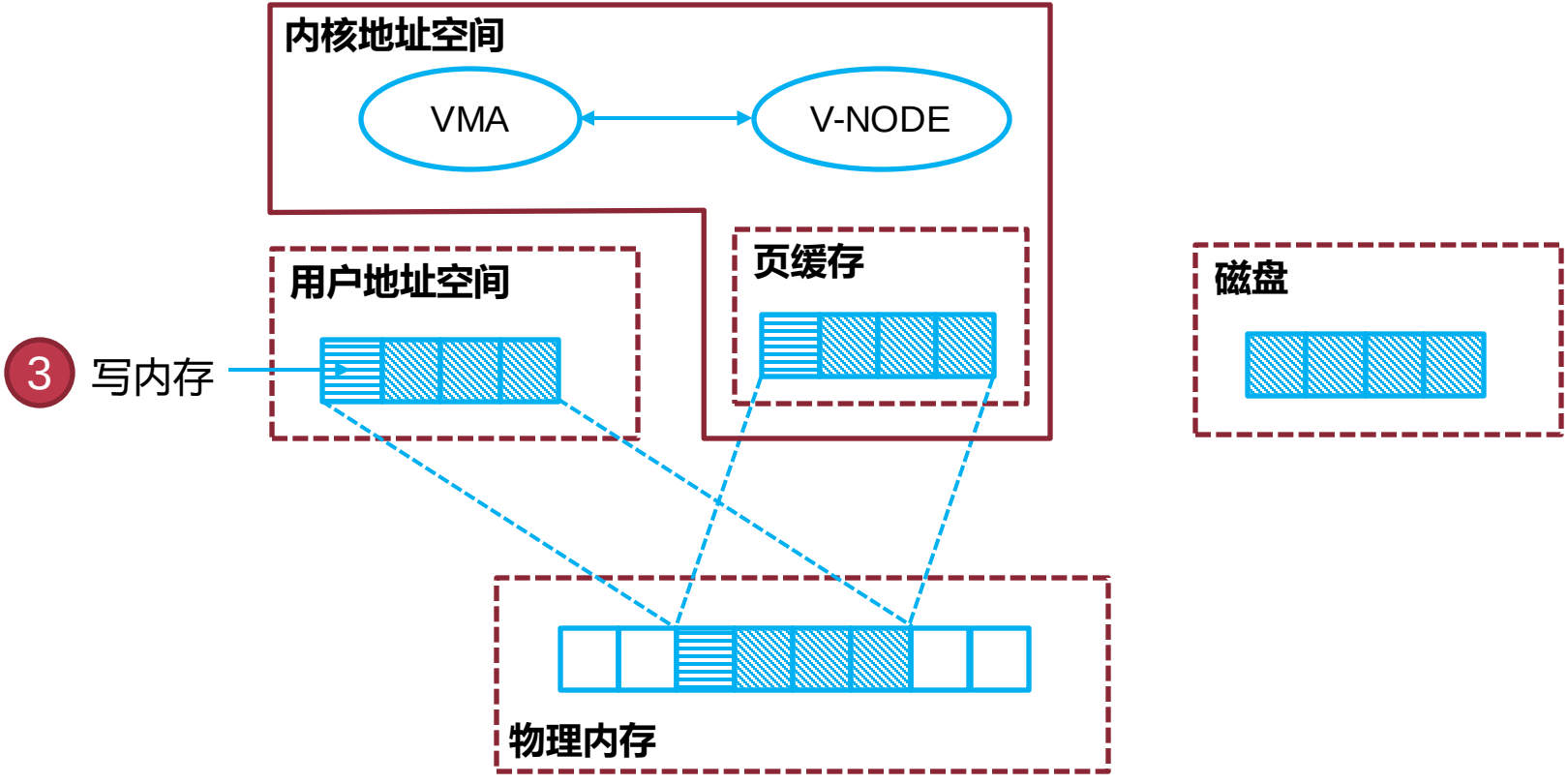
MMAP



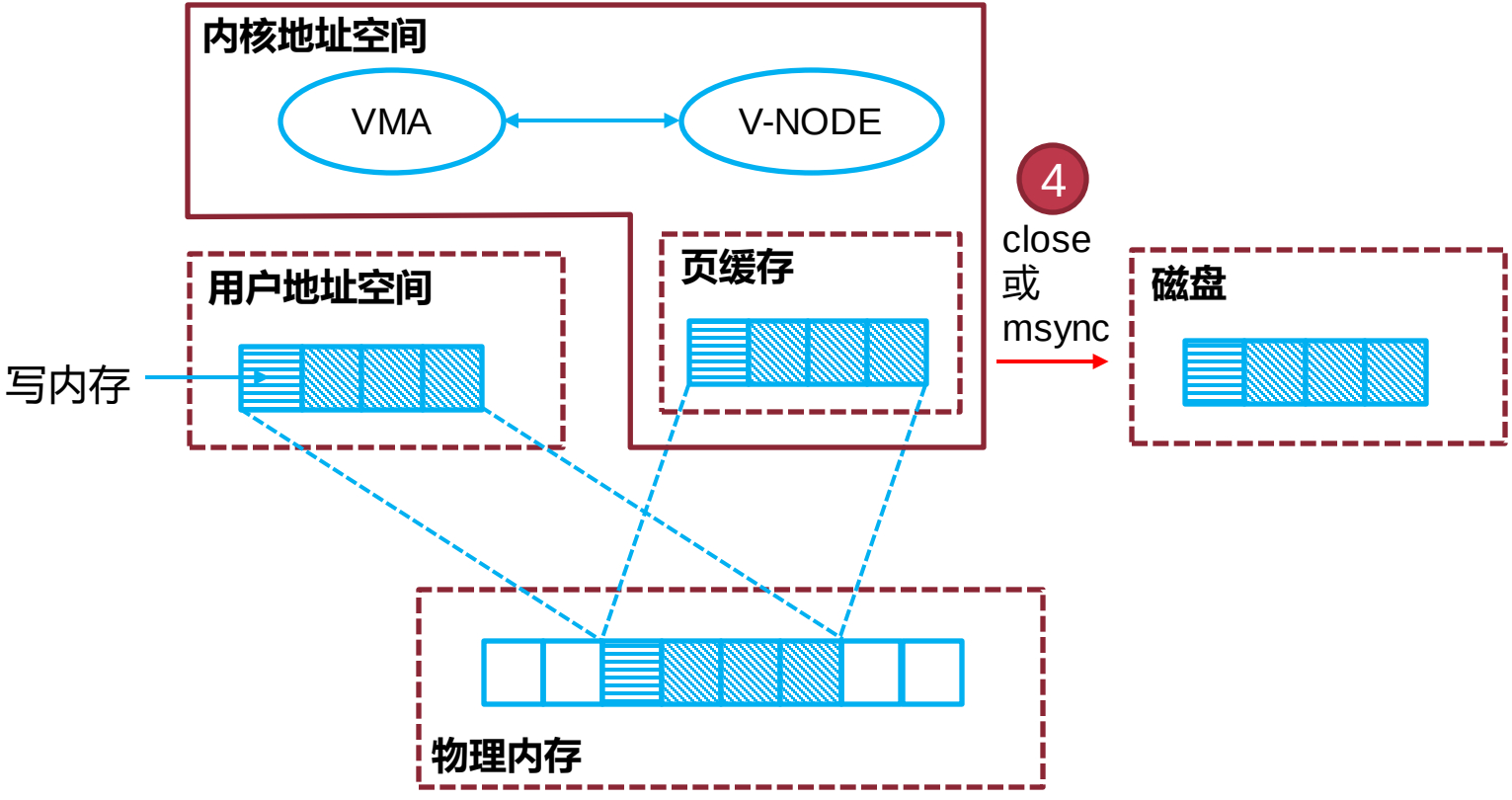
MMAP



MMAP



MMAP



MMAP 的优化

- **Prefault**
  - 每次 page fault 载入连续的多个页，减少 fault 次数
- **MAP_POPULATE**
  - 通过一个包含 MAP_POPULATE 的 flags 参数，可以在调用 mmap 时就预取所有的页，此后访问不会 fault

思考

- **问：mmap匿名映射与文件映射的区别是什么?**
  - 没有backup file，内存的初始值从哪里来?
- **问：如果OS仅采用立即映射，还需要VMA么?**
  - VMA记录的信息和页表记录的信息有何不同?
- **问：demand-paging是否可通过网络来实现?**
  - 如果都通过网络，本地是否还需要磁盘?

小知识：OS可向应用提供灵活的内存管理API

- **madvise**
  - int madvise(void *addr, size_t length, int advice)
  - 将用户态的一些语义信息发给内核以便于优化
    - 例如：将madvise和mmap搭配，在使用数据前告诉内核这一段数据需要使用，**建议**OS提前分配物理页，减少缺页异常开销
- **mprotect**
  - int mprotect(void *addr, size_t len, int prot);
  - 改变一段内存的权限
    - 例如：JIT动态生成的二进制代码，需将内存由"可写"改"为可执行"

