第4季

RISC-V内存管理

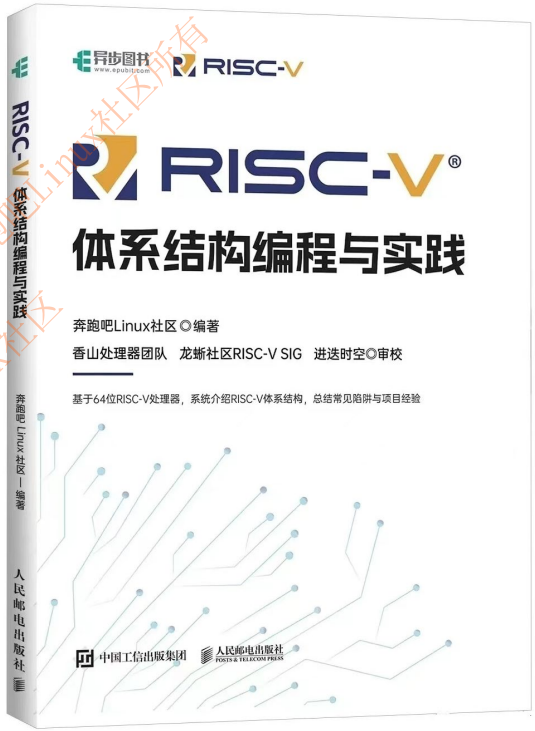
本节课主要内容

- 本章主要内容
  - 内存管理背景知识
  - Sv39页表
  - Sv48页表
  - PMA与PMP
  - 案例分析1：建立恒等映射
  - 案例分析2：图解页表创建过程
  - 6个实验

技术手册：

1. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203
2. SiFive U74-MC Core Complex Manual, 21G2.01.00

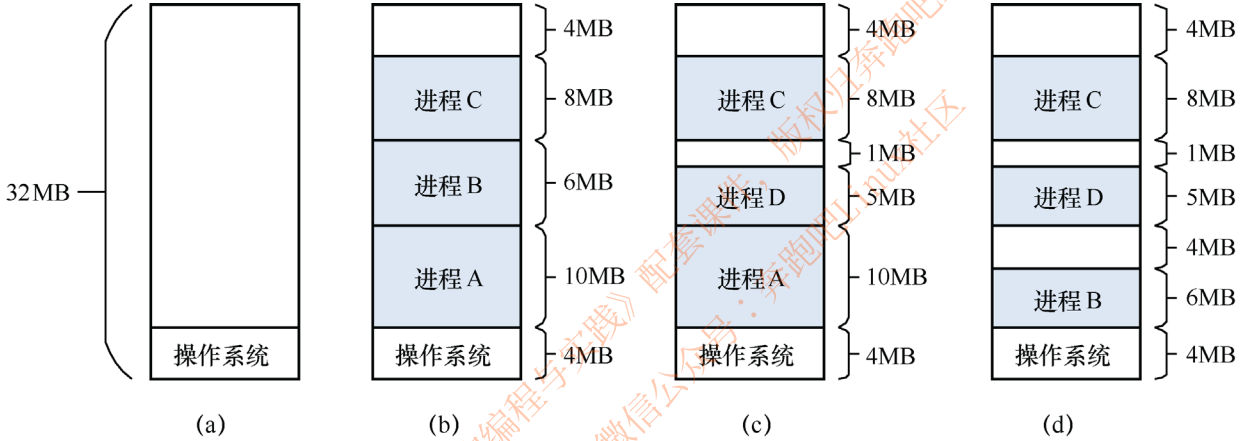
扫码订阅RISC-V视频课程



本节课主要讲解书上第10章内容

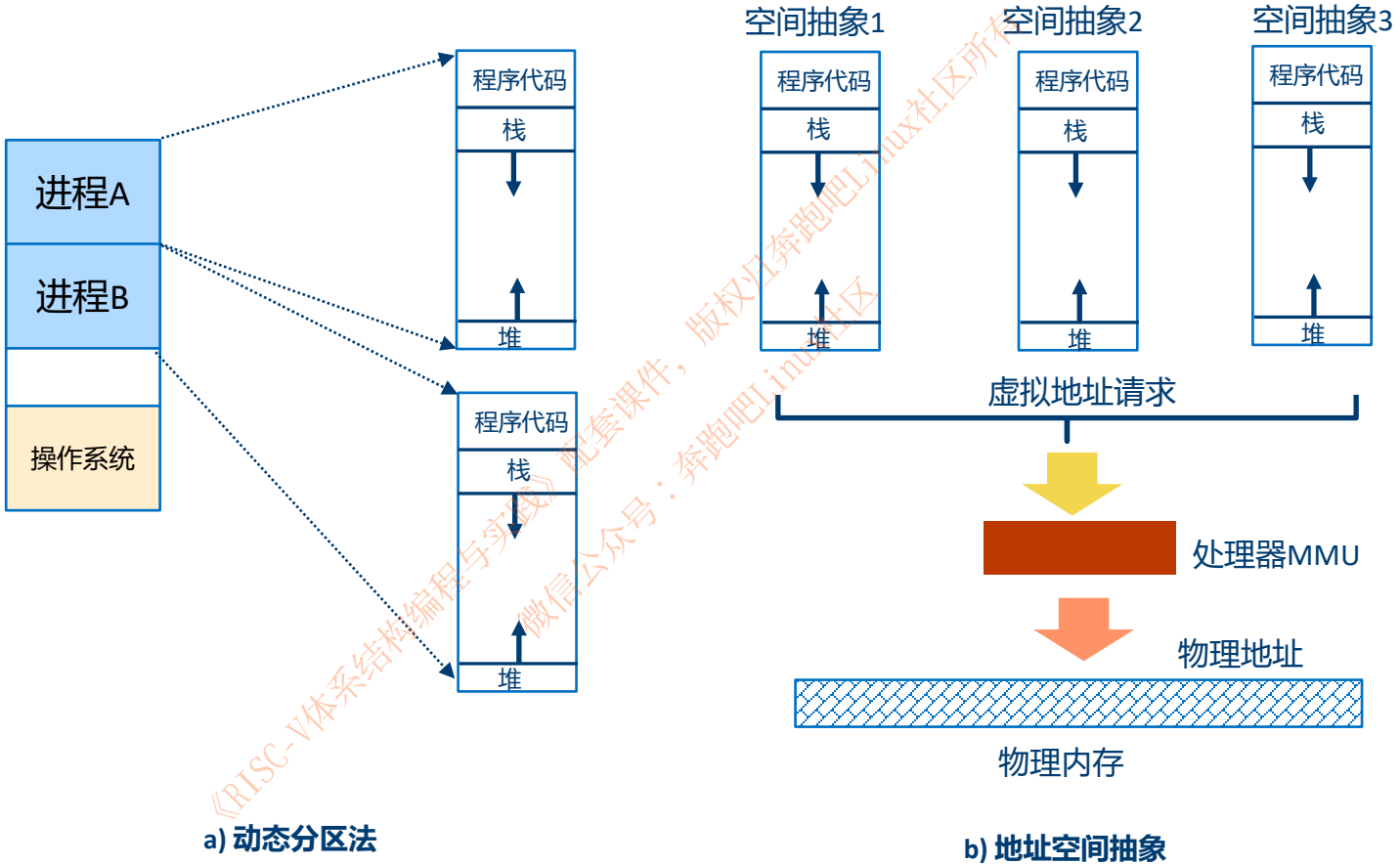
内存管理的发展历史

- 固定分区法和动态分区法



直接使用物理内存的缺点

- 进程地址空间保护问题。所有的用户进程都可以访问全部的物理内存，所以恶意的程序可以修改其他程序的内存数据，
- 内存使用效率低。如果即将要运行的进程所需要的内存空间不足，就需要选择一个进程进行整体换出，这种机制导致有大量的数据需要换出和换入，效率非常低下。
- 程序运行地址重定位问题。



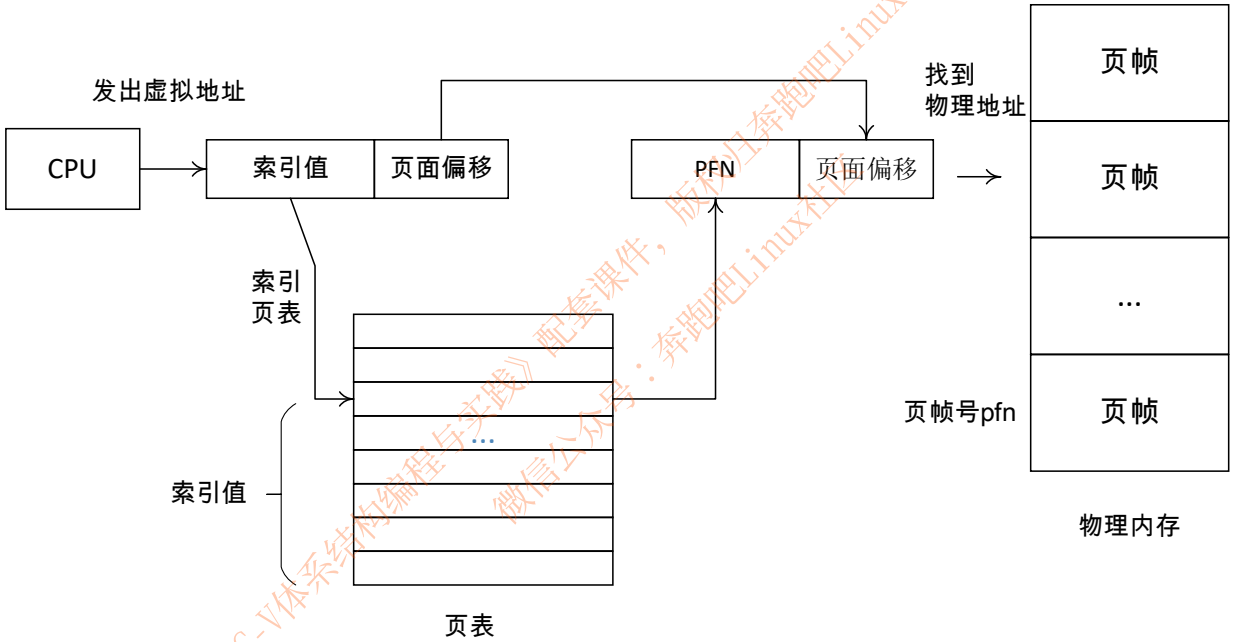
a) 动态分区法

b) 地址空间抽象

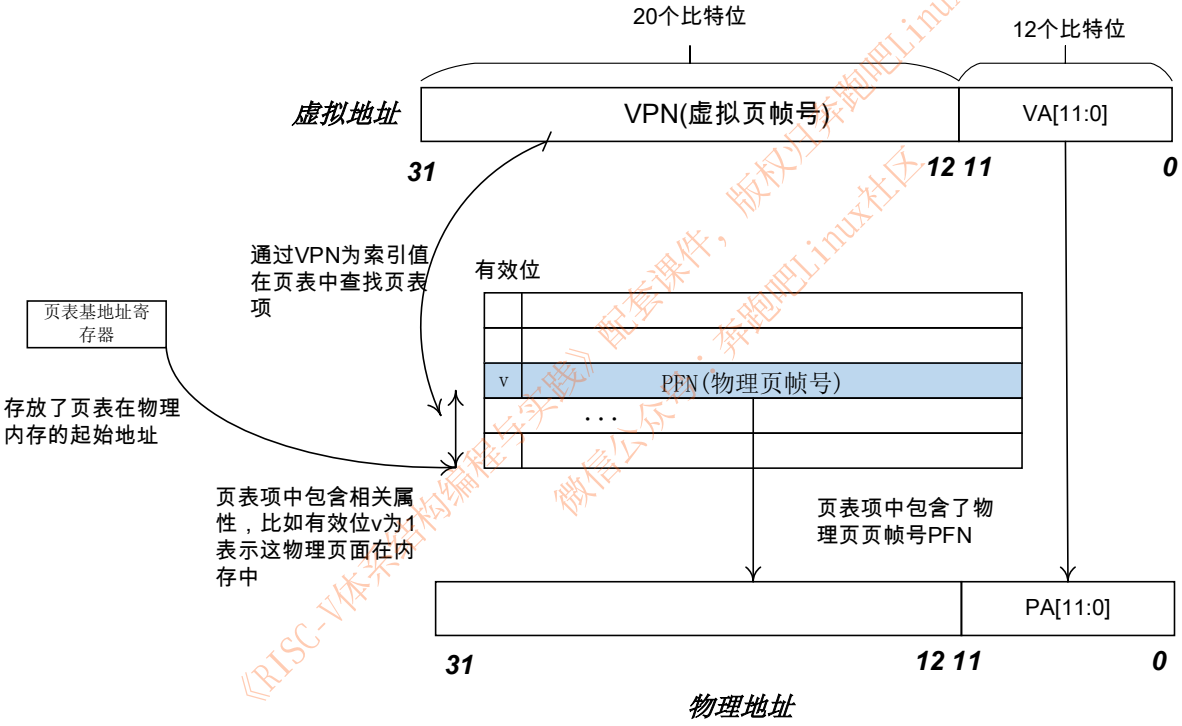
分页机制的基本概念

- 虚拟存储器（Virtual Memory）
- 虚拟地址空间（Virtual Address）
- 物理存储器（Physical Memory）
- 页帧（Page Frame）
- 虚拟页帧号VPN（Virtual Page Frame Number）
- 物理页帧号PFN（Physical Frame Number）
- 页表（Page Table，PT）
- 页表项（Page Table Entry，PTE）

