中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

虚拟内存管理II

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

大纲

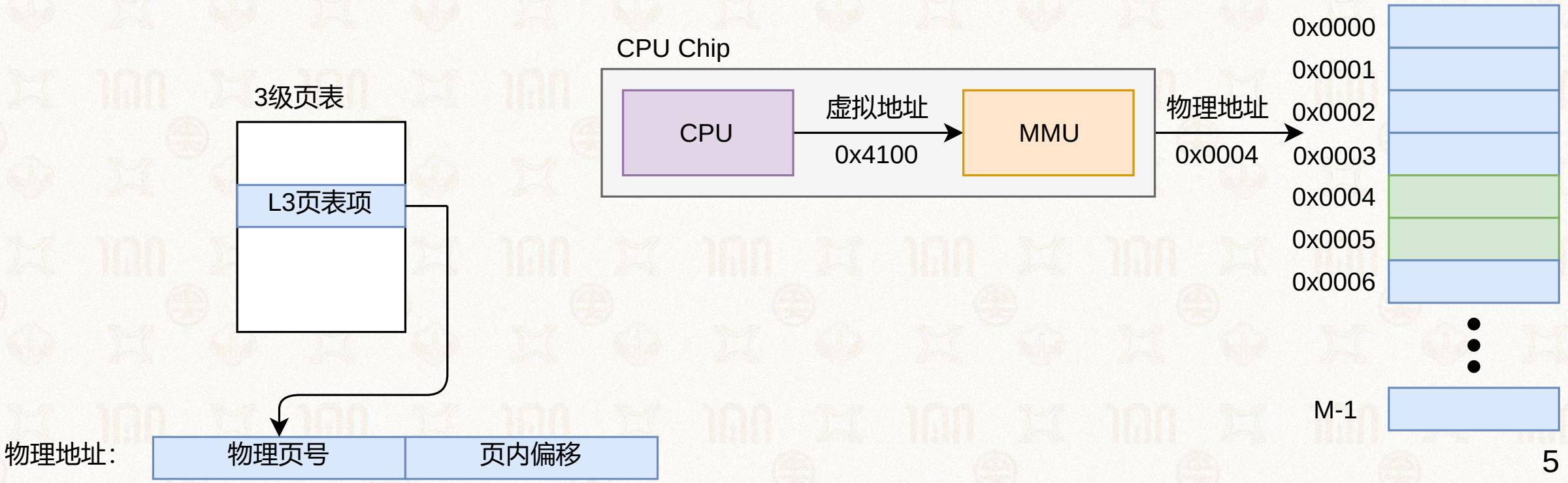
- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

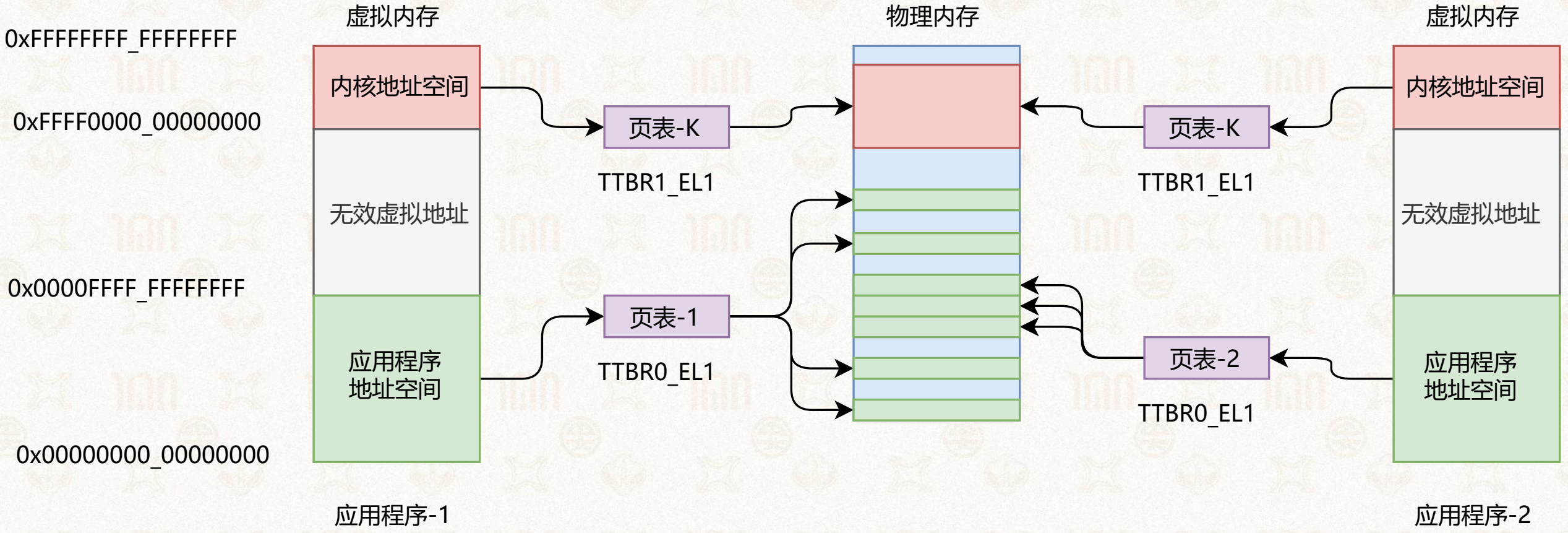
操作系统自己

- 所有汇编语言指令涉及的地址都是虚拟地址
  - 操作系统所有指令应该也是虚拟地址
- 那前面说页表里存的是物理地址是怎么回事？
  - 四级页表虽是由MMU直接使用，但是由操作系统创建和维护的
  - 操作系统怎么填页表里的物理地址？



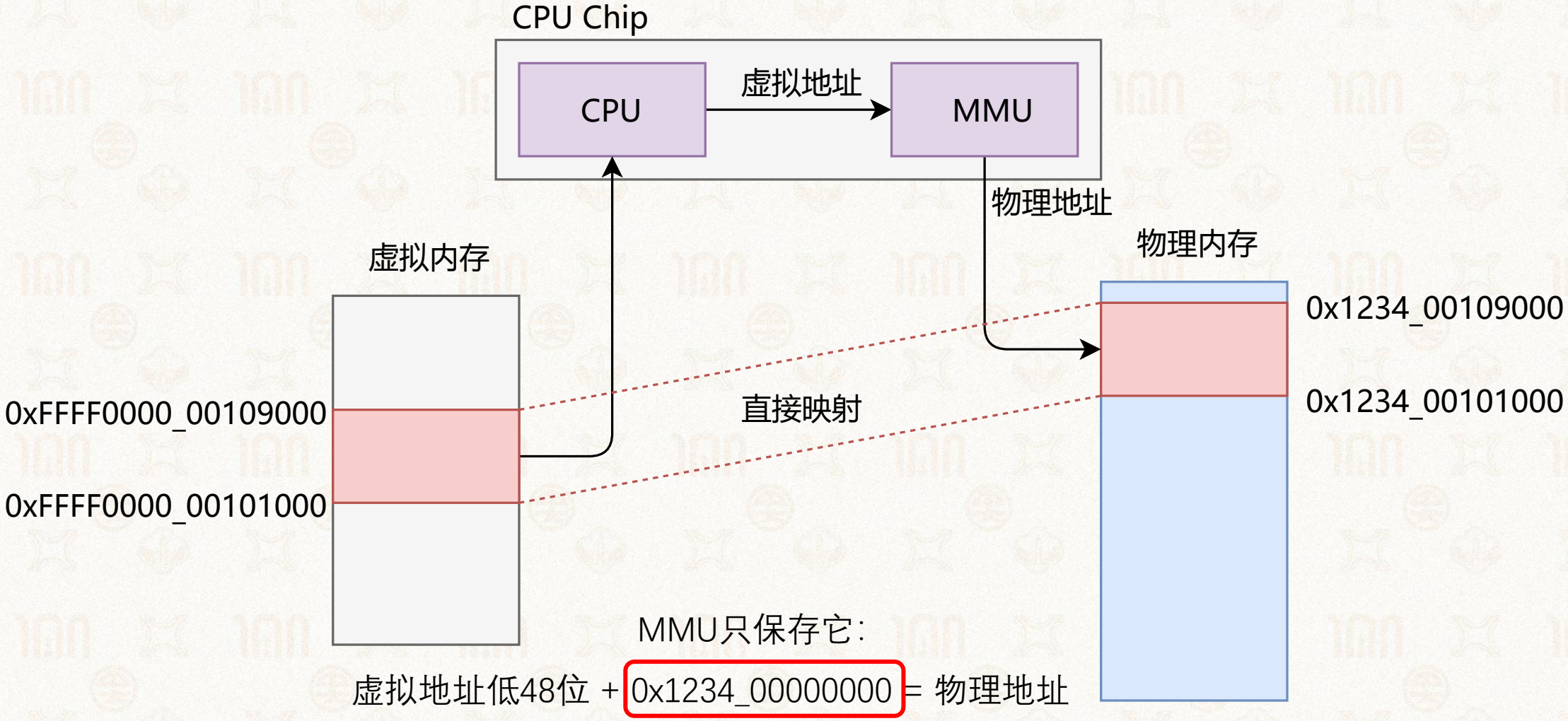
操作系统自己

- 操作系统可以访问所有物理内存
- 操作系统自己还需要页表么？



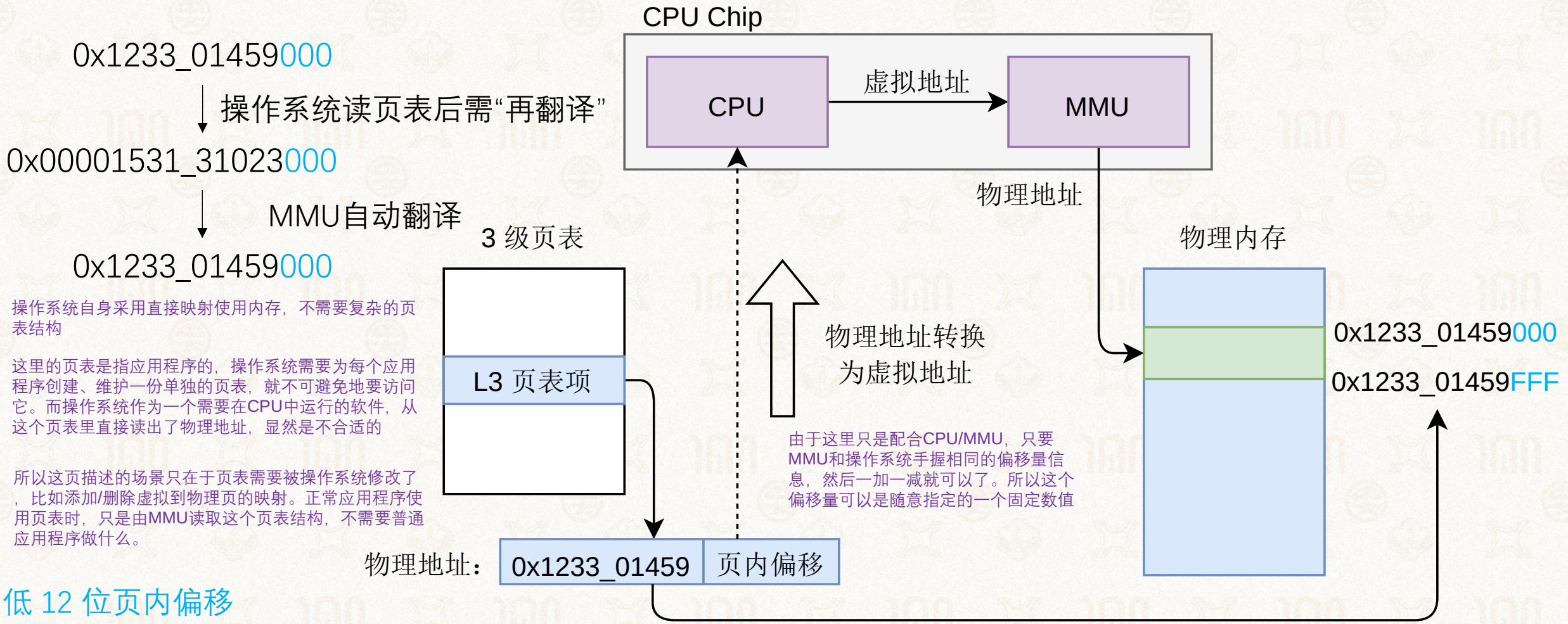
操作系统自己：直接映射(Direct Mapping)

- 操作系统直接使用连续内存空间
- 操作系统虚拟内存地址与物理内存保持简单的线性映射



操作系统自己：直接映射(Direct Mapping)

