Computer Systems

第十二次小班课

助教：罗兆丰

Announcements

- malloclab已经发布, 12/16due, 12/18ddl, 请同学们合理安排时间
  - Start early!
  - Start early!
  - Start early!
  - 满分难度较大

基础知识

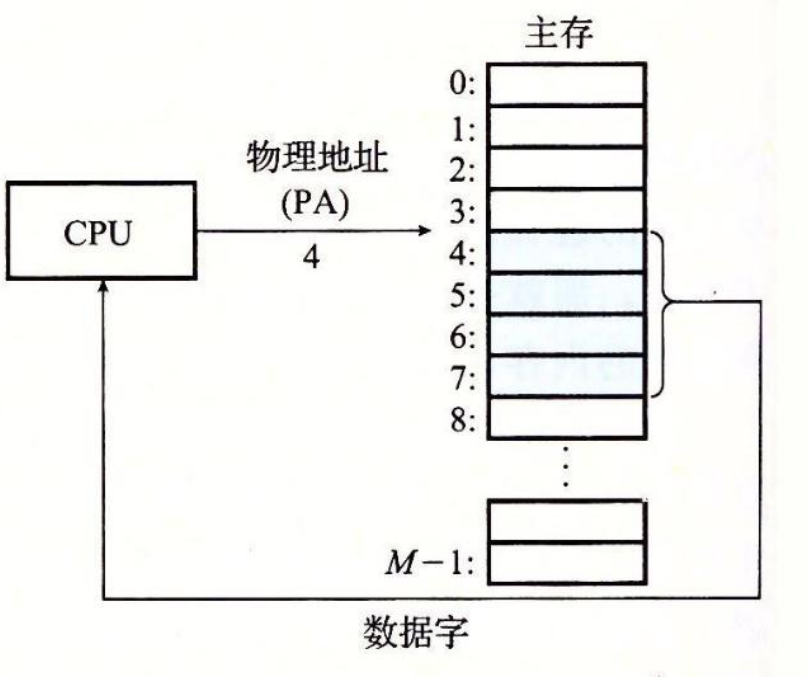
虚拟内存之前曾经使用过的内存机制

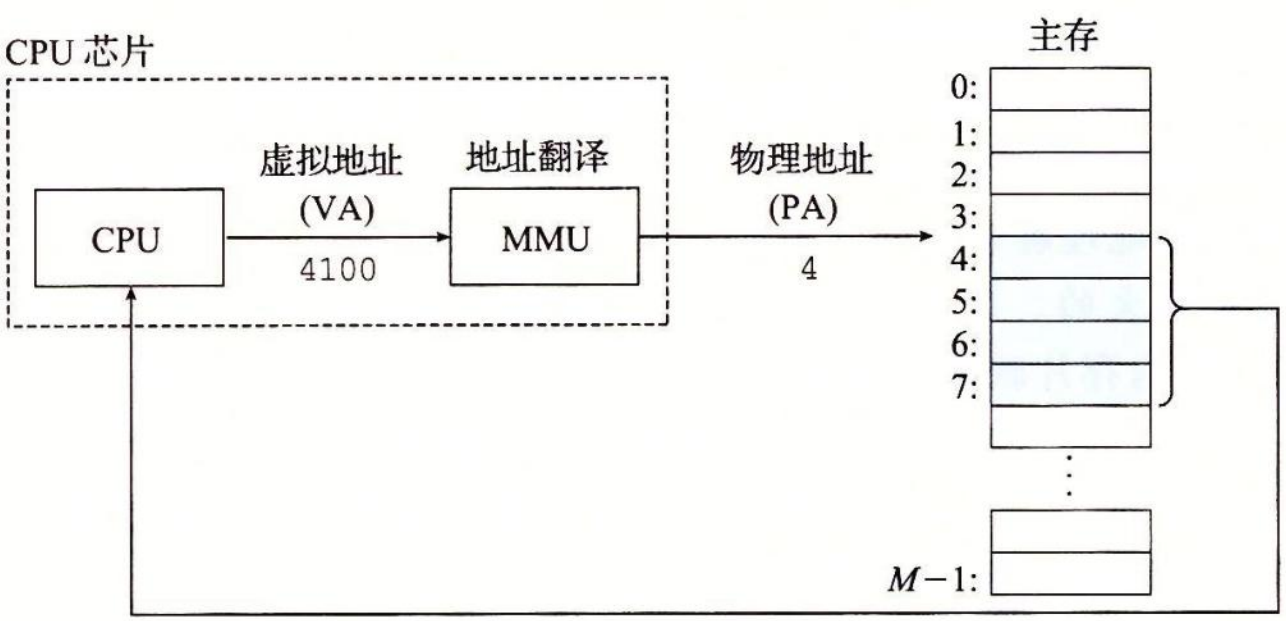
轮到谁谁就占据全部内存
把其他进程扔进磁盘里

把内存划分成块
每个进程用一个块
大小不一定合适

每个进程分配一个段
访存时使用段寄存器寻址
需要碎片整理：复制内存同时修改段寄存器
开销太大

物理寻址，物理地址空间，虚拟寻址，虚拟地址空间





物理地址空间大小：M（不要求是2的幂）

$$\{0,1,2,\cdots,M-1\}$$

设虚拟地址的地址长度为n位
则虚拟地址空间大小为 $N=2^n$.

$$\{0,1,2,\cdots,N-1\}$$

All problems in computer science can be solved by another layer of indirection.

虚拟内存的主要思路：

1.每个进程都有一个虚拟地址空间，程序运行时使用的地址是虚拟地址，在虚拟地址空间上索引得到数据。

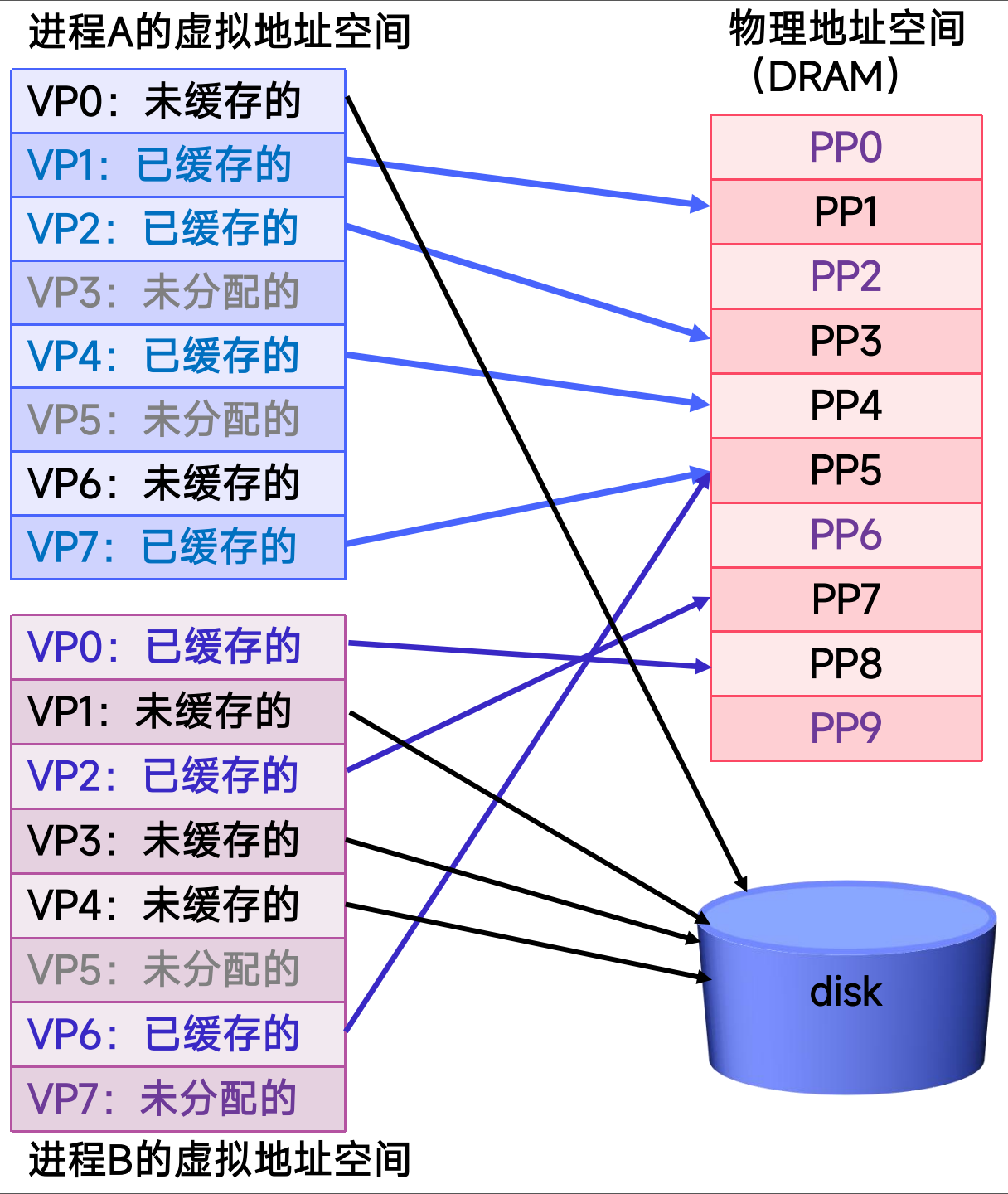
2.不同进程的虚拟地址空间相互隔离，无法直接访问对方数据。

3.虚拟和物理地址空间都被划分的固定大小的页

4.虚拟地址空间由物理内存和磁盘作为支持，虚拟内存页可以是不存在的，或者在物理内存上，或者在磁盘上。

术语：

VP：虚拟页
PP：物理页
VA：虚拟地址
PA：物理地址
SRAM：指高速缓存
DRAM：指主存



虚拟内存页的三种状态：

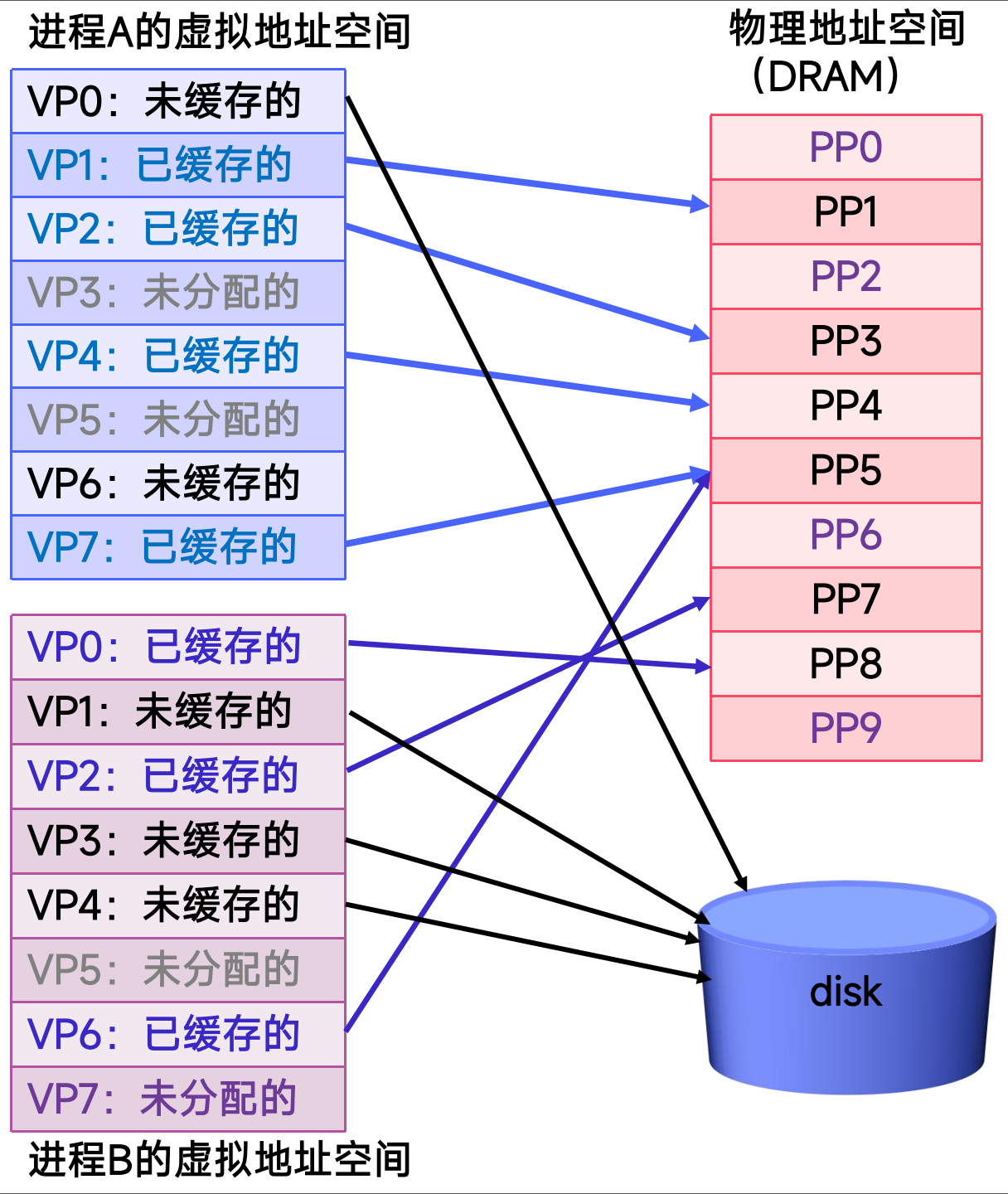
1. 未分配的，没有分配的页，不占用物理内存或者磁盘空间。

2. 已缓存的，虚拟页存在物理内存上。

3. 未缓存的，虚拟页没有存在物理内存上。

思考：

在 Linux 虚拟内存系统中，在实现上是怎么区分未分配的 / 已缓存的 / 未缓存的这三种页的？



页表

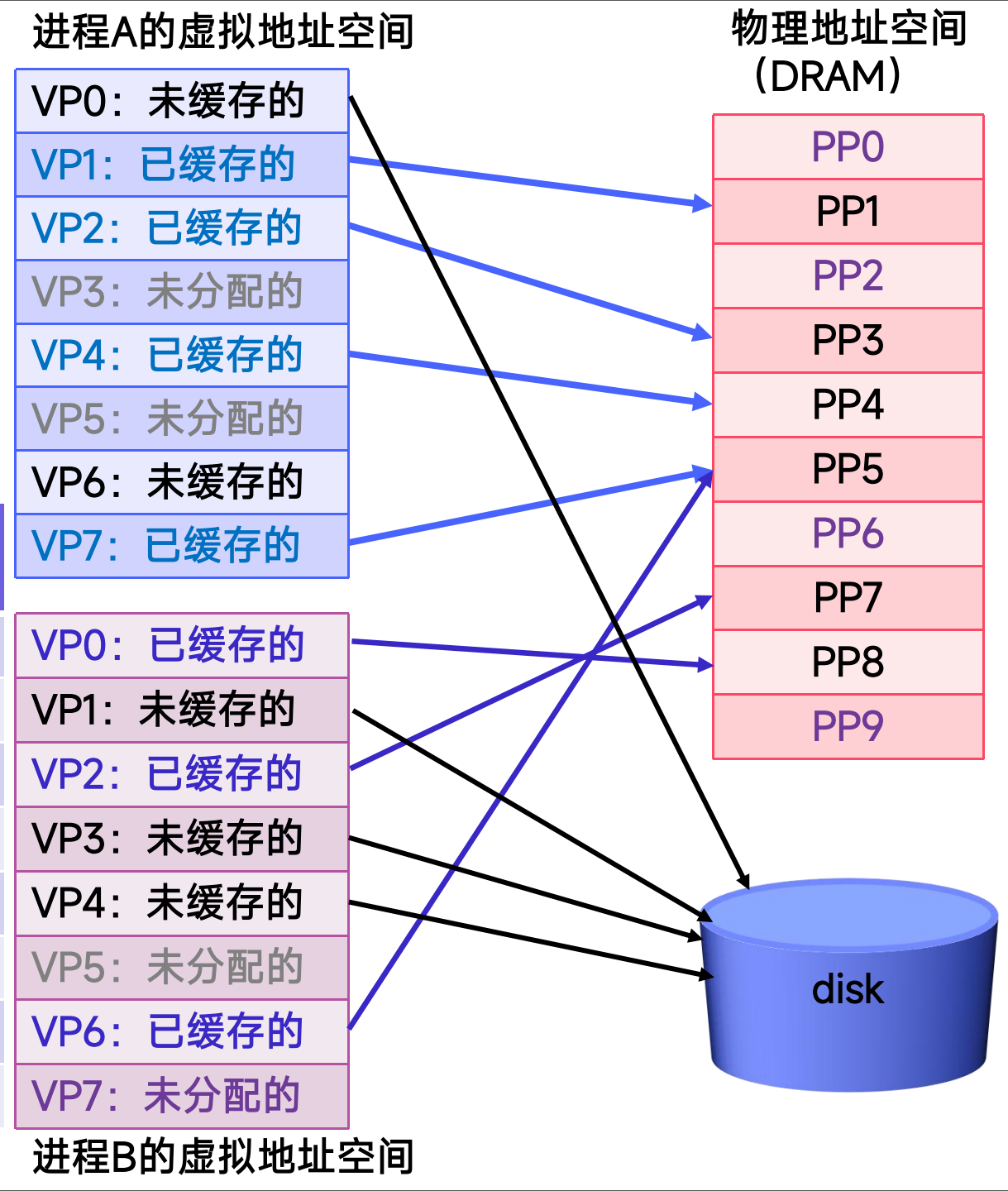
当访问虚拟地址的时候，通过查找页表来确定它的状态和位置

有效位是1：虚拟页在DRAM上，地址指向磁盘位置
有效位是0，地址是NULL：页未分配
地址不是NULL：指向页在磁盘上的哪个位置

（PTE里存储的具体内容稍后介绍）

术语：
PTE：页表条目

有效位	物理页地址或者磁盘位置
1	PP8
0	磁盘XXX处
1	PP7
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	null