虚拟地址到物理地址映射过程



一级页表示意图



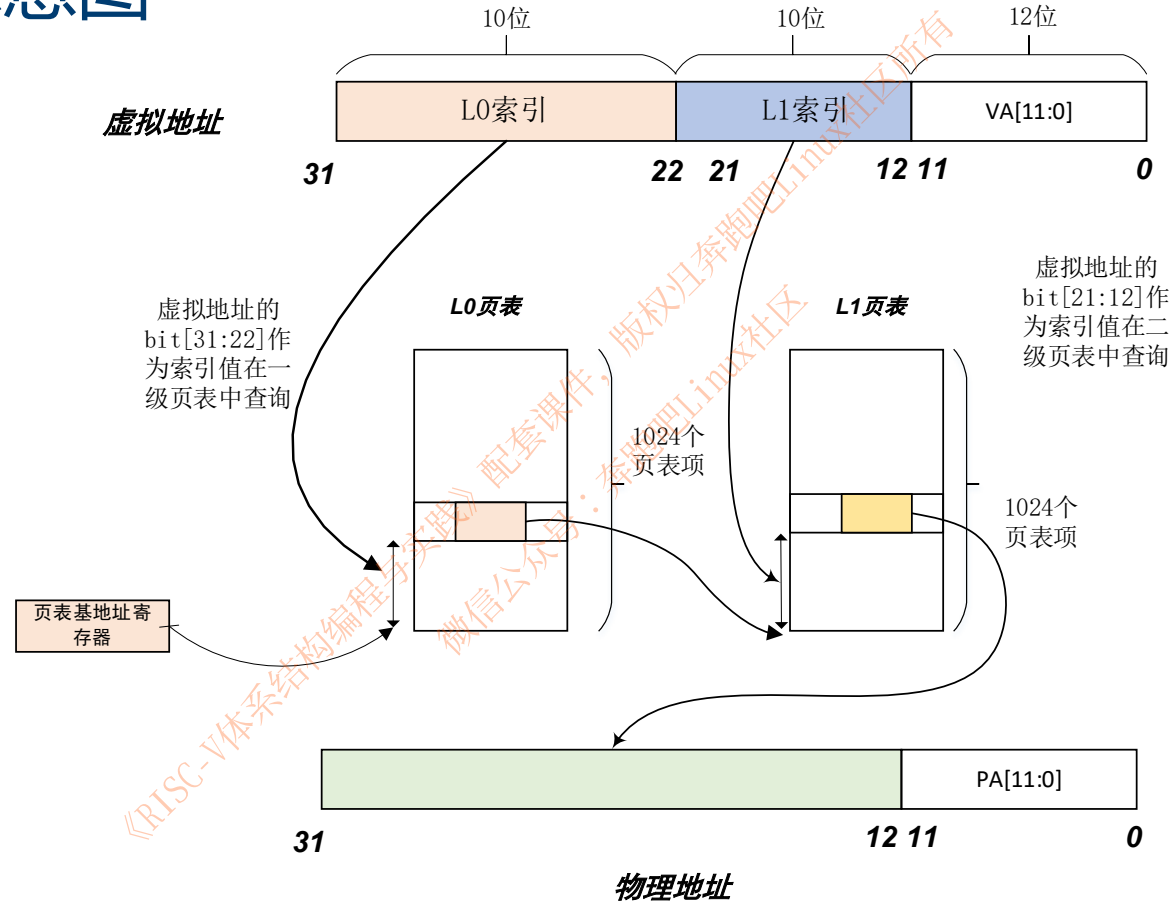
采用一级页表的缺点

- 处理器采用一级页表，虚拟地址空间位宽是32位，寻址范围是4GB大小，物理地址空间位宽也是32比特，最大支持4GB物理内存，另外页面的大小是4KB。为了能映射整个4GB地址空间，那么需要4GB/4KB=1M个页表项，每个页表项占用4字节，则需要**4MB**大小的物理内存来存放这张页表。

- 每个进程拥有了一套属于自己的页表，在进程切换时需要切换页表基地址。如上述的一级页表，每个进程需要为其分配4MB的连续物理内存来存储页表，这是不能接受的，因为这样太浪费内存了。

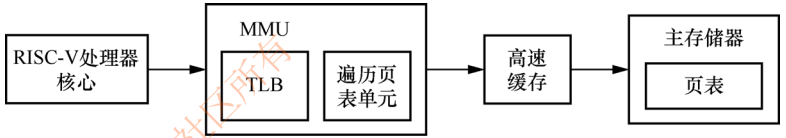
- 多级页表：按需一级一级映射，不用一次全部映射所有地址空间。

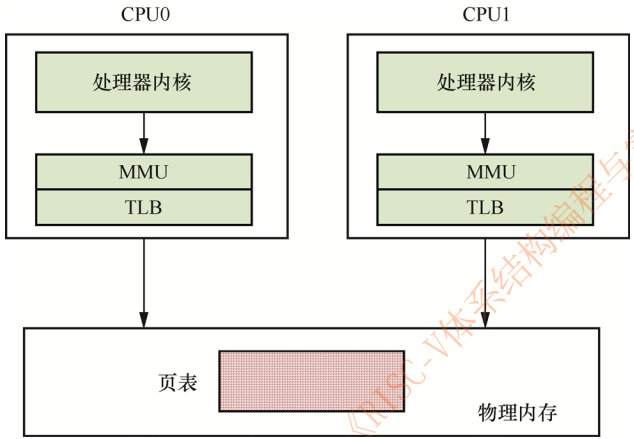
二级页表示意图

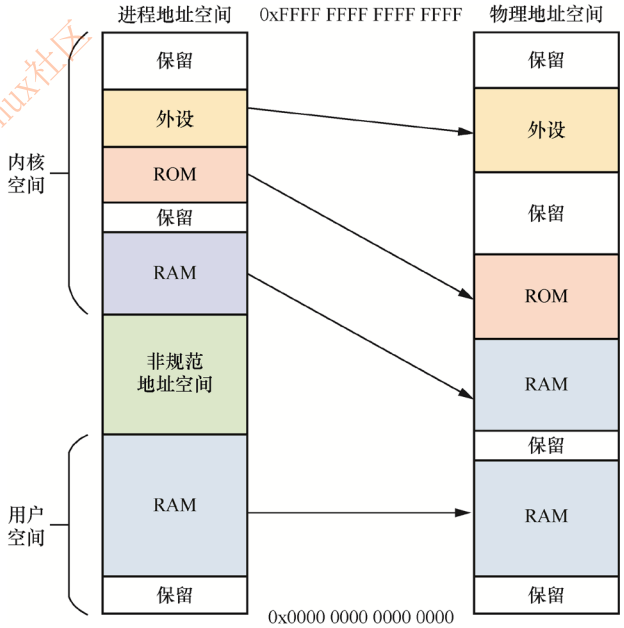


RISC-V内存管理

- MMU包括：页表遍历单元 + TLB
- 在SMP中，每个CPU core内置了MMU和TLB硬件单元
- 进程地址空间映射到物理地址空间







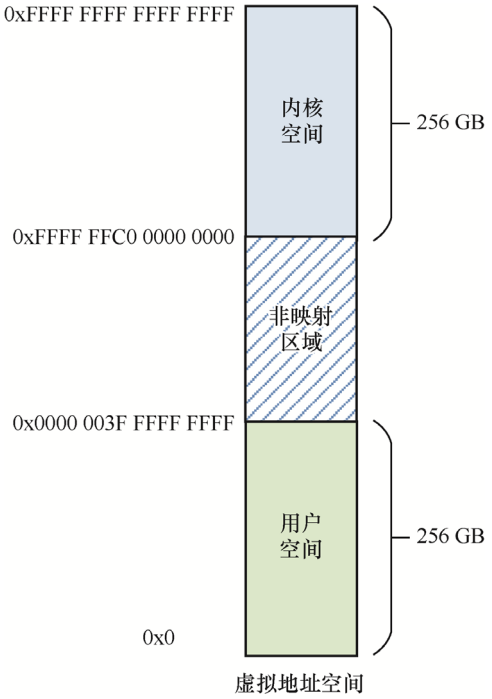
RISC-V页表机制

- Sv32：仅支持32位RSIC-V处理器，二级页表，支持32位虚拟地址转换。
- Sv39：支持64位RSIC-V处理器，三级页表，支持39位虚拟地址转换。
- Sv48：支持64位RSIC-V处理器，四级页表，支持48位虚拟地址转换。

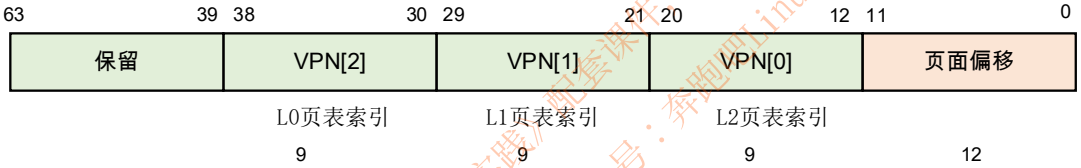
- 支持4 KB大小的页面（page）粒度，也支持2 MB、1 GB大小的块（block）粒度，也称为大页（huge page）

Sv39页表

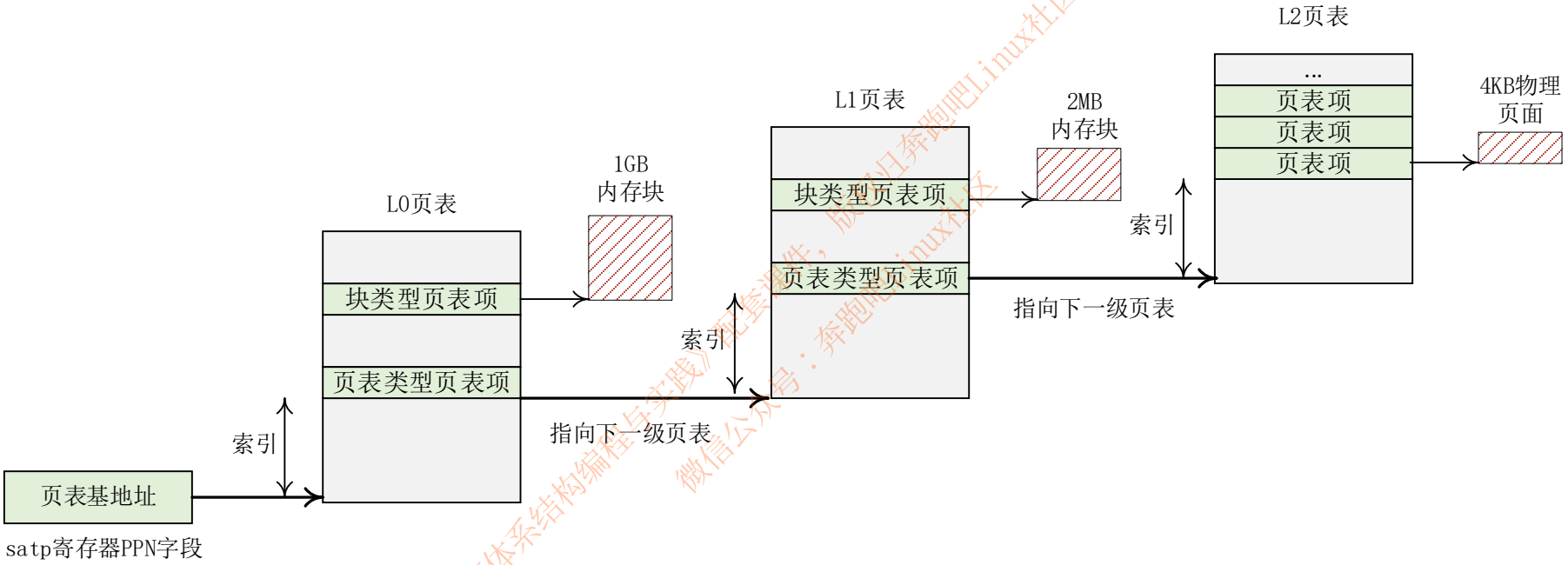
- 39位虚拟地址 -> 56位物理地址
- 64位的虚拟地址中只有低39位用于页表索引，剩余的高位必须和第38位相等
  - 底部256 GB （ Bit[63:38] 全为0），用于用户空间。
  - 高端256GB （ Bit[63:38] 全为1）， 用于内核空间。
  - 中间部分为非映射区域，即Bit[63:38]不全为0或者不全为1。处理器访问该区域会触发缺页异常。