可将物理内存看做是虚拟地址空间的Cache

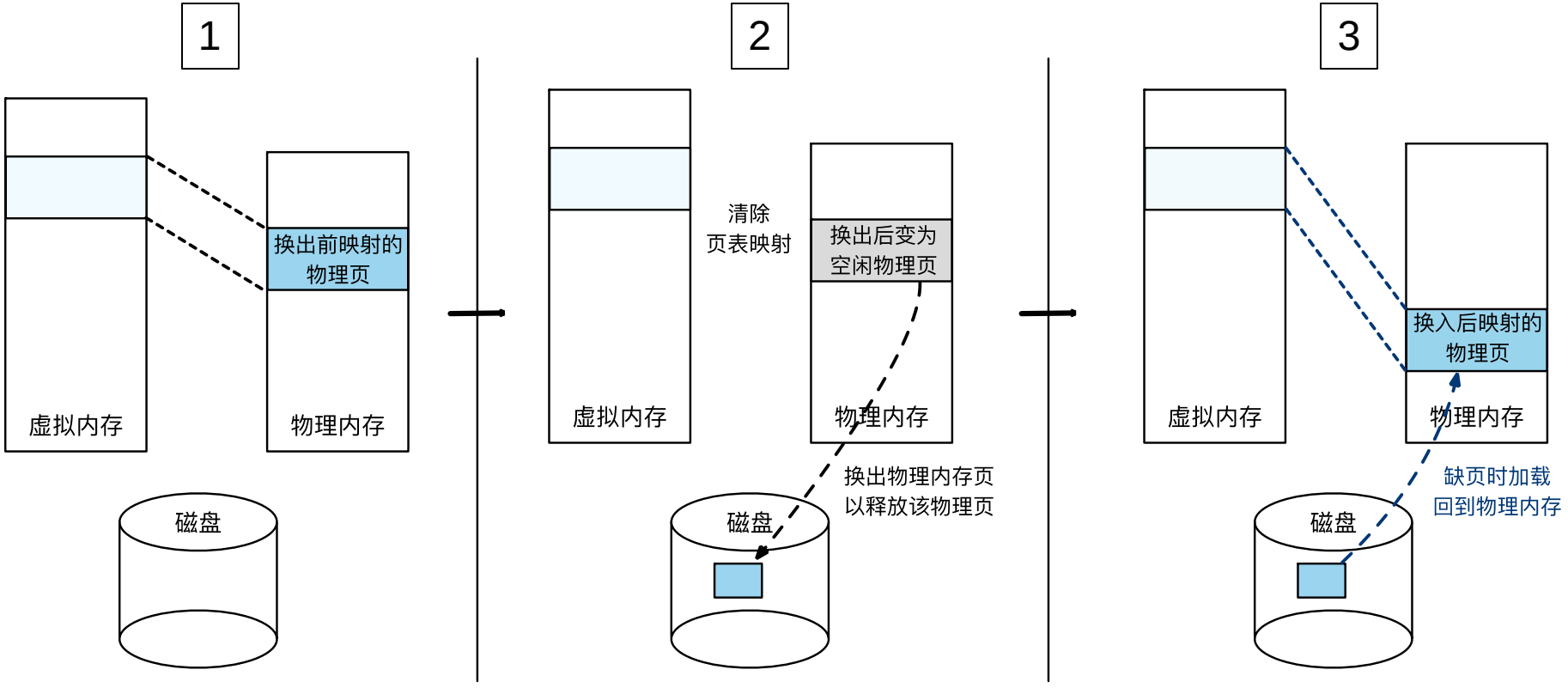
- **情景1:**
  - 两个应用程序各自需要使用 3GB 的物理内存
  - 机器实际上总共只有 4GB 的物理内存

- **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）

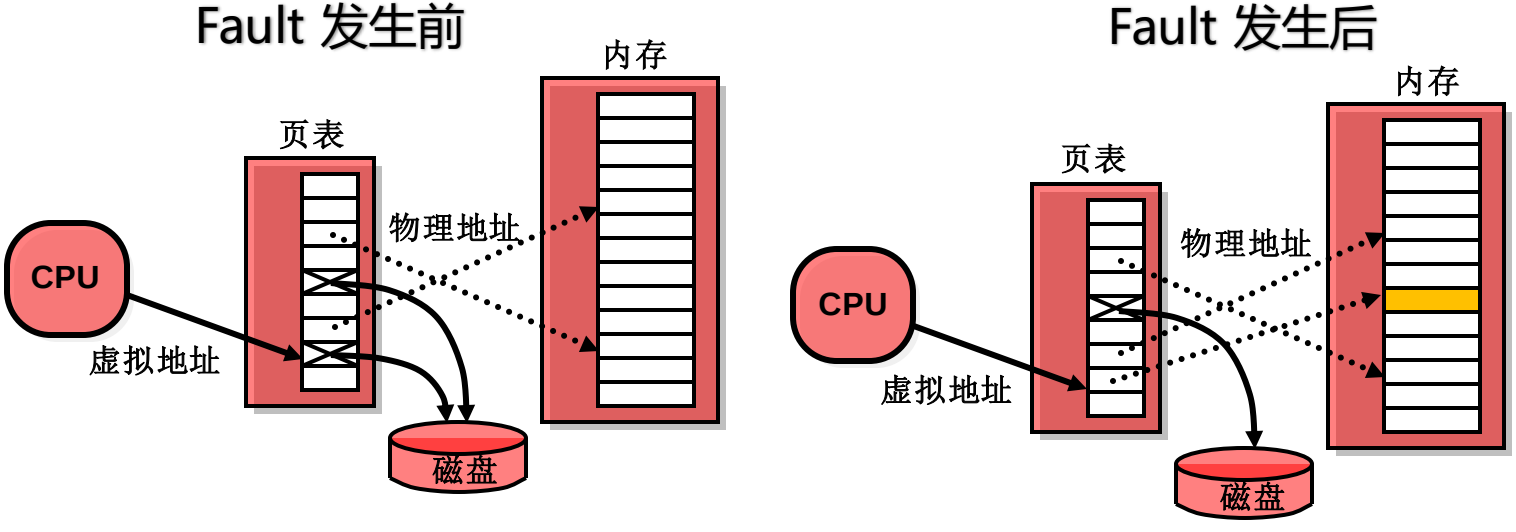
- **换页的基本思想**
  - 用磁盘作为物理内存的补充，且对上层应用透明
  - 应用对虚拟内存的使用，不受物理内存大小限制
- **如何实现**
  - 磁盘上划分专门的Swap分区，或专门的Swap文件
  - 在处理缺页异常时，触发物理内存页的换入换出

简单的换页示例


1
虚拟内存
换出前映射的
物理页
物理内存
磁盘
2
虚拟内存
清除
页表映射
换出后变为
空闲物理页
物理内存
换出物理内存页
以释放该物理页
磁盘
3
虚拟内存
换入后映射的
物理页
物理内存
缺页时加载
回到物理内存
磁盘