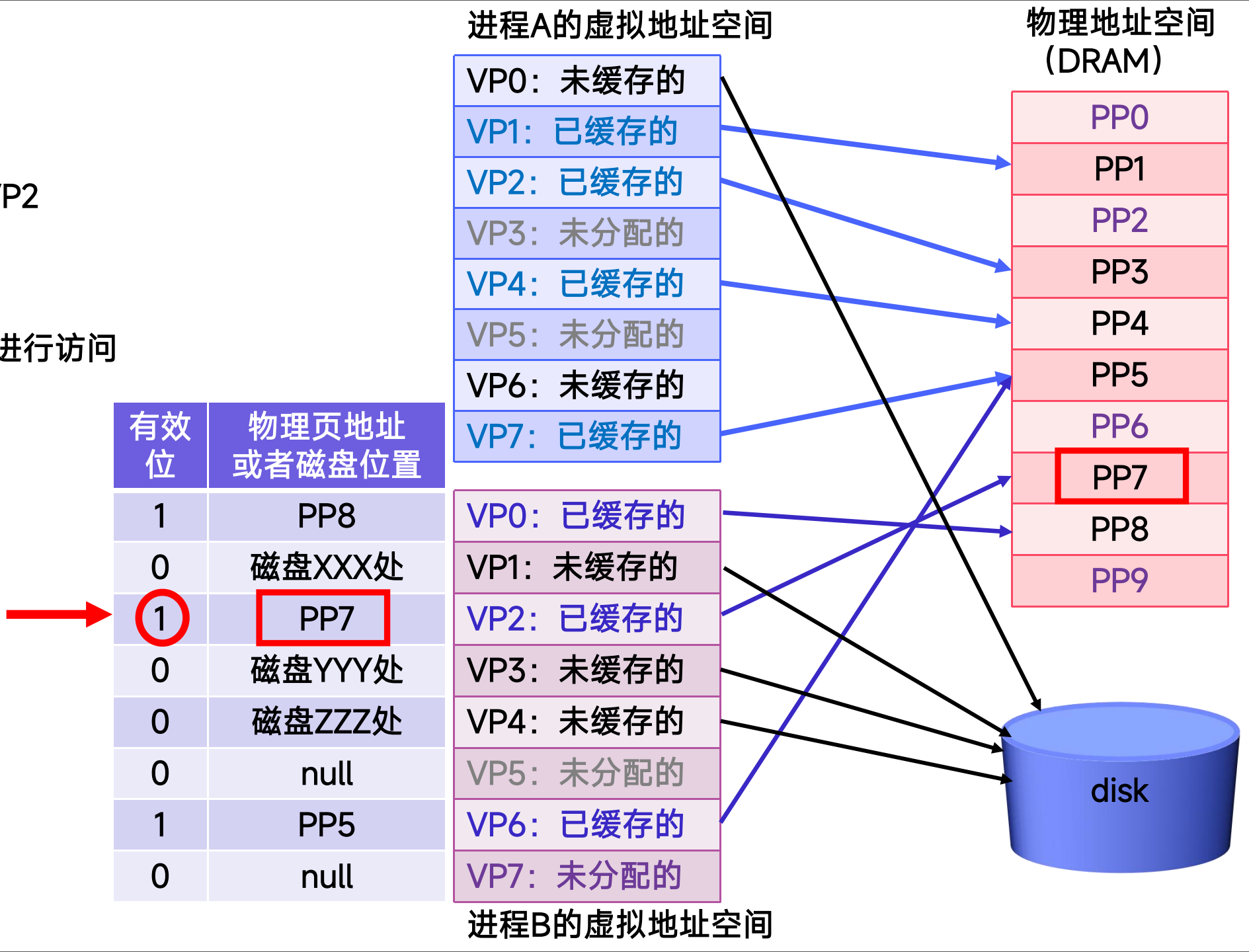
页命中

运行进程B的CPU：访问VP2

查表，发现页有效

从表中取出物理页地址，进行访问

有效位	物理页地址或者磁盘位置
1	PP8
0	磁盘XXX处
1	PP7
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	null



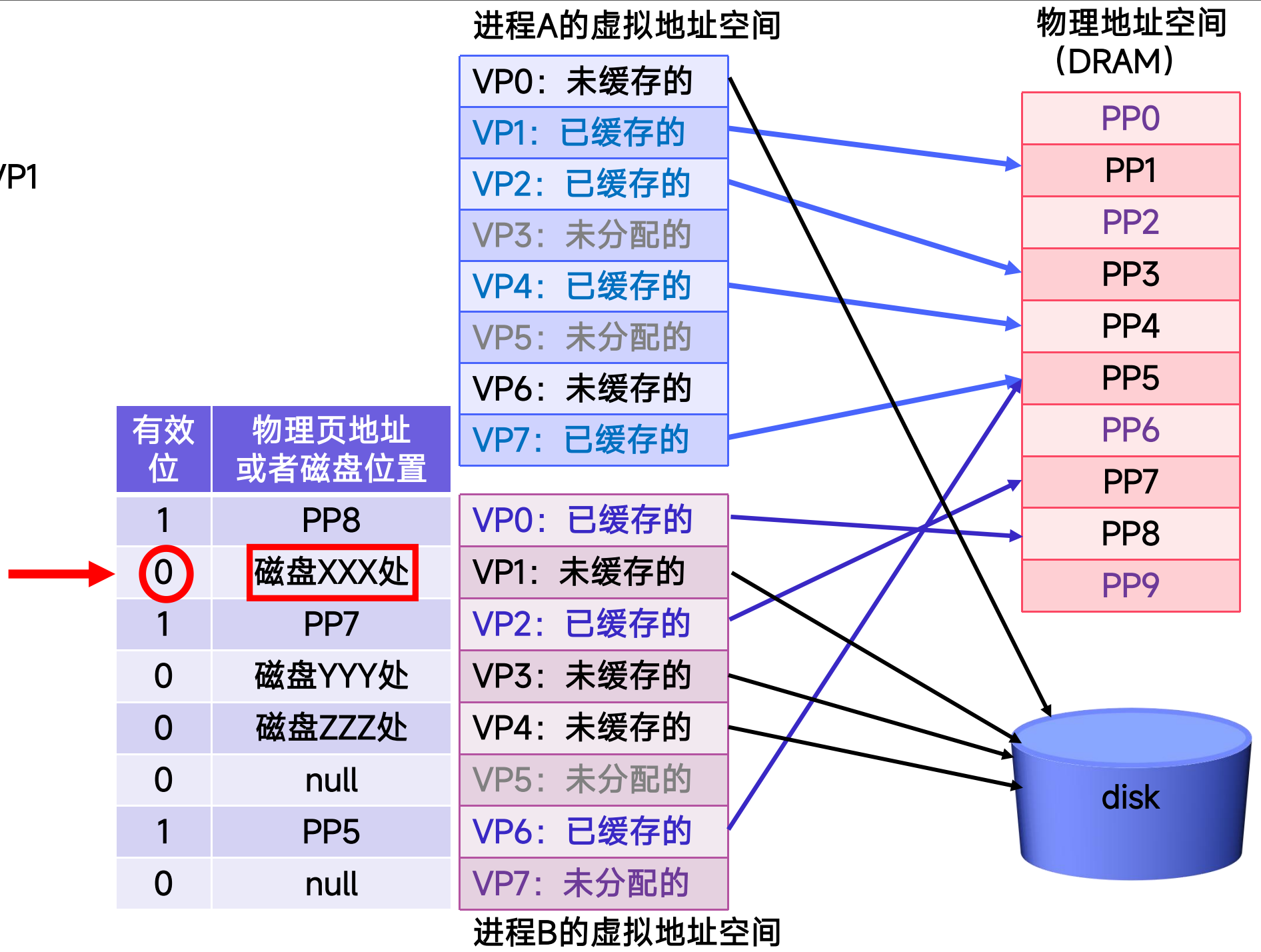
缺页（1）

运行进程B的CPU：访问VP1

查表，发现页无效

触发缺页故障

页在磁盘里，尝试调入



缺页（1）

运行进程B的CPU：访问VP1

查表，发现页无效

触发缺页故障

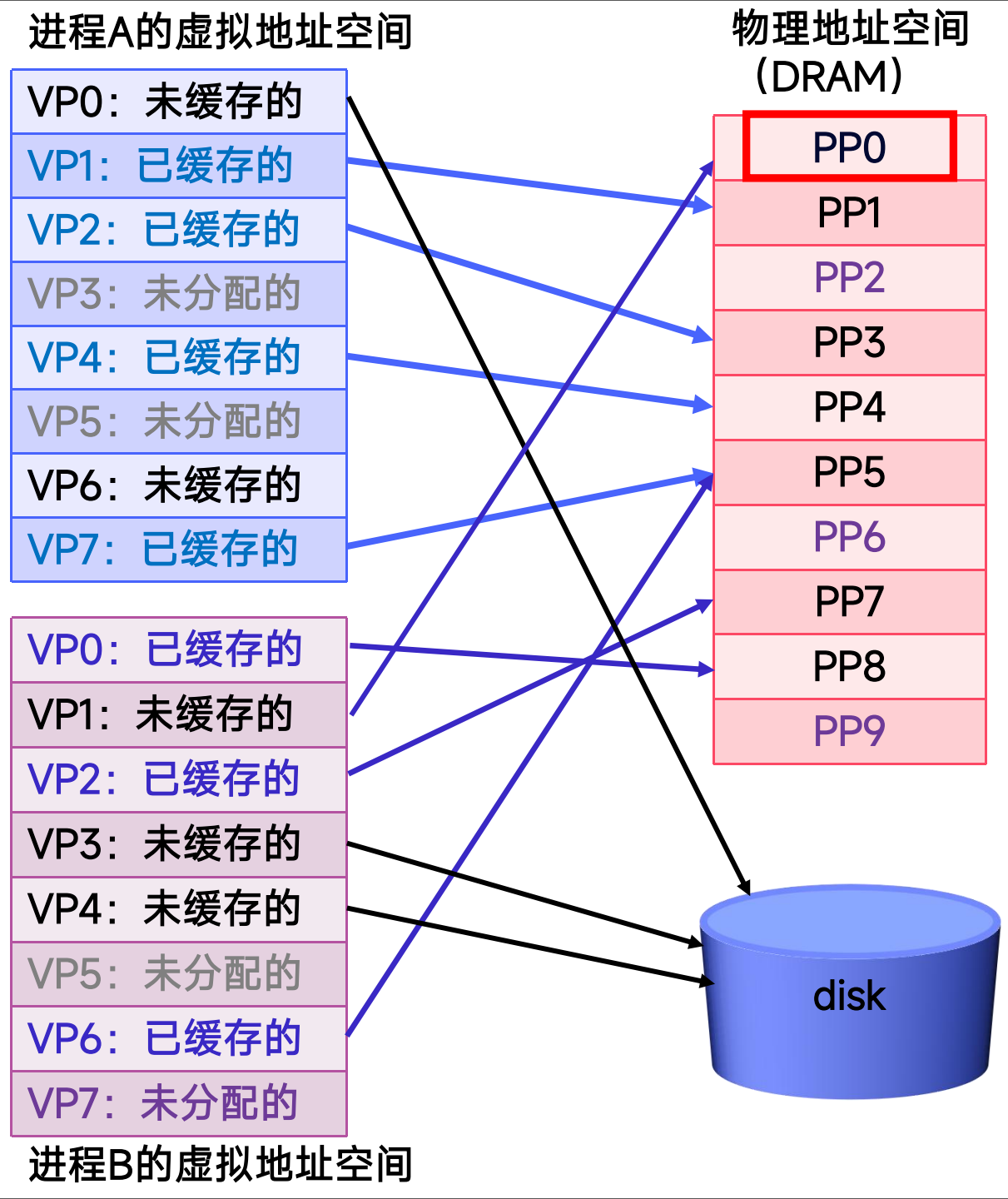
页在磁盘里，尝试调入

将VP1从磁盘中取出，放到DRAM中，并标记VP1的PTE

从缺页故障返回

再次访问VP1，则成功

有效位	物理页地址或者磁盘位置
1	PP8
1	PP0
1	PP7
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	null