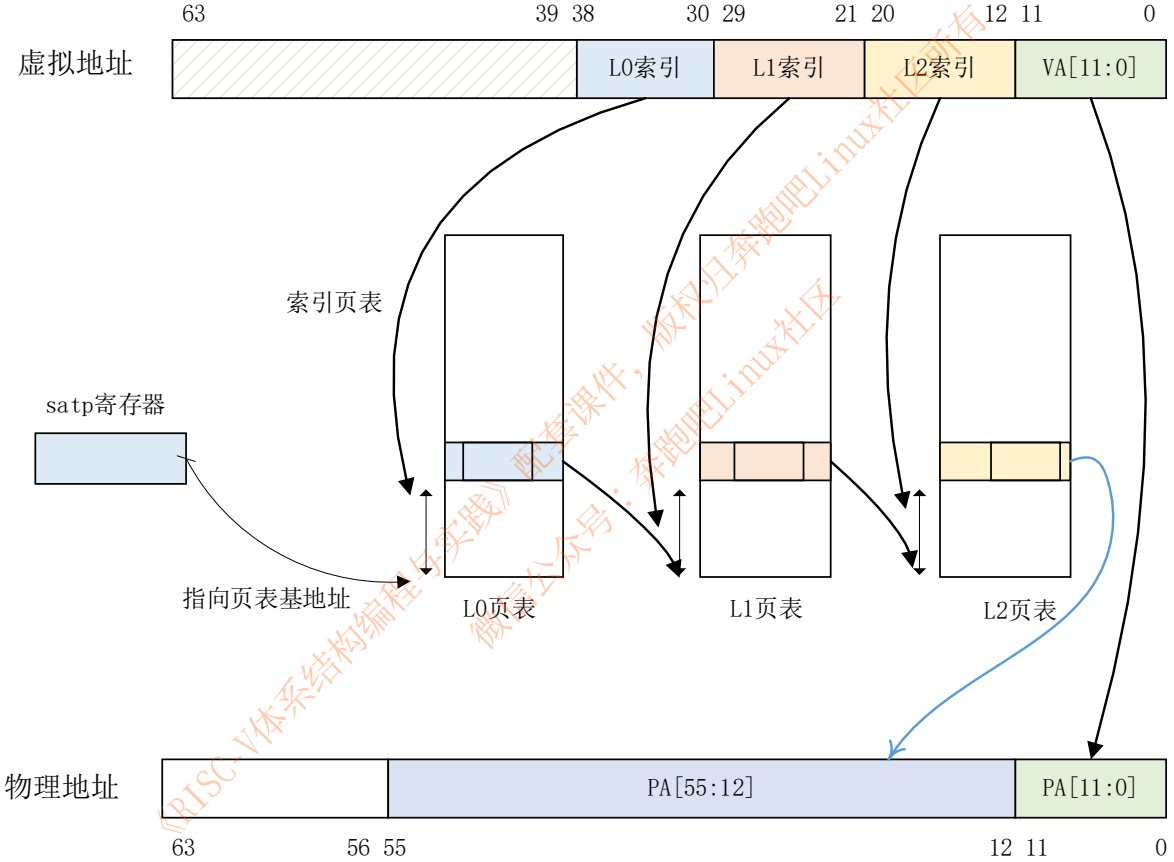
虚拟地址划分



Sv39页表



Sv39页表



MMU查询页表的过程（理想状态）

（1）从satp获取L0页表基地址
（2）查找L0页表
（3）得到L1页表基地址
（4）查询L1页表
（5）得到L2页表基地址
（6）查询L2页表
（7）L2页表的表项里存放着4 KB页面的页帧号，然后加上VA[11:0]，就构成了新的物理地址

MMU查询页表可能出现的异常

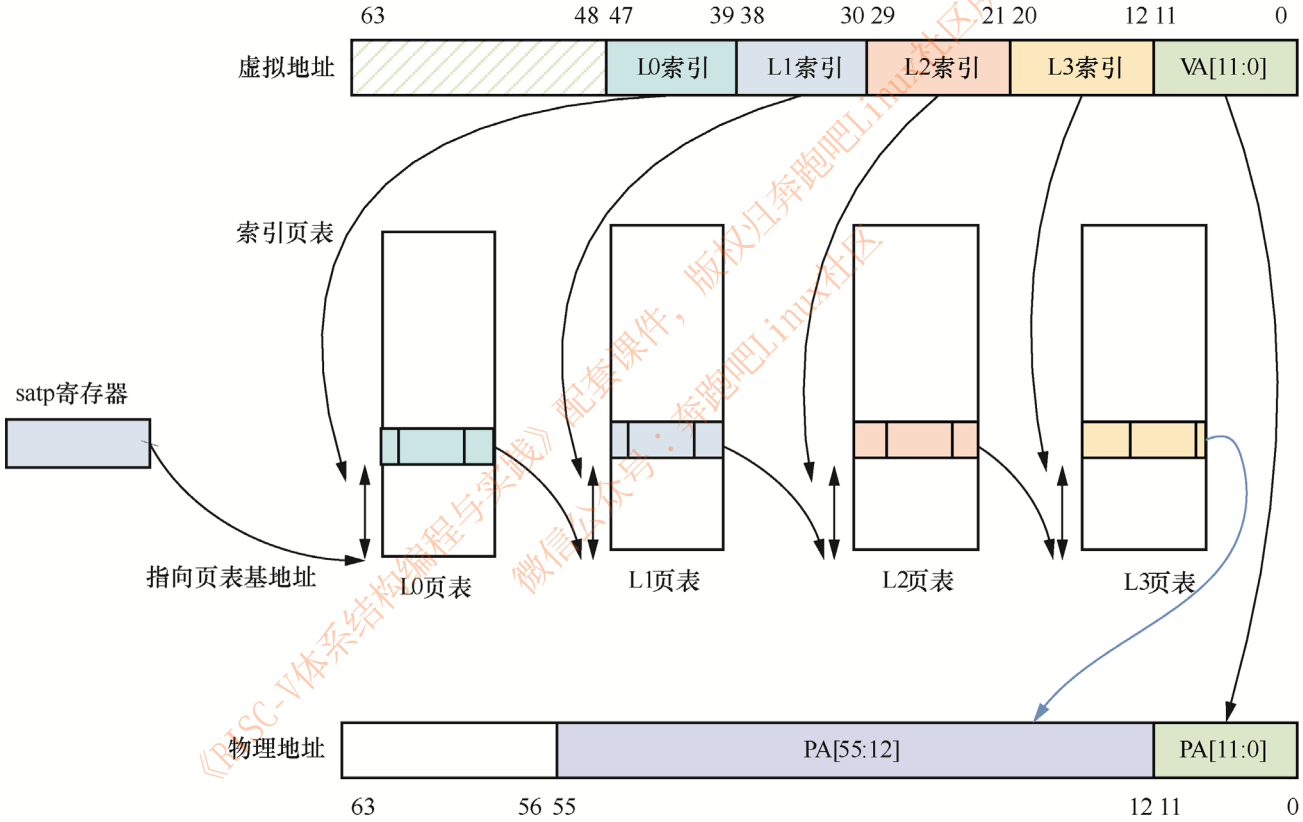
1) PMA或者PMP机制做内存属性相关的检查。触发内存访问异常（access-fault exception）

2) 页表项有效性检查，如发现页表项是无效的，比如V=0或者保留的访问权限（如R=0 && W=1），那么处理器会触发缺页异常。

3) 子叶页表项描述符的权限检查。触发缺页异常。

4) 假设处理器采用软件方式处理A和D标志位，当处理器访问页面时，如果该页面对应的子叶页表项描述符中的访问标志位为0（A=0）或者该访问是存储操作并且脏位为0（D=0），则会触发缺页异常

Sv48页表映射

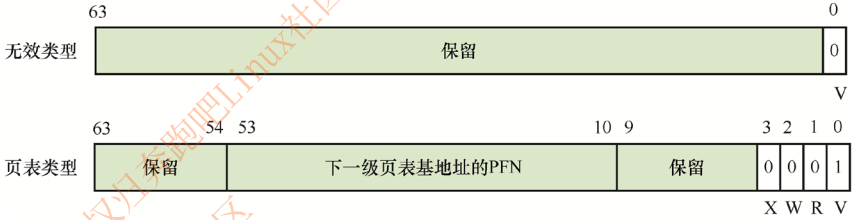


页表项 1

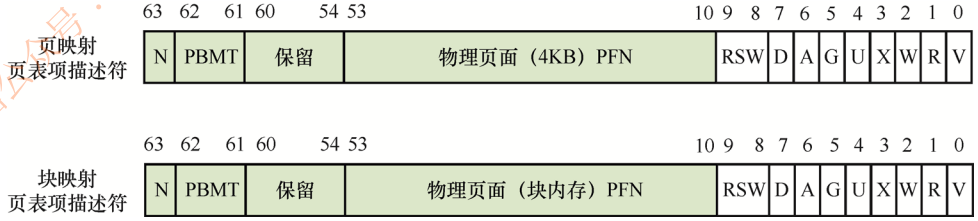
- 页表项：各级页表中的表项
- 使用页表项描述符（page table descriptor）来描述页表项中的内容
- Sv39模式以及Sv48模式，页表项描述符占8字节。格式一样，但是内容不完全一样。

页表项 2

- 无效类型：Bit[0] = 0
- 非子叶类型：Bit[3:1] = 0
  - ✓ 非子叶页表项（non-leaf page table）
  - ✓ 页表项描述符包含指向下一级页表基地址的页帧号。
  - ✓ 页表项描述符中的Bit[3:1]都为0
- 子叶页表类型：Bit[3:1] != 0
  - ✓ 子叶页表项（leaf page table）
  - ✓ 页表项描述符包含指向最终物理地址的字段
  - ✓ 页表项描述符中的Bit[3:1不 为0
- 子叶页表的属性
  - ✓ 低位属性：由Bit[9:0]组成的低位属性
  - ✓ 高位属性：由Bit[63:54]组成的高位属性。



非子叶页表项



子叶页表项

页表项描述符中的低位属性

- 子叶页表的属性
  - ✓ 低位属性：由Bit[9:0]组成的低位属性
  - ✓ 高位属性：由Bit[63:54]组成的高位属性。

表 10.1 页表项描述符中的低位属性

名称	位	描述
V	Bit[0]	有效位。 ❑ 1：表示页表项有效。 ❑ 0：表示页表项无效
R	Bit[1]	可读属性。 ❑ 1：表示页面内容具有可读属性。 ❑ 0：表示页面内容不具有可读属性
W	Bit[2]	可写属性。 ❑ 1：表示页面内容具有可写属性。 ❑ 0：表示页面内容不具有可写属性
X	Bit[3]	可执行属性。 ❑ 1：表示页面内容具有可执行属性。 ❑ 0：表示页面内容不具有可执行属性
U	Bit[4]	用户访问模式。 ❑ 1：用户模式可以访问该页面。 ❑ 0：用户模式不能访问该页面
G	Bit[5]	全局属性，常用于 TLB
A	Bit[6]	访问标志位。 ❑ 1：表示处理器访问过该页面。 ❑ 0：表示处理器没有访问过该页面
D	Bit[7]	脏位。 ❑ 1：表示页面被修改过。 ❑ 0：表示页面是干净的
RSW	Bit[9:8]	预留给系统管理员使用

Svpbmt扩展

- Svpbmt扩展将来用于替代物理内存属性（Physical Memory Attribute，PMA）机制

表 10.2　　页表项高位属性

名称	位	描述
PBMT	Bit[62:61]	用来表示映射页面的内存属性。 ❑ 0：无。 ❑ 1：表示普通内存，关闭高速缓存，支持弱一致性内存模型。 ❑ 2：表示 I/O 内存，关闭高速缓存，支持强一致性内存模型。 ❑ 3：保留
N	Bit[63]	连续块表项

页表项属性 – 访问权限

表 10.3 指定访问权限的字段

X 字段	W 字段	R 字段	说明
0	0	0	页表项指向下一级页表项描述符
0	0	1	只读属性页面
0	1	0	保留
0	1	1	可读、可写页面
1	0	0	只可执行的页面
1	0	1	可读、可执行页面
1	1	0	保留
1	1	1	可读、可写、可执行页面

在没有相应权限的页面中进行读、写或者执行代码等操作会触发缺页异常。

- ✓ 如果在没有可执行权限的页面中预取指令，触发预取缺页异常（fetch page fault）。
- ✓ 如果在没有读权限的页面加载数据，触发加载缺页异常（load page fault）。
- ✓ 如果在没有写权限的页面里写入数据，触发存储缺页异常（store page fault）。

