虚拟内存

操作系统管理页表映射

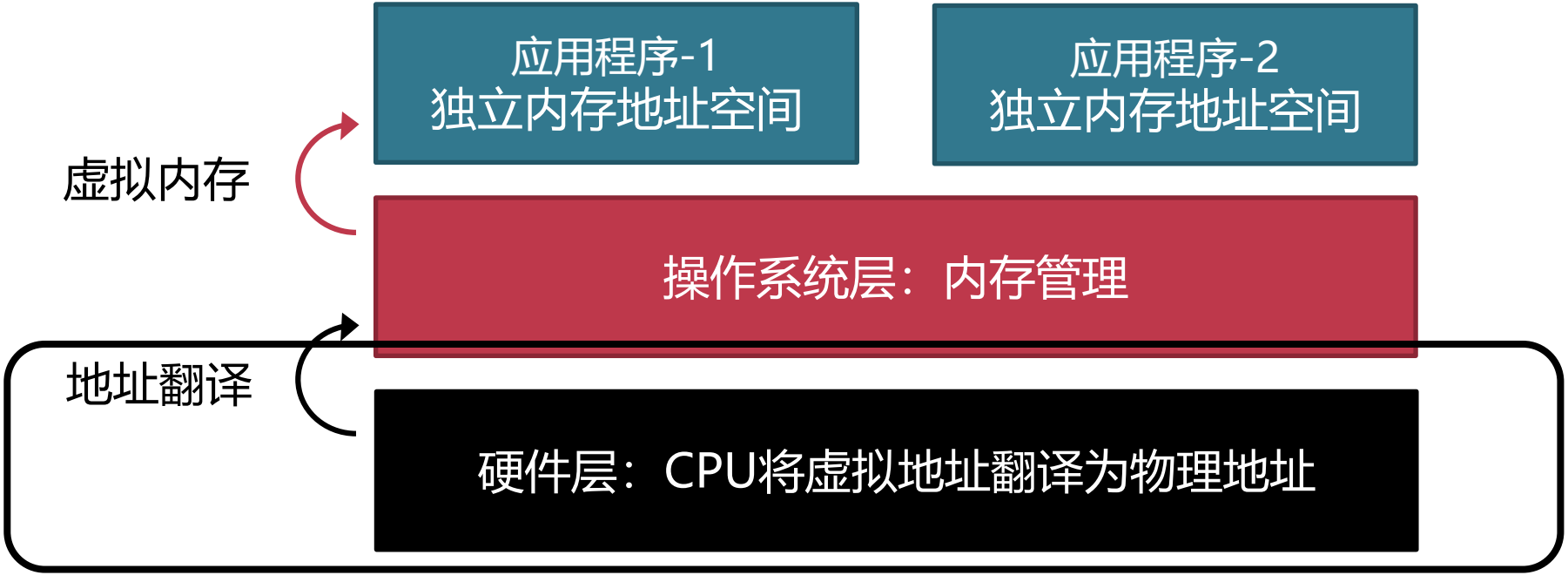
上海交通大学

https://www.sjtu.edu.cn

版权声明

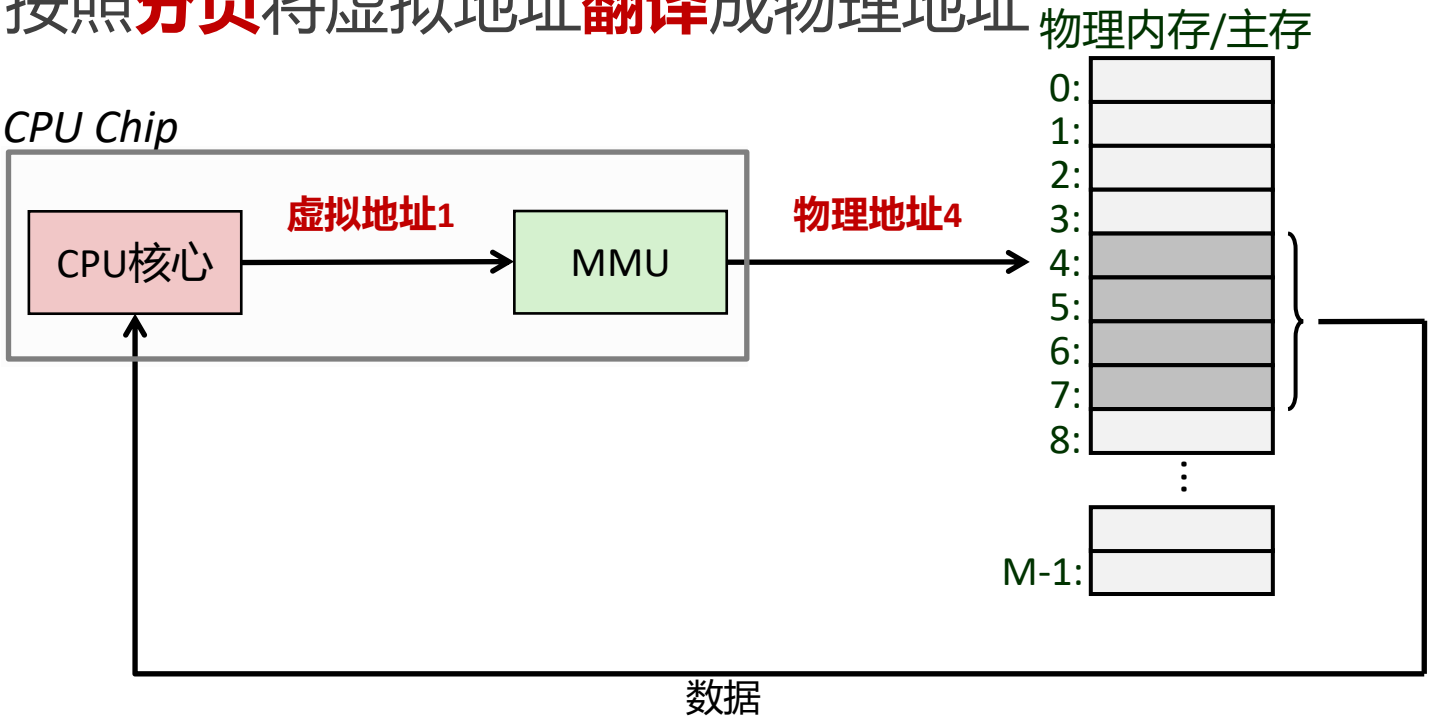
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 资料来自上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