缺页（2）

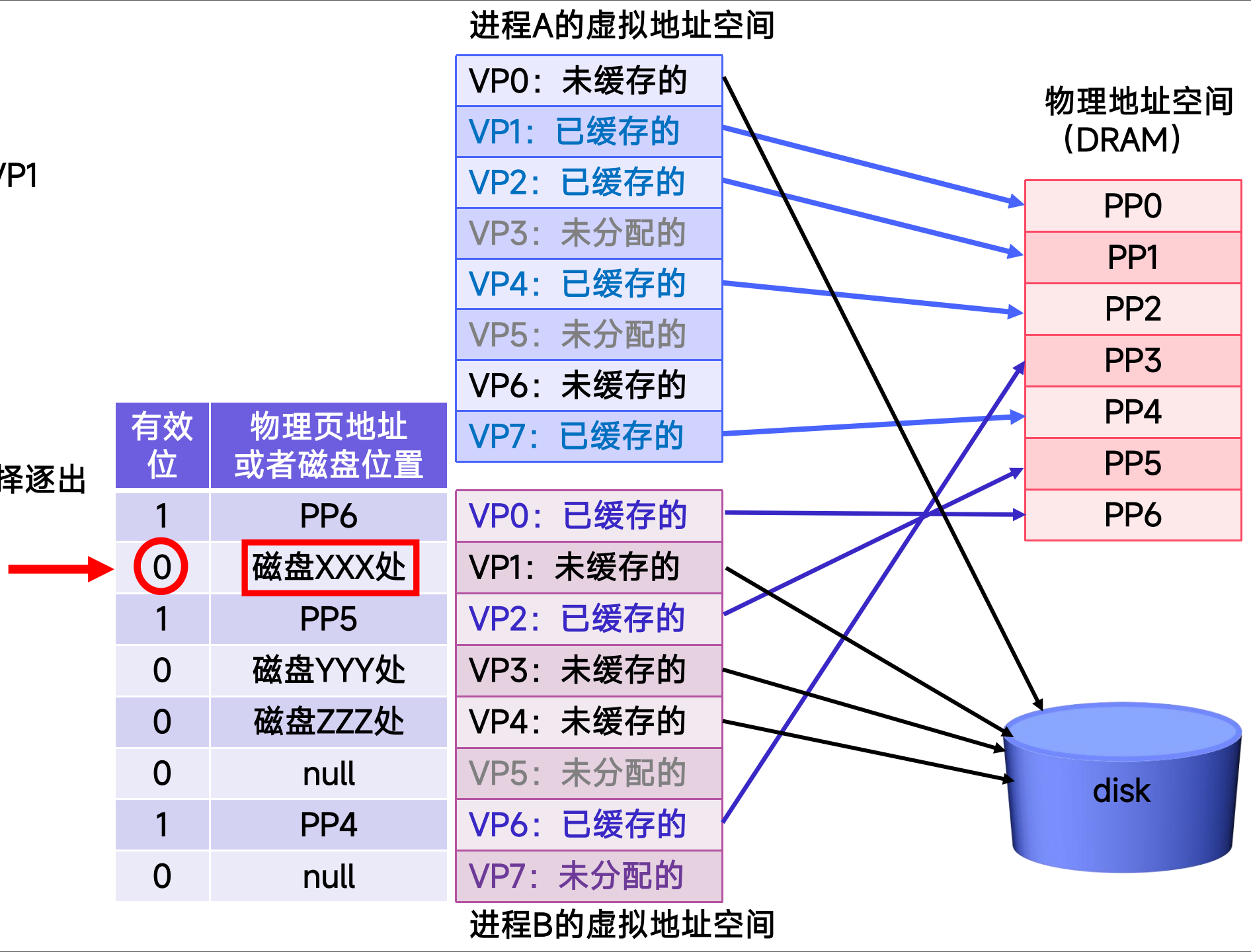
运行进程B的CPU：访问VP1

查表，发现页无效

触发缺页故障

页在磁盘里，尝试调入

发现DRAM缓存已满，选择逐出页，假设选择PP6（VP0）



缺页（2）

运行进程B的CPU：访问VP1

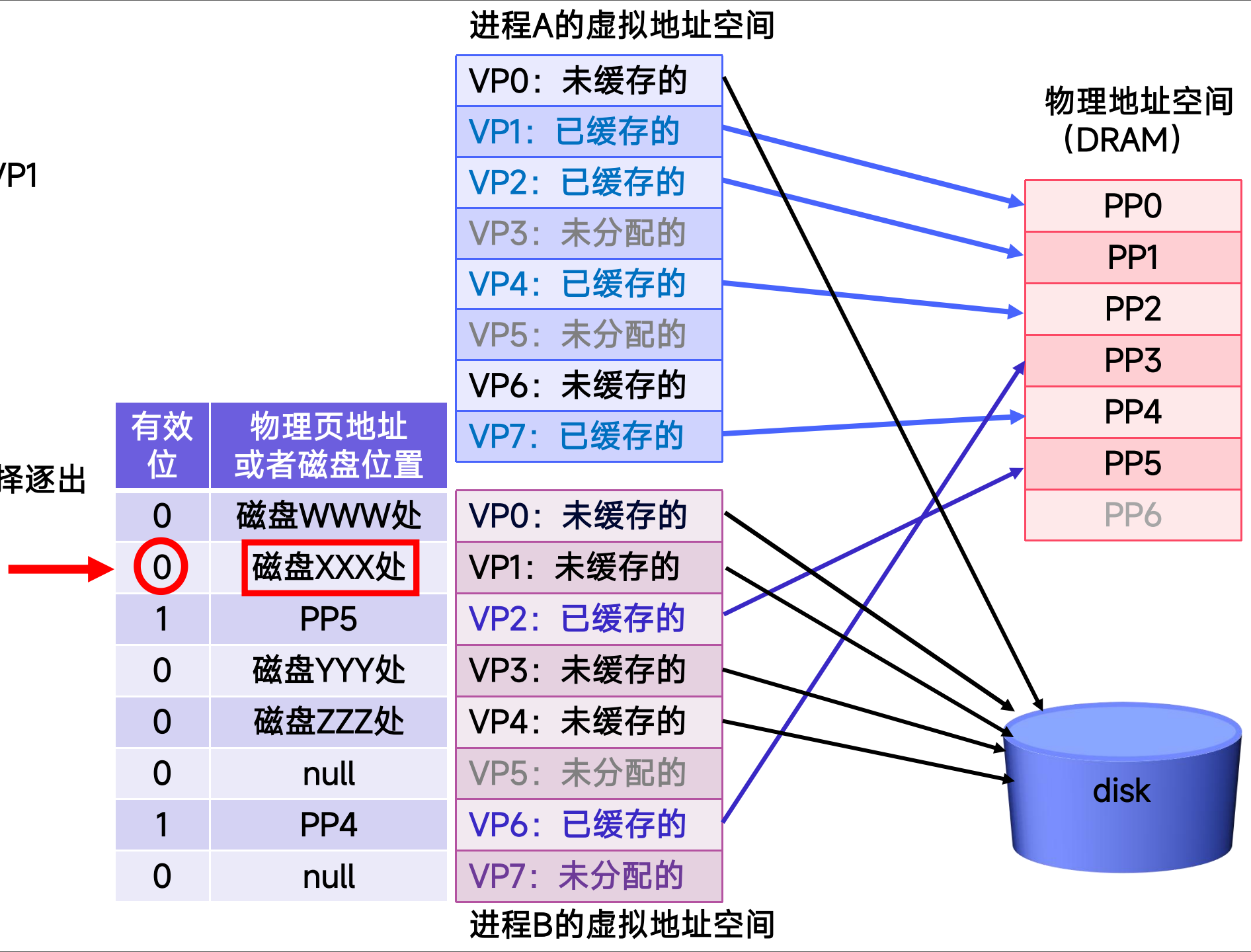
查表，发现页无效

触发缺页故障

页在磁盘里，尝试调入

发现DRAM缓存已满，选择逐出页，假设选择PP6（VP0）

将VP0放入磁盘
并标记VP0的PTE



缺页（2）

运行进程B的CPU：访问VP1

查表，发现页无效

触发缺页故障

页在磁盘里，尝试调入

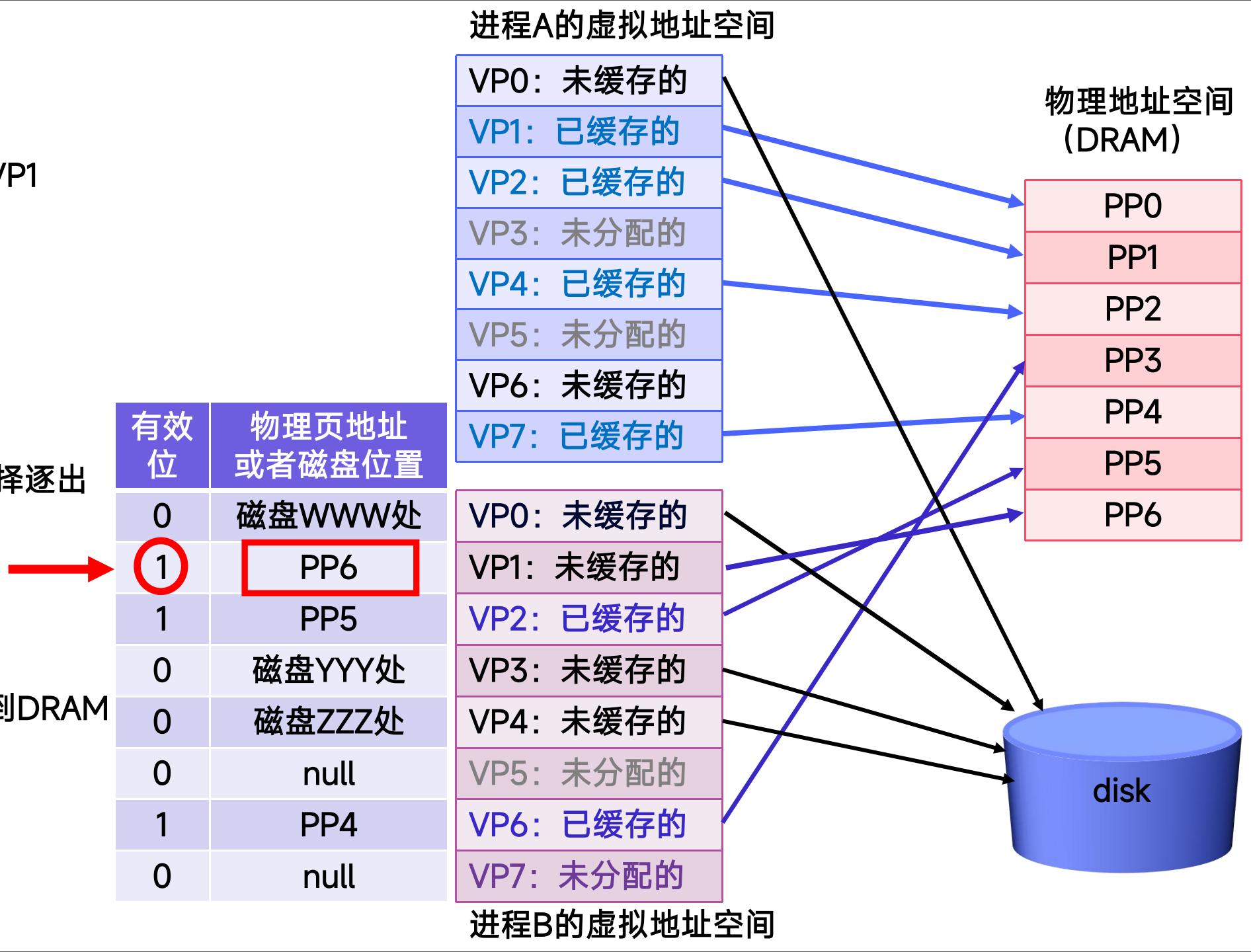
发现DRAM缓存已满，选择逐出页，假设选择PP6（VP0）

将VP0放入磁盘
并标记VP0的PTE

将VP1从磁盘中取出，放到DRAM中，并标记VP1的PTE

从缺页故障返回

再次访问VP1，则成功



缺页（3）

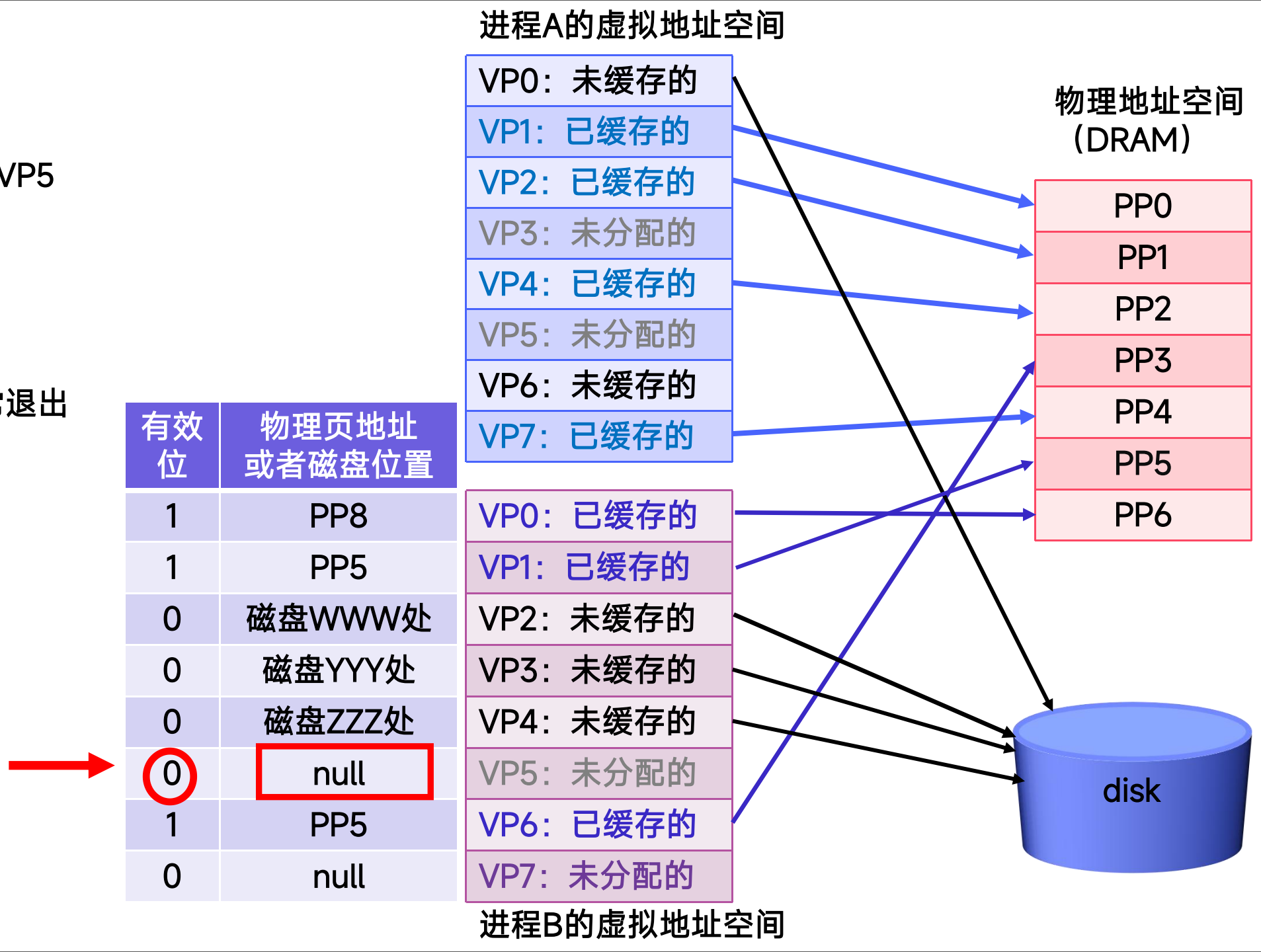
运行进程B的CPU：访问VP5

查表，发现页无效

触发缺页故障

页是未分配的，程序异常退出

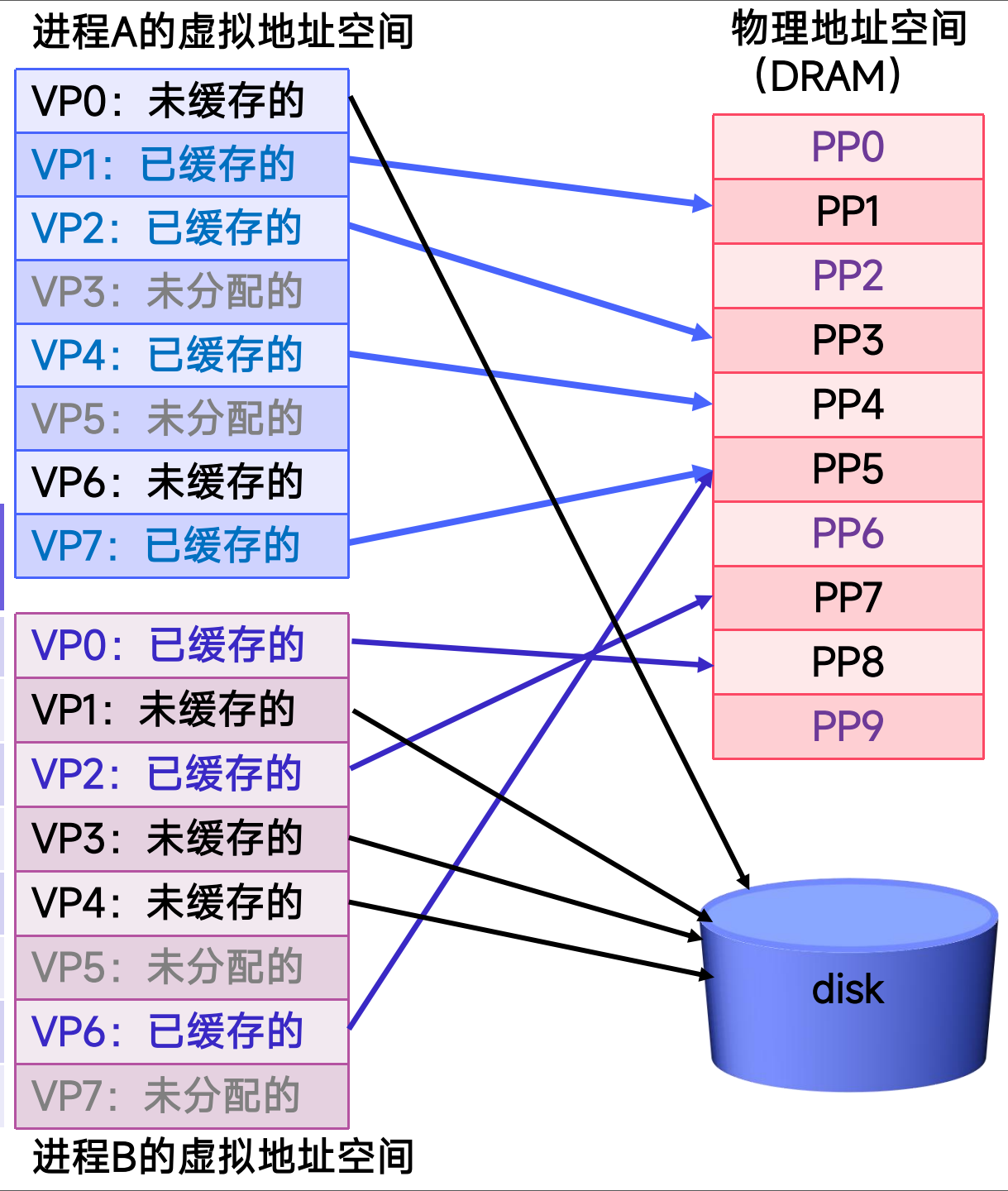
有效位	物理页地址或者磁盘位置
1	PP8
1	PP5
0	磁盘WWW处
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	null