- 应用程序页表里保存的是物理地址，CPU只接收虚拟地址，怎么办？



大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

内存区域分布

- 实验课所示，为什么会有这么多区域？

```
yxsu@os:~$ cat /proc/6130/maps
645360083000-645360084000 r--p 00000000 08:02 795759   /home/yxsu/hello-vm.o
645360084000-645360085000 r-xp 00001000 08:02 795759   /home/yxsu/hello-vm.o
645360085000-645360086000 r--p 00002000 08:02 795759   /home/yxsu/hello-vm.o
645360086000-645360087000 r--p 00002000 08:02 795759   /home/yxsu/hello-vm.o
645360087000-645360088000 rw-p 00003000 08:02 795759   /home/yxsu/hello-vm.o
6453736a6000-6453736c7000 rw-p 00000000 00:00 0        [heap]
740906a00000-740906a28000 r--p 00000000 08:02 273617   /usr/lib/x86_64-linux-gnu/libc.so.6
740906a28000-740906bb0000 r-xp 00028000 08:02 273617   /usr/lib/x86_64-linux-gnu/libc.so.6
740906bb0000-740906bff000 r--p 001b0000 08:02 273617   /usr/lib/x86_64-linux-gnu/libc.so.6
740906bff000-740906c03000 r--p 001fe000 08:02 273617   /usr/lib/x86_64-linux-gnu/libc.so.6
740906c03000-740906c05000 rw-p 00202000 08:02 273617   /usr/lib/x86_64-linux-gnu/libc.so.6
740906c05000-740906c12000 rw-p 00000000 00:00 0
740906c8e000-740906c91000 rw-p 00000000 00:00 0
740906ca0000-740906ca2000 rw-p 00000000 00:00 0
740906ca2000-740906ca6000 r--p 00000000 00:00 0        [vvar]
740906ca6000-740906ca8000 r-xp 00000000 00:00 0        [vdso]
740906ca8000-740906ca9000 r--p 00000000 08:02 273435   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
740906ca9000-740906cd4000 r-xp 00001000 08:02 273435   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
740906cd4000-740906cde000 r--p 0002c000 08:02 273435   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
740906cde000-740906ce0000 r--p 00036000 08:02 273435   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
740906ce0000-740906ce2000 rw-p 00038000 08:02 273435   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd98ef0000-7ffd98f11000 rw-p 00000000 00:00 0        [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

虚拟内存的起始地址

虚拟内存的终止地址

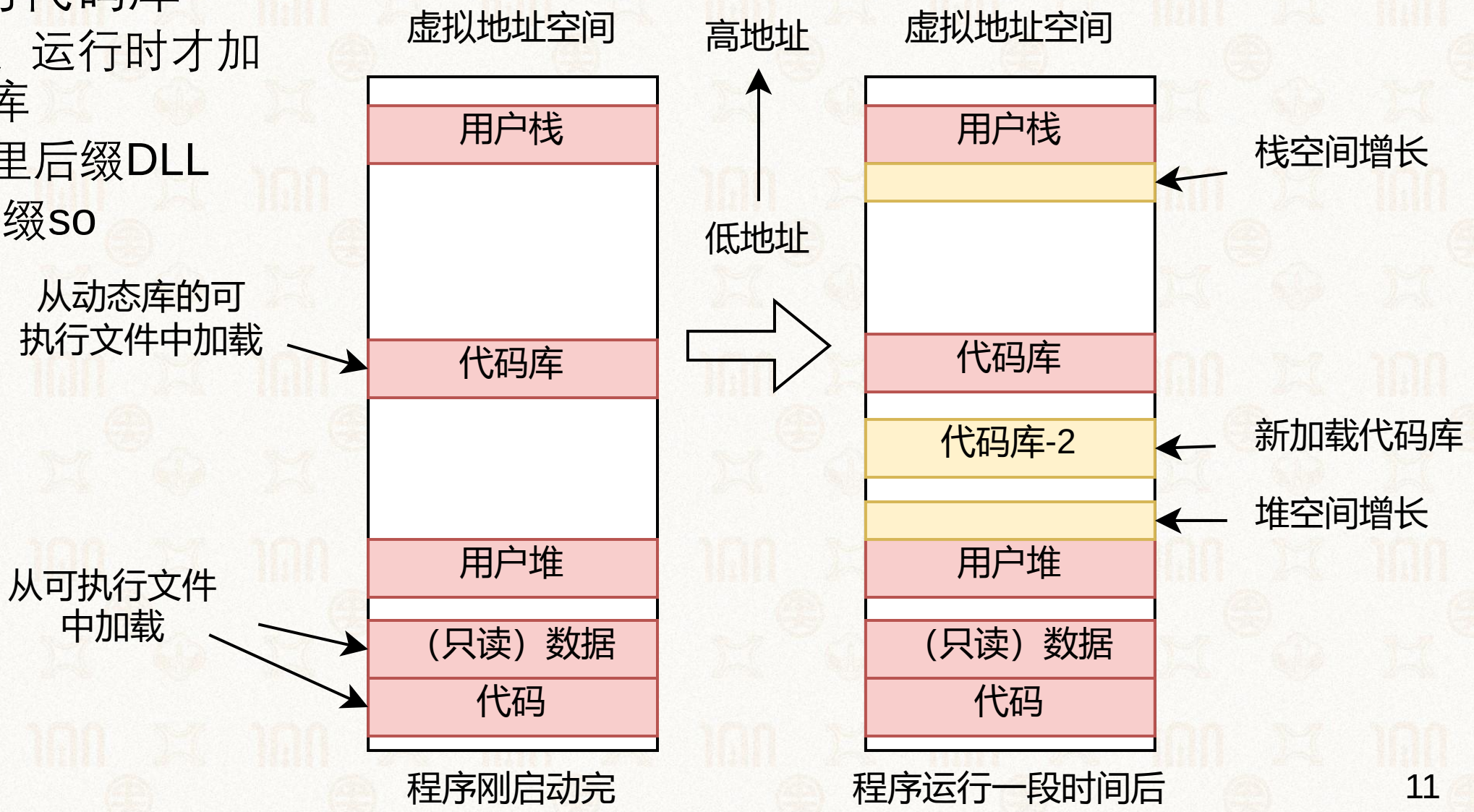
区域的权限标记

这段区域从可执行文件中加载时，它在原可执行文件中的位置（文件头开始的偏移量）

动态加载的代码库

内存区域分布

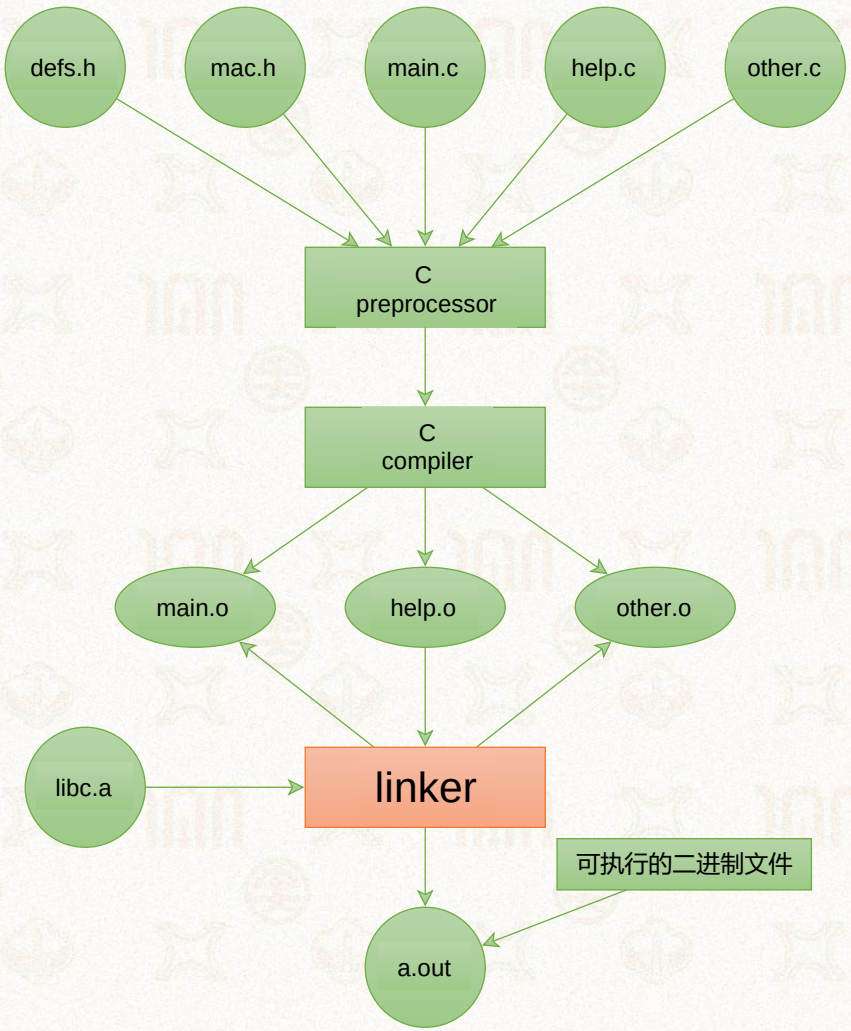
- 动态加载的代码库
  - 程序启动、运行时才加载的指令库
  - Windows里后缀DLL
  - Linux里后缀so



大型c/c++工程的编译过程

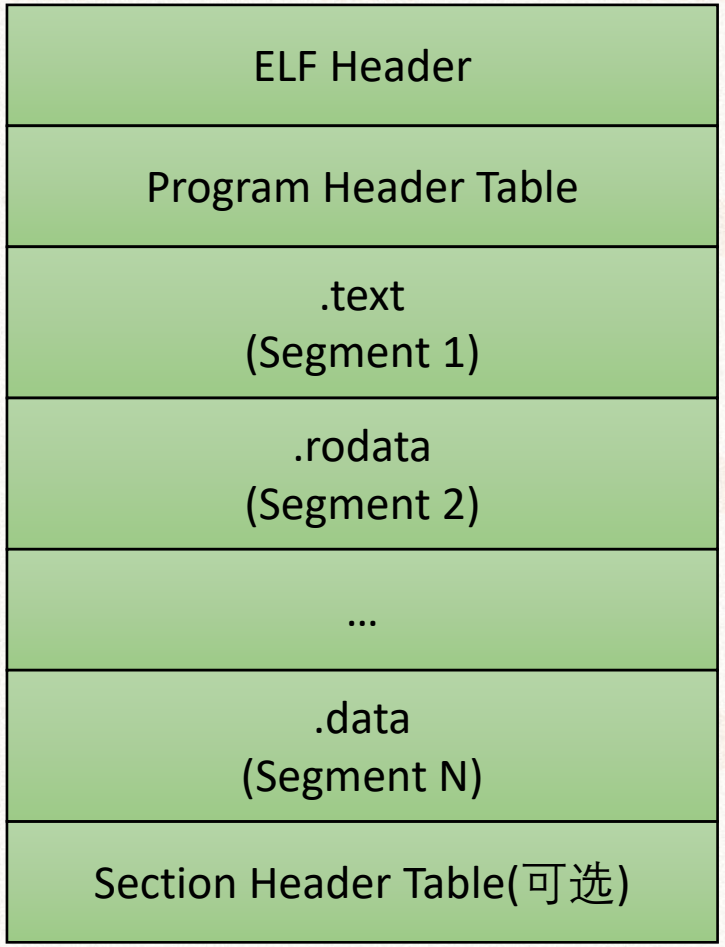
- 为什么要有链接器(Linker)?
- 不仅需要链接外部库文件
- 还需要补全可执行文件的信息:
  - 头部信息（基本信息）
  - 各代码段位置、标签
    - 像函数名一样

因为程序本身就是分段编译、加载的



可执行文件 (ELF)

- ELF可执行可链接文件
- 常见于:
  - Linux/Android系统的可执行文件
  - 共享库(.so / .a)
  - 目标文件(.o)
- 组成
  - ELF头部(ELF header)
  - 多个程序段(program section)
    - 每个程序段都是一个连续的二进制块
    - （硬件或软件）加载器将它们作为代码或数据加载到指定地址的内存中并开始执行



ELF头部文件结构体定义

```
typedef struct elf64_hdr {
  unsigned char e_ident[EI_NIDENT]; /* ELF "magic number" */
  Elf64_Half e_type;
  Elf64_Half e_machine;
  Elf64_Word e_version;
  Elf64_Addr e_entry;         /* Entry point virtual address */
  Elf64_Off e_phoff;          /* Program header table file offset */
  Elf64_Off e_shoff;          /* Section header table file offset */
  Elf64_Word e_flags;
  Elf64_Half e_ehsize;
  Elf64_Half e_phentsize;
  Elf64_Half e_phnum;
  Elf64_Half e_shentsize;
  Elf64_Half e_shnum;
  Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

e_ident：包含了Maigc Number和其它信息，共16字节：
0~3：前4字节为Magic Number，固定为ELFMAG。
4（EI_CLASS）：ELFCLASS32代表是32位ELF，ELFCLASS64 代表64位ELF。
5（EI_DATA）：ELFDATA2LSB代表小端，ELFDATA2MSB代表大端。
6（EI_VERSION）：固定为EV_CURRENT（1）。
7（EI_OSABI）：操作系统ABI标识（实际未使用）。
8（EI_ABIVERSION）：ABI版本（实际 未使用）。
9~15：对齐填充，无实际意义。