Page Faults

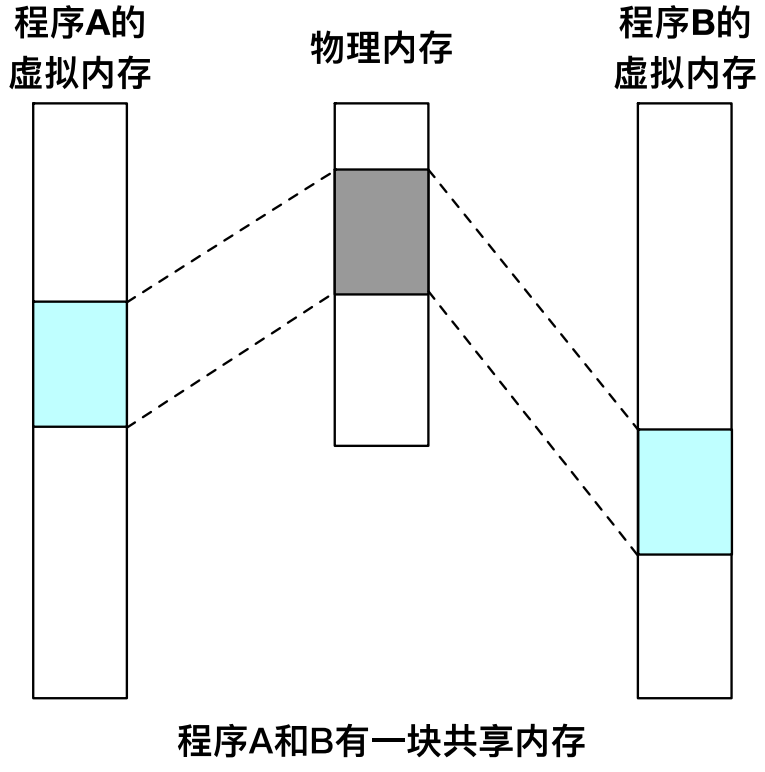
- 换页（Swapping）
- 页面分配（Paging）
- 页面按需分配（Demand paging）