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

回顾：地址翻译

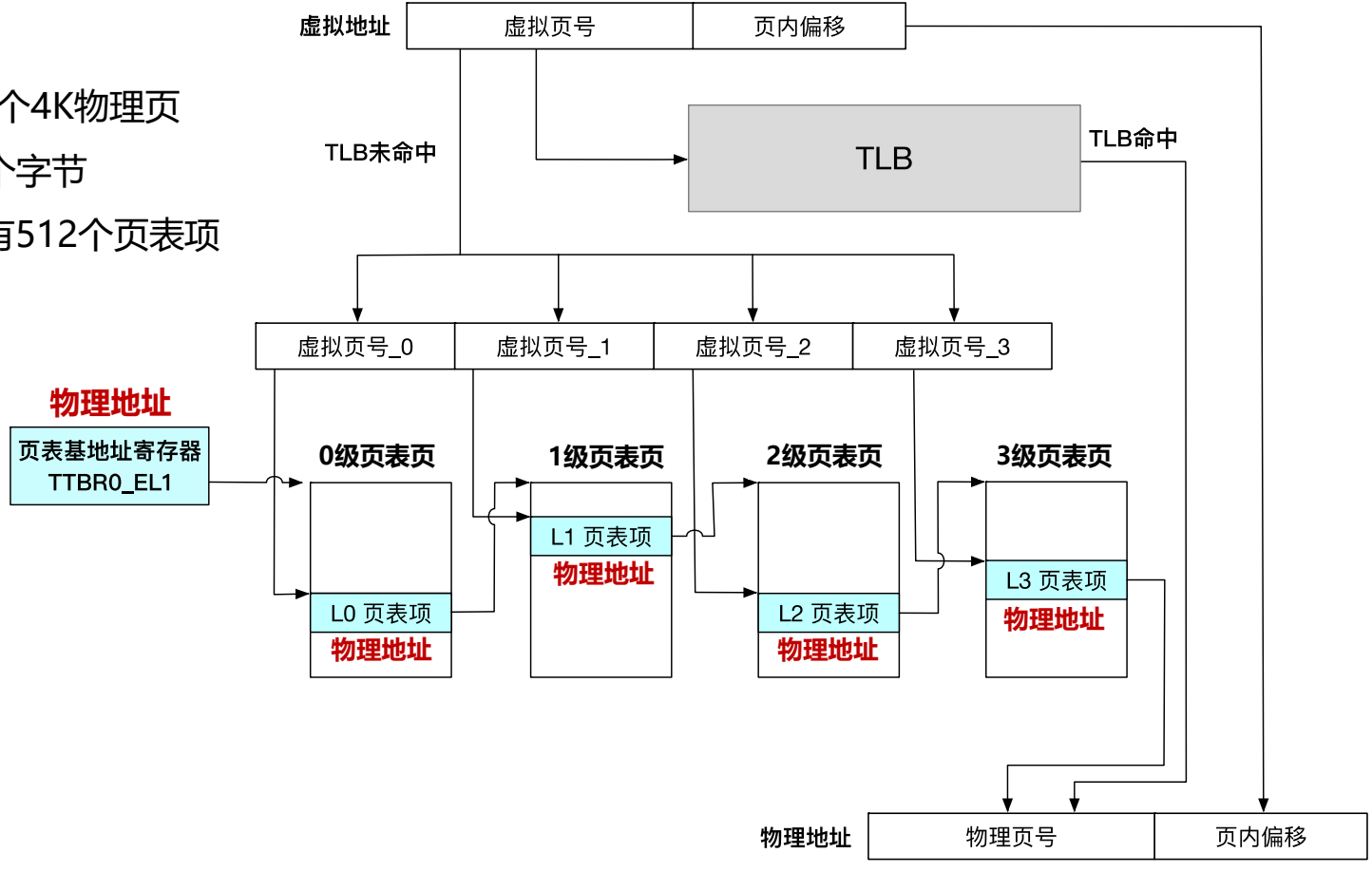
- **Memory Management Unit**
  - 按照**分页**将虚拟地址**翻译**成物理地址