https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/elf.h

可执行程序的文件结构（头信息）

- ELF头部记录文件的结构

- 通过命令获得头部信息：
  readelf -h hello-vm.o

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x80000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136536 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         9
  Section header string table index: 8
```

可执行程序的文件结构（段信息）

yxsu@os:~$ readelf -S hello-vm.o
There are 31 section headers, starting at offset 0x36e8:

Section Headers:

[Nr] Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[0]	NULL	0000000000000000	00000000	0000000000000000	0000000000000000		0	0	0
[1] .interp	PROGBITS	0000000000000318	00000318	000000000000001c	0000000000000000	A	0	0	1
[2] .note.gnu.pr[...]	NOTE	0000000000000338	00000338	0000000000000030	0000000000000000	A	0	0	8
[6] .dynsym	DYNSYM	00000000000003d8	000003d8	00000000000000c0	0000000000000018	A	7	1	8
[7] .dynstr	STRTAB	0000000000000498	00000498	0000000000000094	0000000000000000	A	0	0	1
[12] .init	PROGBITS	0000000000001000	00001000	000000000000001b	0000000000000000	AX	0	0	4
[16] .text	PROGBITS	0000000000001080	00001080	000000000000013f	0000000000000000	AX	0	0	16
[18] .rodata	PROGBITS	0000000000002000	00002000	0000000000000053	0000000000000000	A	0	0	8
[21] .init_array	INIT_ARRAY	0000000000003db0	00002db0	0000000000000008	0000000000000008	WA	0	0	8
[23] .dynamic	DYNAMIC	0000000000003dc0	00002dc0	00000000000001f0	0000000000000010	WA	7	0	8
[25] .data	PROGBITS	0000000000004000	00003000	000000000000001a	0000000000000000	WA	0	0	8
[26] .bss	NOBITS	000000000000401a	0000301a	0000000000000006	0000000000000000	WA	0	0	1
[27] .comment	PROGBITS	0000000000000000	0000301a	000000000000002b	0000000000000001	MS	0	0	1
[28] .symtab	SYMTAB	0000000000000000	00003048	0000000000000390	0000000000000018	29	18	8	
[29] .strtab	STRTAB	0000000000000000	000033d8	00000000000001f5	0000000000000000	0	0	1	

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
D (mbind), l (large), p (processor specific)

段在虚拟内存中的地址

段在ELF文件中的位置（偏移量）

段的大小

段中每个条目的大小（像数组元素大小）

可执行程序的文件结构（常见的段信息）

- .text:
  - 保存程序代码，是由一条条的机器指令组成的。
- .data:
  - 保存初始化的全局变量或静态变量数据
  - 定义在函数内部的局部非静态变量不在该段中存储
- .rodata:
  - 保存只读数据，包括一些不可修改的常量数据，例如全局常量、char *str = "apple"中的字符串常量等
- .bss:
  - 记录未初始化的全局或静态变量，例如int a
  - 由于在运行期间未初始化的全局变量被初始化为0，因此链接器只记录地址和大小，而不是占用实际空间。

应用程序的加载与执行

- 一个程序启动需要有两步：

- 加载(load)
  - 将程序的ELF文件按照链接规则从存储器上按照每个段的加载内存地址（Load Memory Address，LMA）拷贝到内存上指定的地址

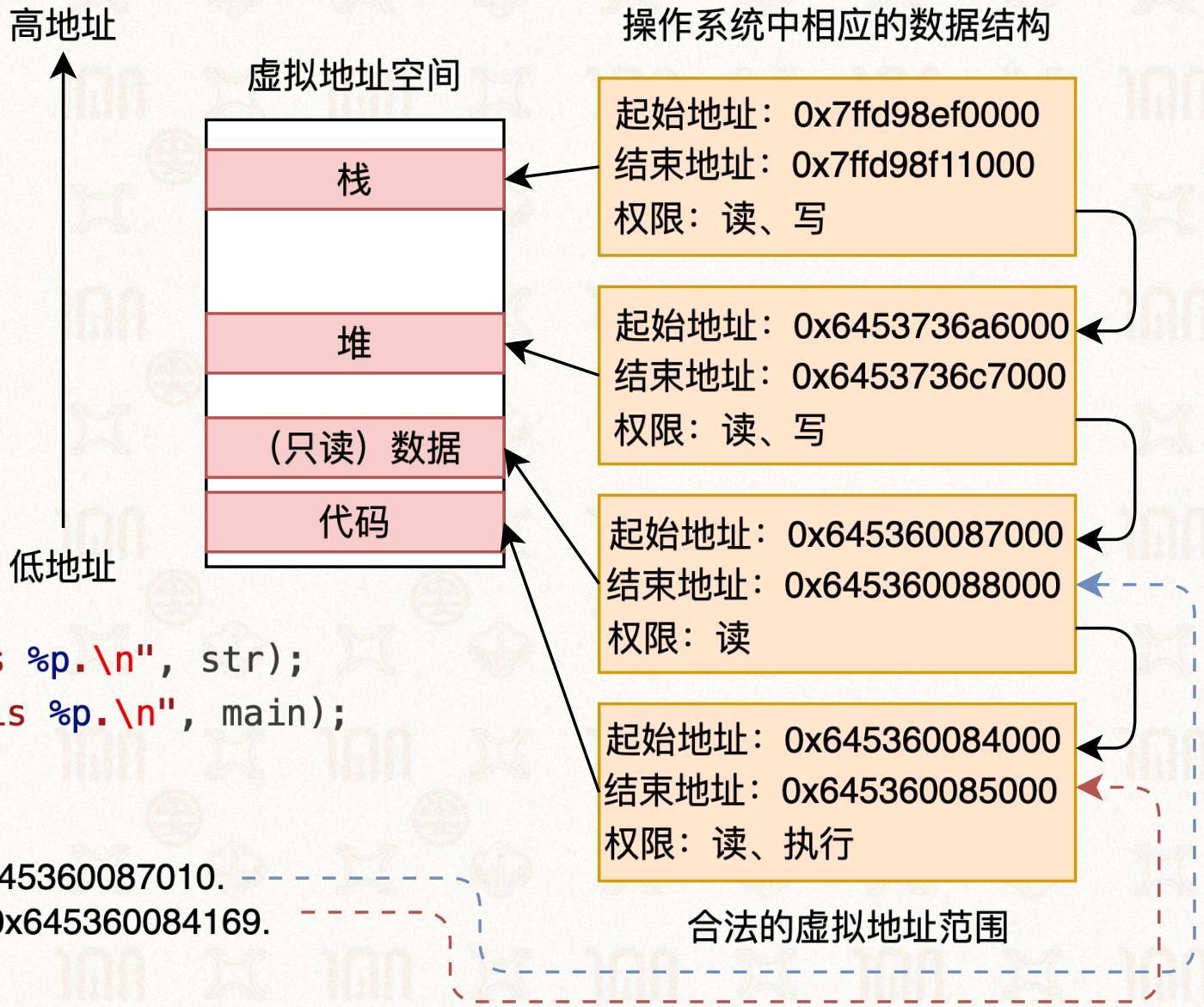
- 执行(execute)
  - 将ELF文件中的段“放到”虚拟内存地址（Virtual Memory Address，VMA）
  - 然后开始真正执行ELF文件中的代码

内存区域分布

- 应用程序主要的内存段布局
  - 从可执行程序文件复制得来

```c
#include "stdio.h"

char str[] = "Hello VM!";

int main(void) {
    printf("%s\n", str);
    printf("The (data) address of str is %p.\n", str);
    printf("The (code) address of main is %p.\n", main);
    return 0;
}
```

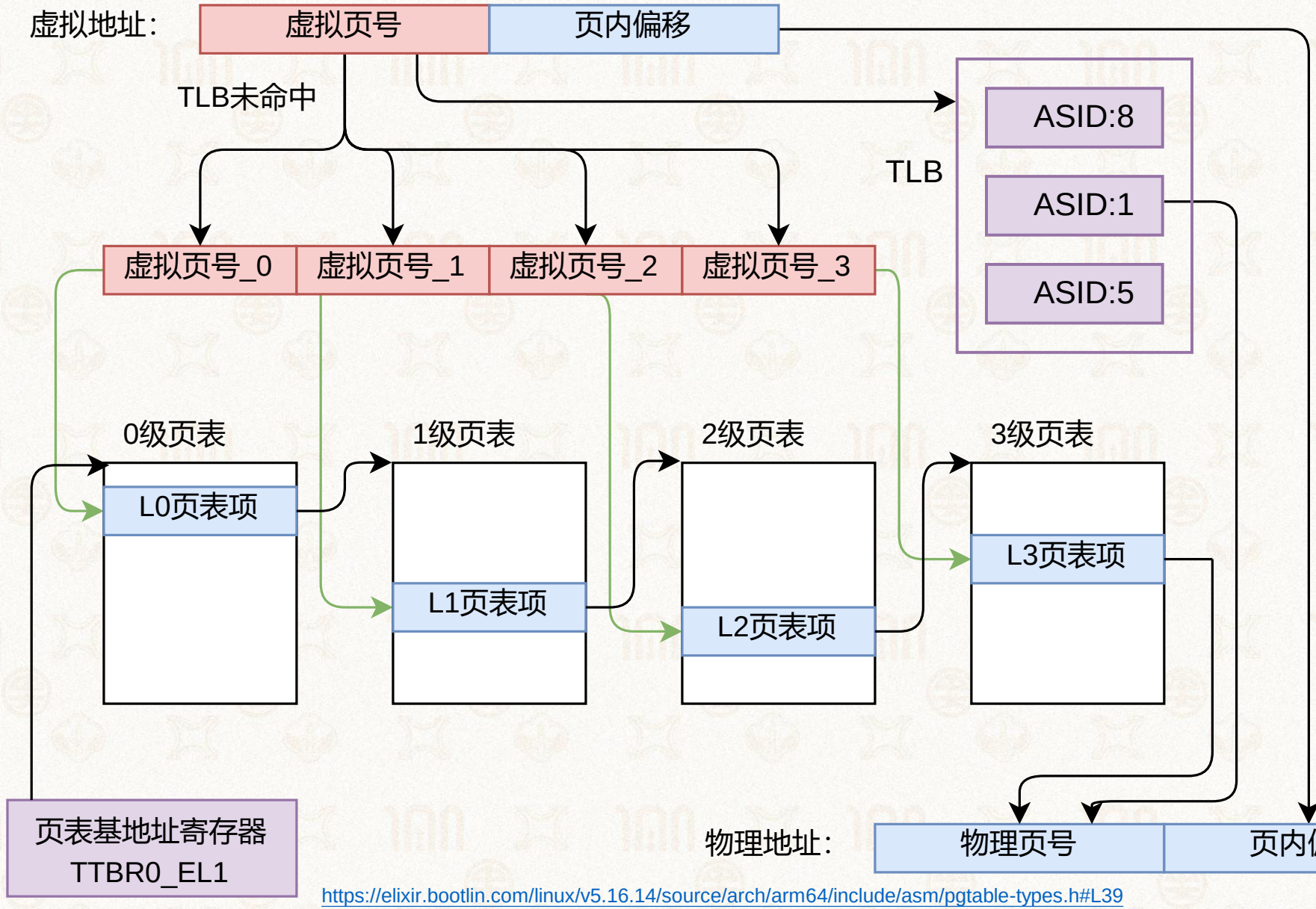
```
yxsu@os:~$ Hello VM!
The (data) address of str is 0x645360087010.
The (code) address of main is 0x645360084169.
```



大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

统一页表框架



- Linux内核中:
  - 0级页表: 页全局目录 (Page Global Directory, PGD)
  - 1级页表: 页上层目录 (Page Upper Directory, PUG)
  - 2级页表: 页中间目录 (Page Middle Directory, PMG)
  - 3级页表: 直接页表 (Page Table, PT)
- Linux 4.11后可以再加一级, 页四级目录 (Page 4th Directory, P4D)

https://elixir.bootlin.com/linux/v5.16.14/source/arch/arm64/include/asm/pgtable-types.h#L39

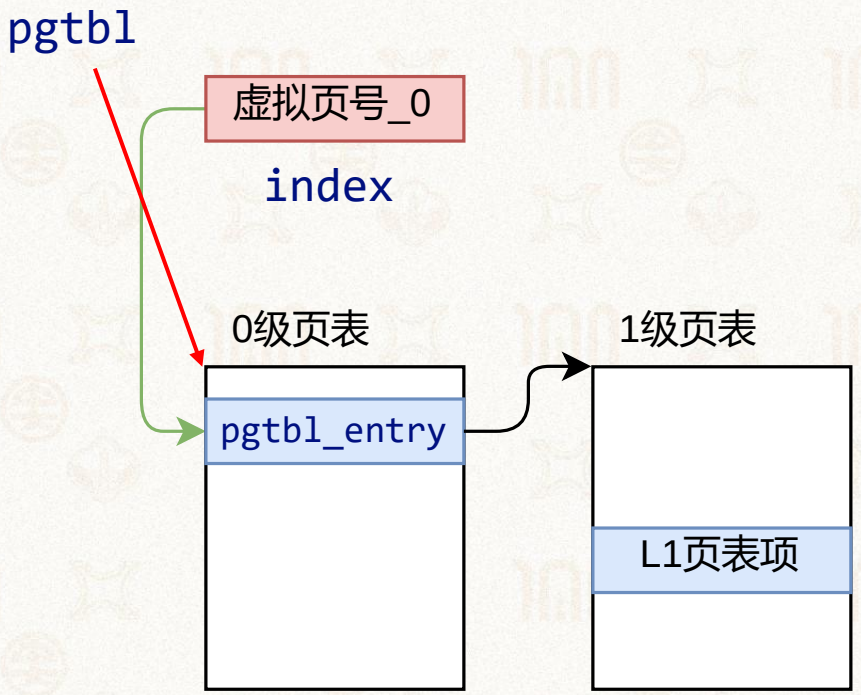
和页表管理相关的代码

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

void add_mapping(u64 pgtbl, u64 va, u64 pa);
void delete_mapping(u64 pgtbl, u64 va);
```