分配页面

假如调用malloc导致堆增长，需要对VP7进行分配

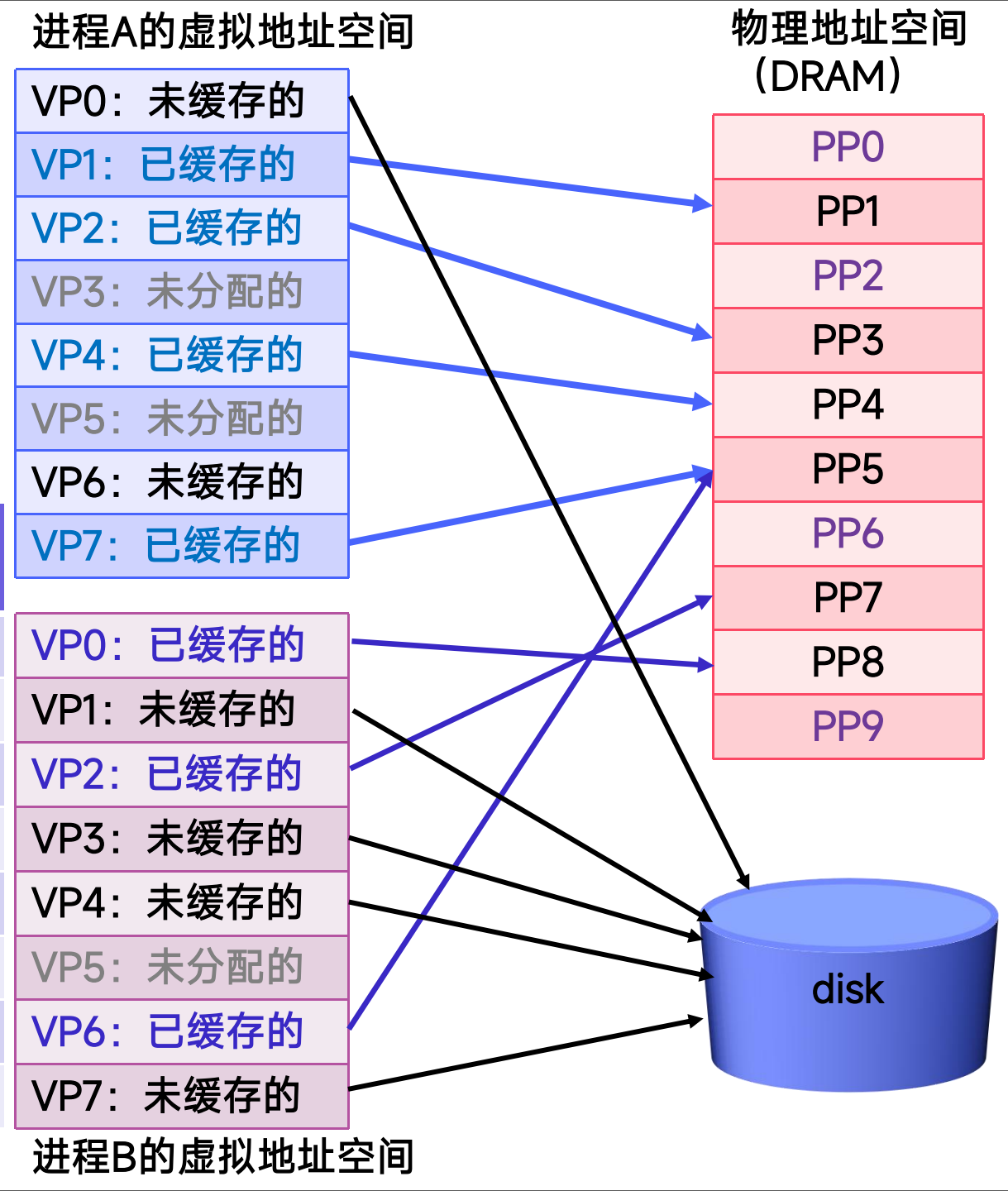
有效位	物理页地址或者磁盘位置
1	PP8
0	磁盘XXX处
1	PP7
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	null



分配页面

假如调用malloc导致堆增长，需要对VP7进行分配

有效位	物理页地址或者磁盘位置
1	PP8
0	磁盘XXX处
1	PP7
0	磁盘YYY处
0	磁盘ZZZ处
0	null
1	PP5
0	磁盘VVV处



虚拟内存作为缓存的工具

可以认为虚拟内存空间本来是在磁盘上的，而将DRAM视作是虚拟内存缓存

DRAM缓存的特点：

访问磁盘的时间开销很大，所以DRAM不命中（缺页）的惩罚很大，因此：

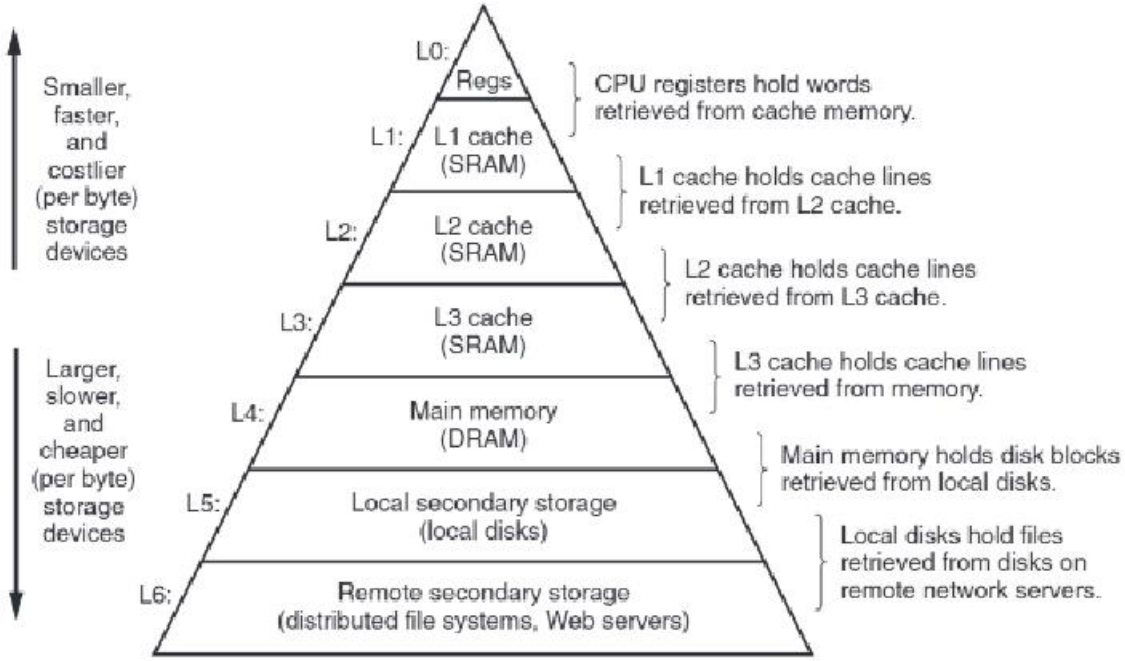
- DRAM是全相联的缓存，即任何虚拟页可以存在任何物理页上
- 页面大小很大，通常4K-2M
- 写策略采用写回而不是直写

虚拟内存的性能不会很差

- 由于程序有局部性，虚拟页面的常驻集合较小。

虚拟内存大小可以超过DRAM缓存大小

- 因为不常用的页面会被从DRAM上逐出



虚拟内存作为内存管理的工具

虚拟内存为每个进程提供了一个抽象屏障，使其感觉不到物理内存的限制。

每个程序都认为它拥有整个地址空间，而不必担心物理内存的大小或位置。

从而简化了内存管理

具体来说，它可以：

- 简化链接
- 简化加载
- 简化共享
- 简化内存分配

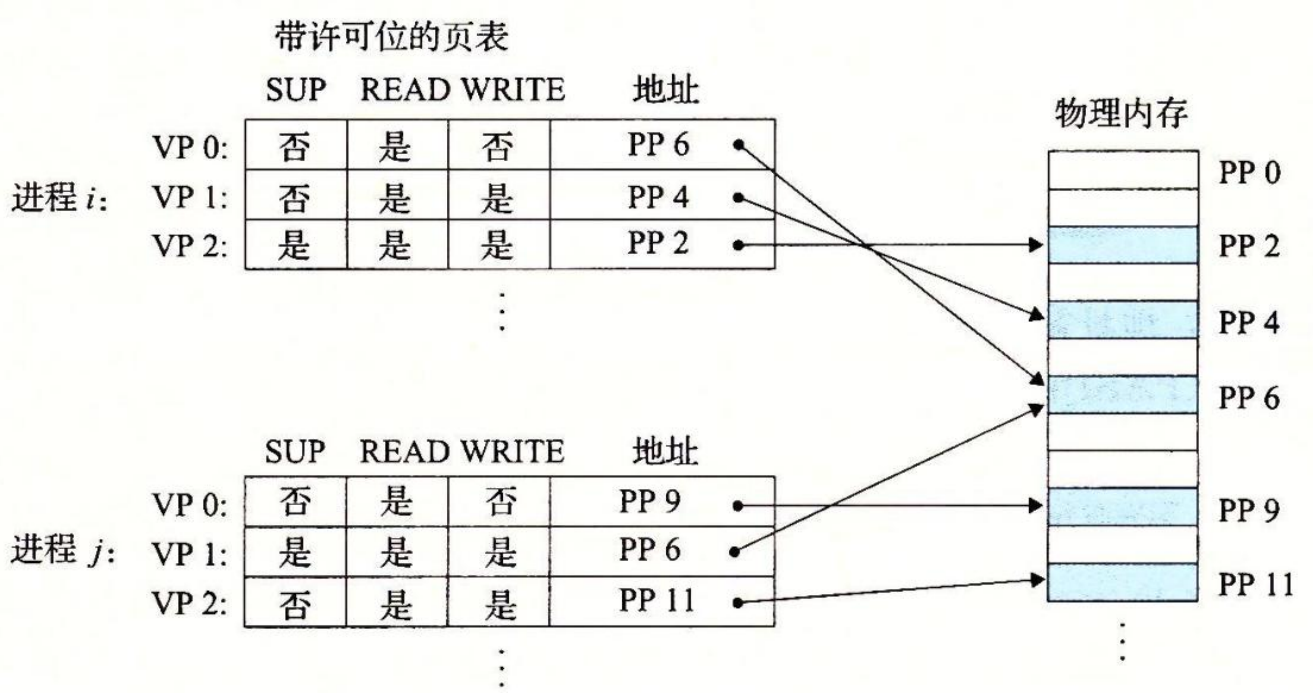
虚拟内存作为内存保护的工具

为了安全和秩序，操作系统必须对进程访问内存做出限制，比如：

- 不允许一个进程直接访问另一个进程的地址空间
- 不允许一个进程在用户模式下访问内核地址空间
- 不允许一个进程修改只读内存区域（如.text，.rodata）