访问标志位与脏标志位 1

- 页表项属性中有一个访问字段A（access），用来指示页面是否被访问过。
  - ✓ 如果A字段为1，表示页面已经被CPU访问过。
  - ✓ 如果A字段为0，表示页面还没有被CPU访问过。
- 页表项属性中的脏标志位（D）表示页面内容被写入或者修改过。
- 软件方式更新A和D标志位的方式如下：
  - ✓ 当CPU尝试访问页面并且A标志位为0时，会触发缺页异常，然后软件就可以设置A标志位为1。
  - ✓ 当CPU尝试修改或者写入页面并且D标志位为0时，会触发缺页异常，然后软件就可以设置D标志位为1。
- 硬件方式更新A和D标志位的方式如下：
  - ✓ 当CPU尝试访问页面并且该页面的A标志位为0时，CPU自动设置A标志位。
  - ✓ 当CPU修改或者写入页面并且该页面的D标志位为0时，CPU自动设置D标志位。
- 当采用硬件方式时，页表项（PTE）的更新必须是**原子的**，即CPU会原子地更新整个页表项，而不是仅仅更新某个标志位。

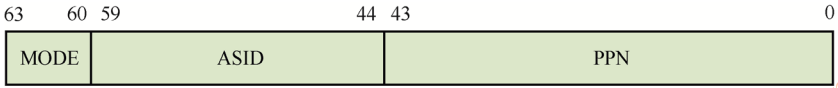
访问标志位与脏标志位 2

- 为什么需要A和D位?
- 操作系统的页面回收机制需要A和D来辅助。
- 操作系统使用访问标志位有如下好处。
  - ✓ 用来判断某个已经分配的页面是否被操作系统访问过。如果访问标志位为0，说明这个页面没有人访问过。
  - ✓ 用于操作系统中的页面回收机制。

连续页块

- 子叶页表项描述符中的N字段就用来实现TLB优化功能。
- 使用连续块页表项位的条件如下。
  - ✓ 连续的页面必须有相同的内存属性，即子叶页表项描述符中Bit[5:0]必须相同。
  - ✓ 必须有2N个连续的页面。

地址转换寄存器： satp



- PPN字段：存储了L0页表基地址的页帧号。
- ASID字段：进程地址空间标识符（Address Space IDentifier，ASID）
- MODE字段：用来选择地址转换的模式。

表 10.4　32 位 RISC-V 处理器的模式选择

模式	值	说明
Bare	0	没有实现地址转换功能
Sv32	1	实现 32 位虚拟地址转换（分页机制）

表 10.5　64 位 RISC-V 处理器的模式选择

模式	值	说明
Bare	0	没有实现地址转换功能
保留	1～7 的整数	保留
Sv39	8	实现 39 位虚拟地址转换（分页机制）
Sv48	9	实现 48 位虚拟地址转换（分页机制）
Sv57	10	保留，用于将来实现 57 位虚拟地址转换（分页机制）
Sv64	11	保留，用于将来实现 64 位虚拟地址转换（分页机制）

物理内存属性PMA（Physical Memory Attributes）

- PMA描述内存映射中的每个地址区域访问的属性
- 通常RISC-V处理器内置一个PMA，当ITLB、DTLB以及页表遍历单元获得物理地址之后，PMA检测器会做物理地址权限和属性检查
- PMA一般是在芯片设计阶段就固定下来，不能修改。

表 10.6　U74 处理器支持的 PMA 内存端口

内存端口	访问权限	支持属性
普通内存端口	可读、可写、可执行	支持原子内存操作和 LR/SC 指令、数据高速缓存、指令高速缓存以及指令预测
外设端口	可读、可写、可执行	支持原子内存操作、指令高速缓存
系统端口	可读、可写、可执行	指令高速缓存

表 10.7　U74 处理器的部分内存映射

起始地址	结束地址	PMA	说明
0x200 0000	0x200 FFFF	RWA	CLINT
0x201 0000	0x0201 3FFF	RWA	L2 高速缓存控制器
0xC00 0000	0xFFF FFFF	RWA	PLIC
0x2000 0000	0x3FFF FFFF	RWXIA	外设端口（512 MB）
0x4000 0000	0x5FFF FFFF	RWXI	系统端口（512 MB）
0x8000 0000	0x10 7FFF FFFF	RWXIDA	普通内存端口（64 GB）

物理内存保护（Physical Memory Protection）

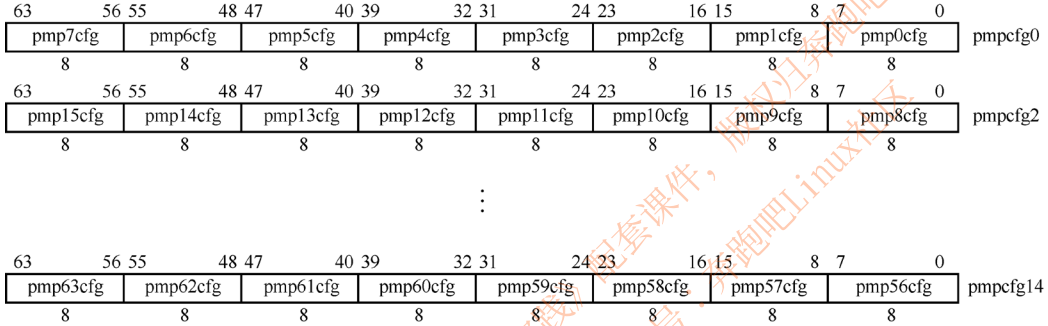
- M模式具有最高特权，拥有访问系统全部资源的权限
- S/U模式默认对任何内存区域都没有访问权限，需要配置PMP

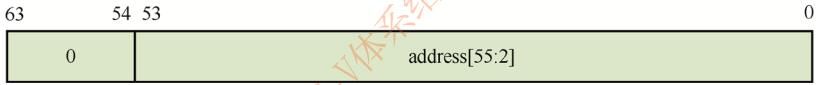
- 什么时候PMP会做检查呢?
  - M模式下只有当PMP表项中的L字段置位才会去检查PMP
  - S/U模式每次访问都会去检查PMP
    - ✓ 当MPRV为1并且MPP为S/U模式，在任意处理器模式下数据访问都需要做PMP检查。
    - ✓ MPRV为0并且在U模式或者S模式下的指令预取和数据访问。
  - 在MMU遍历页表的过程中也会做PMP检查。

- 64个PMP表项：8位宽的字段，用pmpNcfg表示，N表示表项数

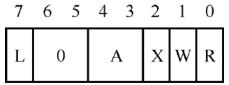
PMP表项

- 64个PMP表项：8位宽的字段，用pmpNcfg表示，N表示表项数

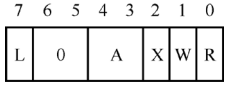




PMP地址寄存器



PMP表项



PMP表项

表 10.8 每个表项的说明

字段	位	说明
R	Bit[0]	可读权限。 ❑ 0：没有读权限。 ❑ 1：具有读权限
W	Bit[1]	可写权限。 ❑ 0：没有写权限。 ❑ 1：具有写权限
X	Bit[2]	可执行权限。 ❑ 0：没有执行权限。 ❑ 1：具有执行权限
A	Bit[4:3]	地址匹配模式。 ❑ 0：表示关闭 PMP 表项对应的检查。 ❑ 1：TOR 模式。 ❑ 2：NA4 模式，即表示 PMP 表项对应的地址范围仅为 4 字节。 ❑ 3：NAPOT 模式
L	Bit[7]	锁定状态。 ❑ 0：表示 PMP 表项没有锁定，对 M 模式不起作用，仅对 S 模式和 U 模式起作用。 ❑ 1：表示 PMP 表项锁定，对所有处理器模式（包括 M 模式）都起作用

PMP地址表示方法 - TOR

- TOR表示法：由前一个PMP表项的地址寄存器代表的起始地址（假设为pmpaddr(i－1)）和当前PMP表项的地址寄存器代表的起始地址（假设为pmpaddri）共同决定

$$\text{pmpaddr}(i-1) \leqslant y < \text{pmpaddr}(i)$$

- 如果当前PMP表项是第0个表项并且A字段为TOR，那么地址空间的下界被认为是0。此时，当前PMP表项代表的地址范围为：

$$0 \leqslant y < \text{pmpaddr}(i)$$

PMP地址表示方法： NAPOT

- NAPOT表示法：采用2n自然对齐的方式，其地址范围计算方式是从PMP地址寄存器第0位开始计算连续为1的个数n，地址的长度为2^n+3
  - ✓ 如果PMP地址寄存器的值为yyyy…yyy0，即LSZB个数为0，则地址空间为从yyyy…yyy0开始的2^3 ，即8 B。
  - ✓ 如果PMP地址寄存器的值为yyyy…yy01，即LSZB个数为1，则该地址空间为从yyyy…yy00开始的2^(1+3)，即16 B。
  - ✓ 如果PMP地址寄存器的值为yy01…1111，即LSZB个数为n，则该地址空间为从yy00…0000开始的2^(n+3) 。

- 例子：假设一个地址区间的起始地址为0x4000 0000，大小为1 MB，这个地址区间的PMP属性为可读、可写、可执行，请计算pmpaddr0寄存器的值以及pmpcfg0寄存器的值（假设目前只有一个PMP表项）。

（1）由于PMP地址寄存器记录的是地址的Bit[55:2]，因此地址需要右移2位，即0x4000 0000 >> 2 = 0x1000 0000。
（2）地址区间的大小为1 MB，即0x10 0000，它为220，因此LSZB为20。
（3）由于PMP地址空间大小的计算公式为2n+3字节，因此LSZB要减去3，即17。
（4）pmpaddr0 = 0x1000 0000 | 0b01 1111 1111 1111 1111 = 0x1001 FFFF。
（5）由于PMP属性为可读、可写、可执行，并且采用NAPOT模式，因此pmpcfg0寄存器的值为0x1F。

NAPOT例子

- 例子：以0x4000 0000为基地址，不同PMP地址大小对应的PMP地址寄存器的值的计算过程

表 10.9　不同 PMP 地址大小对应的 PMP 地址寄存器的值的计算过程

基地址	PMP 地址的长度	LSZB	PMP 地址寄存器的值
0x4000 0000	8 B	0	(0x1000 0000 | 0B0)
0x4000 0000	32 B	2	(0x1000 0000 | 0B011)
0x4000 0000	4 KB	9	(0x1000 0000 | 0B01 1111 1111)
0x4000 0000	64 KB	13	(0x1000 0000 | 0B01 1111 1111 1111)
0x4000 0000	1 MB	17	(0x1000 0000 | 0B01 1111 1111 1111 1111)

PMP使用例子

- 假设同一个地址0x8000 0000在pmp0cfg和pmp1cfg表项中有重叠。

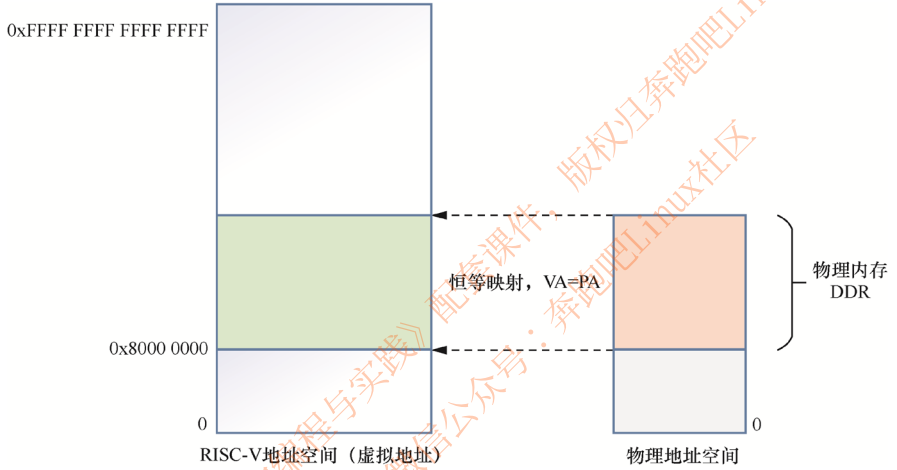
```
pmp0cfg: 0x8000 0000-0x8004 0000
pmp1cfg: 0x0000 0000-0xffff ffff
```

S模式的软件访问0x8000 0000地址，会触发加载访问访问异常

注意事项：

- 如果同一个地址对应多个PMP表项，那么PMP表项编号最小的表项优先级最高。
- PMP只能在M模式下配置
- PMP检查是基于地址范围的

案例分析1：在BenOS里实现恒等映射



恒等映射：虚拟地址 = 物理地址

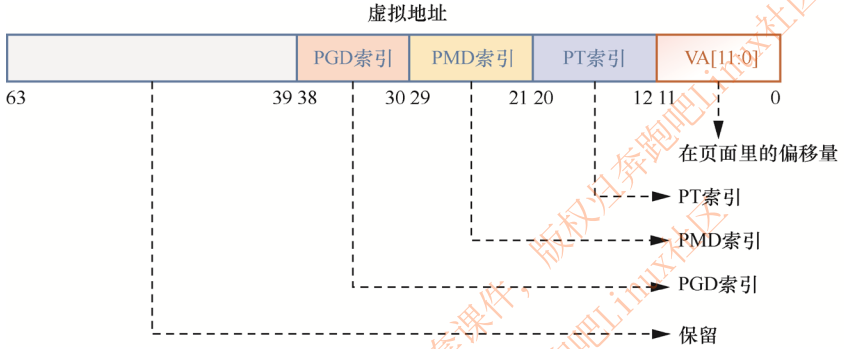
为什么要恒等映射?