获取下一级页表

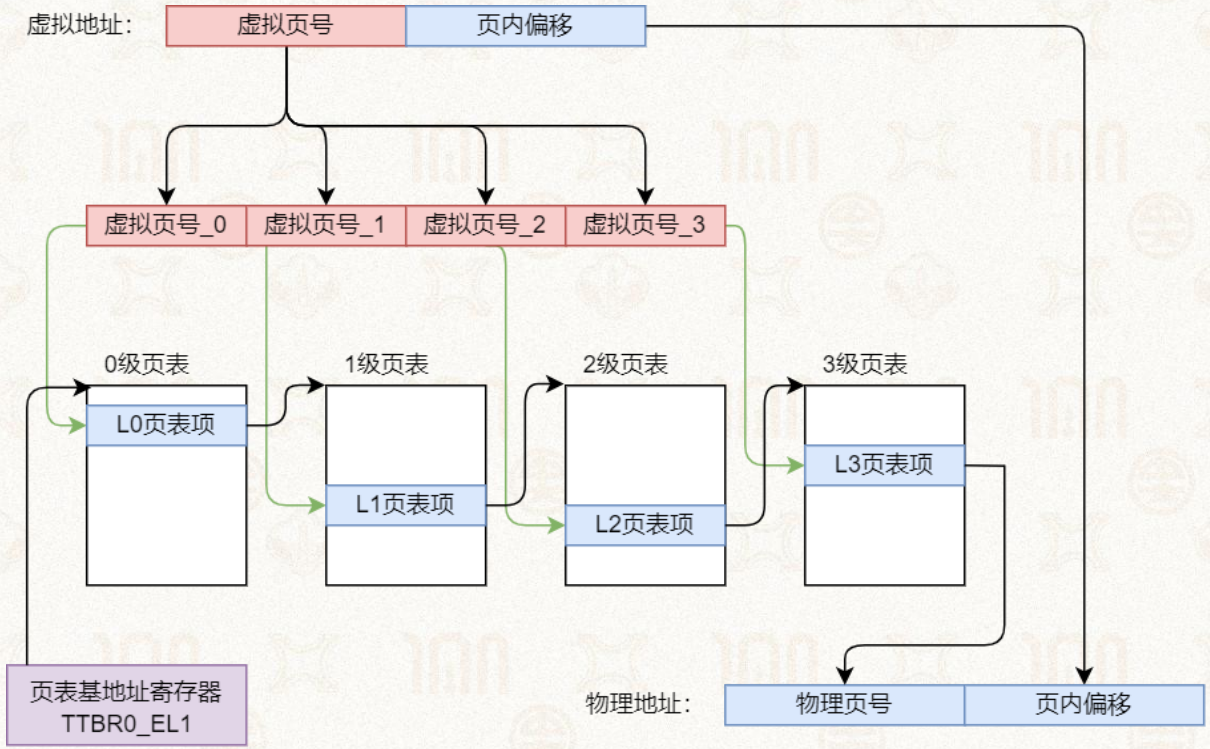
```
u64 get_next_pgtbl_page(u64 *pgtbl, u32 index) {
    u64 pgtbl_entry;

    pgtbl_entry = pgtbl[index];

    if (pgtbl_entry == 0) {
        // 如果没有相应的页表页（页表空洞），则分配页表页
        pgtbl_entry = alloc_pgtbl_page();
        pgtbl_page[index] = pgtbl_entry | some_permission;
    }

    // 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
    return paddr_to_vaddr(pgtbl_entry);
}
```

需要注意这个“再翻译”



添加页表映射

```
// 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;
}
```



删除页表映射

```
void delete_mapping(u64 pgtbl, u64 va) {
    // 类似 add_mapping，在 pgtbl 页表中，
    // 逐级查找虚拟地址 va 对应的页表项，
    // 若页表项存在，则将该页表项清空
    // ...
    // 利用硬件提供的精准刷新虚拟地址相应 TLB 项的指令
    flush_tlb(pgtbl, va);
}
```

大纲

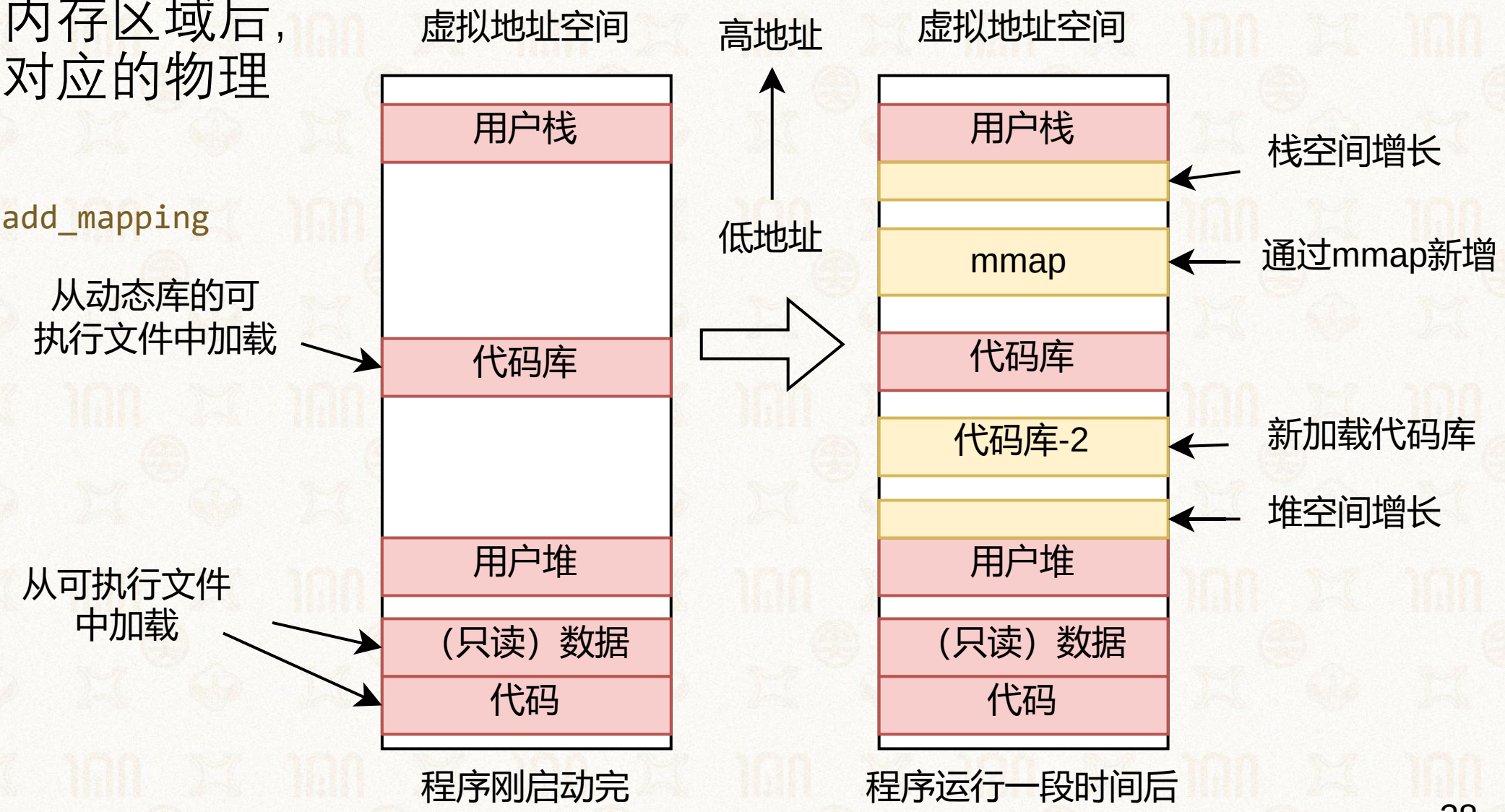
- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

立即映射：mmap

```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;
}
```

立即映射：mmap

- 声明连续内存区域后，马上分配对应的物理内存
- 马上调用`add_mapping`



立即映射：mmap实现

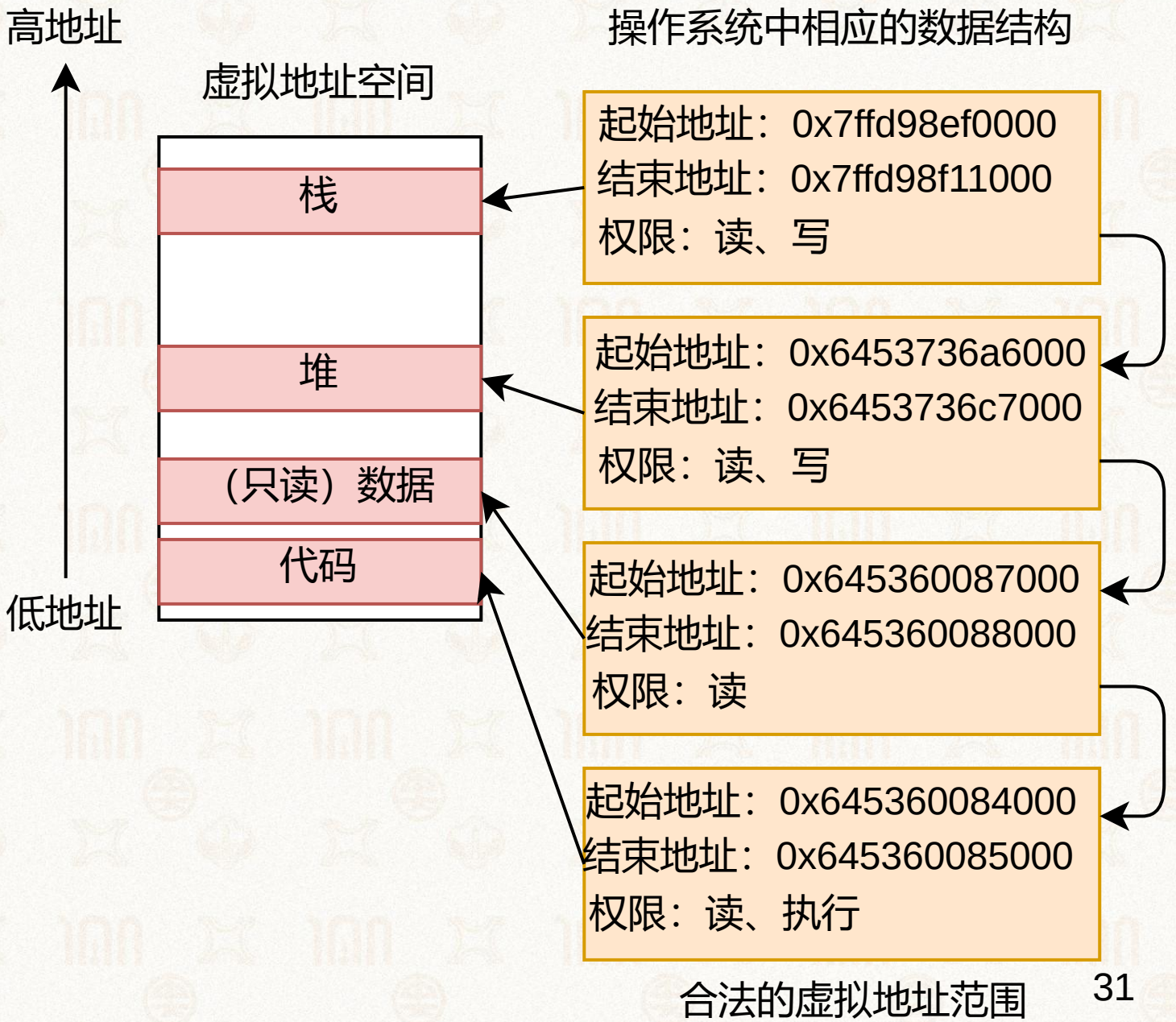
```
// 参数 addr 和 length 分别是虚拟内存区域起始地址和长度
// 伪代码忽略边界条件检查等
void sys_mmap(u64 addr, u64 length, ...) {
    u64 page_num;
    u64 pa;
    u64 pgtbl;

    // 总共需要映射的页面数量，PAGE_SIZE 是 4K
    page_num = length / PAGE_SIZE;

    // 获取当前进程的页表
    pgtbl = get_current_process_pgtbl();

    // 为每个虚拟页分配物理页，并在页表中添加映射
    while (page_num > 0) {
        pa = alloc_page();
        add_mapping(pgtbl, addr, pa);
        addr += PAGE_SIZE;
    }
}
```

大纲

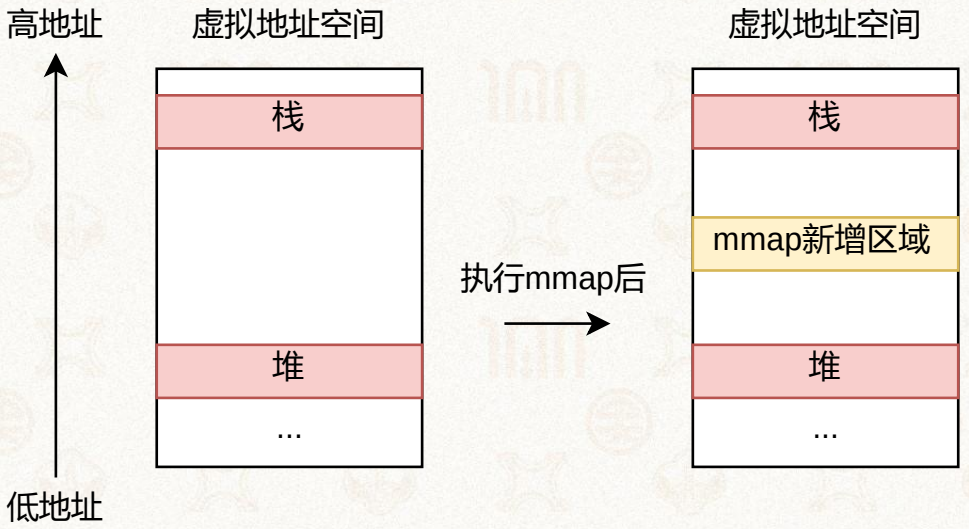
- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

延迟映射

- 先声明内存区域，待真正使用时才创建到物理内存页的映射
- 系统为每个程序保存若干内存区域
- 更加节约物理内存



延迟映射实现



操作系统中记录的
程序虚拟内存区域

起始地址：0x7ffd98ef0000
结束地址：0x7ffd98f11000
权限：读、写

起始地址：0x6453736a6000
结束地址：0x6453736c7000
权限：读、写

操作系统中记录的
程序虚拟内存区域

起始地址：0x7ffd98ef0000
结束地址：0x7ffd98f11000
权限：读、写

起始地址：0x700261200000
结束地址：0x7002638d9000
权限：读、写

起始地址：0x6453736a6000
结束地址：0x6453736c7000
权限：读、写

```c
struct process {
    // 上下文
    struct context *ctx;
    // 虚拟内存
    struct vmspace *vmspace;
};
```

```c
struct vmspace {
    // 页表基地址
    u64 pgtbl;
    // 若干虚拟内存区域组成的链表
    list vmregions;
};

// 表示一个虚拟内存区域
struct vmregion {
    // 起始虚拟地址
    u64 start;
    // 结束虚拟地址
    u64 end;
    // 访问权限;
    u64 perm;
};
```