OS内存管理中的更多机制

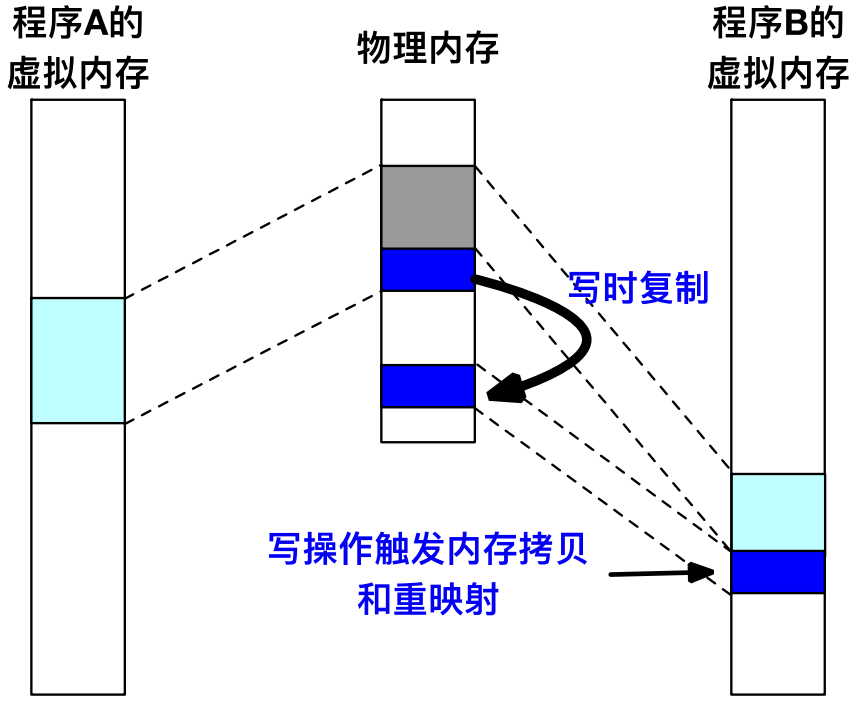
共享内存

- **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

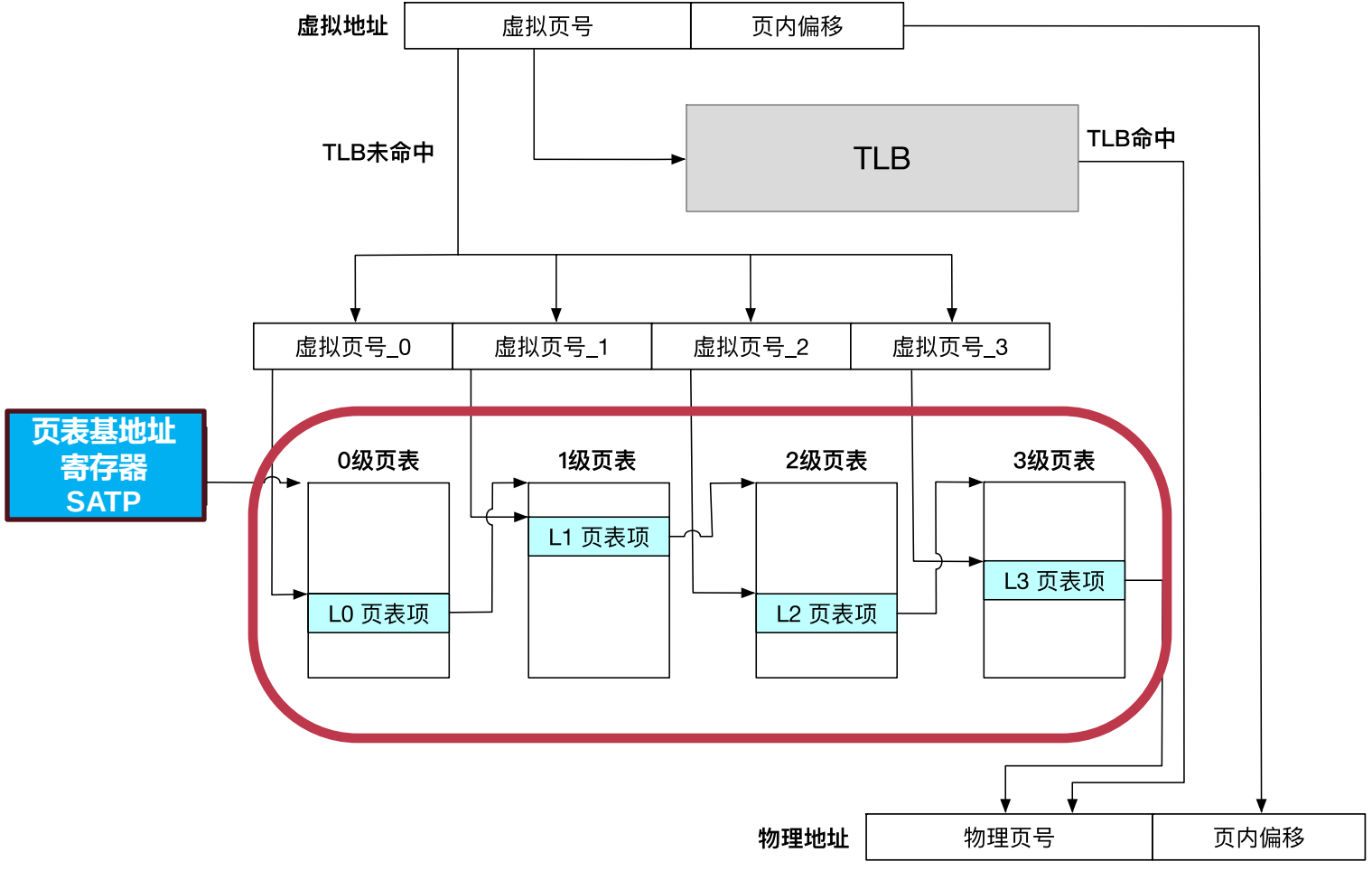
内存压缩

- **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，压缩的优点和缺点有哪些？
- **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

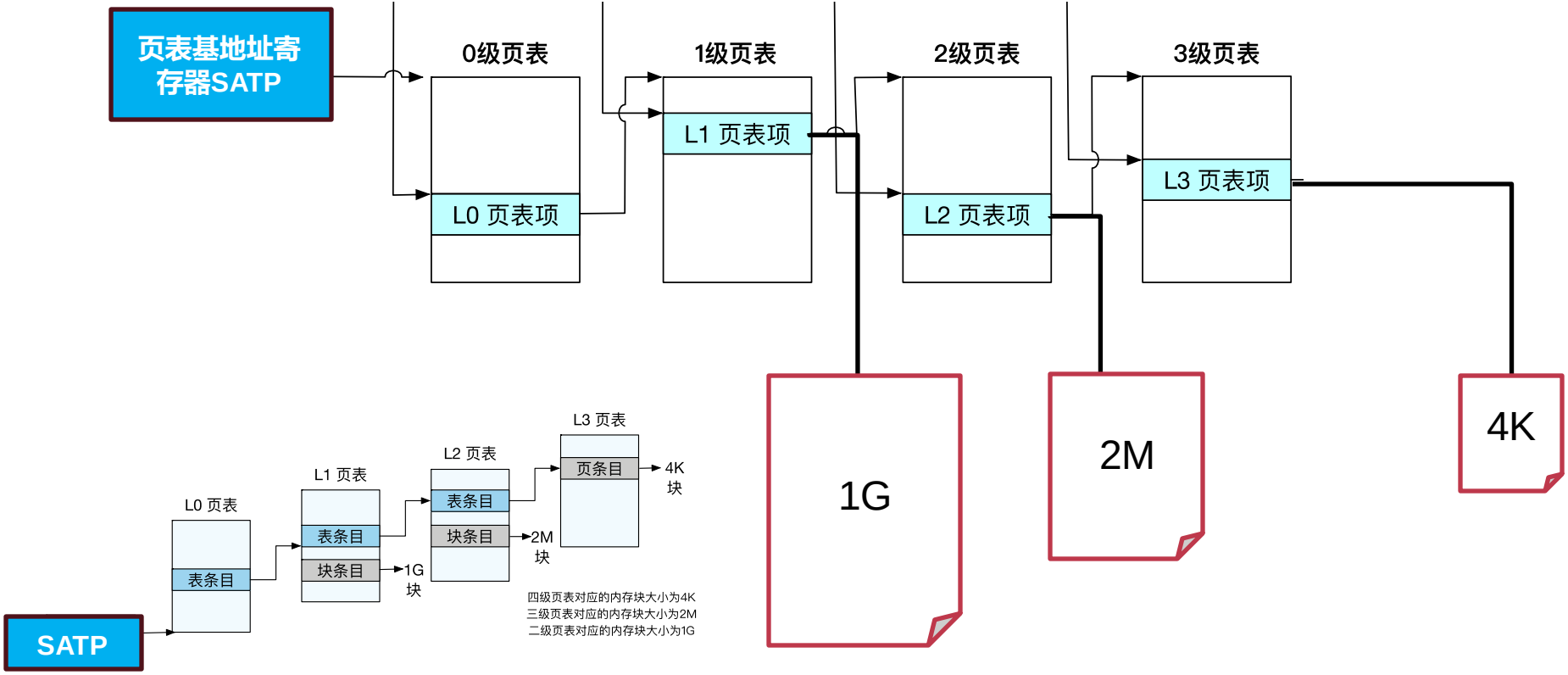
大页：再次回顾4级页表


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
SATP
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SATP
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
1G
2M
4K
L0 页表
表条目
L1 页表
表条目
块条目
1G 块
L2 页表
表条目
块条目
2M 块
L3 页表
页条目
4K 块
SATP
四级页表对应的内存块大小为4K
三级页表对应的内存块大小为2M
二级页表对应的内存块大小为1G

