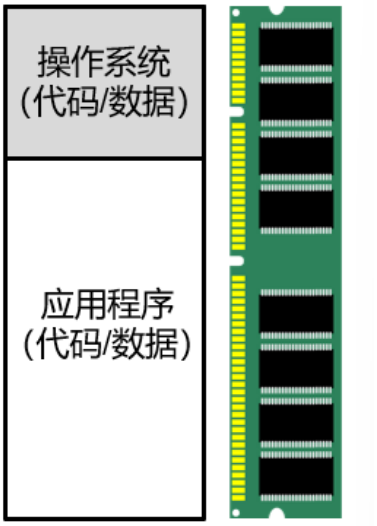
Lecture5 Virtual Memroy

发展的历史

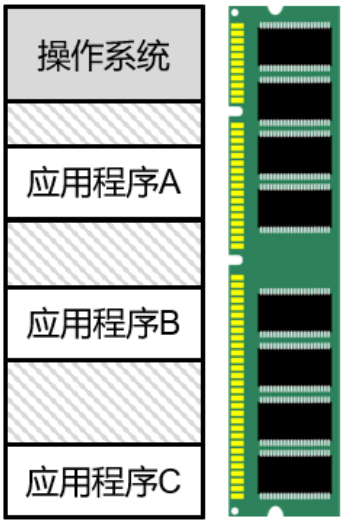
最早期的计算机系统

- **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

- **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）



IBM 360内存隔离：Protection Key

> 无需虚拟内存也可以实现隔离

1. 内存被划分为大小为2KB的内存块
2. 每个内存块有一个4-bit的key，保存在寄存器中
3. 每个进程对应于一个key
4. 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

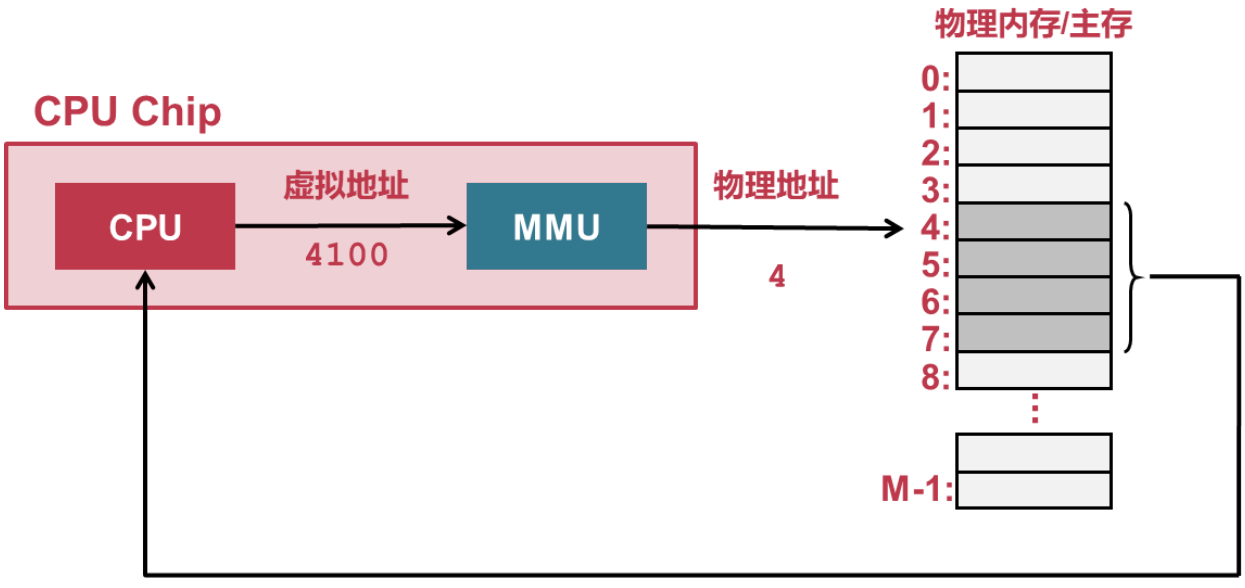
使用物理地址的缺点

1. 干扰性：一个应用会因其它应用的加载而受到影响
2. 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
3. 安全性：一个应用可以通过自身的内存地址，猜测其他应用的位置