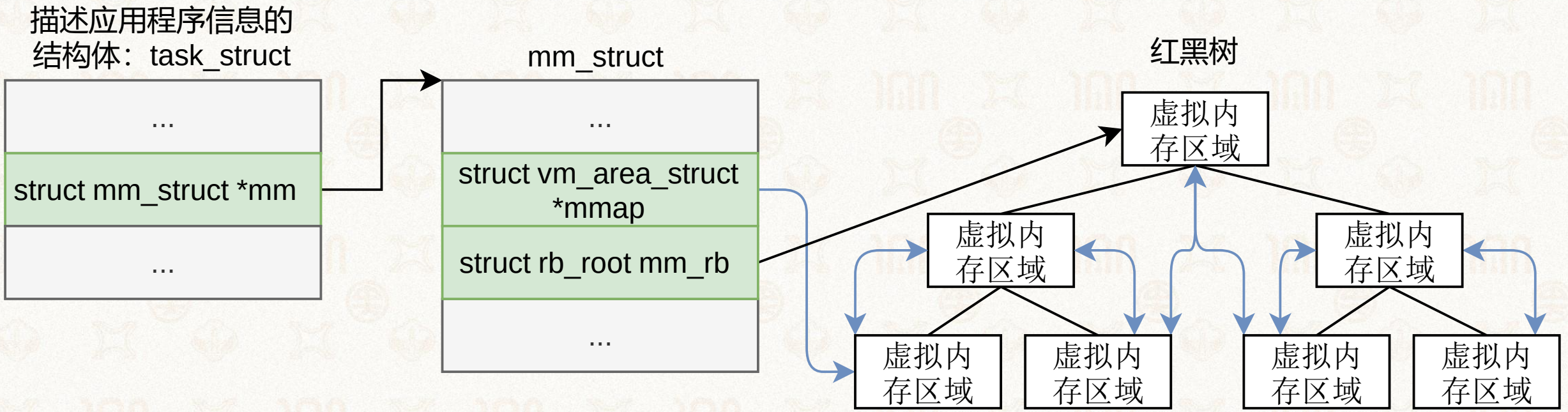
延迟映射缺点

- 应用程序第一次访问虚拟页时会触发缺页异常
- 操作系统介入
  - 降低性能

```
void page_fault_handler(u64 fault_va, ...) {
    u64 va, pa;
    list vmregions;
    struct vmregion vmr;
    u64 pgtbl;

    va = ROUND_DOWN(fault_va, PAGE_SIZE);

    vmregions = get_current_process_vmregions();
    for_each vmr in vmregions:
        if va in [vmr.start, vmr.end):
            check_perm();
            pa = alloc_page();
            pgtbl = get_current_process_pgtbl();
            add_mapping(pgtbl, addr, pa);
}
```

Linux描述虚拟地址空间的结构体： mm_struct

- 指向第一级页表：pgd_t * pgd
- 虚拟内存区域链表：struct vm_area_struct *mmap;
- 虚拟内存区域红黑树：struct rb_root mm_rb;

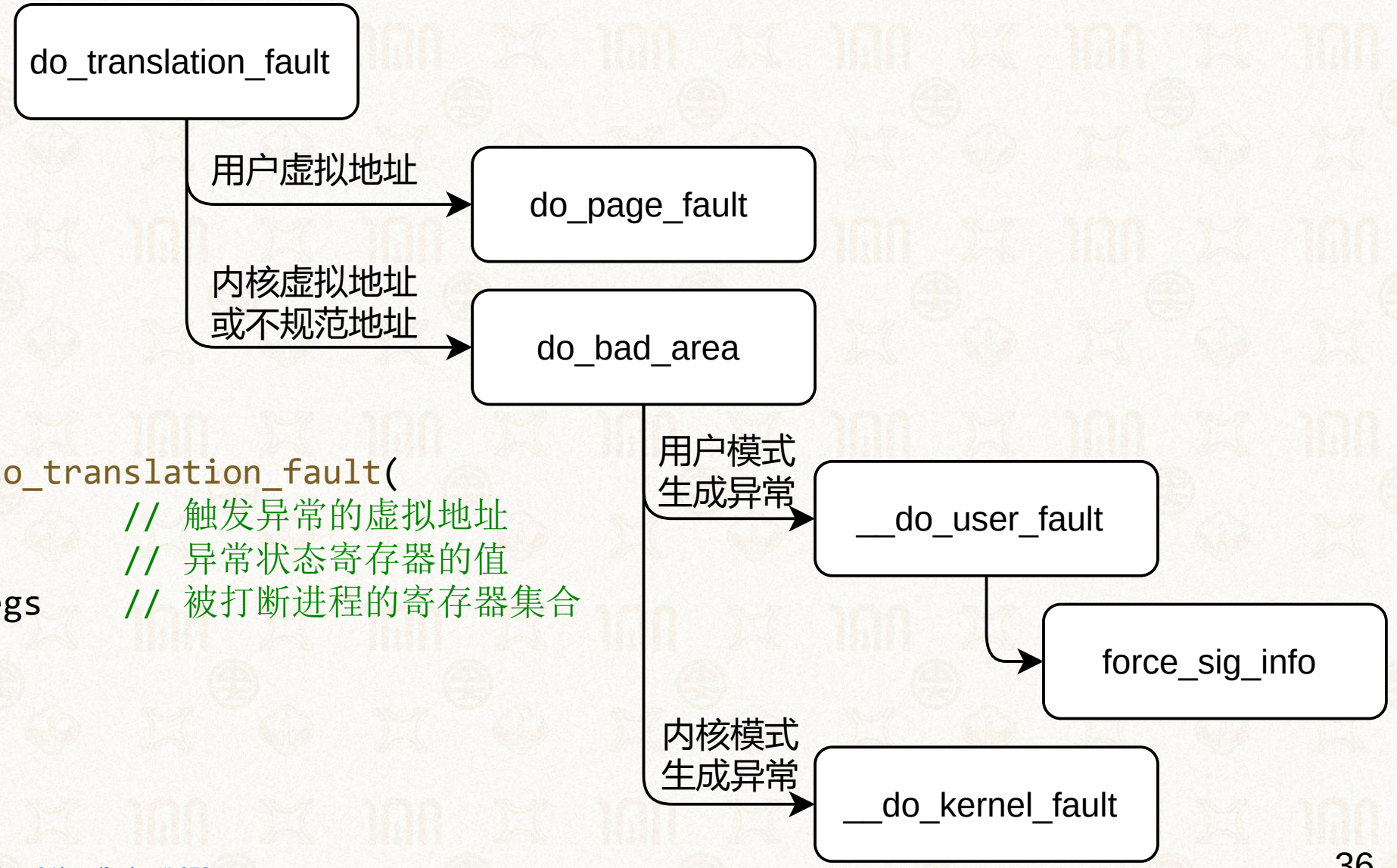


https://elixir.bootlin.com/linux/v6.0/source/include/linux/mm_types.h#L486

大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

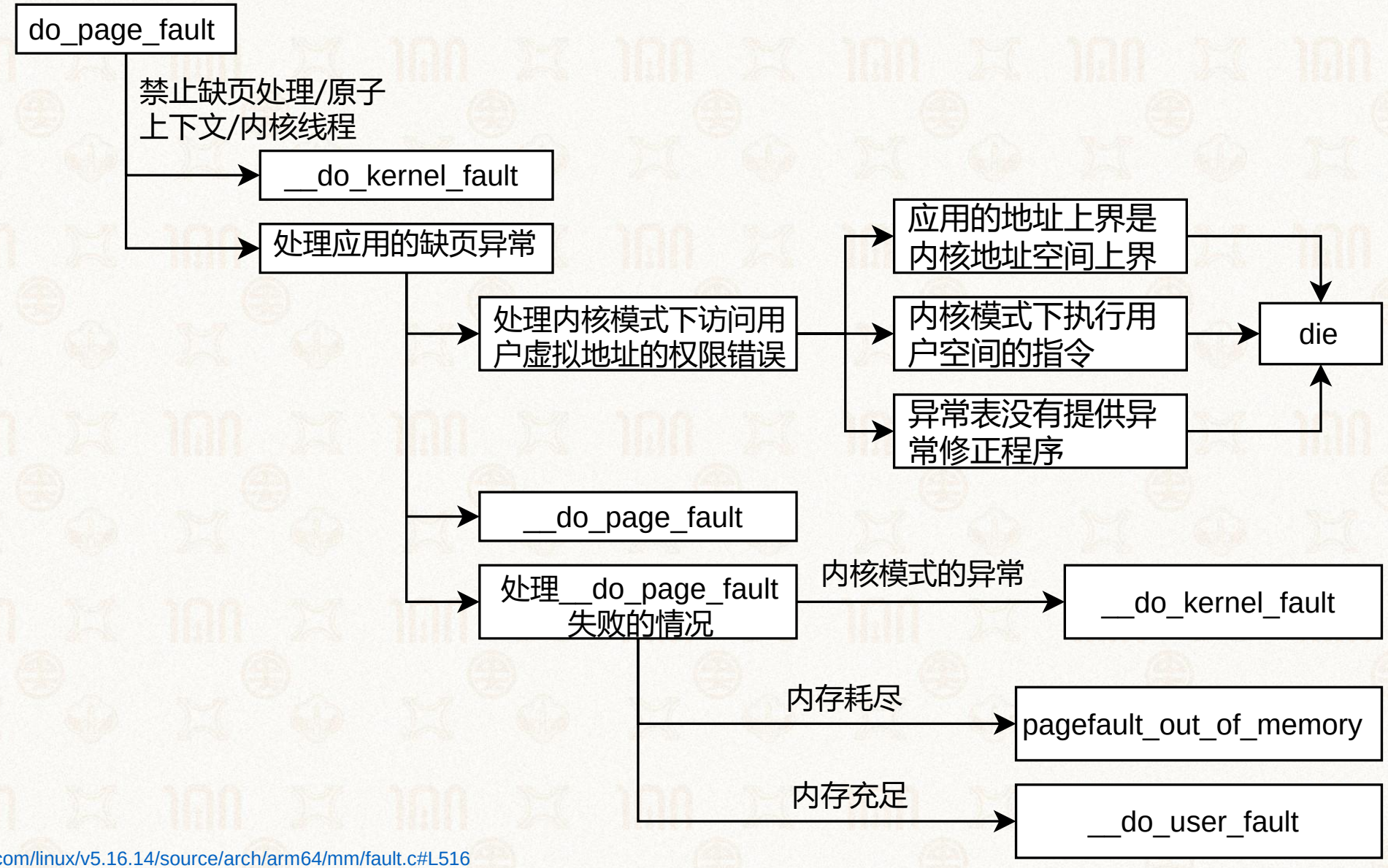
Linux内核处理缺页异常：do_translation_fault




```
static int __kprobes do_translation_fault(
    unsigned long far,       // 触发异常的虚拟地址
    unsigned int esr,        // 异常状态寄存器的值
    struct pt_regs *regs     // 被打断进程的寄存器集合
) {}
```

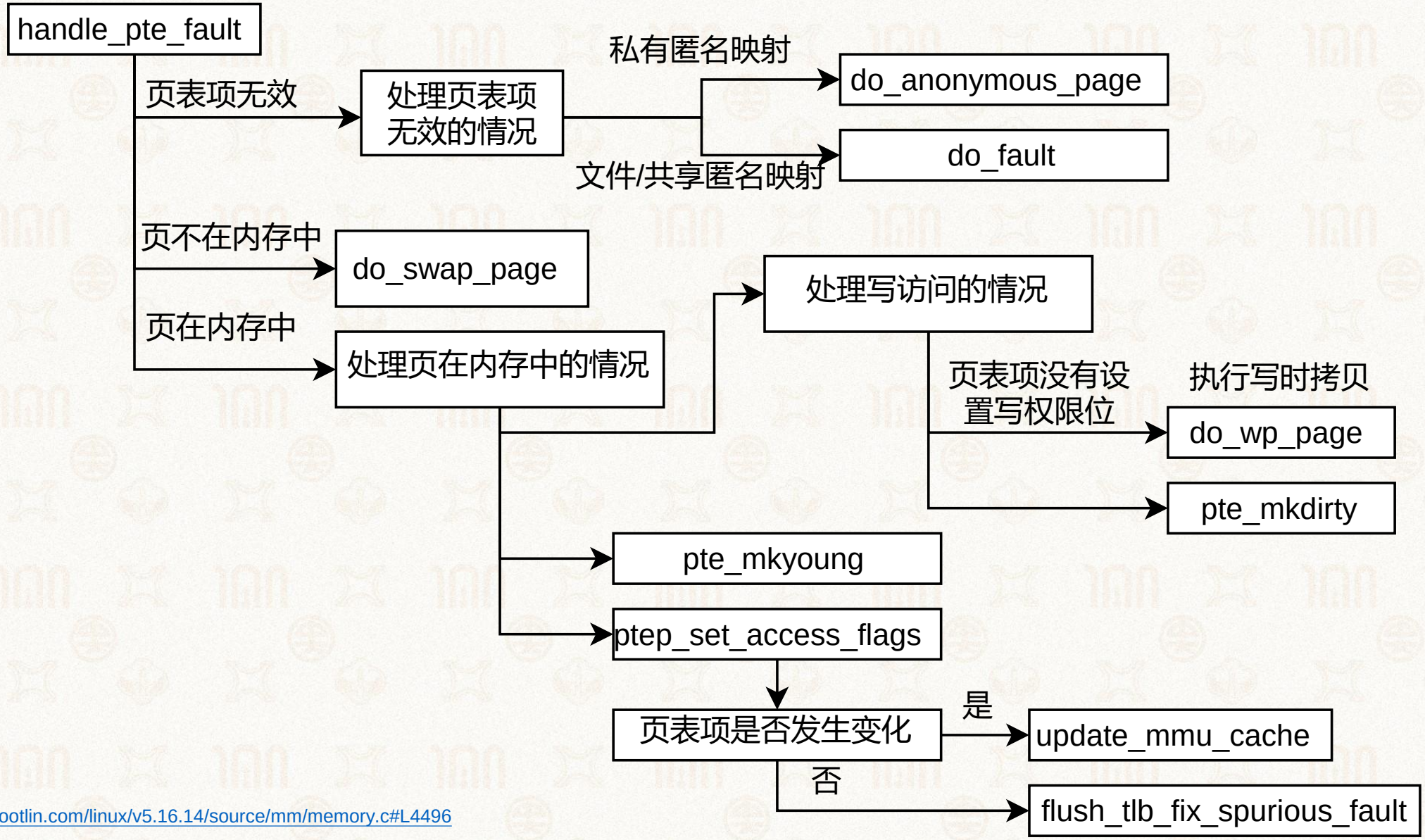
https://elixir.bootlin.com/linux/v5.16.14/source/arch/arm64/mm/fault.c#L673