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

RISC-V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）

RV64支持页面大小

- **RV64**
  - SV39 的 512GiB 地址空间划分为 2^9个 1 GiB 大小的大页。
  - 每个大页被进一步划分为 2^9 个巨页，这些巨页大小为 2 MiB
  - 每个巨页再进一步分为 2^9个 4 KiB 大小的基页

思考

- **什么情况适合使用大页?**
- **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？哪些场景适合?
- **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
    - 更多可参考 Software Fault Isolation

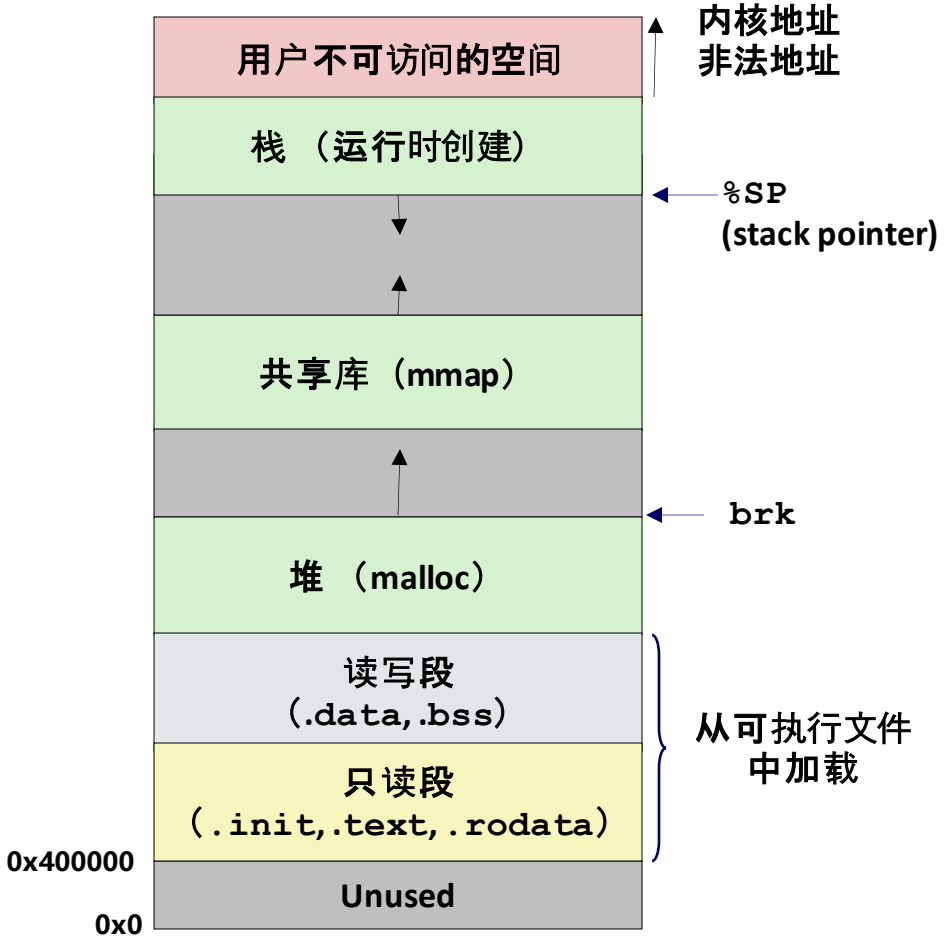
小结：虚拟内存机制的优势

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
- **更强的隔离与更细粒度的权限控制**
  - 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