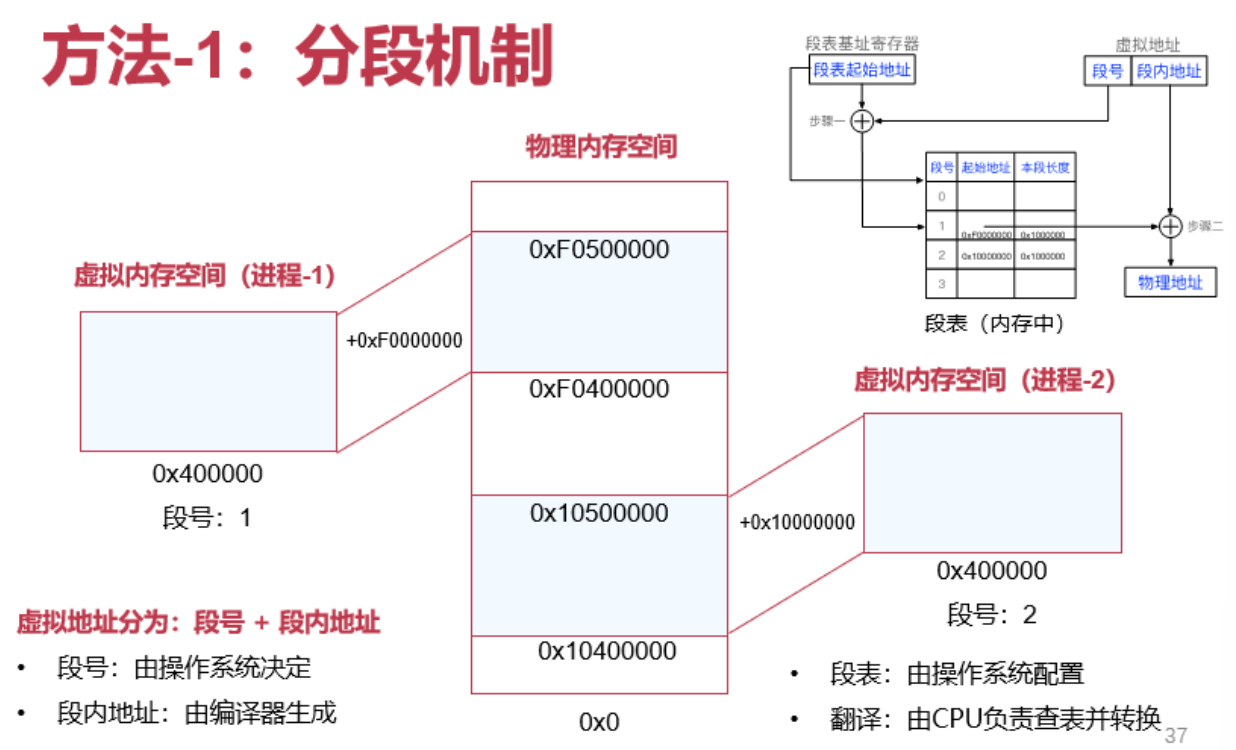
虚拟地址 VS. 物理地址

- **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



分段机制



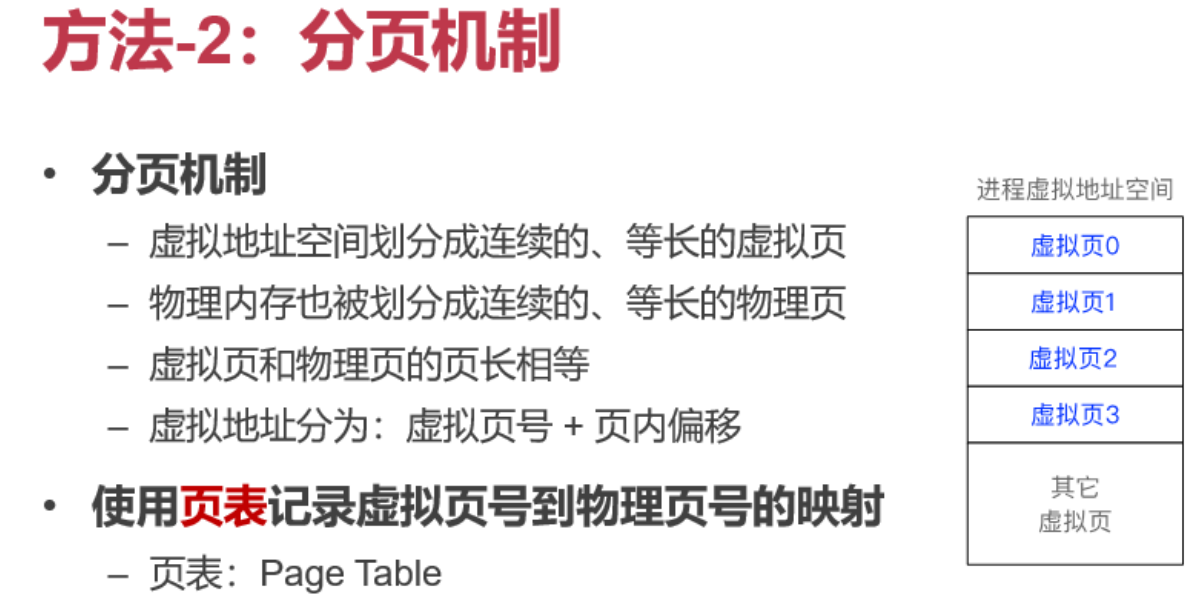
> 相比于分页机制，最大的不同是每一段可以有不同的长度

- 分段机制的问题
  1. 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
  2. 存在问题：内存利用率

     外部碎片&内部碎片

分页机制



- 分页机制的特点
    1. 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
    2. 主存资源易于管理，利用率更高