Linux内核处理缺页异常：do_page_fault



https://elixir.bootlin.com/linux/v5.16.14/source/arch/arm64/mm/fault.c#L516

Linux内核处理缺页异常：handle_pte_fault



https://elixir.bootlin.com/linux/v5.16.14/source/mm/memory.c#L4496

缺页异常与 Segmentation Fault

- 虚拟地址需要分配后才能使用
  - 分配但未使用，第一次访问时触发缺页异常
  - 未分配也未使用，第一次访问时触发Segmentation Fault

```c
#include <stdio.h>

int main() {
    char *p = NULL;
    printf("%s\n", p);
    return 0;
}
```

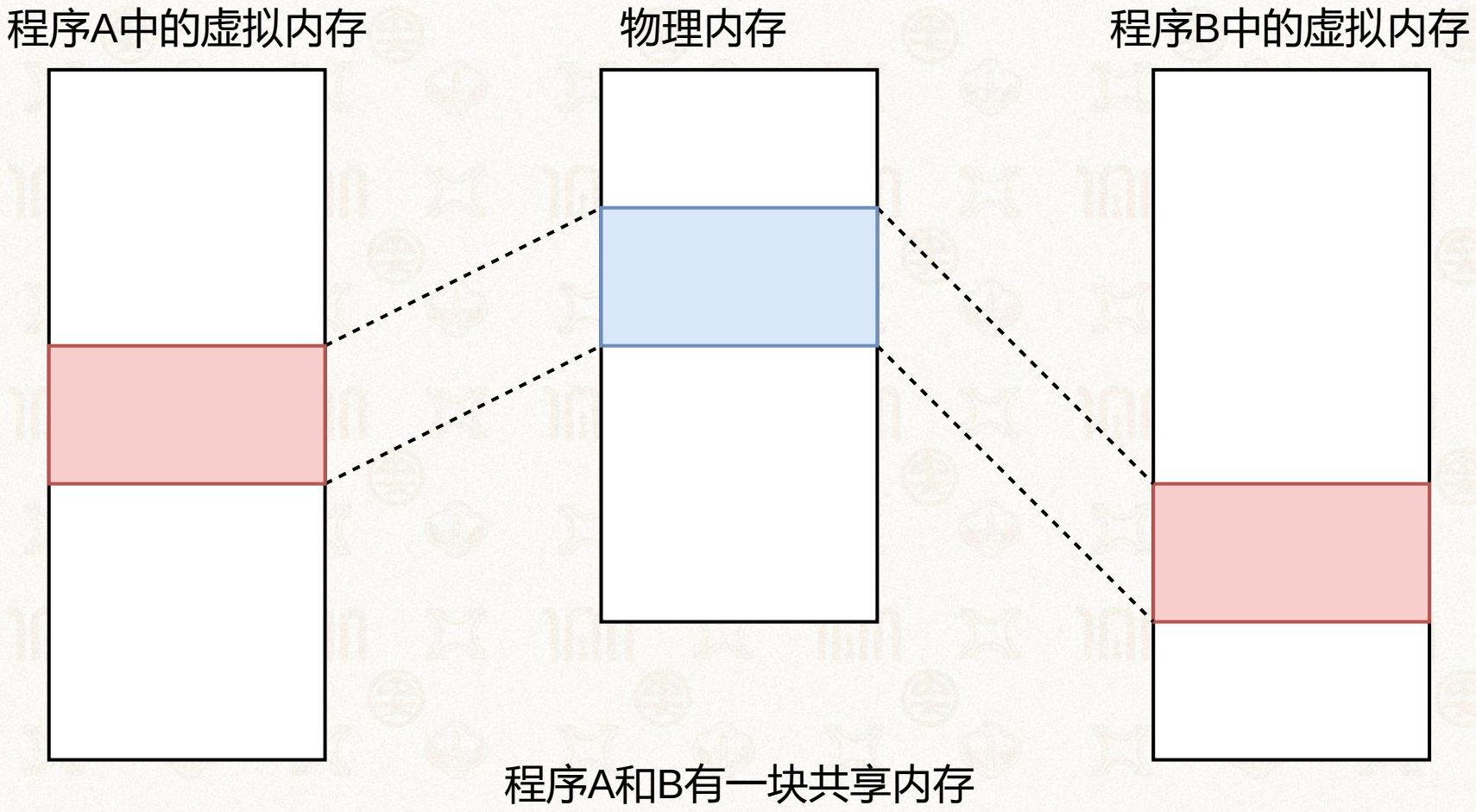
执行后显示：
Segmentation fault

大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

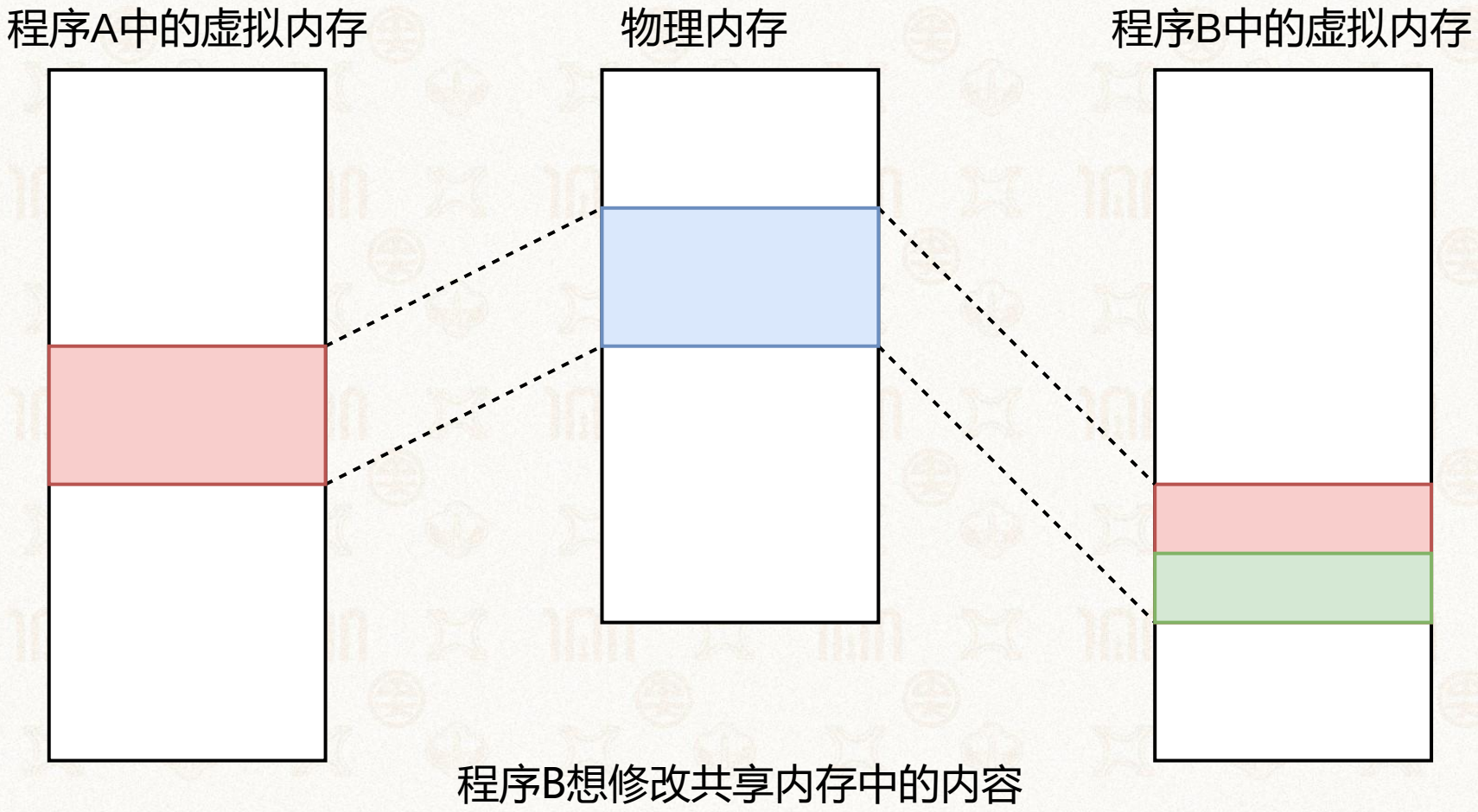
写时拷贝（copy-on-write）

- 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



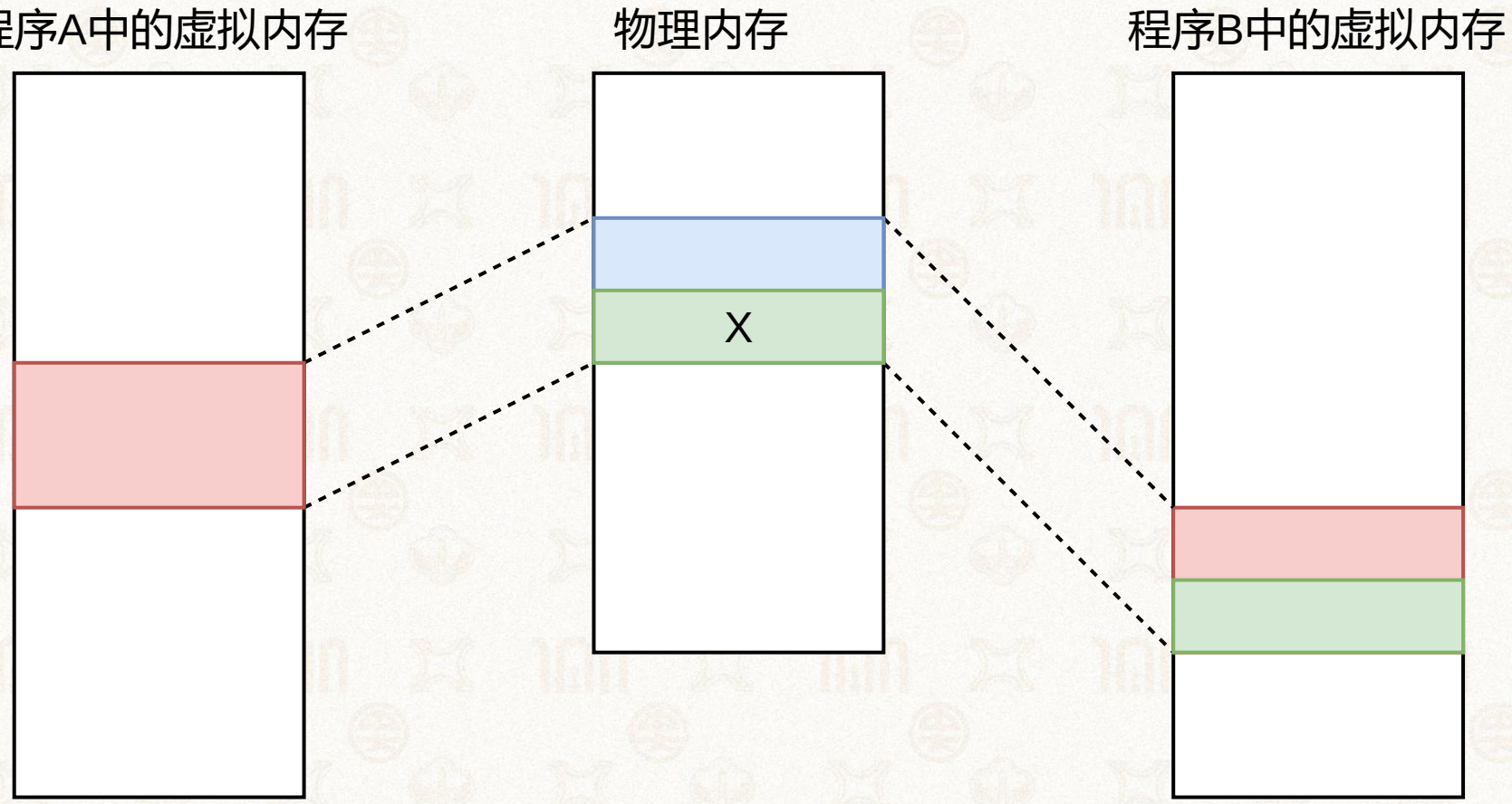