为了打开MMU不会出问题:

1. 在关闭MMU情况下，处理器访问的地址都是物理地址。当MMU打开时，处理器访问的地址变成了虚拟地址。

2. 现代处理器都是多级流水线架构，处理器会提前预取多条指令到流水线中。当打开MMU时，处理器已经提前预取了多条指令，并且这些指令是以物理地址来进行预取的。当打开MMU指令执行完成，处理器的MMU功能生效，那么之前提前预取的指令以虚拟地址来访问，到MMU单元去查找对应的物理地址。因此，这里是为了保证处理器在开启MMU前后可以连续取指令。

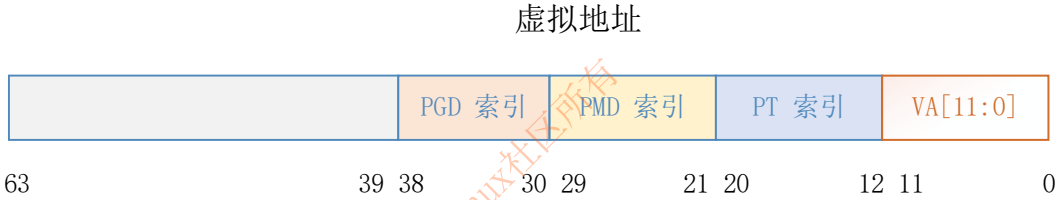
页表



采用与Linux内核类似的页表定义方式，采用以下3级分页模型：

- 页全局目录（Page Global Directory，PGD） -> L0页表
- 页中间目录（Page Middle Directory，PMD）-> L1页表
- 页表（Page Table，PT）-> L2页表

页表相关的宏




```
/* PGD */
#define PGDIR_SHIFT     30
#define PGDIR_SIZE      (1UL << PGDIR_SHIFT)
#define PGDIR_MASK      (~(PGDIR_SIZE - 1))
#define PTRS_PER_PGD    (PAGE_SIZE / sizeof(pgd_t))
```

```
/* PMD */
#define PMD_SHIFT       21
#define PMD_SIZE        (1UL << PMD_SHIFT)
#define PMD_MASK        (~(PMD_SIZE - 1))
#define PTRS_PER_PMD (1 << (PGD_SHIFT - PMD_SHIFT))
```

```
/* PTE */
#define PTE_SHIFT 12
#define PTE_SIZE (1UL << PTE_SHIFT)
#define PTE_MASK (~(PTE_SIZE-1))
#define PTRS_PER_PTE (1 << (PMD_SHIFT - PTE_SHIFT))
```

页表属性

```
#define _PAGE_PRESENT    (1 << 0)
#define _PAGE_READ       (1 << 1)
#define _PAGE_WRITE      (1 << 2)
#define _PAGE_EXEC       (1 << 3)
#define _PAGE_USER       (1 << 4)
#define _PAGE_GLOBAL     (1 << 5)
#define _PAGE_ACCESSED   (1 << 6)
#define _PAGE_DIRTY      (1 << 7)
#define _PAGE_SOFT       (1 << 8)
```

在BenOS里根据内存属性划分不同类型的页面。

- PAGE_KERNEL：操作系统内核中的普通内存页面。
- PAGE_KERNEL_READ：操作系统内核中只读的普通内存页面。
- PAGE_KERNEL_READ_EXEC：操作系统内核中只读的、可执行的普通页面。
- PAGE_KERNEL_EXEC：操作系统内核中可执行的普通页面。

表 10.10　　页面属性

页面属性	对应的字段
可读	V、A、U、R 字段
可读、可写	V、A、U、R、W 字段
可执行	V、A、U、X 字段
可读、可执行	V、A、U、R、X 字段
可读、可写、可执行	V、A、U、R、W、X 字段

页表项描述符

```
typedef unsigned long long u64;

typedef u64 pteval_t;
typedef u64 pmdval_t;
typedef u64 pgdval_t;

typedef struct {
    pteval_t pte;
} pte_t;
#define pte_val(x)  ((x).pte)
#define __pte(x)  ((pte_t) { (x) })

typedef struct {
    pmdval_t pmd;
} pmd_t;
#define pmd_val(x)  ((x).pmd)
#define __pmd(x)  ((pmd_t) { (x) })

typedef struct {
    pgdval_t pgd;
} pgd_t;
#define pgd_val(x)  ((x).pgd)
#define __pgd(x)  ((pgd_t) { (x) })
```

- 页表项都是64位宽
- pgd_t表示一个PGD页表项
- pmd_t表示一个PMD页表项
- pte_t表示一个页表项

PGD页表

```
SECTIONS
{
      /*
       * 设置 BenOS 的加载入口地址为 0x8020 0000
       */
      . = 0x80200000,

      ...

      /*
       * 数据段
       */
      _data = .;
      .data : { *(.data) }
      . = ALIGN(4096);
      idmap_pg_dir = .;
      . += 4096;
      _edata = .;

      ...
}
```

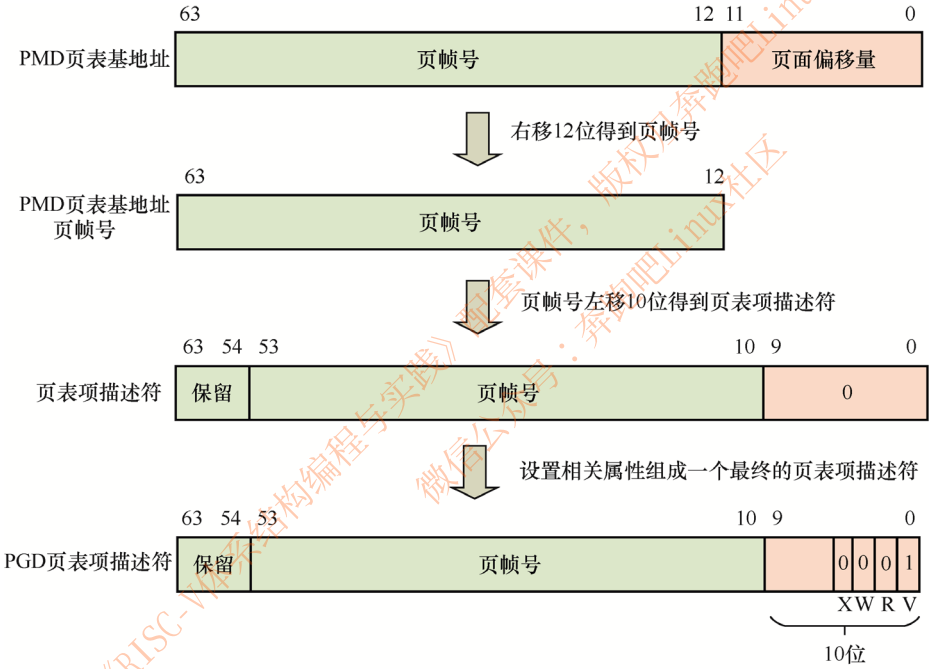
在链接脚本的数据段中预留4 KB大小给PGD页表。

```
void __create_pgd_mapping(pgd_t *pgdir, unsigned long phys,
      unsigned long virt, unsigned long size,
      unsigned long prot,
      unsigned long (*alloc_pgtable)(void),
      unsigned long flags)
```

__create_ pgd_ mapping()函数逐步创建页表

- pgdir表示PGD页表的基地址
- phys表示要映射物理内存的起始地址
- virt表示要映射的虚拟内存的起始地址
- size表示要创建的映射的总大小
- prot表示要创建的映射的内存属性
- alloc_pgtable用来分配下一级页表的内存分配函数
- flags传递给页表创建过程中的标志位

创建PGD页表项的过程



打开和测试MMU

```
1    .global enable_mmu_relocate
2    enable_mmu_relocate:
3        la a2, idmap_pg_dir
4        srl a2, a2, PAGE_SHIFT
5        li a1, SATP_MODE_39
6        or a2, a2, a1
7        sfence.vma
8        csrw satp, a2
9        ret
```

设置satp寄存器打开MMU

```
13   /*
14    * 访问一个没有建立映射的地址
15    *
16    * 存储/AMO 页面异常
17    */
18   static int test_access_unmap_address(void)
19   {
20       unsigned long address = DDR_END + 4096;
21
22       *(unsigned long *)address = 0x55;
23
24       printk("%s access 0x%x done\n", __func__, address);
25
26       return 0;
27   }
28
29   static void test_mmu(void)
30   {
31       test_access_map_address();
32       test_access_unmap_address();
33   }
```

分别访问一个经过恒等映射和没有经过恒等映射的内存地址

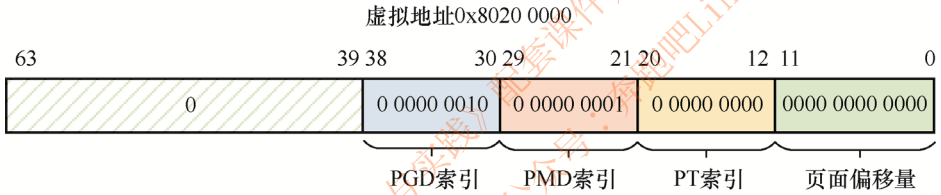
最终运行结果

```
rlk@master:benos$ make run
...
test_access_map_address access 0x87fff000 done
Oops - Store/AMO page fault
Call Trace:
[<0x0000000080202acc>] test_access_unmap_address+0x1c/0x42
[<0x0000000080202afe>] test_mmu+0xc/0x1a
[<0x0000000080202d1c>] kernel_main+0xa6/0xac
sepc: 0000000080202acc ra : 0000000080206f10 sp : 0000000080206fb0
 gp : 0000000000000000 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000080206fe0
 s1 : 0000000080200010 a0 : 0000000000000000 a1 : 0000000000000010
 a2 : ffffffffffffffff a3 : 0000000080206ed0 a4 : 0000000000000055
 a5 : 0000000088001000 a6 : 0000000000000000 a7 : 0000000000000061
 s2 : 8000000000006800 s3 : 0000000080200000 s4 : 0000000082200000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 00000000800120e8
 s8 : 000000008020002e s9 : 000000000000007f s10: 0000000000000000
 s11: 0000000000000000 t3 : 45b0206f91166285 t4: 0000000080017ee0
 t5 : 0000000000000027 t6 : 0000000000000000
sstatus:0x8000000200006120 sbadaddr:0x0000000088001000 scause:0x000000000000000f
Kernel panic
```