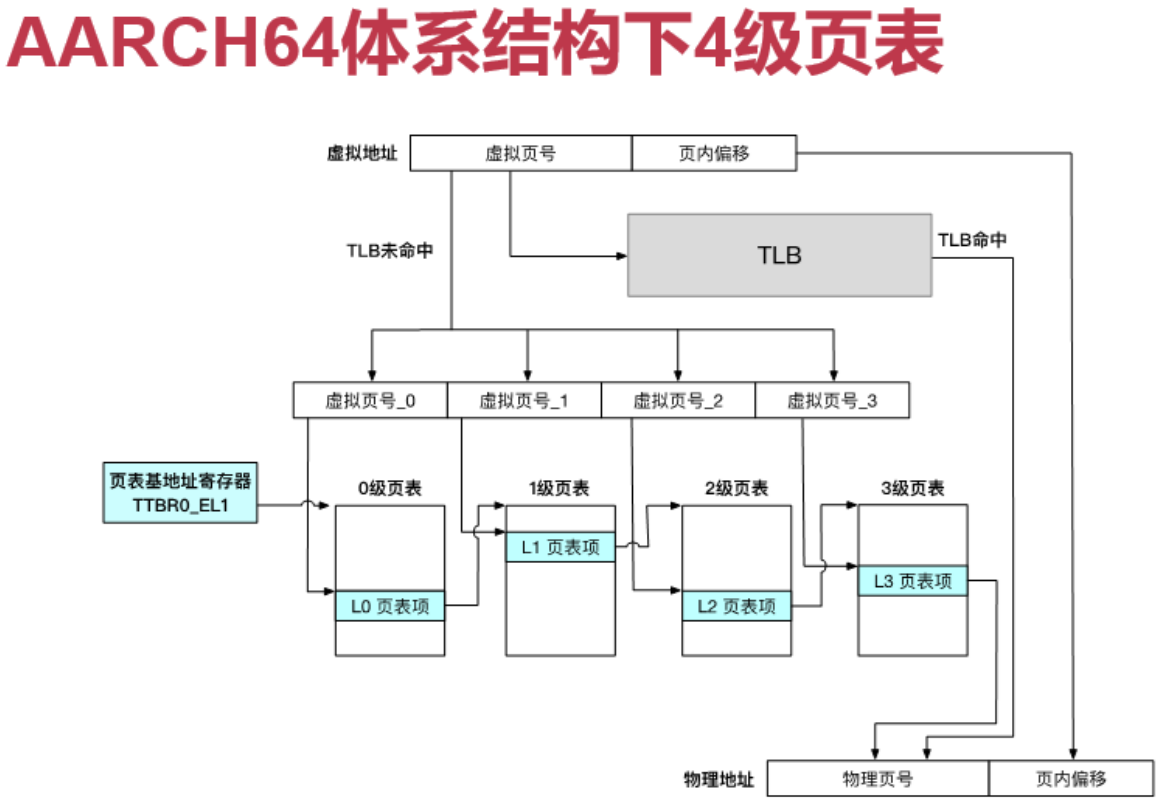
       按照固定页大小分配物理内存，能大大降低外部碎片和内部碎片

ARM64的页表格式

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

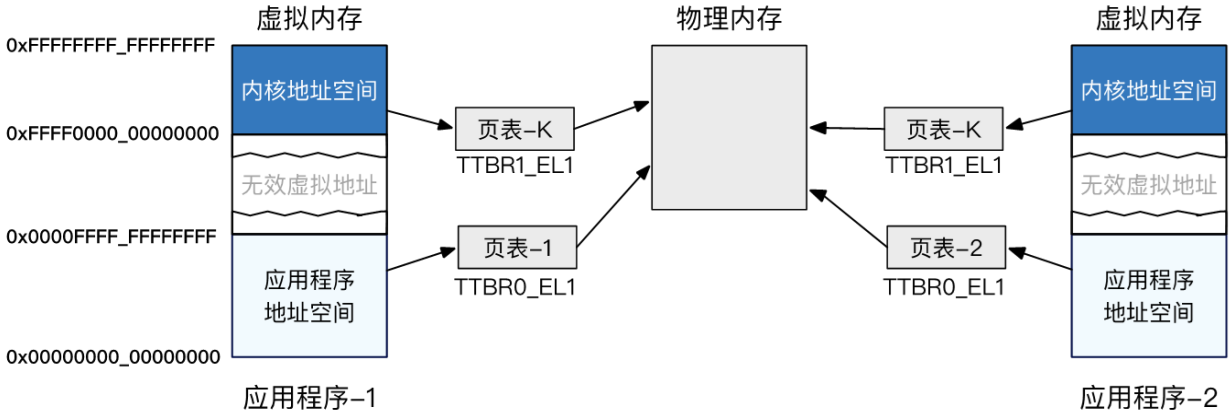
> 若某级页表的某条目为空，那么该条目对应的下一级页表便无需存在



> 64位虚拟地址的分布：
>
> 1. 16 (前面不用的位，全0/全1，应用程序用0，内核用)
> 2. 9 * 4 （四级页表，每一级的页表索引大小为9-bit）
> 3. 12 （页内偏移）

页表基地址寄存器

> AARCHC64：两个寄存器TTBR0_EL1&TTBR1_EL2
>
> X86-64：一个寄存器CR3(Control Register 3)



TLB：页表的Cache

> TLB: Translation Lookaside Buffer

地址翻译的加速器

- **TLB位于CPU内部，是页表的缓存**

  > 缓存 `虚拟页号->物理页号` 的映射关系

> **有限数目**的TLB缓存项

- 在地址翻译的过程中，MMU首先查询TLB

TLB清空：TLB Flush

- TLB使用虚拟地址索引，当**OS切换页表的时候要全部刷掉**

  > 1. 切换进程的时候一定要刷
  > 2. user与kernel切换的时候，在现有的知识框架下不需要刷，因为地址空间没有发生变化，还是一个进程(但是最近有一个meltdown的问题，以后再讲)

- 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，系统调用不用切换

- x86-64只有唯一的CR3，内核映射到应用页表的高地址，避免syscall时TLB清空的开销

如何降低TLB清空导致的开销

- **新的硬件特性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会将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，不再需要刷掉TLB，提高性能
  - 修改页表映射后，仍需刷掉TLB（为什么？）

> 先去匹配VA，再去找ASID
>
> Q：修改页表映射后，仍需刷掉TLB，这是为什么?
>
> A：因为修改的是页表翻译（va->pa）的过程，而不是页表的内容