写时拷贝（copy-on-write）

- 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



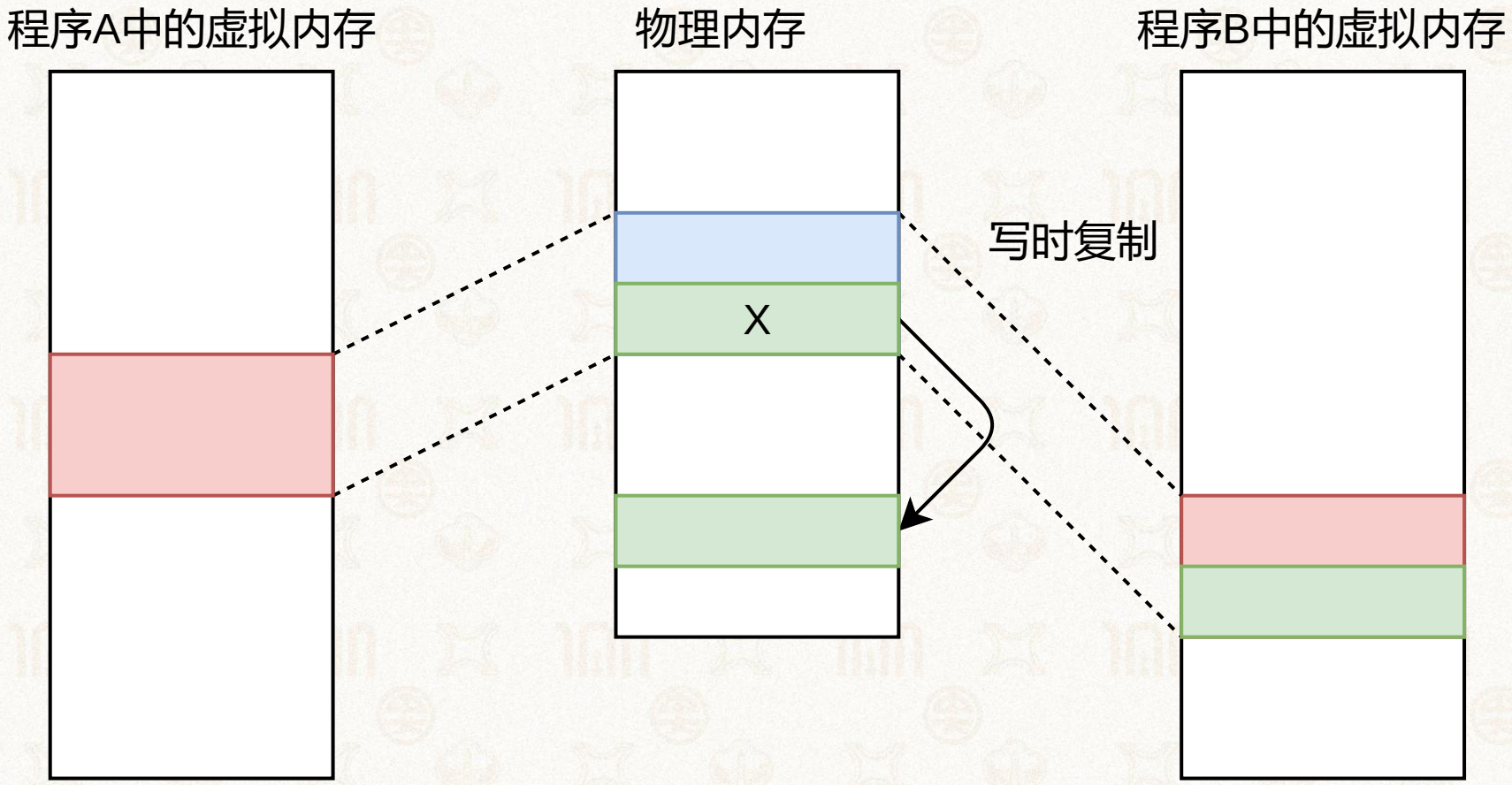
写时拷贝（copy-on-write）

- 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
- 此时禁止在共享物理空间内直接写入，
  - 页表项里有权限控制



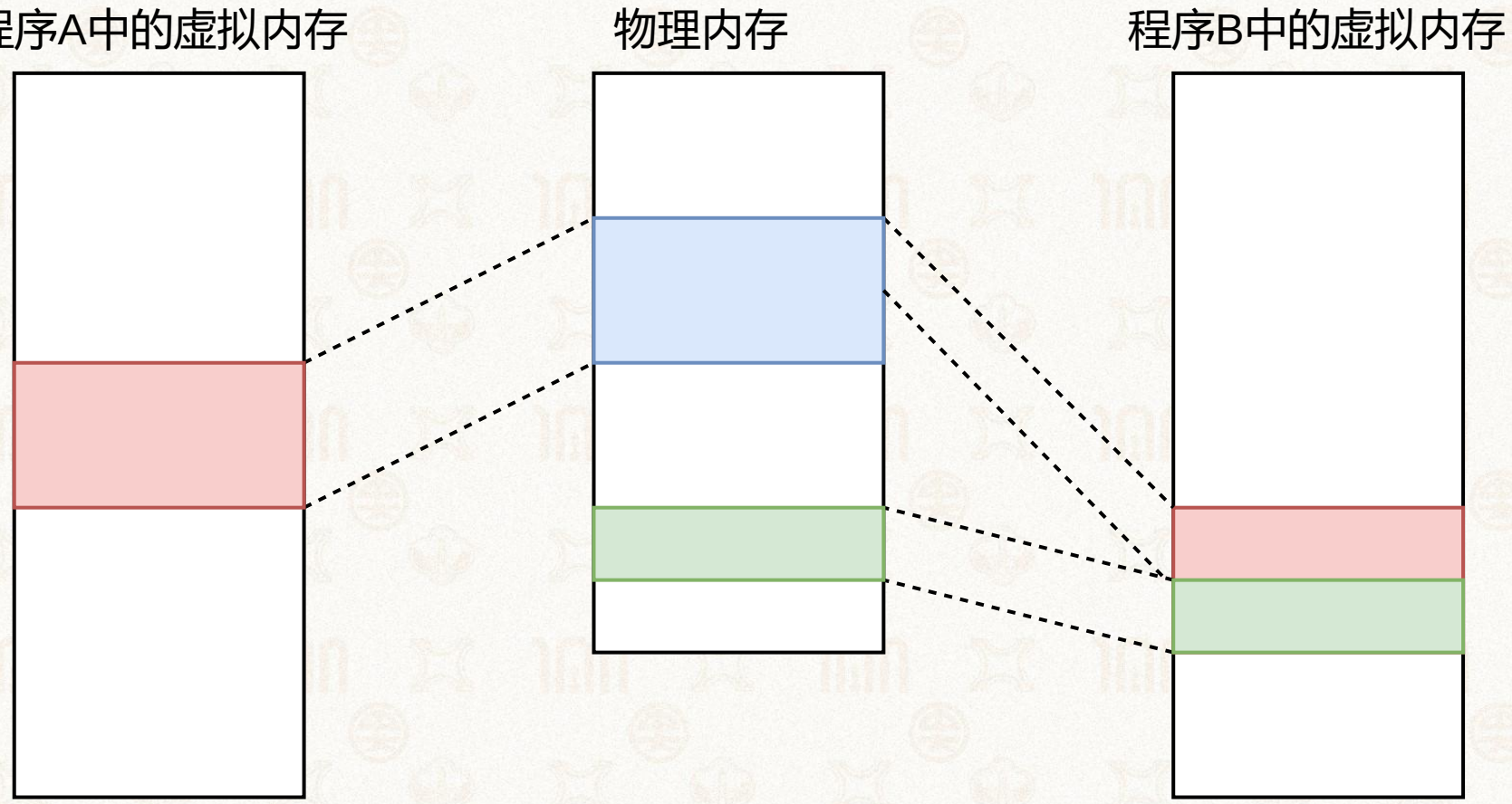
写时拷贝（copy-on-write）

- 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
- 写操作触发内存拷贝



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
- 写操作触发重映射



以写时拷贝的方式共享内存

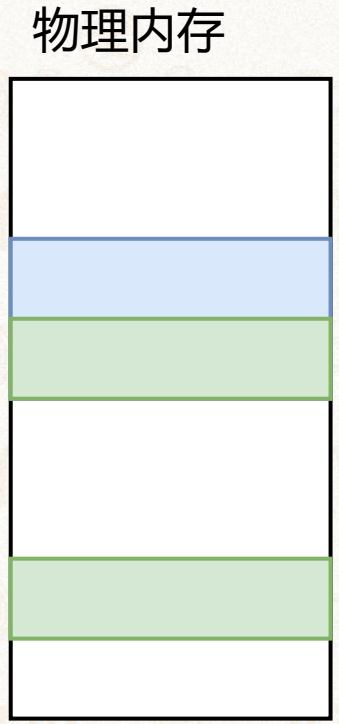
大纲

- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

内存去重

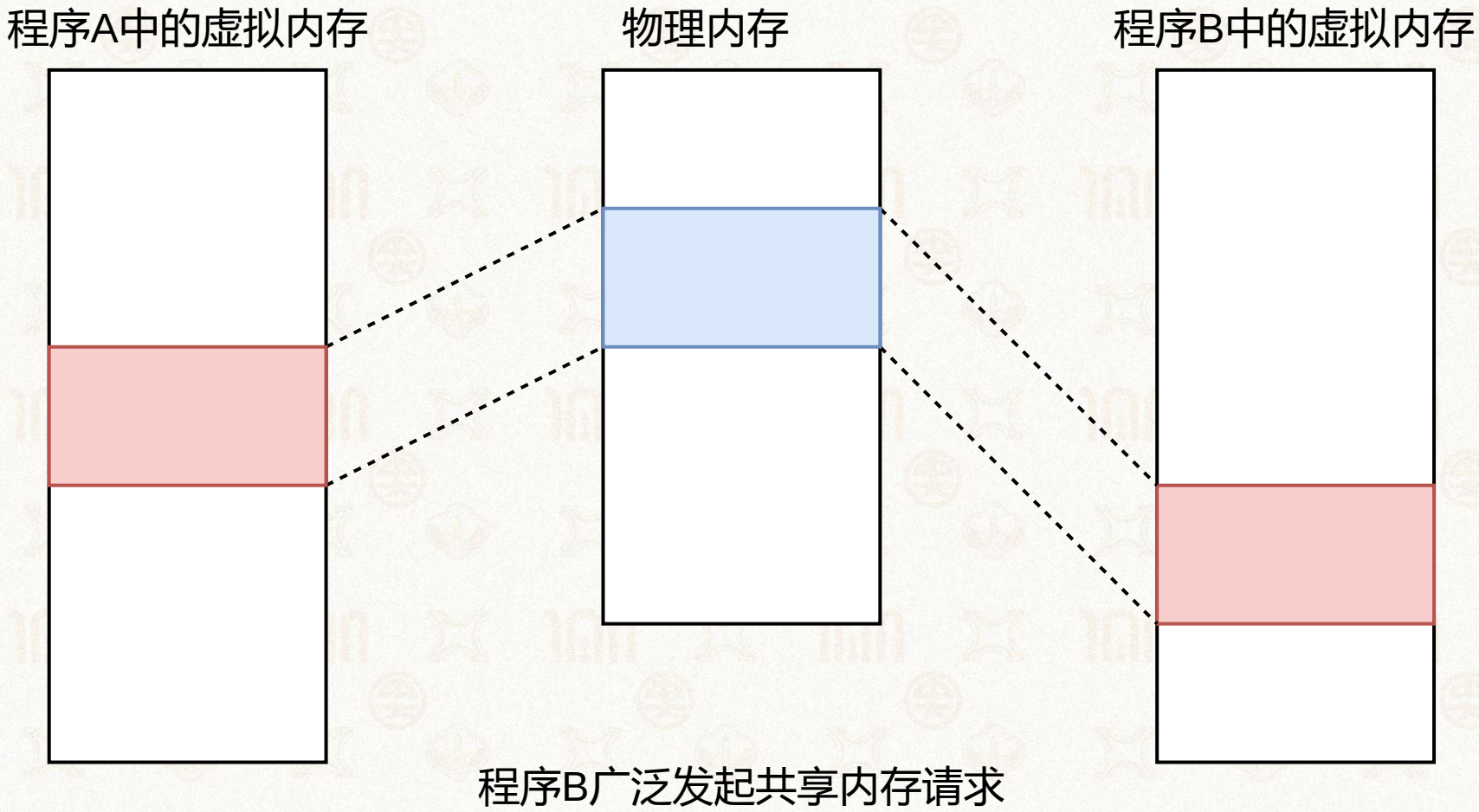
- memory deduplication
  - 基于写时拷贝机制
  - 在内存中扫描发现具有相同内容的物理页面
  - 执行去重
  - 操作系统发起，对用户态透明