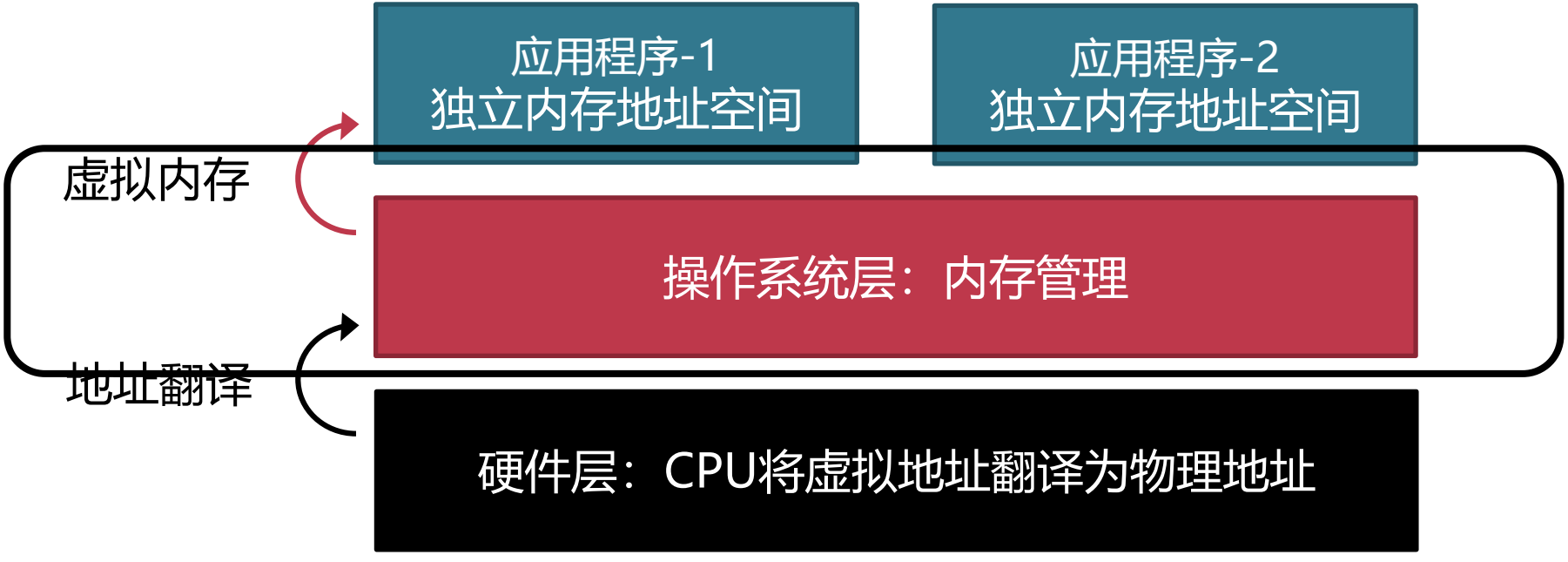
回顾：AARCH64体系结构下4级页表

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

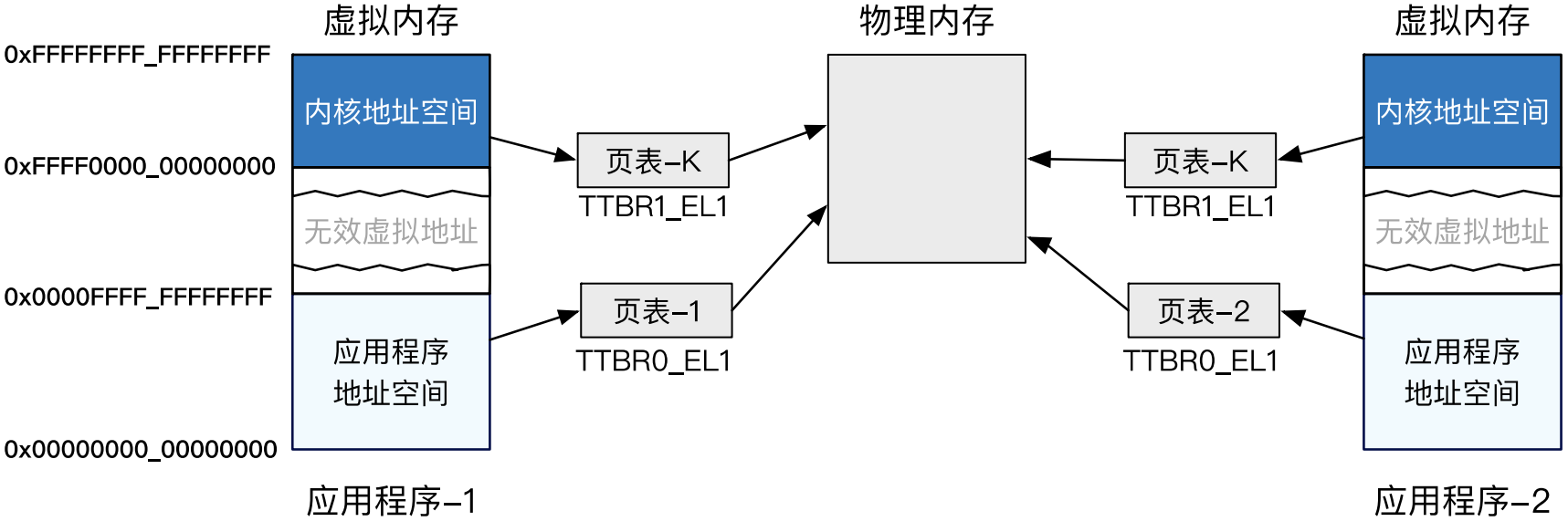


内容提纲


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

设置页表映射是操作系统的职责

- TTBR0_EL1和TTBR1_EL1分别指向内核和应用页表



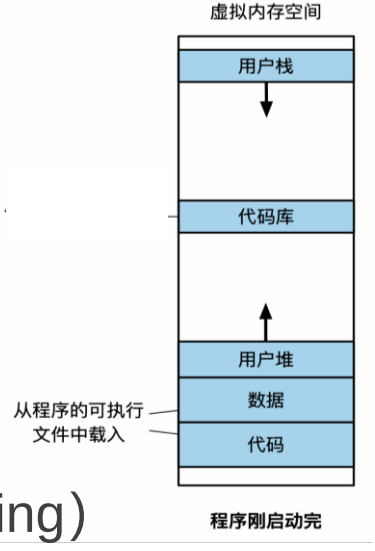
何时设置页表映射

- **操作系统自己使用的页表**
  - 在启动时填写
  - 映射全部物理内存
    - 虚拟地址 = 物理地址 + 固定偏移（**直接映射，Direct Mapping**）
    - 思考：为什么需要直接映射?

- **应用进程的页表**
  - 何时设置?