TLB与多核

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

虚拟内存：段和VMA

> 内存比CPU cache更新的状况：TLB的flush&DMA/RDMA

> Q：DMA的non-cachable memory拷到新的memory位置后没有inconsistency吗？（没搞明白，下次上课去问）
>
> A：

Segmentation Fault

> 访问虚拟内存中未被分配的地址

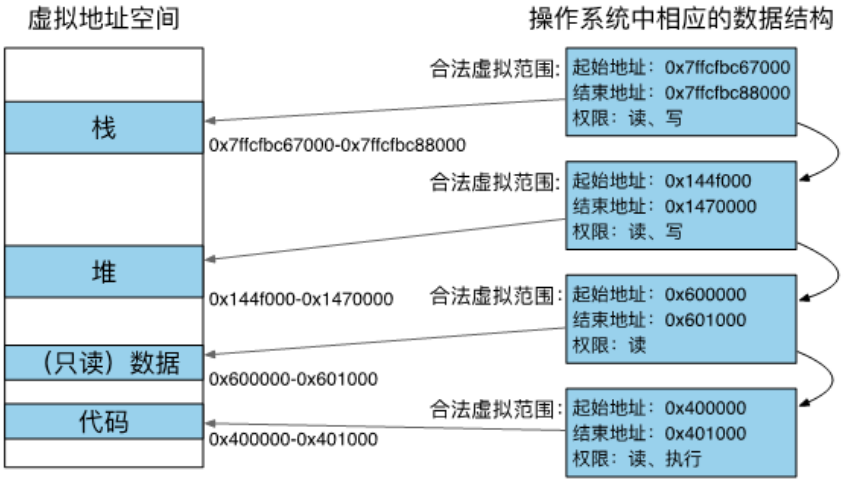
- **OS采用段来管理虚拟地址**
  - 段内连续，段与段之间非连续
  - 合法虚拟地址段：代码、数据、堆、栈
  - 非法虚拟地址段：未映射
    - 一旦访问，则触发segfault
- **思考：为什么要用段来管理?**

> Q：能不能遍历访问虚拟内存中所有的地址空间?
>
> A：因为不是所有的虚拟地址空间都是可以访问的，需要先malloc一块空间（VMA会检查是否访问了未被分配的虚拟地址空间）
>
> **malloc之后并不会真的分配物理地址，唯一的变化存在VMA里**

VMA是如何添加的

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



- 途径1：OS在创建应用程序的时候分配

> R/O数据&代码&栈

- 途径2：应用程序主动向OS发出请求

> 1. brk()：扩大/缩小堆区域
> 2. 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

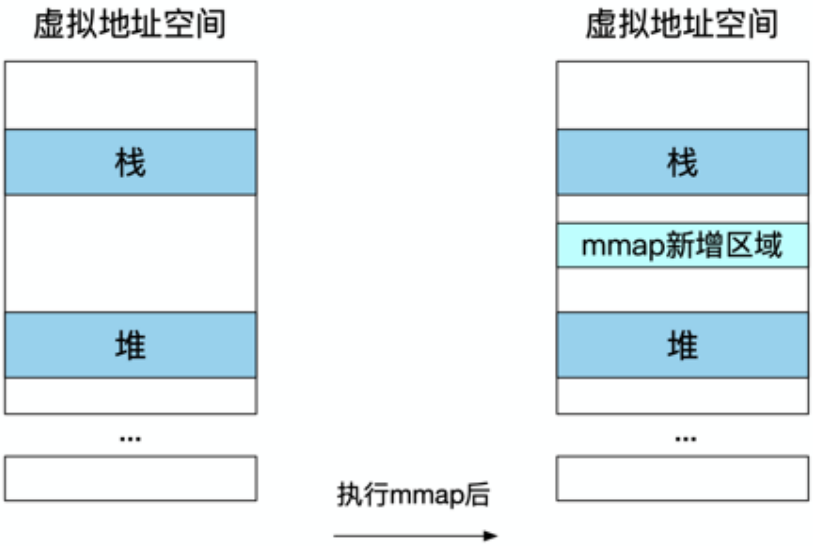
mmap: 分配一段虚拟内存区域

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, // 起始地址
    size_t length, // 长度
    int prot, // 权限，例如PROT_READ
    int flags, // 映射的标志，例如MAP_PRIVATE
    int fd, // -1 或者是有效fd
    off_t offset) // 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

执行mmap后，VMA的变化



- **执行 `mmap` 后并没有真正的分配物理内存，这时所有的变化都存在VMA中**

> Q：mmap匿名映射与文件映射的区别是什么？
>
> A：匿名映射可以没有文件
>
> Q：如果OS仅采用立即映射，还需要VMA吗？
>
> A：不需要了，页表采用的也是立即映射，此时VMA相当于一个索引加速查找，不是很必要
>
> Q：demand-paging是否可通过网络来实现？
>
> A：可以，现在网卡都可能比硬盘要快

VMA和页表是否冗余?