虚拟内存将每个程序的地址空间隔离开来

通过在PTE中加入权限位，来判断此次访问是否合法



地址翻译

术语表

VPN：虚拟页号

VPO：虚拟页偏移

PPN：物理页号

PPO：物理页偏移

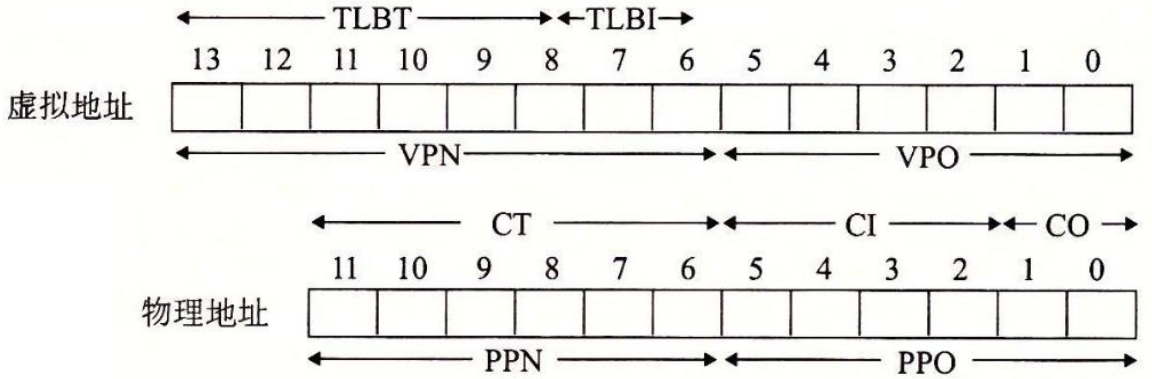
TLBT：TLB标记

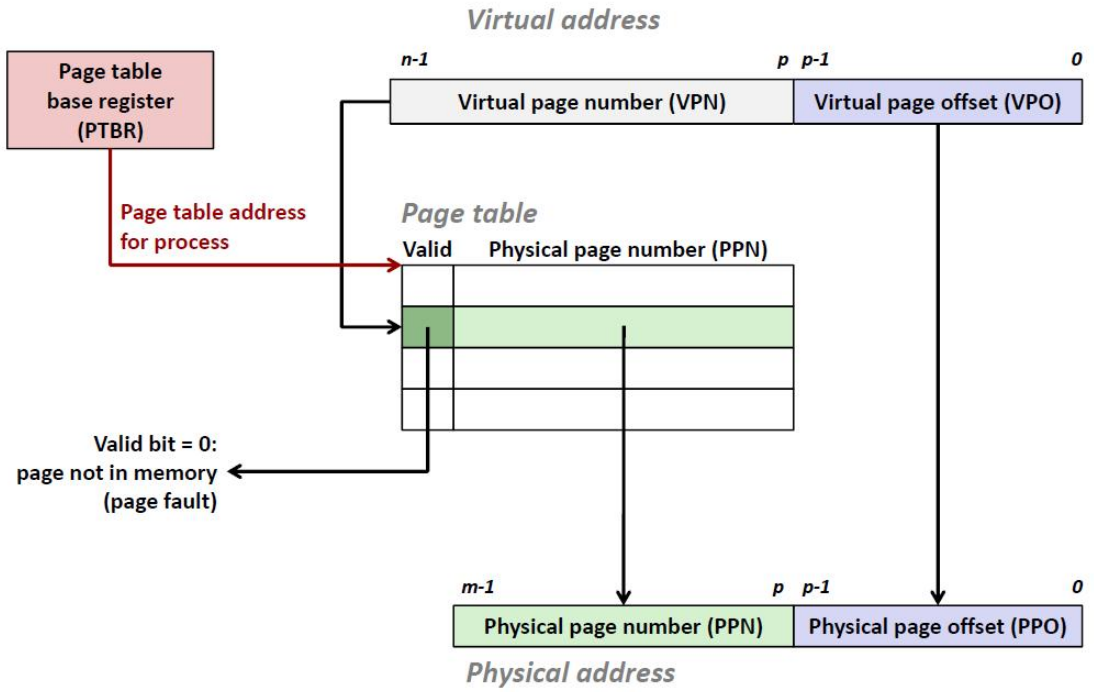
TLBI：TLB组索引

CT：缓存标记

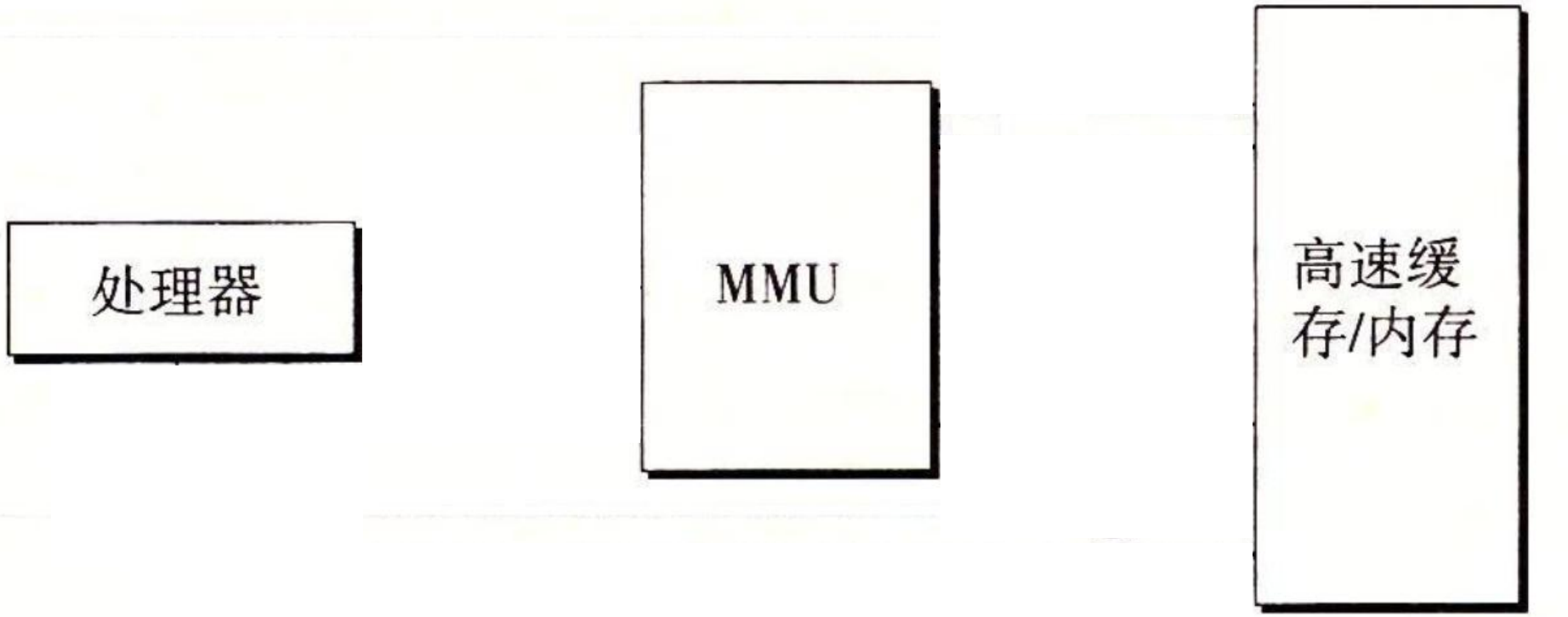
CI：缓存组索引

CO：缓存块偏移

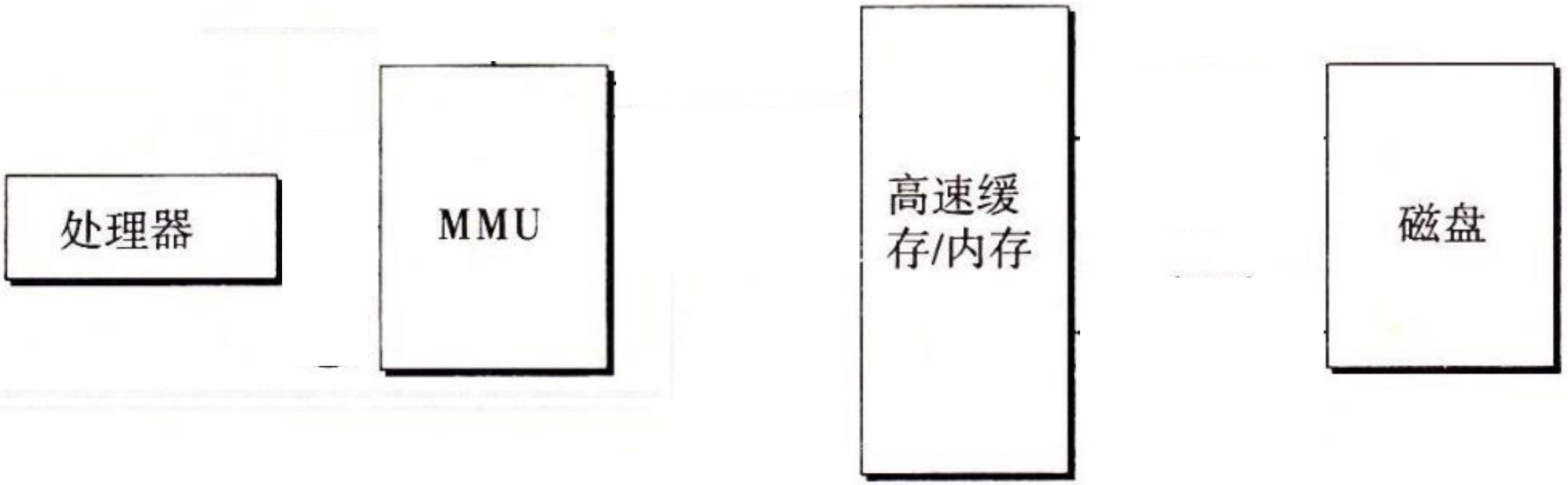




地址翻译——页命中



地址翻译——缺页时页面调入

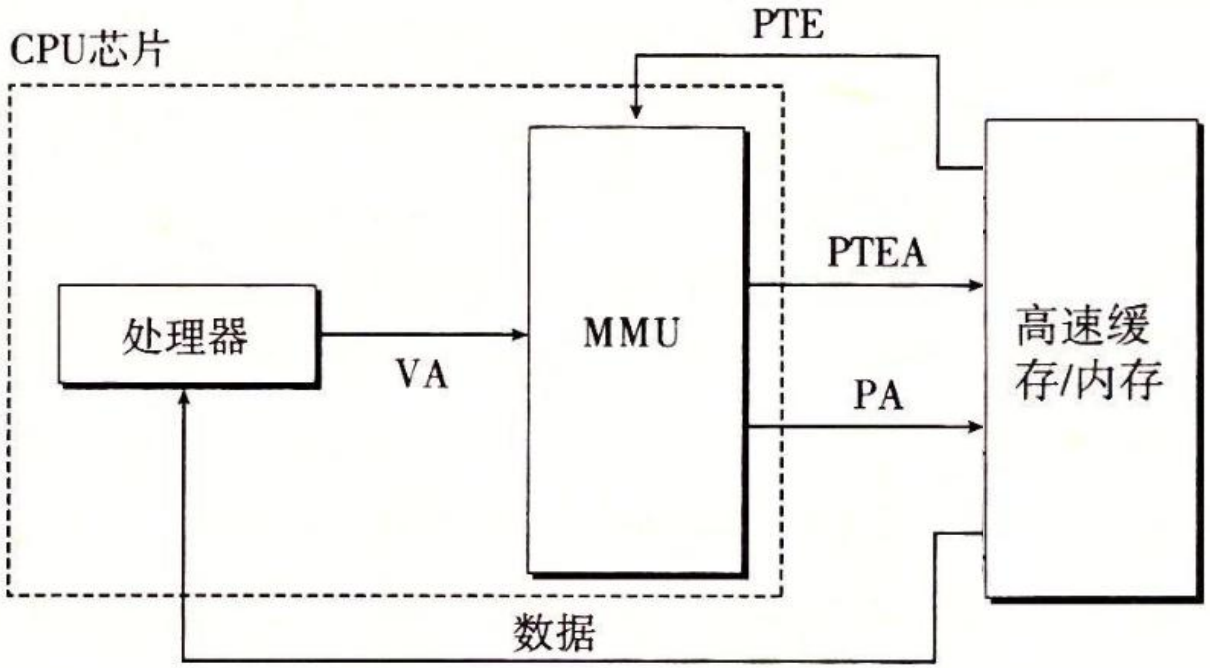


地址翻译——结合高速缓存

页和页表本身都能被缓存在高速缓存中

访问时两者都可能发生不命中

物理寻址的高速缓存：
地址翻译发生在高速缓存查找之前



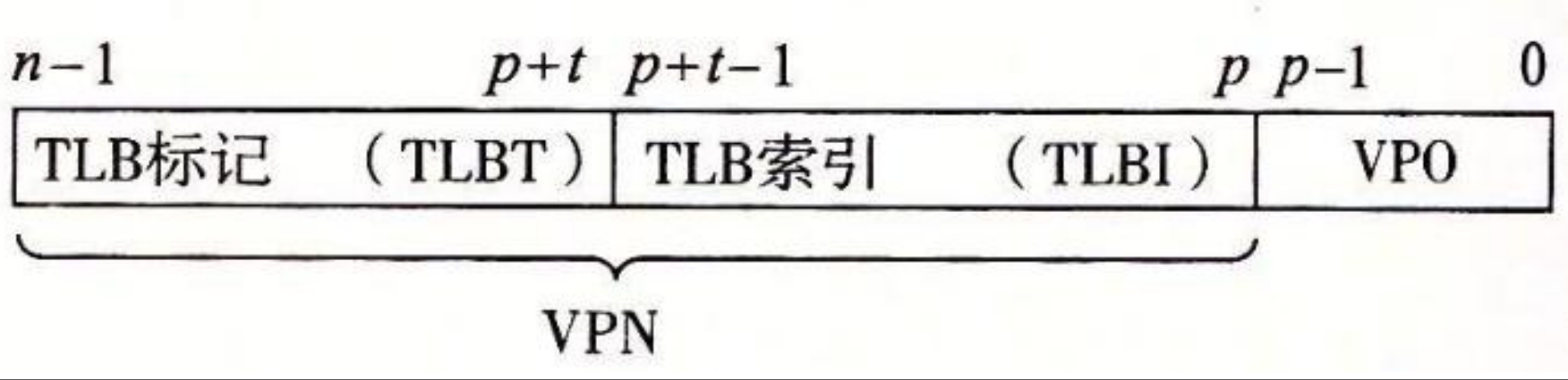
地址翻译——使用TLB进行加速

我们会很频繁的访问PTE，因此需要加快访问PTE的速度
TLB，快表，是一种在CPU中的缓存，用来缓存页表项

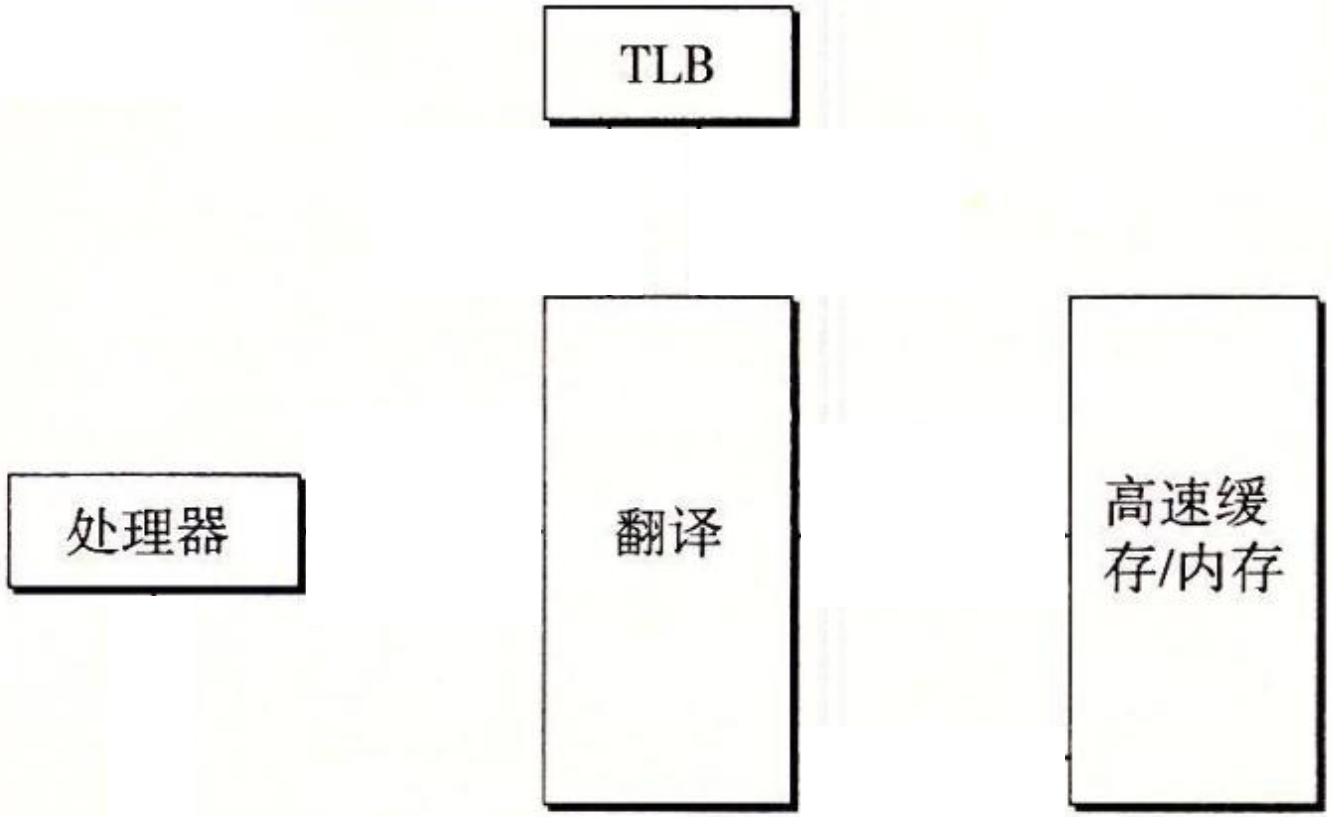
结构是组相联高速缓存，但是每一行是一个PTE
将VPN划分为标记（TLBT）和组索引（TLBI）
（访问TLB时一次访问一行，没有块偏移，VPO不是块偏移而是访问页面时的页偏移）