何时填写进程页表：立即映射

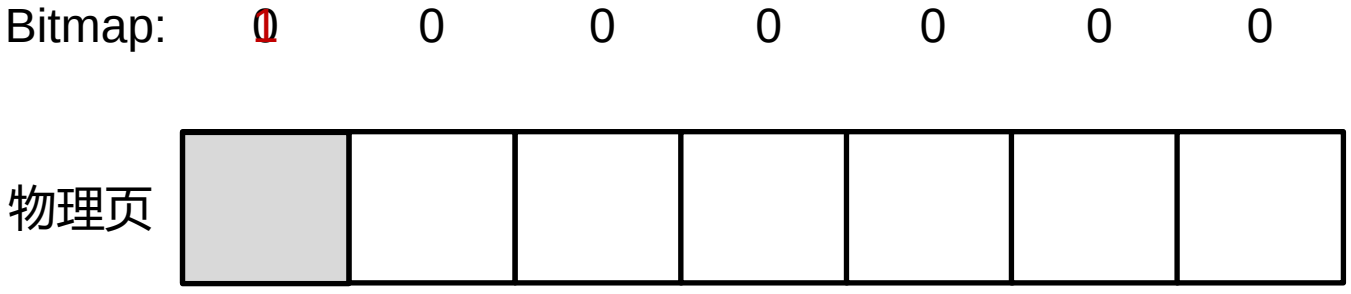
- **创建进程时，OS按照虚拟内存区域填写进程页表**
  - 例如，代码段和数据段
  - 具体步骤：
    - 步骤-1: 分配物理页（alloc_page）
    - 步骤-2: 把应用代码/数据从磁盘加载到物理页中
    - 步骤-3: 添加虚拟页到物理页的映射（add_mapping）
    - 步骤-4: 未加载完毕，回到步骤-1



分配物理页的简单实现

- **操作系统用位图记录物理页是否空闲**
  - 0：空闲；1：已分配

alloc_page()接口的实现



操作系统如何填写进程页表

- 操作系统在进程结构体中保存页表基地址

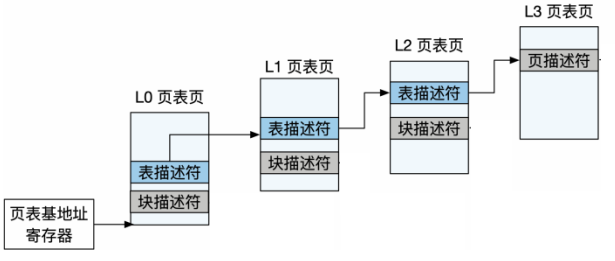
```
1  struct process {
2    // 上下文
3    struct context *ctx;
4    // 页表基地址（物理地址）
5    u64 pgtbl;
6    ...
7  };
8
9  void add_mapping(struct process *, u64 va, u64 pa);
10 void delete_mapping(struct process *, u64 va);
```

填写进程页表的代码实现

```c
// 在进程页表中添加虚拟地址 va 到物理地址 pa 的映射
void add_mapping(struct process *p, u64 va, u64 pa)
{
  u64 *pgtbl_page;
  u32 index;

  // 获取 0 级页表页的起始地址：即为页表基地址
  // 每个页表页占据 4K，包含 512 个页表项
  pgtbl_page = (u64 *)paddr_to_vaddr(p->pgtbl);

  // 获取虚拟地址在 0 级页表页中的页表项索引
  index = L0_INDEX(va);
  // 获取 1 级页表页的起始地址
  pgtbl_page = get_next_pgtbl_page(pgtbl_page, index);

  // 获取虚拟地址在 1 级页表页中的页表项索引
  index = L1_INDEX(va);
  // 获取 2 级页表页的起始地址
  pgtbl_page = get_next_pgtbl_page(pgtbl_page, index);

  // 获取虚拟地址在 2 级页表页中的页表项索引
  index = L2_INDEX(va);
  // 获取 3 级页表页的起始地址
  pgtbl_page = get_next_pgtbl_page(pgtbl_page, index);

  // 获取虚拟地址在 3 级页表页中的页表项索引
  index = L3_INDEX(va);
  // 在 3 级页表页的页表项中填写物理地址 paddr
  pgtbl_page[index] = pa | some_permission;
}
```

问：paddr_to_vaddr如何实现?

```c
struct process {
  // 上下文
  struct context *ctx;
  // 页表基地址（物理地址）
  u64 pgtbl;
  ...
};
```



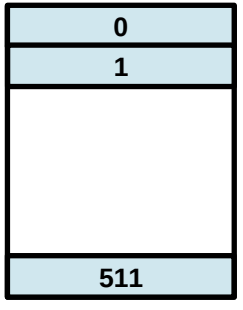
填写进程页表的代码实现

```
1  u64 get_next_pgtbl_page(u64 *pgtbl, u32 index)
2  {
3    u64 pgtbl_entry;
4
5    pgtbl_entry = pgtbl[index];
6
7    if (pgtbl_entry == 0) {
8      // 如果没有相应的页表页（页表空洞），则分配页表页
9      pgtbl_entry = alloc_pgtbl_page();
10     pgtbl_page[index] = pgtbl_entry | some_permssion;
11   }
12
13   // 页表项中存储的是物理地址，而操作系统在运行时使用虚拟地址
14   return paddr_to_vaddr(pgtbl_entry);
15 }
```

问：参数pgtbl是虚拟地址还是物理地址？

问：alloc_pgtbl_page如何实现？

页表页：512个页表项组成的数组



表描述符：指向下一级页表

0级、1级、2级页表项

页描述符：指向4K页

3级页表项

立即映射的弊端

- **立即映射是一种操作系统可以选择的页表填写策略**
  - 在初始化进程虚拟地址空间时，直接在进程页表中添加各虚拟内存区域的映射

- **潜在弊端**
  - 以关卡/副本类型游戏加载为例：只玩1关，加载1000关
    - **物理内存资源浪费**
    - **非必要时延**

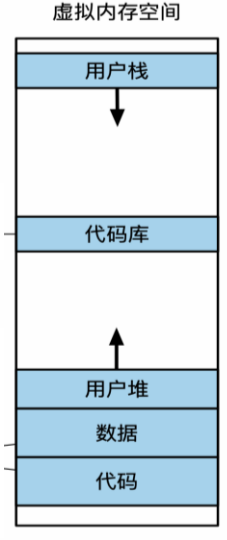
操作系统填写进程页表的另一种策略

延迟映射/按需映射

延迟映射

- **解决立即映射弊端的直观想法**
  - 操作系统**按进程实际需要分配**物理页和填写页表，避免分配的物理页实际不被用到的情况

- **主要思路：解耦虚拟内存分配与物理内存分配**
  - 先记录下为进程分配的虚拟内存区域
  - 当进程实际访问某个虚拟页时，CPU 会触发缺页异常
  - 操作系统在缺页异常处理函数中添加映射