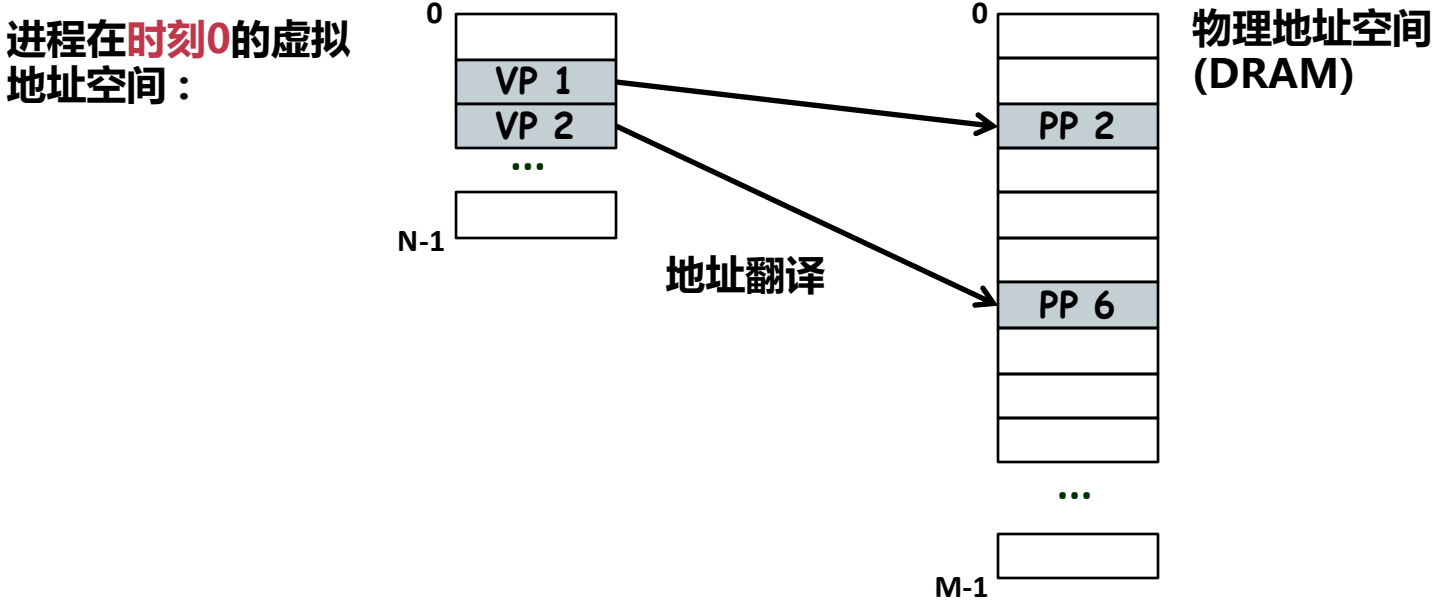
每个进程拥有独立的虚拟地址空间

- **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



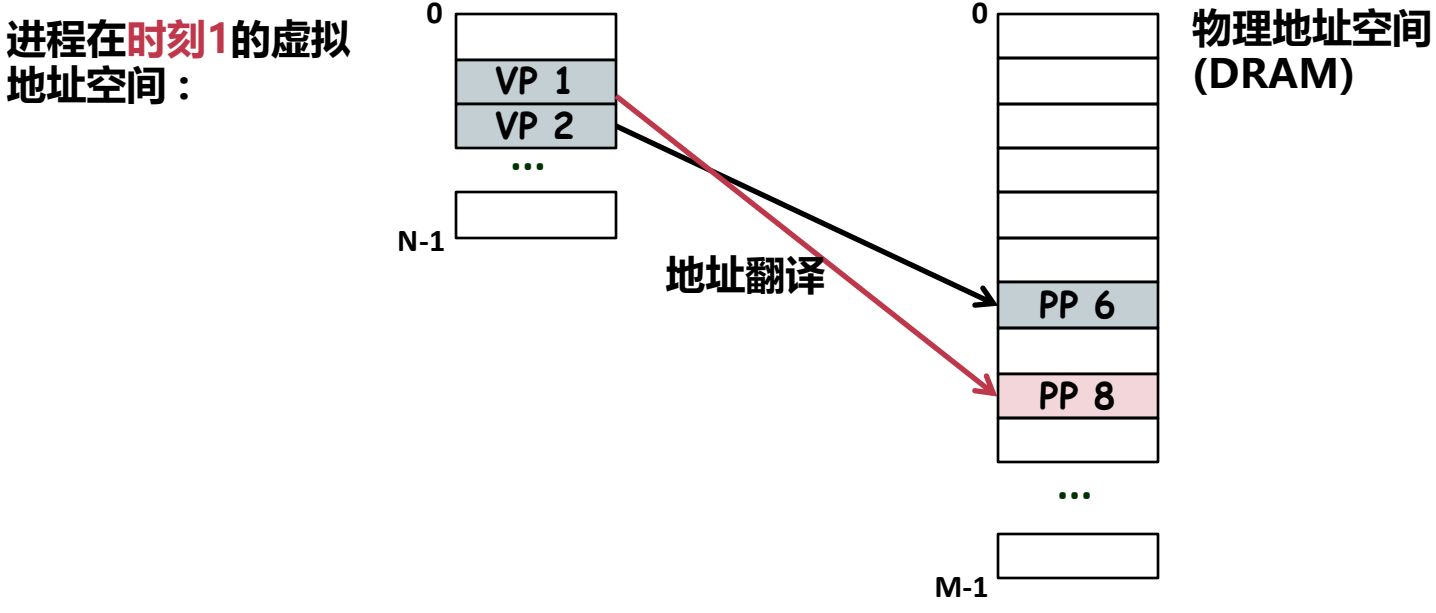
灵活的虚拟内存-物理内存映射

- 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



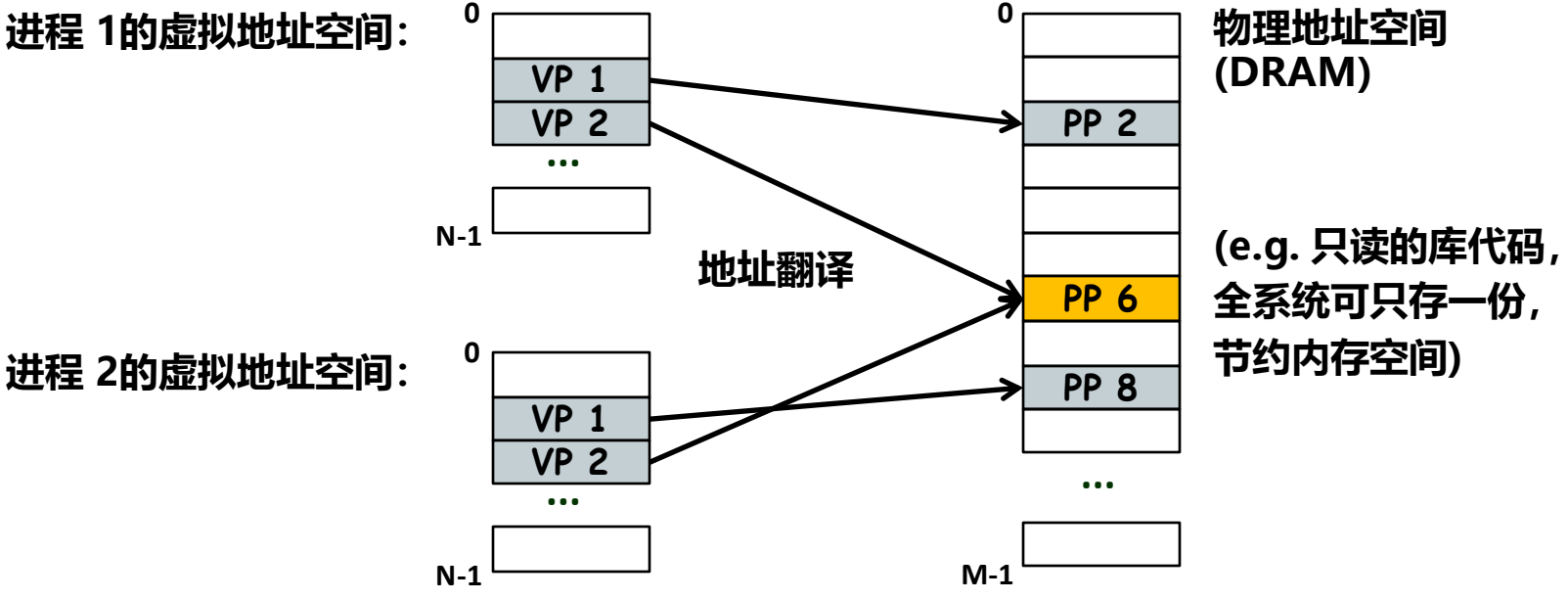