由于不同的进程使用不同的页表，故在上下文切换时，TLB会被刷新（全部标记为无效）

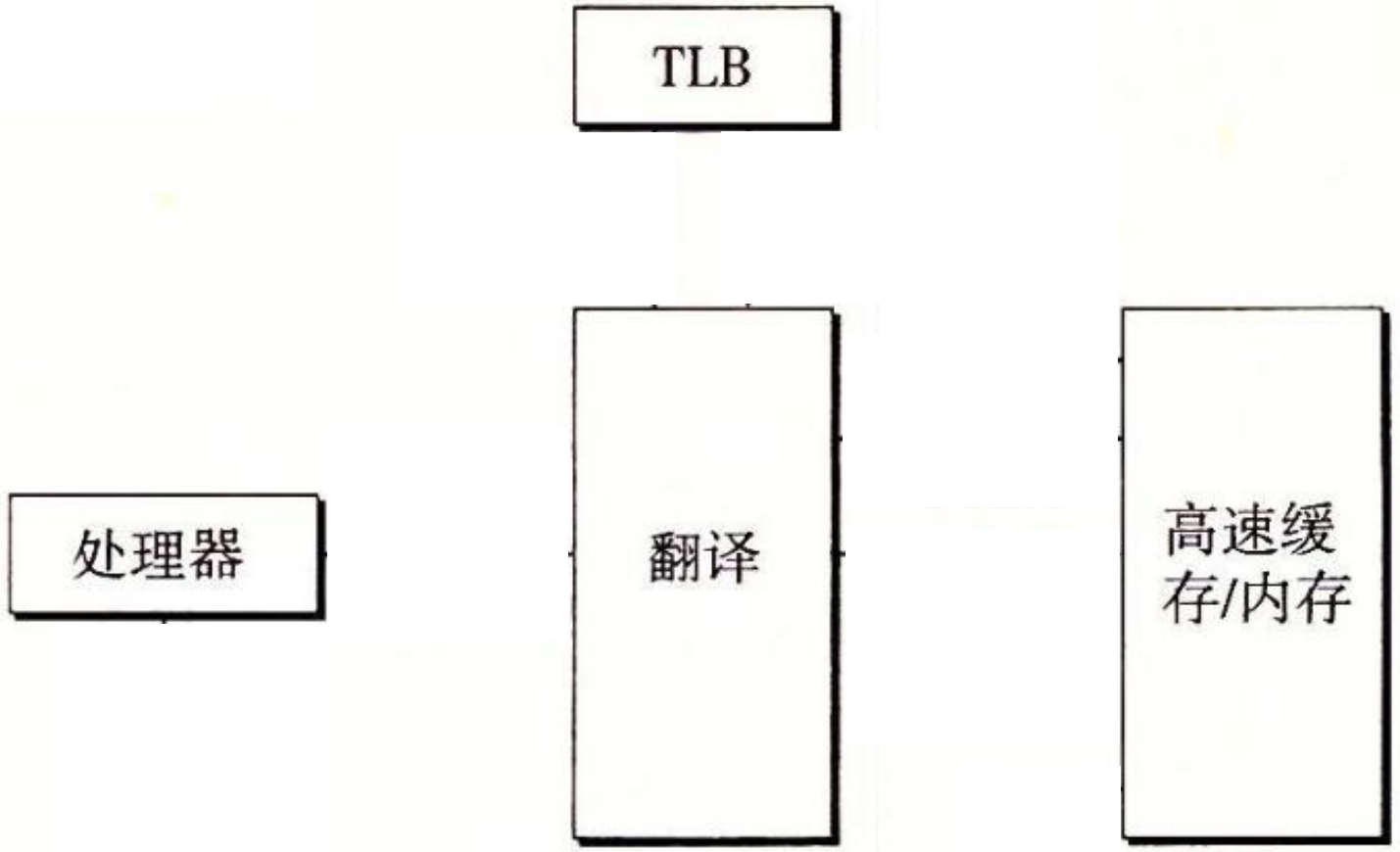
一个虚拟地址：



地址翻译——TLB命中



地址翻译——TLB不命中



问题

Q：地址翻译是硬件行为还是软件行为？

A：地址翻译的过程是由硬件完成的
　　每当程序需要访问一个虚拟地址的时候，硬件（比如 CPU、MMU）自动就执行前两页课件展示的一系列行为，包括访问TLB、访问高速缓存/内存等，来完成从虚拟地址到物理地址的转换。

Q：那软件在虚拟内存系统中发挥了什么作用呢？

A：与虚拟内存相关的数据结构需要由软件（操作系统内核代码）来维护。
　　比如，页表的创建、修改等都是由操作系统负责的。

地址翻译——多级页表

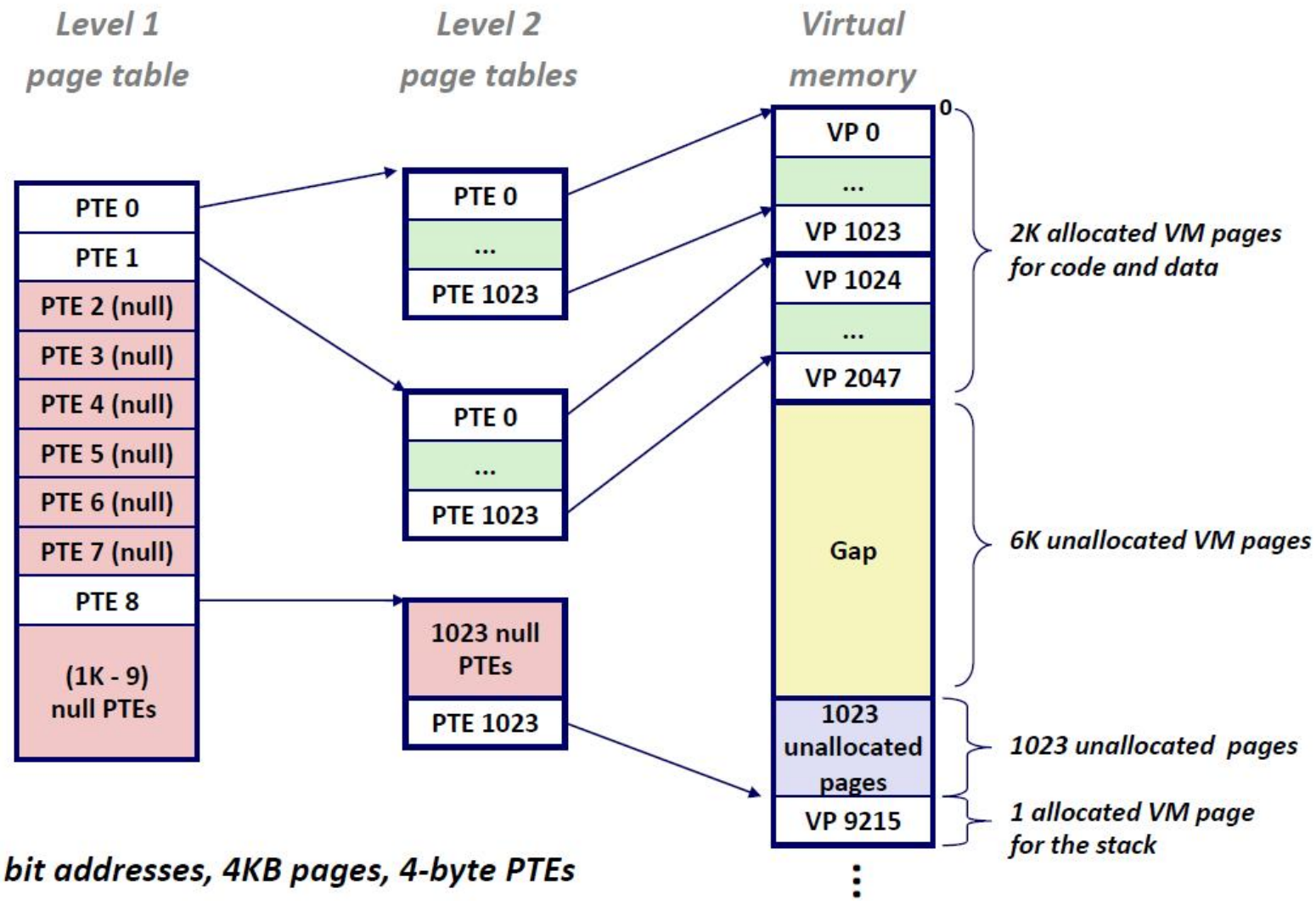
问题：页表需要常驻在物理内存中，占据一片连续的区域
在32位系统中，每个进程需要4MB的页表
在64位系统中，页表更大且无法存储

观察发现：
大部分的虚拟地址空间都是未分配的，因此大部分的PTE也都是未分配的，有一部分页表的PTE全都是未分配的，这种页表就没必要存储

解决方案：多级页表.

从数据结构的方面考虑，单级页表->连续数组，多级页表->树，只需要分配需要的页表（树节点）

右图为二级页表实例，其中第二级只分配了3个表就可以满足要求，而不是1024个



地址翻译——多级页表

一般情况下，每个页表正好占据一个页

VPN被分成k份

先从页表地址寄存器获得第一级页表的物理地址

- 利用VPN1在一级页表上索引，找到一个PTE，获得二级页表的物理地址
- 利用VPN2在二级页表上索引，找到一个PTE，获得三级页表的物理地址
- ……
- ……
- 利用VPNk在k级页表上索引，找到一个PTE，获得要查找的页的物理地址

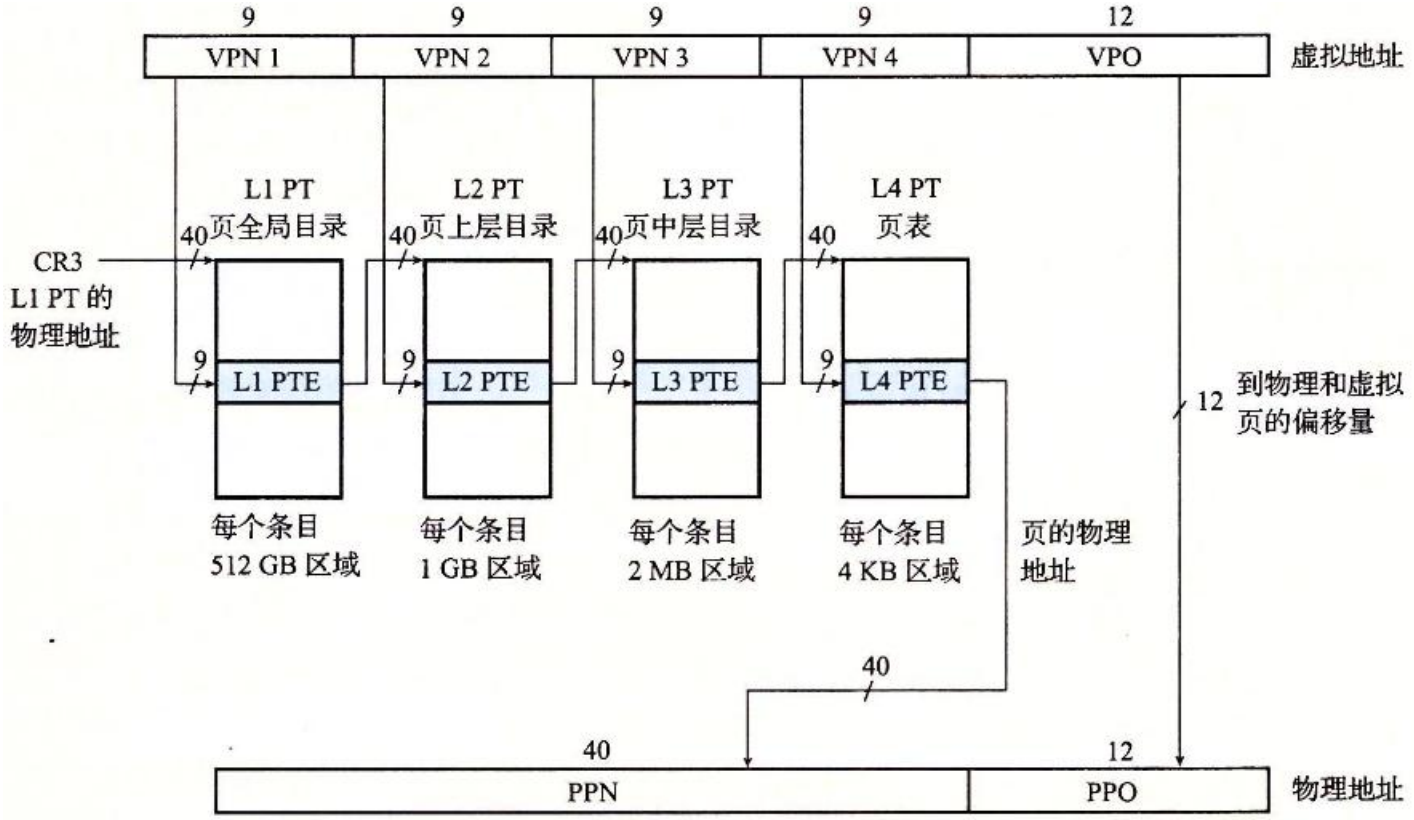


图 9-25 Core i7 页表翻译(PT：页表，PTE：页表条目，VPN：虚拟页号，VPO：虚拟页偏移，PPN：物理页号，PPO：物理页偏移量。图中还给出了这四级页表的 Linux 名字)

Core i7 地址翻译

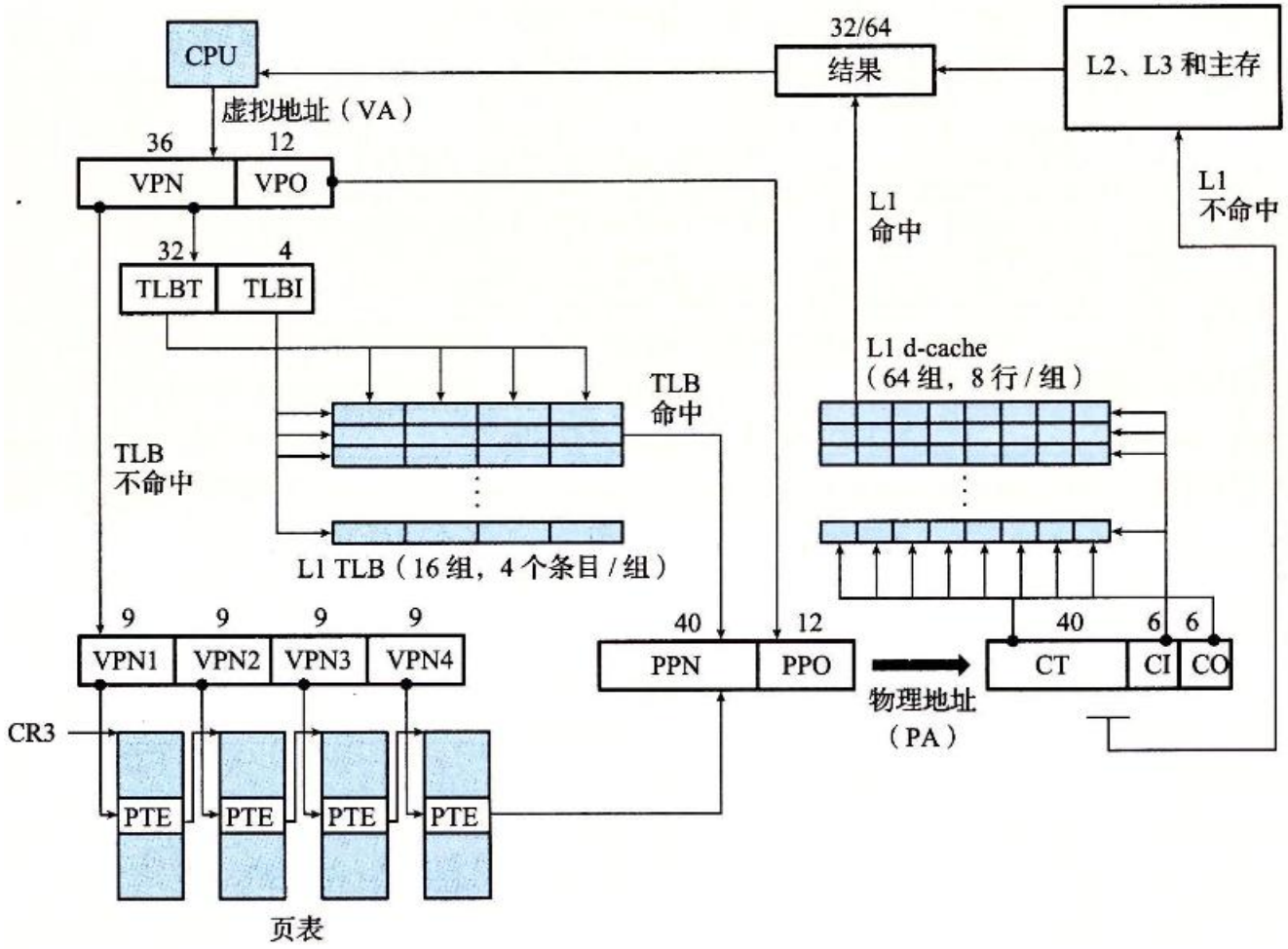
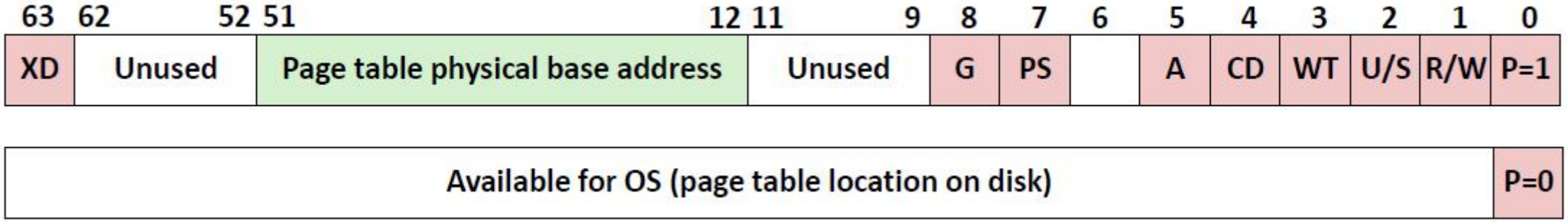