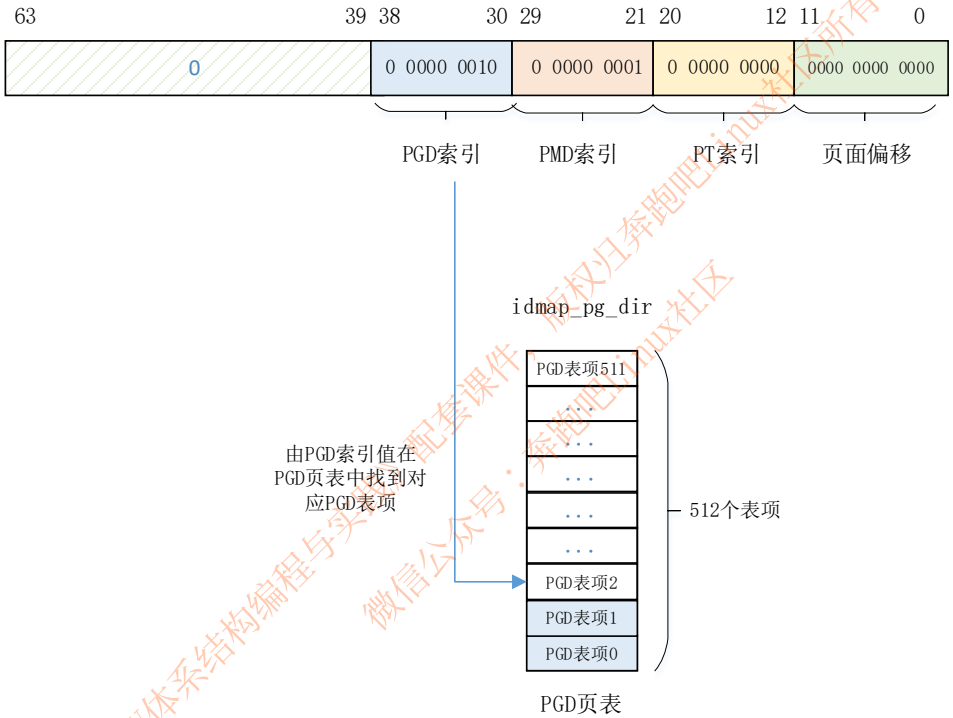
sbadaddr：表示该地址为未映射的地址

案例分析2：图解页表创建过程

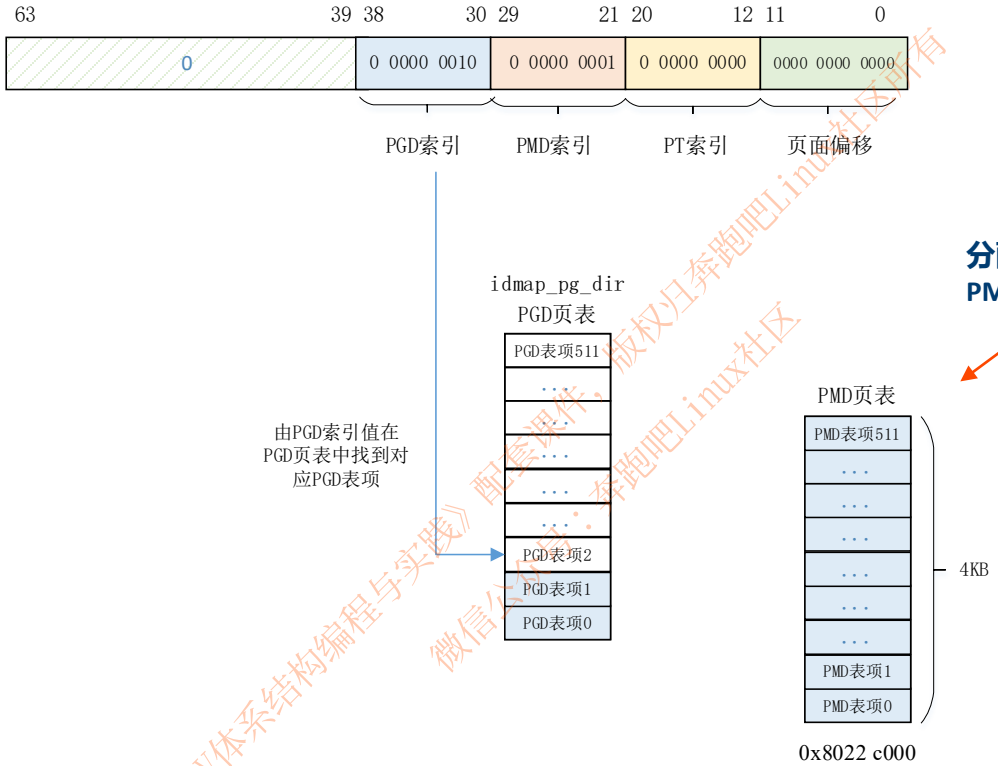
假设PA=0x8020 0000 ， VA=0x8020 0000，映射大小为4 KB，PGD页表的基地址idmap_pg_dir为0x8020 8000



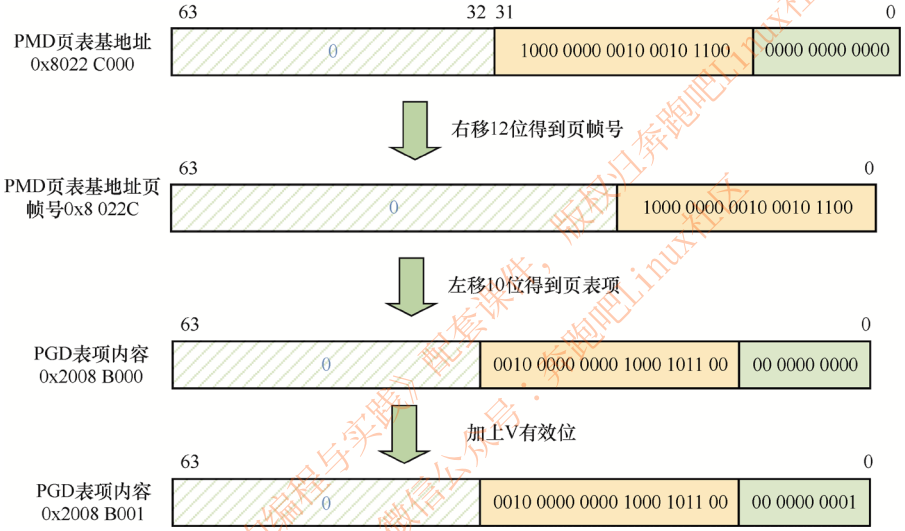
1.填充和创建PGD页表项



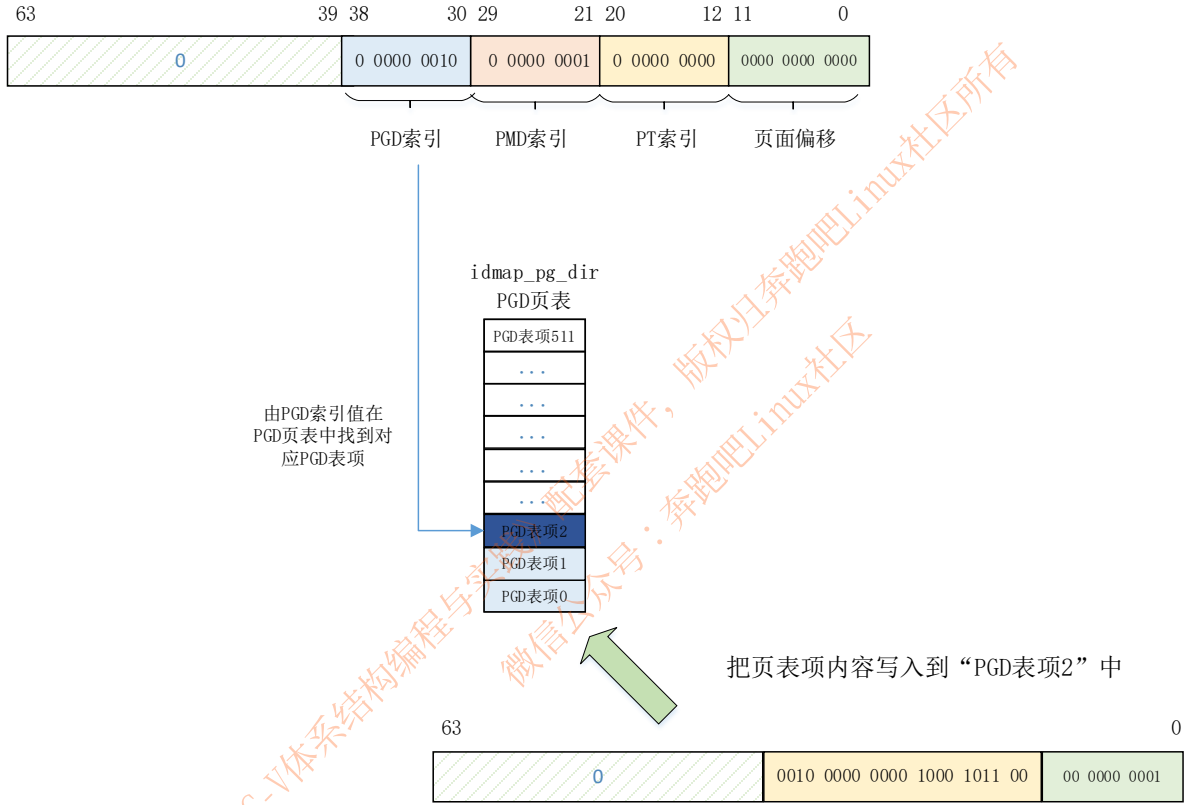
虚拟地址0x8020 0000对应的PGD索引为2，在PGD页表中找到页表项，即**PGD页表项2**