回顾：缺页异常

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

- V（Valid）：当MMU查询页表项时，若V=0，则触发缺页异常

操作系统需要区分合法/非法缺页异常

```
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)
```

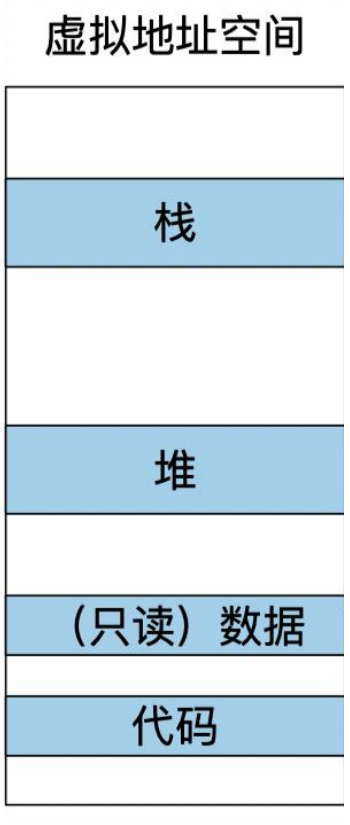
如果改变第5行会怎么样？比如：

char *p = 0x40000;

char *p = 0x400000000;

操作系统记录为进程分配的虚拟内存区域

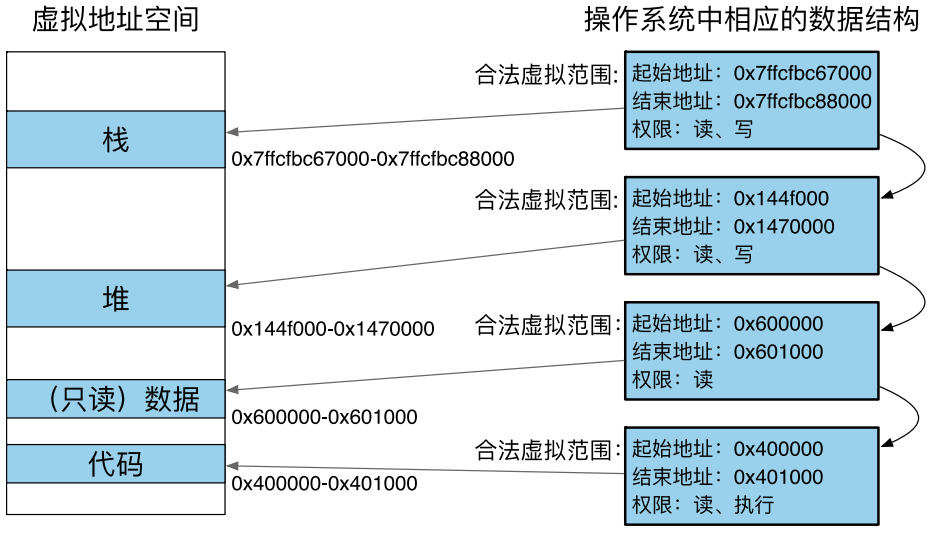
- **虚拟地址空间**
  - 若干非连续的**虚拟内存区域**
    - 每个虚拟内存域中的虚拟地址都是进程可用的
      - 相同的访问权限
    - 例如：代码、数据、堆、栈
    - 非法虚拟地址，访问触发CPU异常
      - 操作系统会报segfault



小知识：segfault （软件行为） vs. pagefault （硬件异常）

合法虚拟地址信息的记录方式

- **记录进程已分配的虚拟内存区域**
  - 在Linux中对应 vm_area_struct（VMA）结构体
  - 在ChCore-Lab中对应vmregion（vmr）结构体



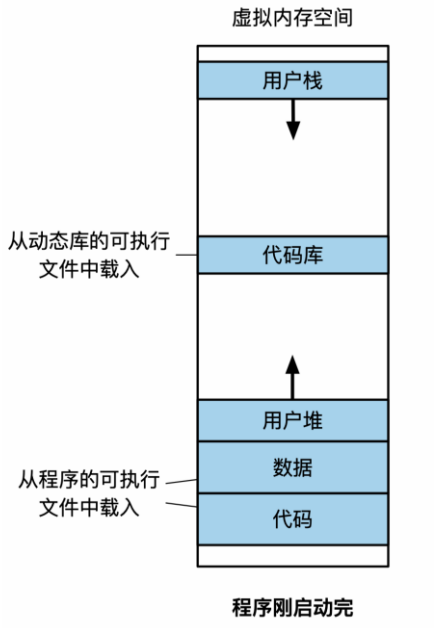
进程结构体的变化

```
1  struct process {
2    // 上下文
3    struct context *ctx;
4
5    // 虚拟内存
6    struct vmspace *vmspace;
7
8    ...
9  };
10
11 struct vmspace {
12   // 页表基地址
13   u64 pgtbl;
14
15   // 若干虚拟内存区域组成的链表
16   list vmregions;
17 };
18
19 // 表示一个虚拟内存区域
20 struct vmregion {
21   // 起始虚拟地址
22   u64 start;
23   // 结束虚拟地址
24   u64 end;
25   // 访问权限;
26   u64 perm;
27 };
```