灵活的虚拟内存-物理内存映射

- 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



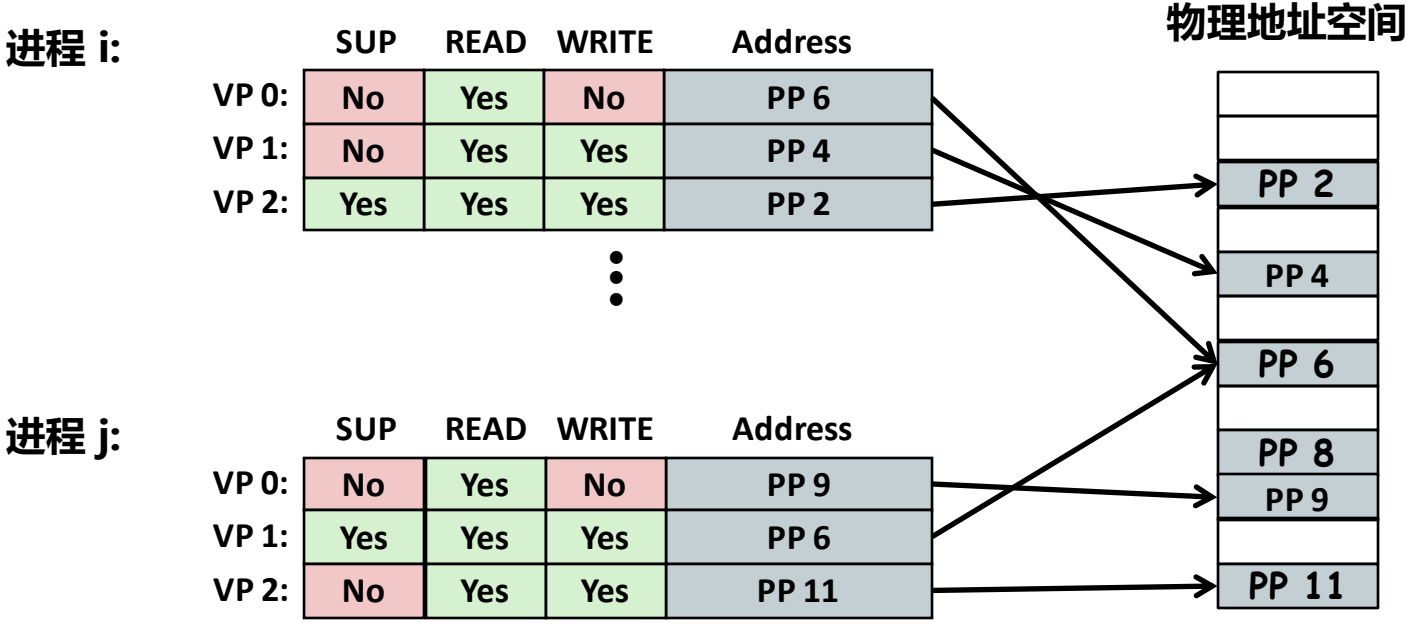
可在不同进程之间共享内存

- 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



基于虚拟内存实现内存保护

- **缺页异常处理函数会在映射前检查权限位**
  - 如果违反权限，会向进程发送 SIGSEGV (segmentation fault) 信号