由于“PGD页表项2”为空，为PMD页表分配一个4 KB页面，假设这个页面的物理地址pmd_phys为0x8022 C000

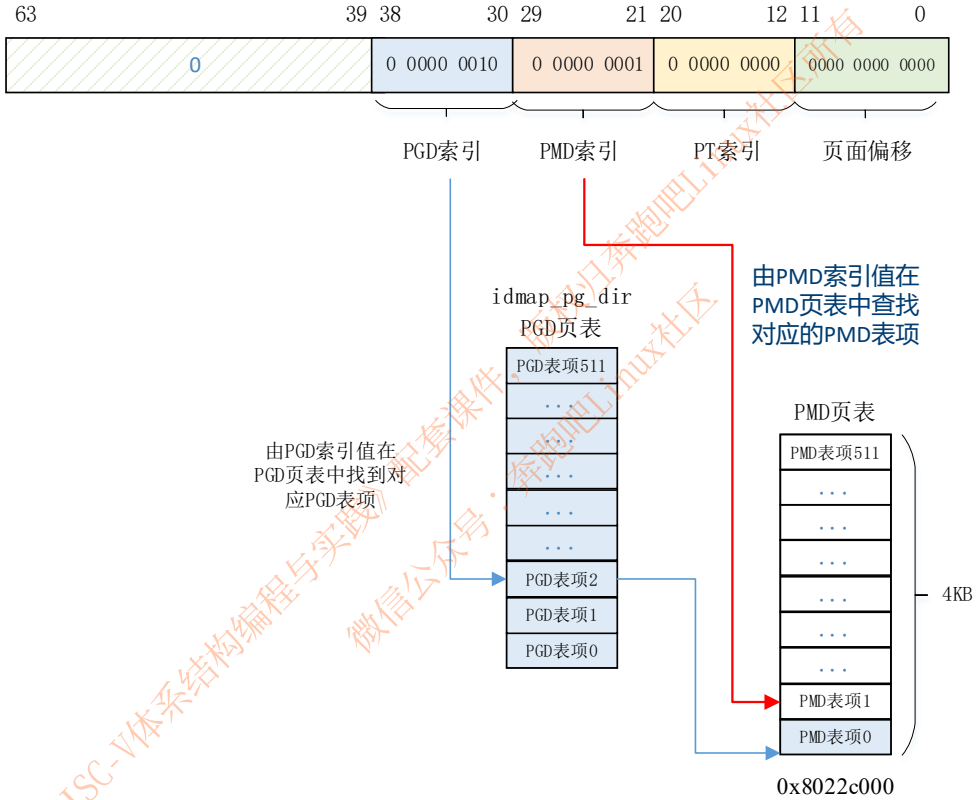


由PMD页表基地址来 构造PGD页表项描述符

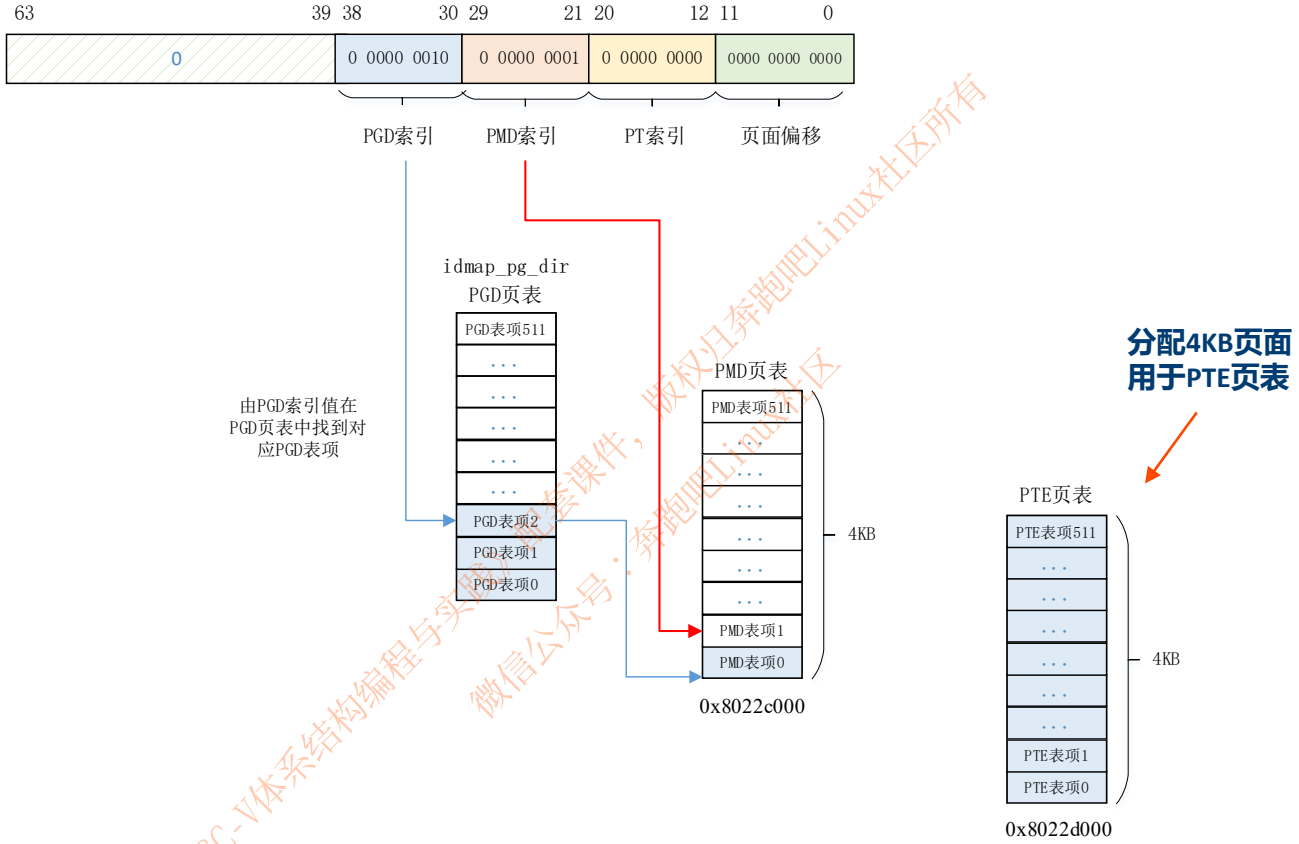


对PGD页表项2的填充

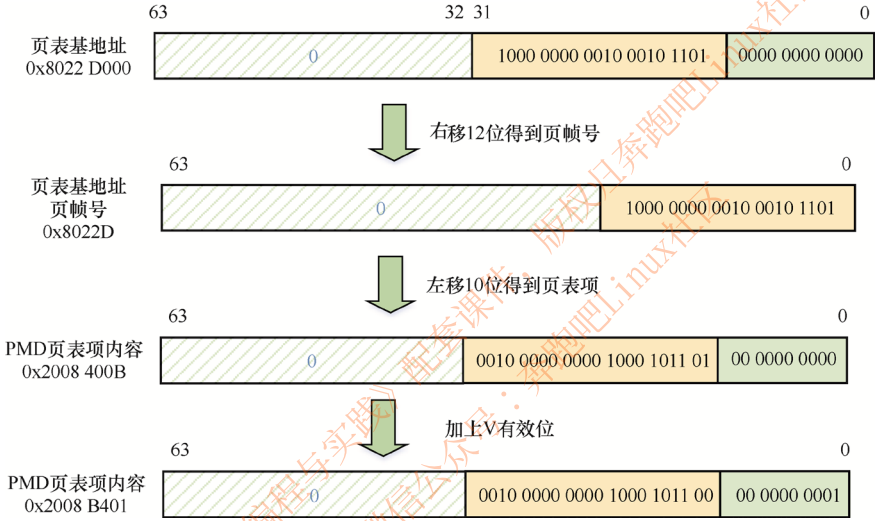
2.填充和创建PMD页表项



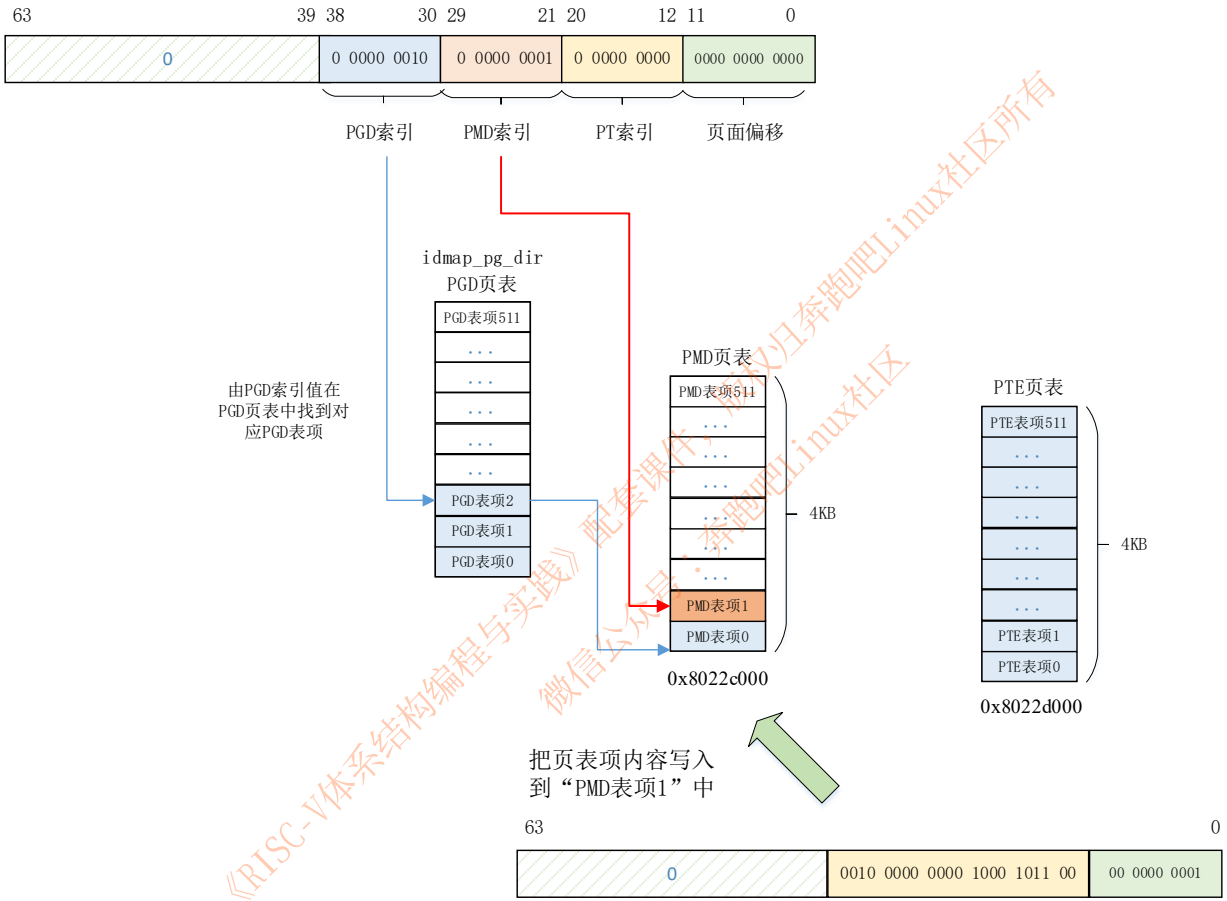
虚拟地址0x8020 0000对应的PMD索引为1，在PMD页表中找到页表项，即**PMD页表项1**