> Q：VMA是否冗余？（in quesition）
>
> A：不冗余，VMA分配的空间不一定有页表的映射，而被分配的页表一定有物理内存的映射，简单理解就是VMA的范围更大，页表的范围更小

延迟映射/按需调页

> 1. **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）
> 2. **立即映射**：每个虚拟页都对应了一个物理内存页

缺页异常

- **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

> Q：如何判断page fault的合法性?
>
> A：OS记录应用程序已分配的虚拟内存端（VMA），不落在VMA区域则为违法

- 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

> Q：如何区分后两种缺页异常?
>
> A：在PTE上设置一个bit

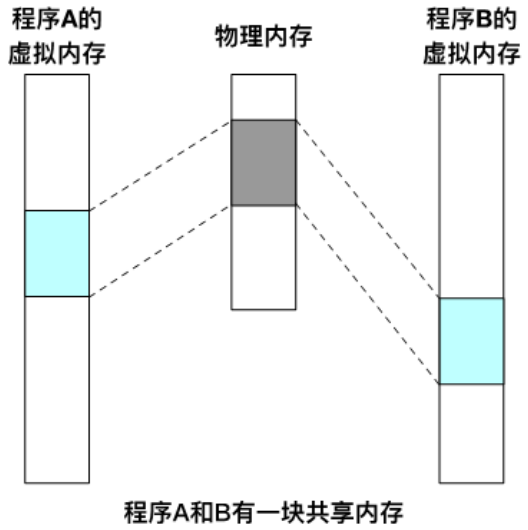
按需分配考虑的权衡

- **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

虚拟内存的扩展功能

共享内存

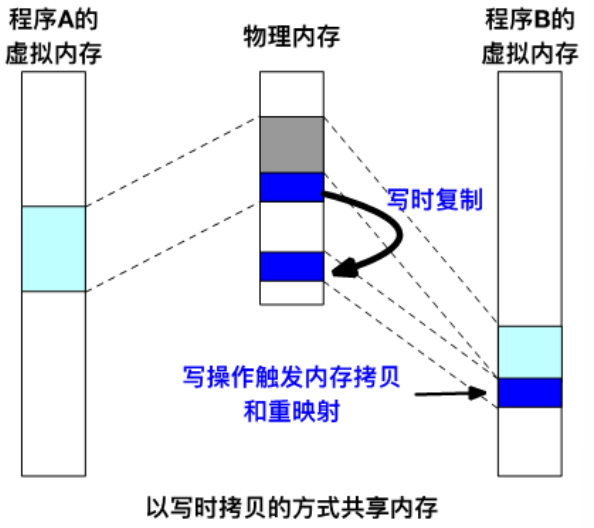
- **基本功能**
  - 节约内存，如共享库
  - 进程通信，传递数据



程序A和B有一块共享内存

写时拷贝

- **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

> 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

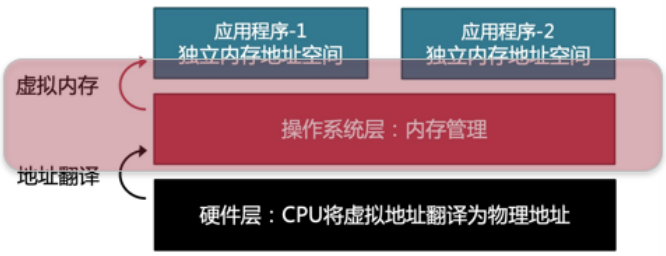
大页的利弊

- 好处
  1. 减少TLB缓存项的使用，提高命中率
  2. 减少页表的级数，提升遍历页表的效率
- 缺点
  1. 未使用整个大页而造成物理内存资源浪费
  2. 增加管理内存的复杂度

Summary

虚拟内存小结

- **填写页表的策略**
  - 立即映射
  - 延迟映射
- **延迟映射实现原理**
  - 硬件基础：缺页异常
  - 软件设计：VMA数据结构
- **虚拟内存的扩展功能**



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

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

> 1. 大数据的场景，更少的TLB Miss
> 2. 只使用物理内存，不适合多用户，适合单用户
>
> 编译器插桩是指由编译器通过插入检查代码（检查访问地址范围是否合法）的方式隔离

课外补充

1. `ttbr0_el1`和`ttbr1_el1`中存放的是**物理地址**
2. PPN中存放的是**物理地址**，否则会无穷递归
3. 内核的作用是管理页表，**MMU是硬件，会自动去做地址转换**。不可能每一次访问内存都需要通过内核态去转换地址。所以内核只是配页表，在过程中会有转换但不是说所有转换都是由内核去完成。