图 9-22　Core i7 地址翻译的概况。为了简化，没有显示 i-cache、i-TLB 和 L2 统一 TLB

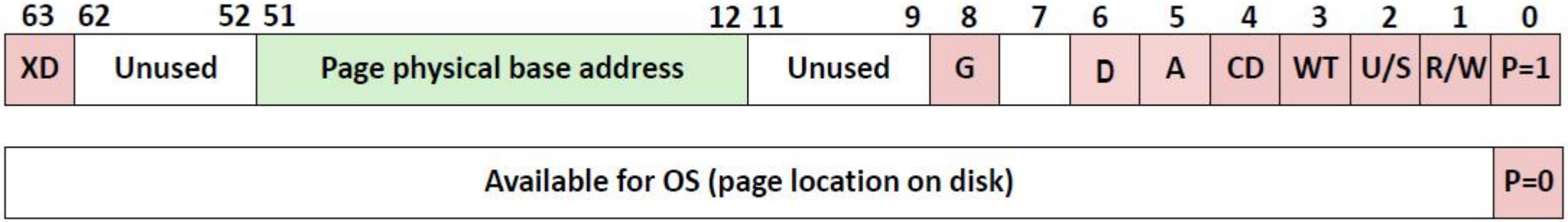
Core i7 一二三级页表的PTE（用来索引下一级页表）



P：子页表在/不在物理内存
R/W: 所有可访问页的只读或读写权限
U/S: 所有可访问页的用户或超级用户（内核）访问权限
WT：子页表直写/写回缓存策略
CD：能/不能缓存子页表
A: 引用位（每次访问都会设置，可以用它实现页替换算法）
PS：4KB或4MB页大小（只对第一层PTE定义）
Base addr：子页表物理地址的高40位（物理页表4KB对齐）
XD：能/不能从这个PTE可访问的所有页中取指令

存储Base Addr的空间用不到64位，高位有空余（因为物理地址没那么大），低位也有空余（因为物理页4K对齐，这些位必是0），可以在这些位上存储一些信息。

Core i7 四级页表的PTE（用来索引物理页）



P：子页在/不在物理内存
R/W: 子页的只读或读写权限
U/S: 子页的用户或超级用户（内核）访问权限
WT：子页直写/写回缓存策略
CD：能/不能缓存子页
A: 引用位（每次访问都会设置，可以用它实现页替换算法）
D: 修改位（写后设置，决定是否写回牺牲页）
G: 全局位（任务切换时，不从TLB中驱逐出去）
Base addr：子页物理地址的高40位（物理页4KB对齐）
XD：能/不能从这个子页中取指令

存储Base Addr的空间用不到64位，高位有空余（因为物理地址没那么大），低位也有空余（因为物理页4K对齐，这些位必是0），可以在这些位上存储一些信息。

问题

Q：“64位处理器”、“48位虚拟地址空间”、“52位物理地址空间”，各自是什么意思？

A：
64位处理器：处理器能够进行 64 bit 的计算
（具体来说，比如算术逻辑运算单元 ALU 支持计算 64 bit 操作数，寄存器能存 64 bit 数据）
48位虚拟地址空间：每个虚拟地址有48个bit
52位物理地址空间：每个物理地址有52个bit

Q：PTE 中存的是物理页号，还是虚拟页号？

A：
PTE 中存的都是物理页号！
例如：一级页表的PTE中存了二级页表所在的页的物理页号
只有知道物理地址，才能把物理地址放到地址总线上去访存

问题

Q：为什么是 48 位虚拟地址空间?

A:
前提1：每个虚拟/物理页大小为 4KB (= 2^12B)
结论1：VPO/PPO 有 12 位

前提2：物理地址有52位，所以每个 PTE 中存的 PPN 有52-12=40位
结论2：每个PTE需要用 8 个 Byte 存

前提3：希望每个页表恰好占满一页
结论3：每个页表页中存放 4KB / 8B = 2^9 个 PTE

前提4：每个页表是 PTE 的数组，索引方式是 VPN
结论4：每一级 VPN 有 9 位

前提5：希望一级页表恰好占满一页
结论5：虚拟地址位数为 n * 9 + 12 （n 为多级页表的级数）

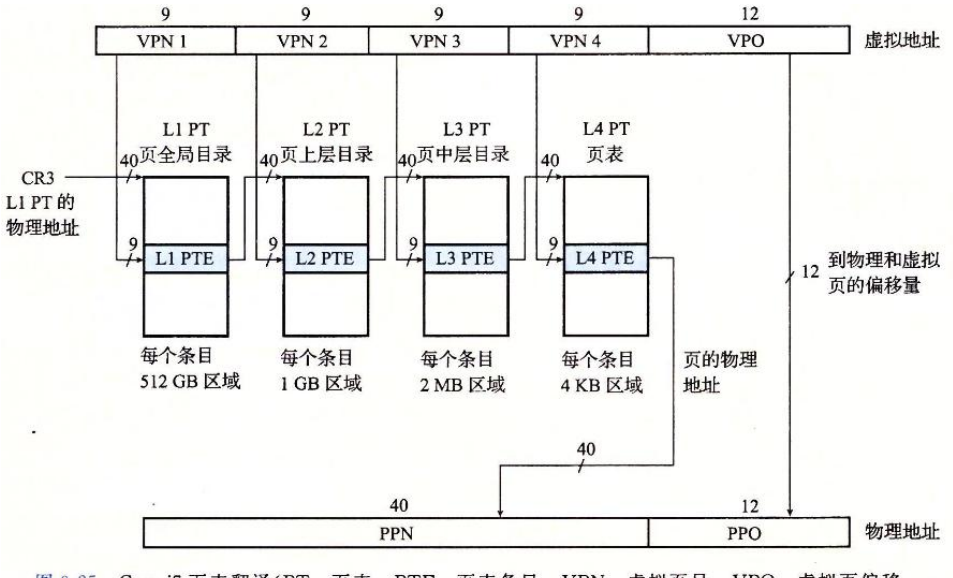


图 9-25 Core i7 页表翻译(PT：页表，PTE：页表条目，VPN：虚拟页号，VPO：虚拟页偏移，PPN：物理页号，PPO：物理页偏移量。图中还给出了这四级页表的 Linux 名字)

4*9+12=48 够用、高效、且美观
照着这个思路，同学们可以算一算为什么在 32 位虚拟地址空间中，使用了 4KB 作为页面大小?

Linux虚拟内存

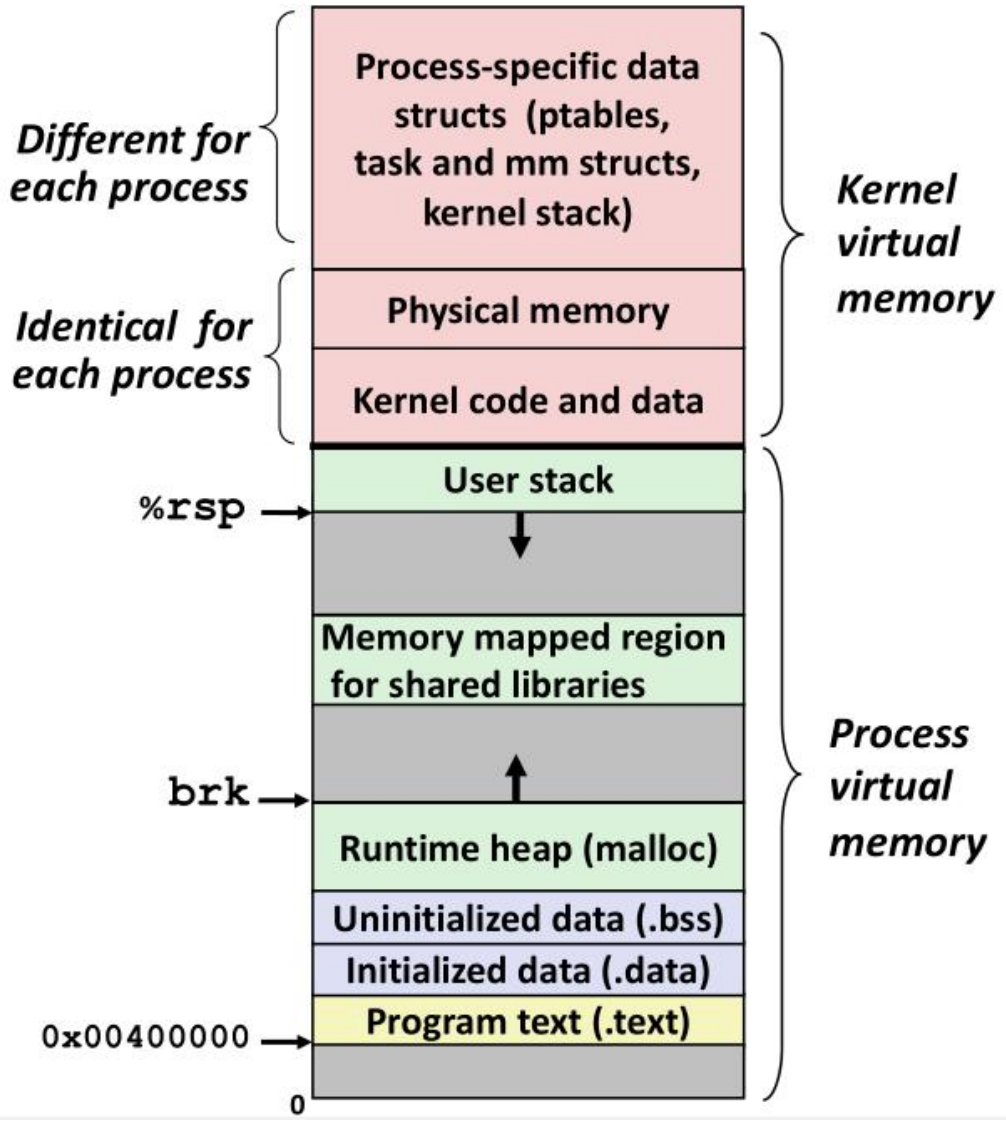
Linux虚拟内存系统

进程相关的结构

- 用户地址空间
- 内核中与进程相关的结构
  - ✓ 包括页表，task和mm结构，内核栈等

进程无关的结构

- 和物理内存直接对应的部分
  - ✓ 便于内核直接访问物理内存
- 内核自身的代码和数据



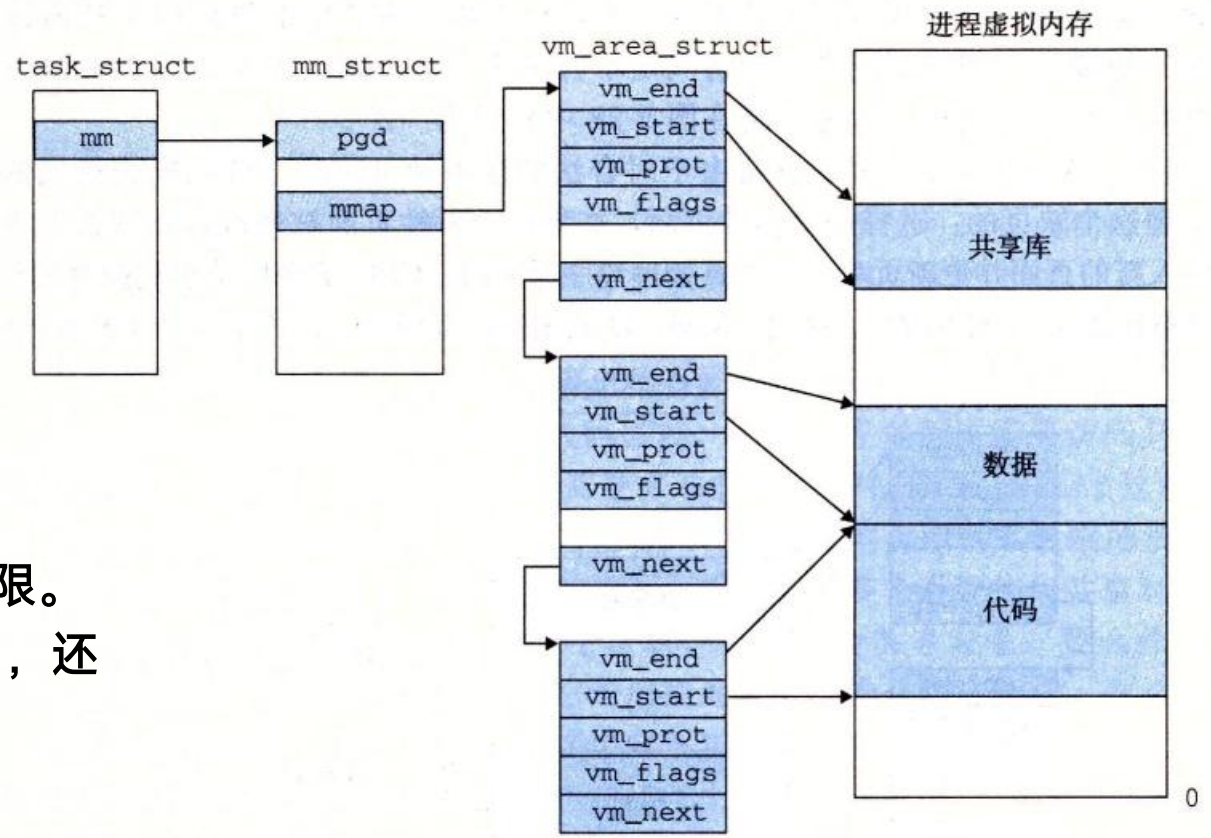
Linux虚拟内存区域

每个进程有一个task_struct，包含PID、指向用户栈的指针、
可执行目标文件的名字、以及程序计数器等。

pgd指向第一级页表(页全局目录)的基址。

mmap指向一个vm_area_structs(区域结构)的链表。

- vm_start:指向这个区域的起始处。
- vm_end:指向这个区域的结束处。
- vm_prot:描述这个区域内包含的所有页的读写许可权限。
- vm_flags:描述这个区域内的页面是与其他进程共享的，还是这个进程私有的(还描述了其他一些信息)。
- vm_next:指向链表中下一个区域结构。



Linux页故障处理

1. 判断地址是不是存在的
   - 和每个段的起始位置作比较，如果没有找到在某段中，则终止

2. 判断访问是不是合法的
   - 查看读/写/执行位

3. 前两个都不是，则页故障是缺页导致的
   - 将页面调入

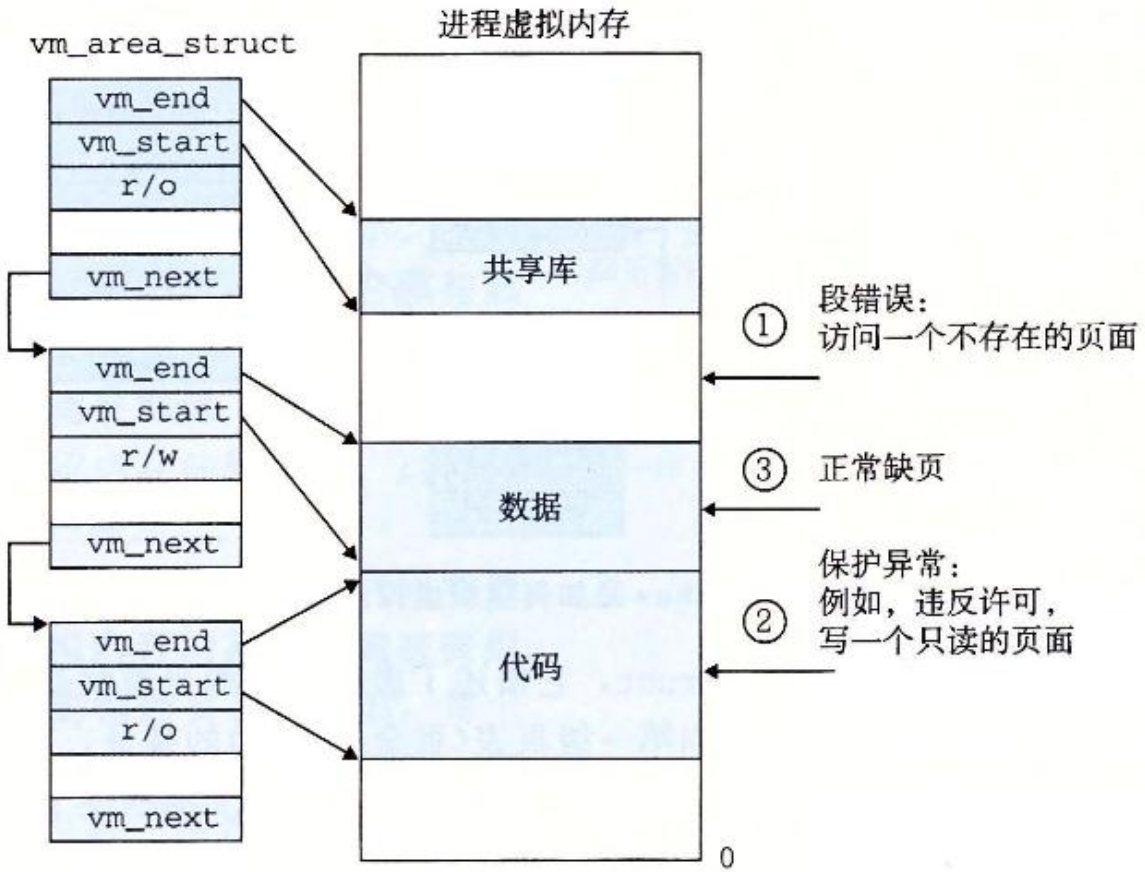


图 9-28　Linux 缺页处理

问题

Q：在 Linux 虚拟内存系统中，在实现上是怎么区分
未分配的 / 已缓存的 / 未缓存的
这三种页的？

A：已缓存的：PTE 被标记了 P 位（有效）
未分配的/未缓存的：PTE 都没有被标记 P 位

但是
未缓存的页对应的地址处在某一个区域结构 vm_area_struct 内
未分配的页不处在任何区域结构 vm_area_struct 内

当硬件进行地址翻译，碰到没有标记 P 位的 PTE 时，都会触发 page fault
内核的 page fault handler 会查询区域结构，从而区分导致 page fault 的原因是访问到了一个未缓存的页还是未分配的页