VMA是如何添加的

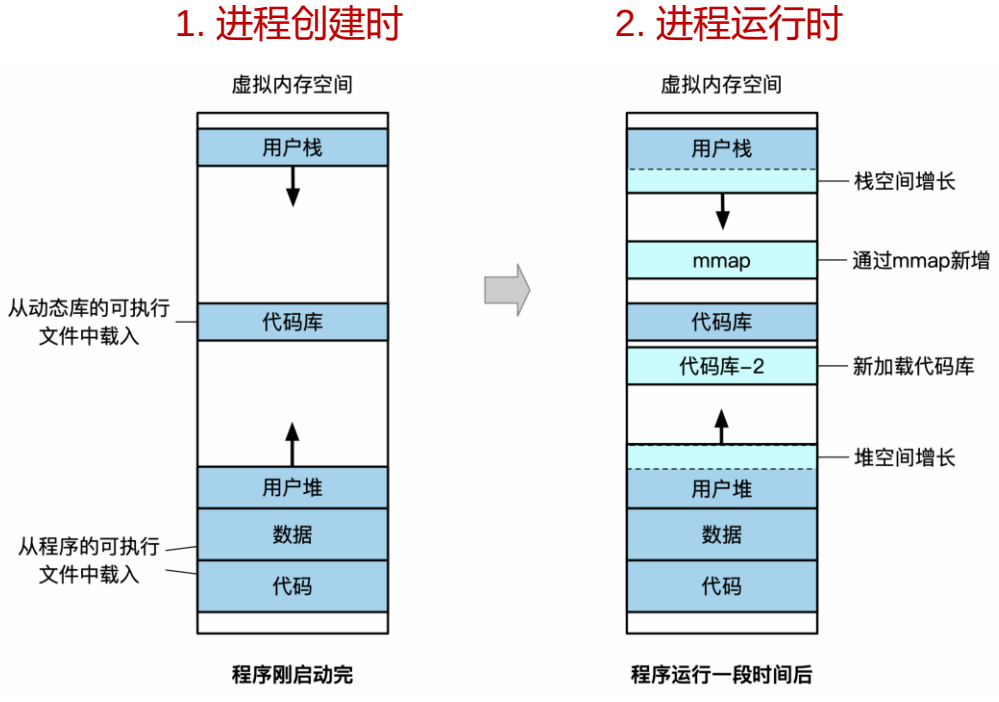
- **途径-1: OS在创建进程时分配**
  - 数据（对应ELF段）
  - 代码（对应ELF段）
  - 栈（初始无内容）

1. 进程创建时



VMA是如何添加的

- **途径-1: OS在创建进程时分配**
  - 数据（对应ELF段）
  - 代码（对应ELF段）
  - 栈（初始无内容）
- **途径-2：进程运行时添加**
  - 堆、栈
  - mmap/munmap
    - 分配内存buffer
    - 加载新的代码库



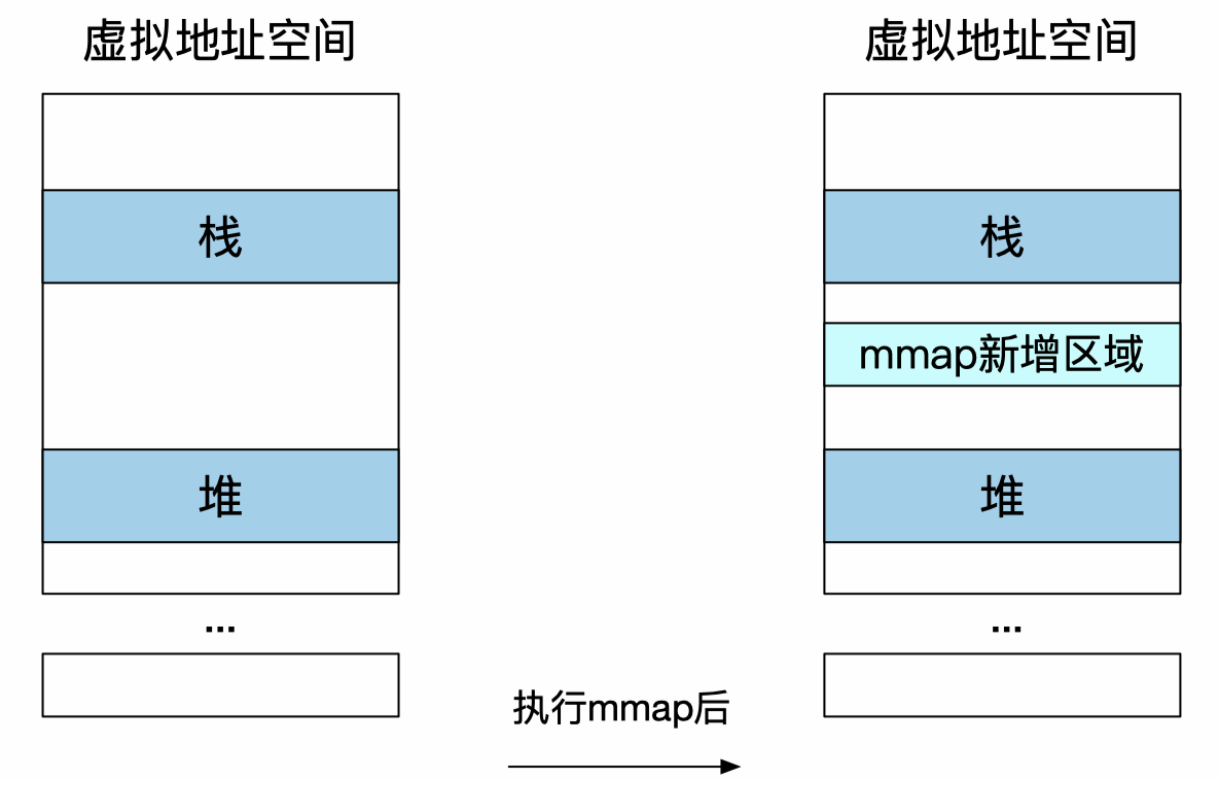
mmap：分配一段虚拟内存区域

- **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
    - VMA中还会包含文件映射等信息
- **也可以不映射任何文件，仅仅新建虚拟内存区域（匿名映射）**