- 典型案例：Linux KSM
  - kernel same-page merging



内存去重潜在安全隐患

- 导致新的side channel
  - 访问被合并的页会导致访问延迟明显
- 潜在攻击
  - 攻击者可以确认目标进程中含有构造数据



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

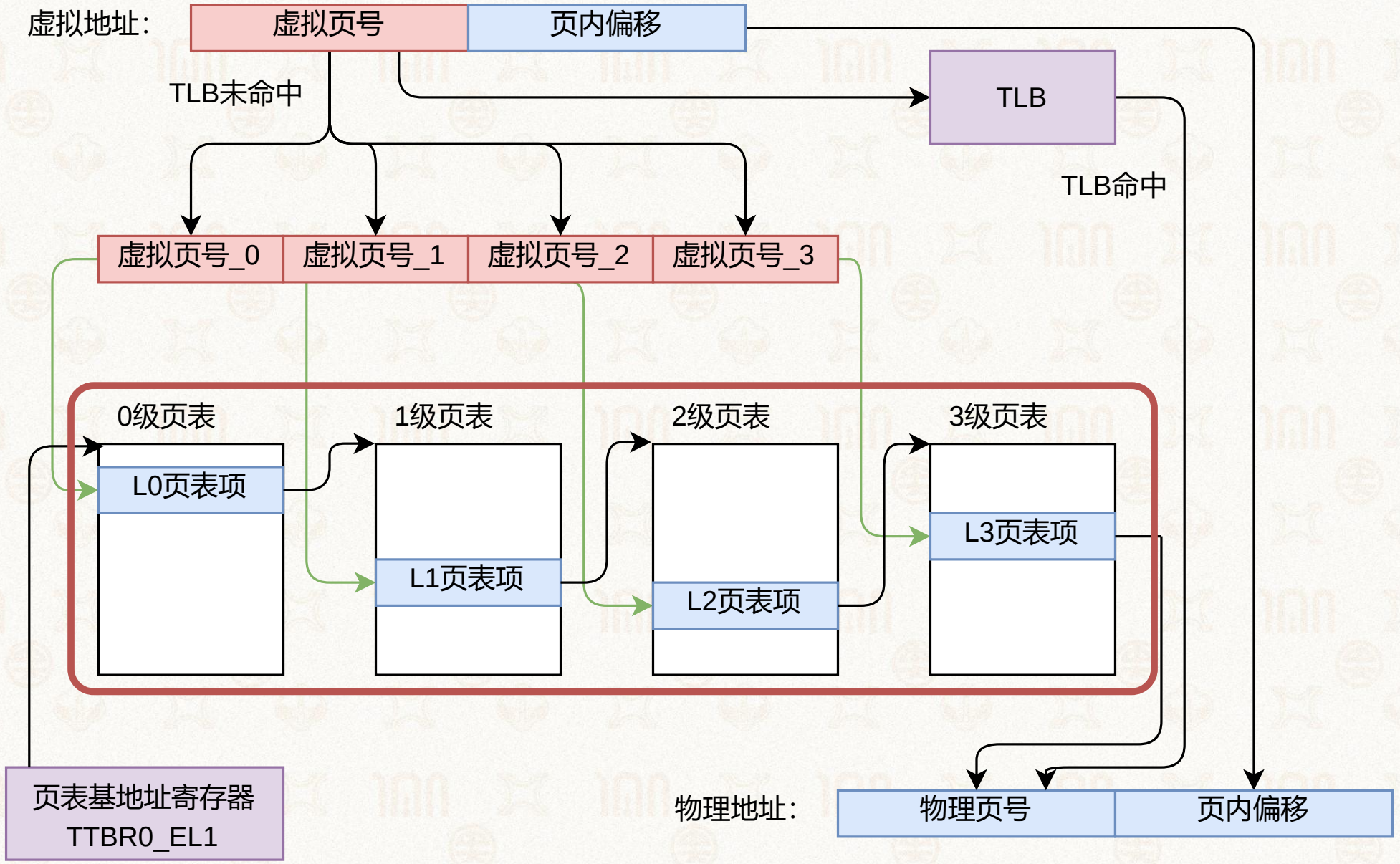
- Windows 10
  - 压缩后的数据仍然存放在内存中
  - 当访问被压缩的数据时，操作系统将其解压即可
  - 思考：对比交换内存页到磁盘？
- Linux
  - zswap：换页过程中磁盘的缓存
  - 将准备换出的数据压缩并先写入 zswap 区域 （内存）
  - 好处：减少甚至避免磁盘I/O；增加设备寿命

  - 等价于：传输多个小文件和传输一个大小类似的大文件，哪个快？

大纲

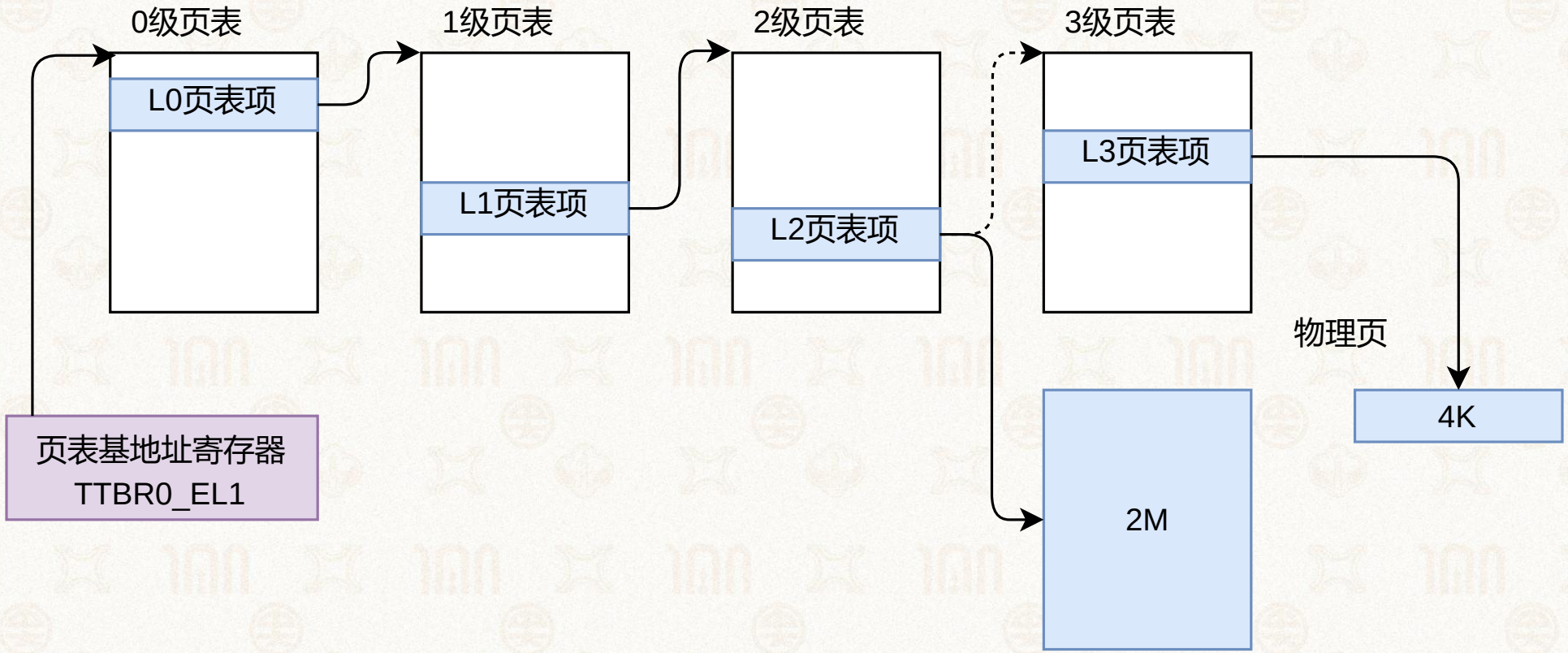
- 直接映射
- 虚拟内存段分布
- 管理页表映射
  - 立即映射
  - 延迟映射
  - 缺页异常
- 扩展功能
  - 共享内存
  - 内存压缩
  - 大页

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
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
页表基地址寄存器
TTBR0_EL1
物理地址:
物理页号
页内偏移

大页(huge page)

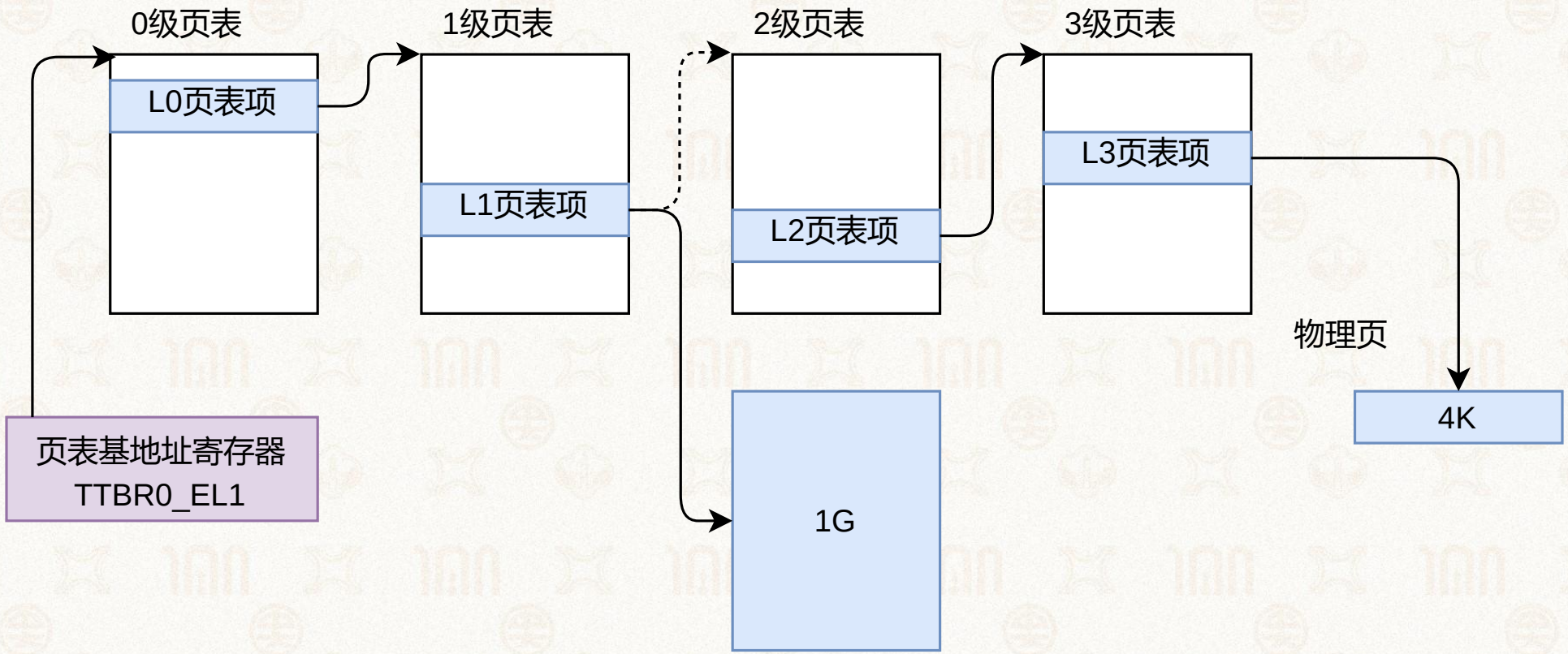
- 在4级页表中，某些页表项只保留两级或三级页表
- L2页表项的第1位
  - 标识着该页表项中存储的物理地址（页号）是指向 L3 页表页（该位是 1）
  - 还是指向一个 2M 的物理页（该位 是 0）



大页(huge page)

- 在4级页表中，某些页表项只保留两级或三级页表
- L1页表项的第1位
  - 类似地，可以指向一个 1G 的物理页

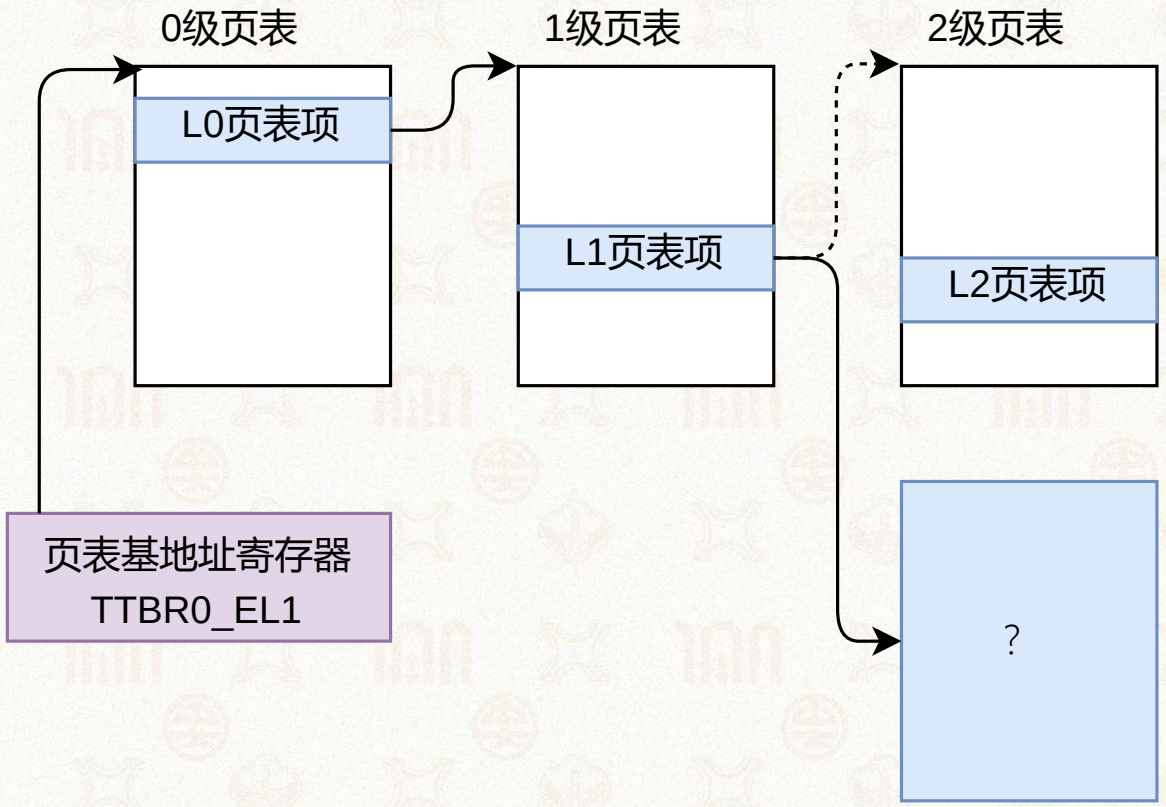
为什么是1G和2M?



单选题 1分

设置

如果只有3级页表映射（36位页号均分为3等份），那么L1页表项指向的大页空间为多少?



A 8M 字节

B 16M 字节

C 2G 字节

D 4G字节

提交

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式的大页分配
  - 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

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
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