内存映射和共享

虚拟内存区域可以和一个磁盘上的对象关联起来，以初始化这个虚拟内存区域的内容

- 普通文件（如execve加载的可执行目标文件）
- 匿名文件（请求二进制0，不是真的在磁盘上）

虚拟页面被初始化后，会与交换空间swap file换来换去

共享对象和私有对象

对于每个进程，对象有两类

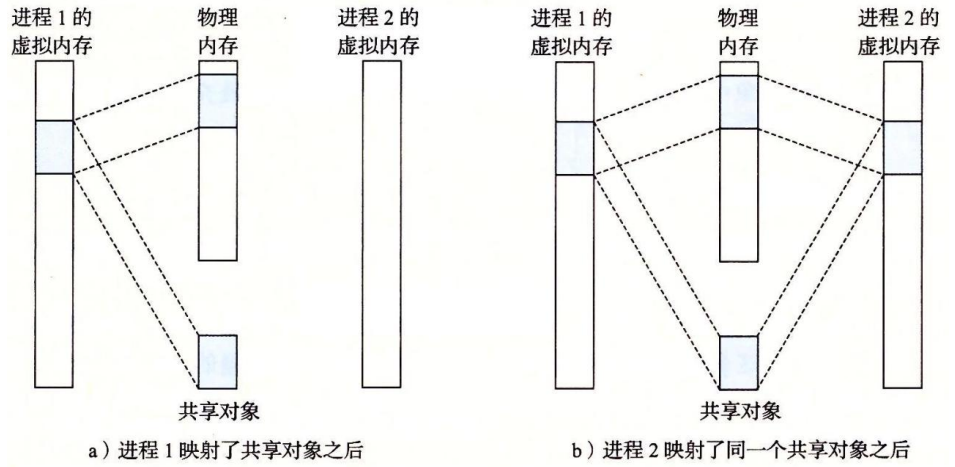
- 共享对象-共享区域：和其他共享这个对象的进程共享
- 私有对象-私有区域：对其他所有进程都不可见

共享对象的映射：

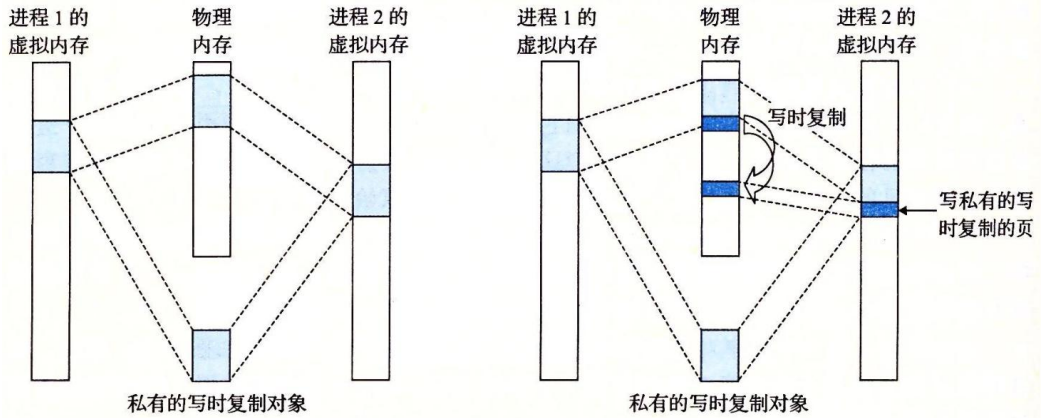
- 多个进程共享同一个对象
- 物理内存中只留一份，各个进程用自己的虚拟地址/VP对应到它

私有对象的映射：

- COW: copy-on-write——保护故障
- 私有对象一开始是只读的
- 当有进程尝试写的时候，被写的页面在物理内存中复制，新复制的页面标记为可写，并且与虚拟页面关联。
- 既实现了私有性，又最大限度的保留了内存



a）进程 1 映射了共享对象之后　　b）进程 2 映射了同一个共享对象之后

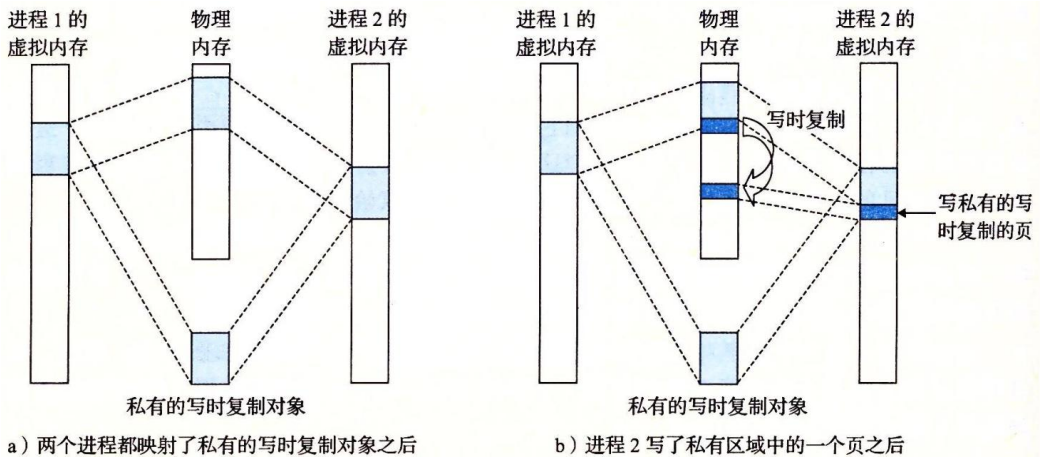


a）两个进程都映射了私有的写时复制对象之后　　b）进程 2 写了私有区域中的一个页之后

fork和execve

fork
- 为新进程创建数据结构，分配pid，创建当前区域结构的原样副本。
- 私有区域的页表条目都被标记为只读
- 私有区域的区域结构被标记为私有的写时复制
- 当任意一个进程试图写私有的写时复制区域中的一个页面时，会创建新页面

execve
- 删除已存在的用户区域
- 映射私有区域
  - ✓ 代码、数据、.bss、栈
- 映射共享区域，
  - ✓ 共享对象（或目标）链接
- 设置程序计数器



a）两个进程都映射了私有的写时复制对象之后　　b）进程 2 写了私有区域中的一个页之后

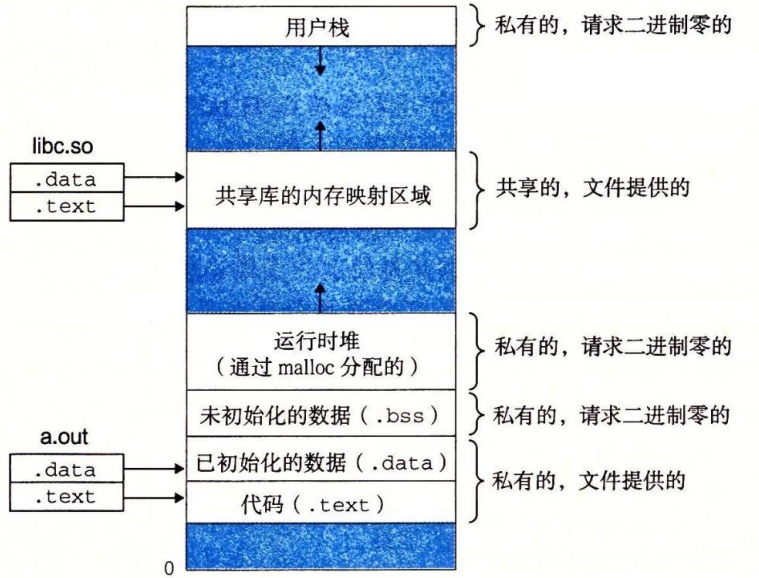


图 9-31　加载器是如何映射用户地址空间的区域的

COW 流程梳理

1. 父进程 F 的地址空间中，有一个可写的私有对象 O （比如 O 是 F 的 .data 段）

“可写”：体现在 O 对应的页的 PTE 中 R/W 位是 1 （可读写）

“私有”：体现在 O 所处的区域的 vm_area_struct 结构有 MAP_PRIVATE 标记

2. 父进程 F 调用 fork()，创建出子进程 S

在父进程和子进程的页表中，O 对应的页的 PTE 中的 R/W 位设置为 0（只读）

在父进程和子进程的 mm_struct 中，把 O 所处的区域的 vm_area_struct 结构都标记为私有的写时复制

3. 当子进程 S 试图写私有对象 O 时

硬件进行地址翻译，发现 PTE 中的 R/W 位是 0 （只读），可是却要进行写操作

硬件触发 page fault 异常

内核运行 page fault handler，查询区域结构，发现 O 所处的区域结构 vm_area_struct 被标记为私有的写时复制

于是 handler 程序在物理内存中创建这个页面的一个新副本，更新进程 S 的页表条目指向这个新的副本，然后在进程 S 的页表中修改 PTE 恢复这个页面的可写权限

问题

Q：硬件在地址翻译的时候会用到“区域”结构（vm_area_struct等）吗？

A：不会！硬件地址翻译的时候只会用到页表。
在软件里才会用到区域结构（比如在内核的 page fault handler 中，要借助 vm_area_struct 来分析错误原因和处理页错误）

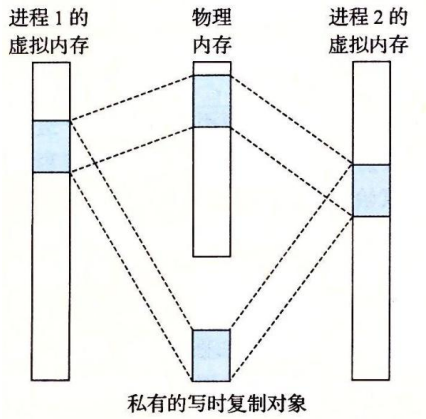
Q：“一个对象是COW的”体现在哪里？

A：1. 其所在页对应的PTE被标记为只读
2. 其所在区域的 vm_area_struct 被标记为私有的写时复制

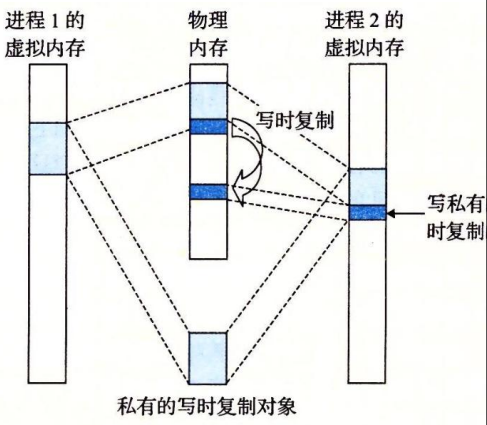
Q：在之前的情景中，如果子进程试图修改一个 COW 的页，那么物理内存中就会出现这个页的两份拷贝。此时，如果父进程又修改了这个 COW 的页，会发生什么？（不要求）

A：在现代 linux 中，当操作系统发现这个 COW 的物理页引用计数为 1 时，会直接原地复用这个物理页，并把页表表项改成可写。不会再做一次拷贝。

附一篇刷到的介绍Linux的COW的知乎文章：
https://zhuanlan.zhihu.com/p/464765151



a）两个进程都映射了私有的写时复制对象之后



b）进程 2 写了私有区域中的一个页之后

mmap和munmap

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

在进程的地址空间中创建映射区域

- addr: 指定映射区域的首地址，通常设为NULL，让系统自动选择合适的地址。
- length: 映射区域的长度（字节数）。
- prot: 保护标志，指定对映射区域的访问权限，如 PROT_READ、PROT_WRITE、PROT_EXEC等。
- flags: 控制映射区域的属性，如 MAP_SHARED、MAP_PRIVATE等。
- fd: 文件描述符，指定要映射的文件，如果是匿名映射，则传入 -1。
- offset: 文件映射的偏移量。

- 成功时返回映射区的起始地址，失败时返回 MAP_FAILED

int munmap(void *addr, size_t length);

解除内存映射

- addr: 映射区域的首地址。
- length: 映射区域的长度。

- 成功时返回 0，失败时返回 -1。

mmap和munmap

code/vm/mmapcopy.c

```c
#include "csapp.h"

/*
 * mmapcopy - uses mmap to copy file fd to stdout
 */
void mmapcopy(int fd, int size)
{
    char *bufp; /* ptr to memory-mapped VM area */

    bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    Write(1, bufp, size);
    return;
}

/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required command-line argument */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

    /* Copy the input argument to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```

code/vm/mmapcopy.c

思考：用 mmap 比直接 read/write 好在哪里?

分享刷到的另一篇知乎文章（我怎么这么喜欢刷知乎）：
https://zhuanlan.zhihu.com/p/616105519?utm_psn=1707538270748209152

12、进程 P1 通过 fork()函数产生一个子进程 P2。假设执行 fork()函数之前，进程 P1 占用了 53 个（用户态的）物理页，则 fork 函数之后，进程 P1 和进程 P2 共占用_____个（用户态的）物理页；假设执行 fork()函数之前进程 P1 中有一个可读写的物理页，则执行 fork()函数之后，进程 P1 对该物理页的页表项权限为_____。上述两个空格对应内容应该是（　　）

A．53，读写　B．53，只读　C．106，读写　D．106，只读

B

练习

(C)3.假设有一台 64 位的计算机的物理页块大小是 8KB,采用三级页表进行虚拟地址寻址,它的虚拟地址的 VPO(Virtual Page Offset,虚拟页偏移)有 13 位,问它的虚拟地址的 VPN(Virtual Page Number,虚拟页号码)有多少位?

A. 20
B. 27
C. 30
D. 33

3.C.本题考查对页表组成的理解.页块大小为 8KB,即 2^13 byte.在 64 位机器上,一个页表条目为 8byte.故共有页表条目 2^10 项,故每一级页表可以表示 10 位地址.因此三级页表存储, 共需要 10*3=30 位.

练习

(B)12.假定整型变量 A 的虚拟地址空间为 0x12345cf0,另一整形变量 B 的虚拟地址 0x12345d98,假定一个 page 的长度为 0x1000 byte,A 的物理地址数值和 B 的物理地址数值关系应该为:

A.A 的物理地址数值始终大于 B 的物理地址数值
B.A 的物理地址数值始终小于 B 的物理地址数值
C.A 的物理地址数值和 B 的物理地址数值大小取决于动态内存分配策略
D.无法判定两个物理地址值的大小

12. B.考察物理地址和虚拟地址的映射关系.同一个 page.

THANK YOU

Come On !