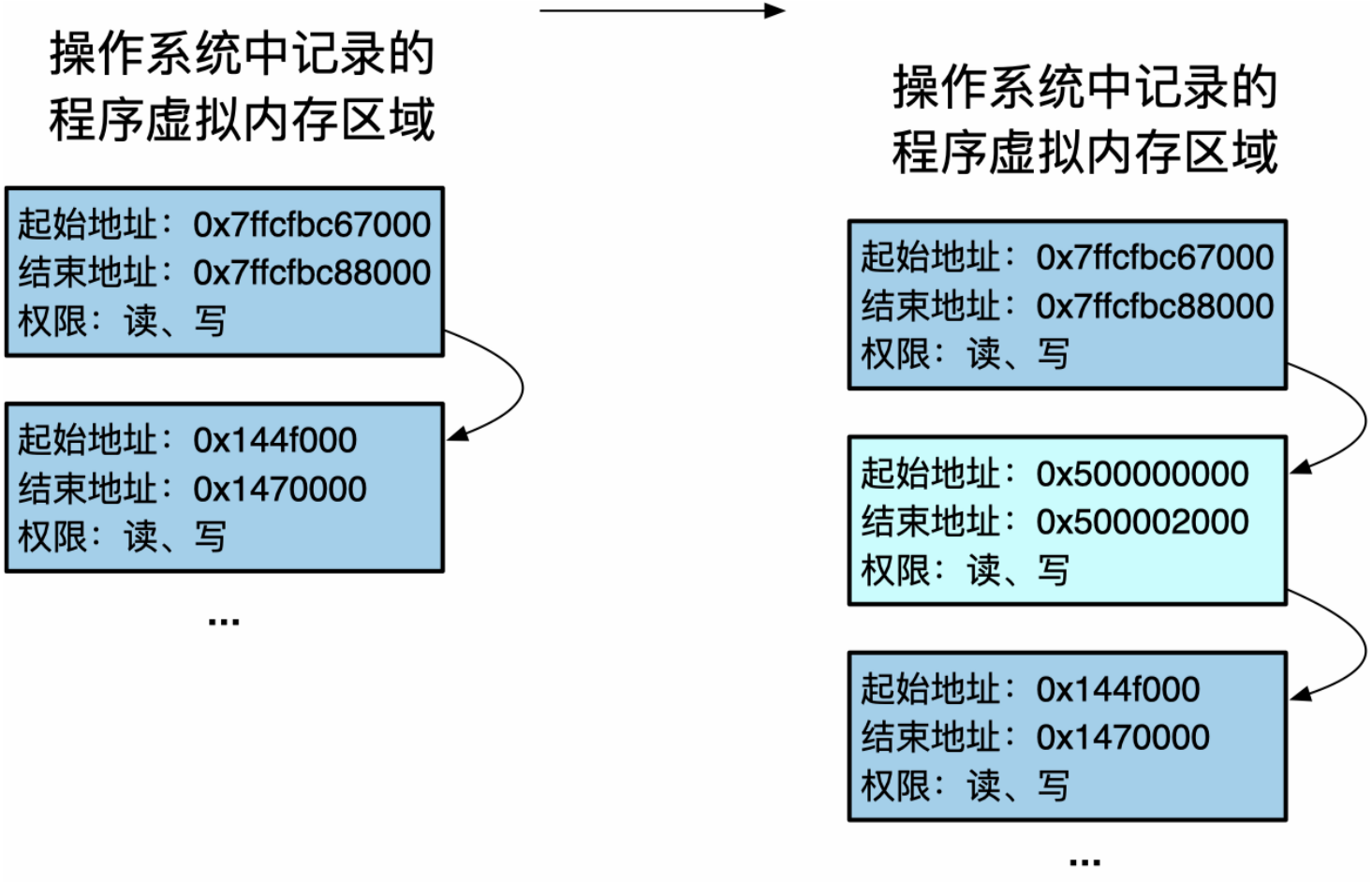
mmap匿名映射示例

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <sys/mman.h>
4
5  // void *mmap(void *addr, size_t length, int prot, int
   ↪  flags, int fd, off_t offset);
6
7  int main()
8  {
9    char *buf;
10
11   buf = mmap((void *)0x500000000, 0x2000, PROT_READ |
     ↪  PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
12   printf("mmap returns %p\n", buf);
13
14   strcpy(buf, "Hello mmap");
15   printf("%s\n", buf);
16
17   return 0;
18 }
19
20 The output after the execution is like :
21 mmap returns 0x500000000
22 Hello mmap
```

示例：执行mmap后，VMA的变化



示例：执行mmap后，VMA的变化



mmap映射文件

```c
int main() {
    int fd;
    struct stat sb;
    off_t len;
    char *addr;

    fd = open("hello.txt", O_RDONLY);
    fstat(fd, &sb);
    len = sb.st_size;
    addr = mmap(NULL, len, PROT_READ, MAP_PRIVATE, fd, 0);

    // 将文件内容输出到屏幕上
    printf("%s", addr);

    // 解除内存映射
    munmap(addr, len);
    close(fd);

    return 0;
}
```

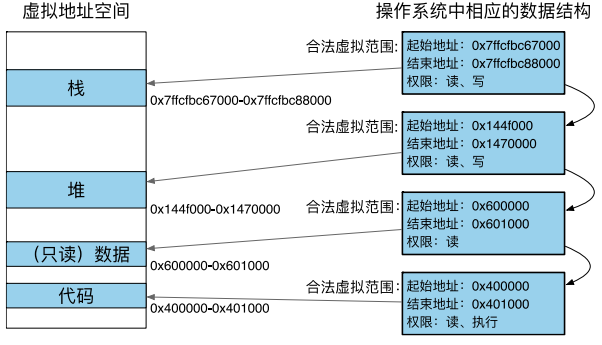
VMA是如何添加的

- **途径2: 进程运行时添加/应用程序主动向OS发起系统调用**
  - 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
  - brk()：扩大、缩小堆区域
  - 栈VMA的可选策略
    - OS为进程初始分配固定大小的栈VMA，在发现stackoverflow之后自动扩大栈VMA
  - 用户态的malloc（API）也可能改变VMA
    - 调用brk，在堆中分配新的内存
    - 调用mmap分配较大区域

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // 使用malloc分配10个整数大小的内存空间
    int* ptr = (int*) malloc(10 * sizeof(int));

    // 存储数据到分配的内存空间中
    for (int i = 0; i < 10; i++) {
        ptr[i] = i + 1;
    }

    // 打印存储的数据
    for (int i = 0; i < 10; i++) {
        printf("%d ", ptr[i]);
    }

    // 释放分配的内存空间
    free(ptr);

    return 0;
}
```

根据VMA判断缺页异常的合法性

- **缺页异常（page fault)**
  - AARCH64：触发（通用的）同步异常（8）
  - 根据ESR信息判断是否为缺页异常
  - 访问的虚拟地址存放在FAR_EL1



- **操作系统的缺页处理函数**
  - FAR_EL1中的值不落在VMA区域内，则为非法
  - 反之，则分配物理页，并在页表中添加映射

问：用什么数据结构组织VMA?