由于“PMD页表项1”为空，新创建一个下一级页表（即PT），为页表分配一个4 KB页面，假设这个页面的物理地址pte_phys为0x8022 d000

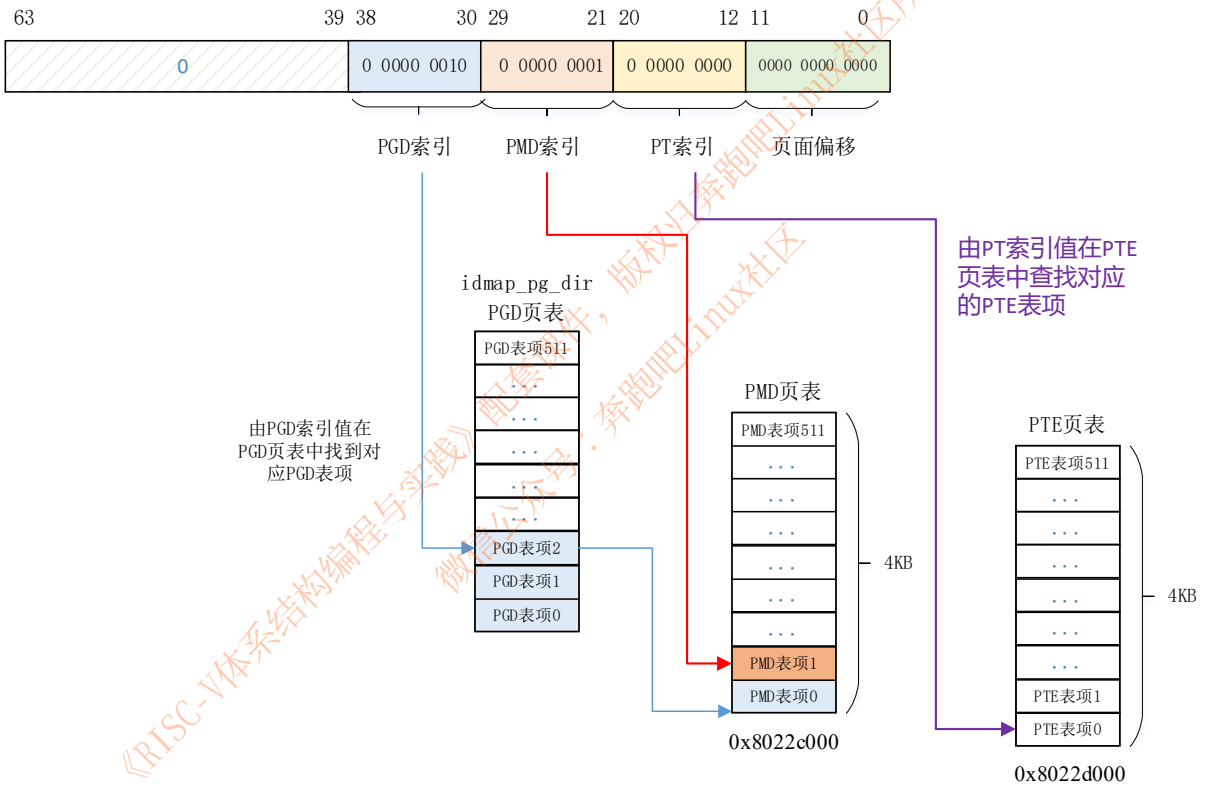


由PTE页表基地址来 构造PMD页表项描述符的内容

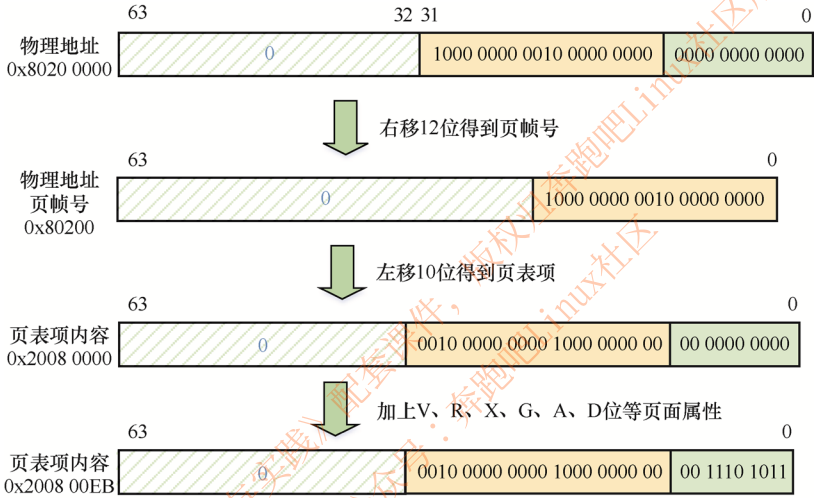


对PMD页表项1的填充

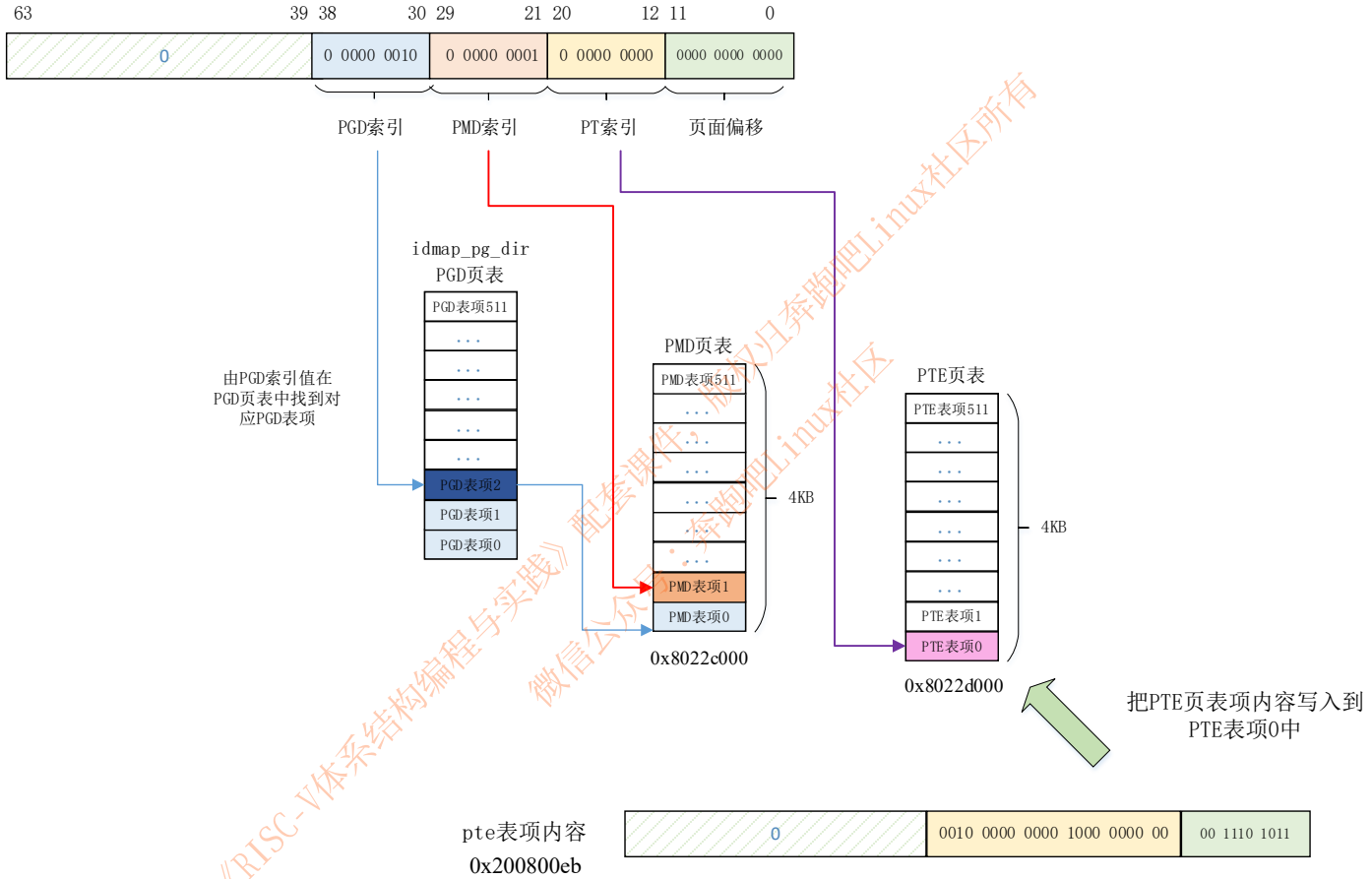
3.填充和创建PT页表项



虚拟地址0x8020 0000对应的PT索引为0，在PTE页表中找到页表项，即**PTE页表项0**

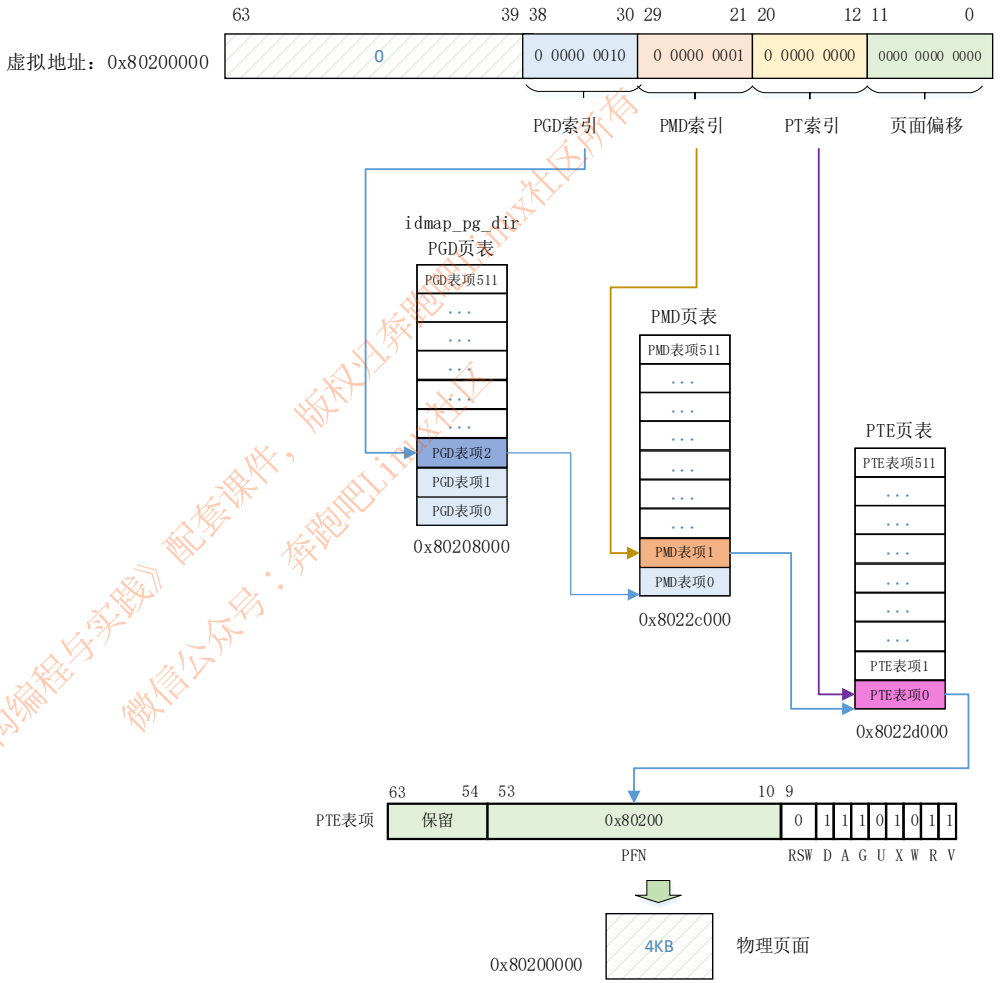


根据物理地址（0x8020 0000）来创建一个页表项描述符的内容



把这个页表项内容写入页表项0中，完成对页表的填充

4. 最终效果图



实验1：建立恒等映射

1．实验目的

熟悉RISC-V处理器中MMU的工作流程。

2．实验要求

（1）在QEMU上完成案例分析1的代码

（2）使用GDB单步调试页表建立过程

```
BenOS image layout:
  .text.boot: 0x80200000 - 0x80200044 (    68 B)
       .text: 0x80200048 - 0x80205000 ( 20408 B)
     .rodata: 0x80205000 - 0x80206b00 (  6912 B)
       .data: 0x80206b00 - 0x80209000 (  9472 B)
        .bss: 0x80209020 - 0x8022b488 (140392 B)
sstatus:0x2
sstatus:0x2, sie:0x222
test_access_map_address access 0x80fff000 done
Oops - Store/AMO page fault
Call Trace:
[<0x0000000080202934>] test_access_unmap_address+0x28/0x5c
[<0x000000008020297c>] test_mmu+0x14/0x2c
[<0x00000000802002cf8>] kernel_main+0xe8/0xf0
sepc: 0000000080202934 ra : 0000000080202980 sp : 0000000080207fc0
 gp : 0000000080209800 tp : 0000000000000000 t0 : 0000000000000005
 t1 : 0000000000000005 t2 : 0000000080200020 t3 : 0000000080207fe0
 s1 : 0000000080200010 a0 : 0000000000000000 a1 : 0000000000000000
 a2 : 0000000000000000 a3 : 00000000802021a4 a4 : 0000000000000055
 a5 : 0000000081001000 a6 : 0000000000000000 a7 : 0000000000000001
 s2 : 0000000000000000 s3 : 0000000000000000 s4 : 0000000000000000
 s5 : 0000000000000000 s6 : 0000000000000000 s7 : 0000000000000000
 s8 : 000000008020003c s9 : 0000000000000000 s10: 0000000000000000
 s11: 0000000000000000 t3 : 00510133000012b7 t4: 0000000000000000
 t5 : 0000000000000000 t6 : 0000000000000000
sstatus:0x0000000000000120  sbadaddr:0x0000000081001000  scause:0x000000000000000f
Kernel panic
```

实验2：为什么MMU无法运行

1．实验目的

（1）熟悉RISC-V处理器中MMU的工作流程。

（2）培养调试和解决问题的能力。

2．实验要求

小明同学把实验1中的create_identical_mapping()函数写成：

```
static void create_identical_mapping(void)
{
        unsigned long start;
        unsigned long end;

        /*map memory*/
        start = (unsigned long)_text_boot;
        end = DDR_END;
        __create_pgd_mapping((pgd_t *)idmap_pg_dir, start, start,
                             end - start, PAGE_KERNEL,
                             early_pgtable_alloc,
                             0);
        printk("map memory done\n");
}
```

他发现系统无法运行，这是什么原因导致的？请使用QEMU与GDB单步调试代码并找出是哪条语句发生了问题。为什么MMU无法运行？

实验3：实现一个MMU页表的转储功能

1．实验目的

（1）熟悉RISC-V处理器中MMU的工作流程。

（2）培养调试和解决问题的能力。

2．实验要求

在实验1的基础上实现一个MMU页表的转储（dump）功能，输出页表的虚拟地址、页表属性等信息，以方便调试和定位问题

```
0x0000000080200000-0x0000000080201000           4K PTE     D A G . X . R V
0x0000000080201000-0x0000000080202000           4K PTE     D A G . X . R V
0x0000000080202000-0x0000000080203000           4K PTE     D A G . X . R V
0x0000000080203000-0x0000000080204000           4K PTE     D A G . X . R V
0x0000000080204000-0x0000000080205000           4K PTE     D A G . X . R V
0x0000000080205000-0x0000000080206000           4K PTE     D A G . . W R V
0x0000000080206000-0x0000000080207000           4K PTE     D A G . . W R V
```

虚拟地址范围

A页面的大小

页表属性:
比如可读、可写、可执行权限等

实验4：修改页面属性

1．实验目的

（1）熟悉RISC-V处理器中MMU的工作流程。
（2）培养调试和解决问题的能力。

2．实验要求

在系统中找出一个只读属性的页面，然后把这个页面的属性设置为可读、可写，使用memset()函数往这个页面写入内容。

本实验的步骤如下。
（1）从系统中找出一个4 KB的只读页面，其虚拟地址为vaddr。
（2）遍历页表，找到vaddr对应的页表项。
（3）修改页表项，为它设置可读、可写属性。
（4）使用memset()修改页面内容。

实验5：使用汇编语言来建立恒等映射

1．实验目的

（1）熟悉RISC-V处理器中MMU的工作流程。
（2）熟悉页表建立过程。
（3）熟悉汇编的使用。

2．实验要求

（1）在实验1的基础上，在汇编阶段使用汇编语言来创建恒等映射，即大小为2 MB的块映射，并且打开MMU。
（2）写一个测试例子来验证MMU是否开启。

实验6：在MySBI中实现和验证PMP机制

1．实验目的

（1）熟悉RISC-V处理器的PMP机制。

2．实验要求

（1）在MySBI中实现PMP配置功能，配置页表的属性为可读、可写、可执行。

```
pmp0cfg: 0x0-0xffff ffff ffff ffff
```

（2）在MySBI中实现如下PMP配置功能，分别配置页表的属性为不可读、不可写、不可执行，页表的属性为可读、可写、可执行。

```
pmp0cfg: 0x8000 0000-0x8004 0000
pmp1cfg: 0x0000 0000-0xffff ffff ffff ffff
```

（3）在BenOS中访问地址0x8000 0000，请观察现象。

在广袤的宇宙与有限的时空中，
能通过文字和视频与你共同学习RISC-V，
是我们无比的荣幸！

笨叔

文字不如声音，声音不如视频

扫描订阅RISC-V视频课程

第4季 奔跑吧Linux社区 视频课程

RISC-V体系结构编程与实践

主讲：笨叔

课程名称	进度	时长（分钟）
第1课：课程介绍（免费）	完成	20
第2课：RISC-V体系结构介绍（免费）	完成	47
第3课：RISC-V处理器微架构（免费）	完成	48
第4课：搭建RISC-V开发环境（免费）	完成	30
第5课：RISC-V指令集（免费）	完成	128
第6课：RISC-V函数调用规范	完成	40
第7课：RISC-V GNU AS汇编器	完成	42
第8课：RISC-V GNU 链接脚本	完成	90
第9课：RISC-V GNU 内嵌汇编	完成	52
第10课：RISC-V异常处理	完成	80
第11课：RISC-V中断处理	完成	52
第12课：RISC-V内存管理	完成	116
第13课：内存管理实验讲解	完成	36
第14课：cache基础知识	完成	78
第15课：缓存一致性	完成	96
第16课：RISC-V TLB管理	完成	54
第17课：RISC-V原子操作	未录制	
第18课：RISC-V内存屏障	未录制	
第19课：BenOS操作系统相关知识	未录制	
第20课：RVV可伸缩矢量计算	未录制	
第21课：RISC-V压缩指令	未录制	
第22课：RISC-V虚拟化	未录制	
		总计17小时

更多精彩内容马上献上....
微信公众号：奔跑吧Linux社区

视频课程持续更新中...