延迟映射 vs. 立即映射

- **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先映射的策略（预测相邻的虚拟页也会被访问，提前映射）
    - 在节约内存和减少缺页异常次数之间取得平衡

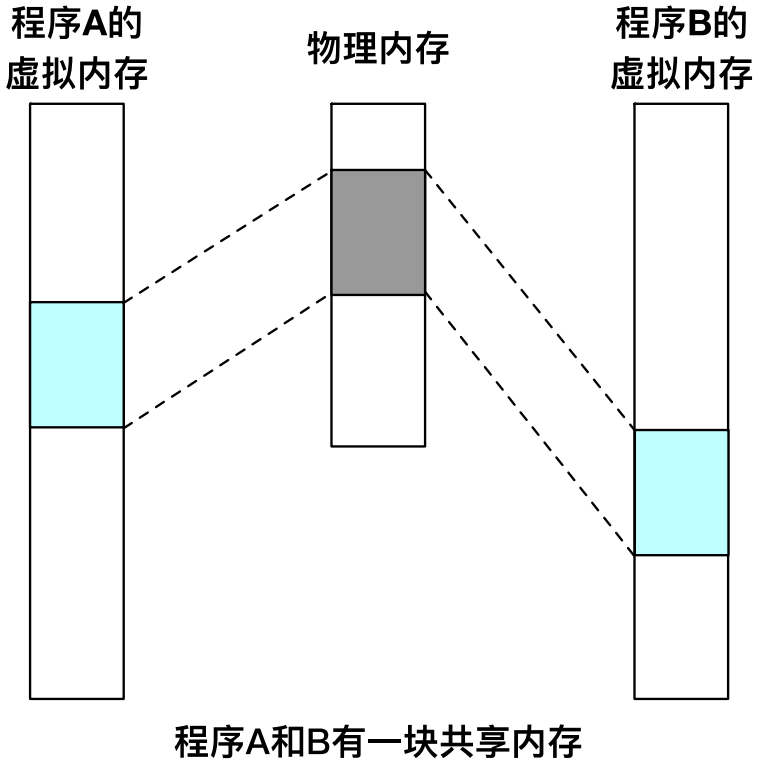
OS可向应用提供灵活的内存管理系统调用

- **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"

虚拟内存的扩展功能

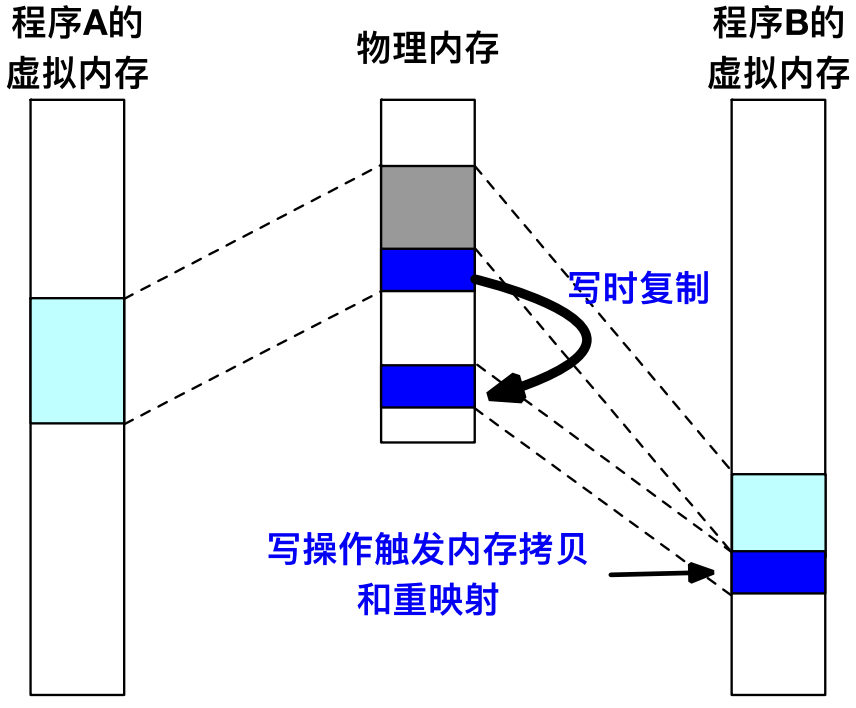
共享内存

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



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

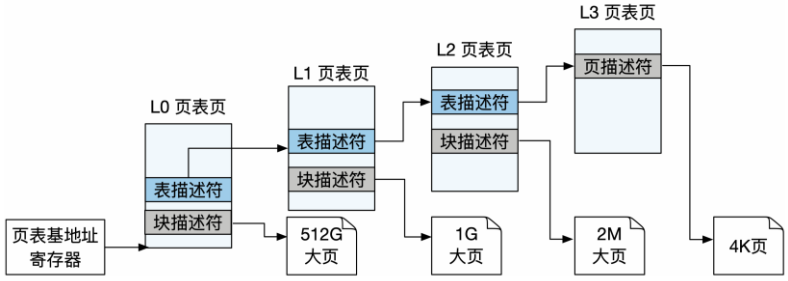
内存压缩

- **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

AARCH64支持多种最小页面大小

- **TCR_EL1可以选择不同的最小页面大小**
  - 3种配置：4K、16K、64K
  - 4K + 大页：2M/1G
  - 16K + 大页：32M（问：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）

总结

- **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**