地址翻译

Address Translation

本章内容

Topic

- 地址翻译

  Address Translation

- 举例：简单的内存系统

  Simple Memory System example

- 案例研究：Core i7 / Linux 内存系统

  Case study: Core i7/Linux Memory System

- 内存映射

  Memory mapping

地址翻译

Address Translation

虚拟内存地址翻译
VM Address Translation

- 虚拟地址空间
  Virtual Address Space

  V = {0, 1, ···, N–1}

- 物理地址空间
  Physical Address Space

  P = {0, 1, ···, M–1}

- 地址翻译
  Address Translation
  - MAP：**V → P U { Φ }**
  - 对于虚拟地址 **a**：
    For virtual address **a**:
    - **MAP (a) = a’** 如果虚拟地址为 **a**的数据存储在物理地址**a’**中 **(a’ ∈P)**
      MAP (a) = a’ if data at virtual address **a** is at physical address **a’** in **P**
    - **MAP (a) = Φ** 如果虚拟地址为 **a**的数据不在物理内存中
      MAP (a) = Φ if data at virtual address **a** is not in physical memory
      - 可能是一个非法地址，也可能是数据存储在磁盘上
        Either invalid or stored on disk

地址翻译
Address Translation

地址翻译中用到的符号
Address Translation Symbols

- 基本参数
  Basic Parameters
  - $N = 2^n$：虚拟地址空间的地址数量
    Number of addresses in virtual address space
  - $M = 2^m$：物理地址空间的地址数量
    Number of addresses in physical address space
  - $P = 2^p$：页大小（字节）
    Page size (bytes)

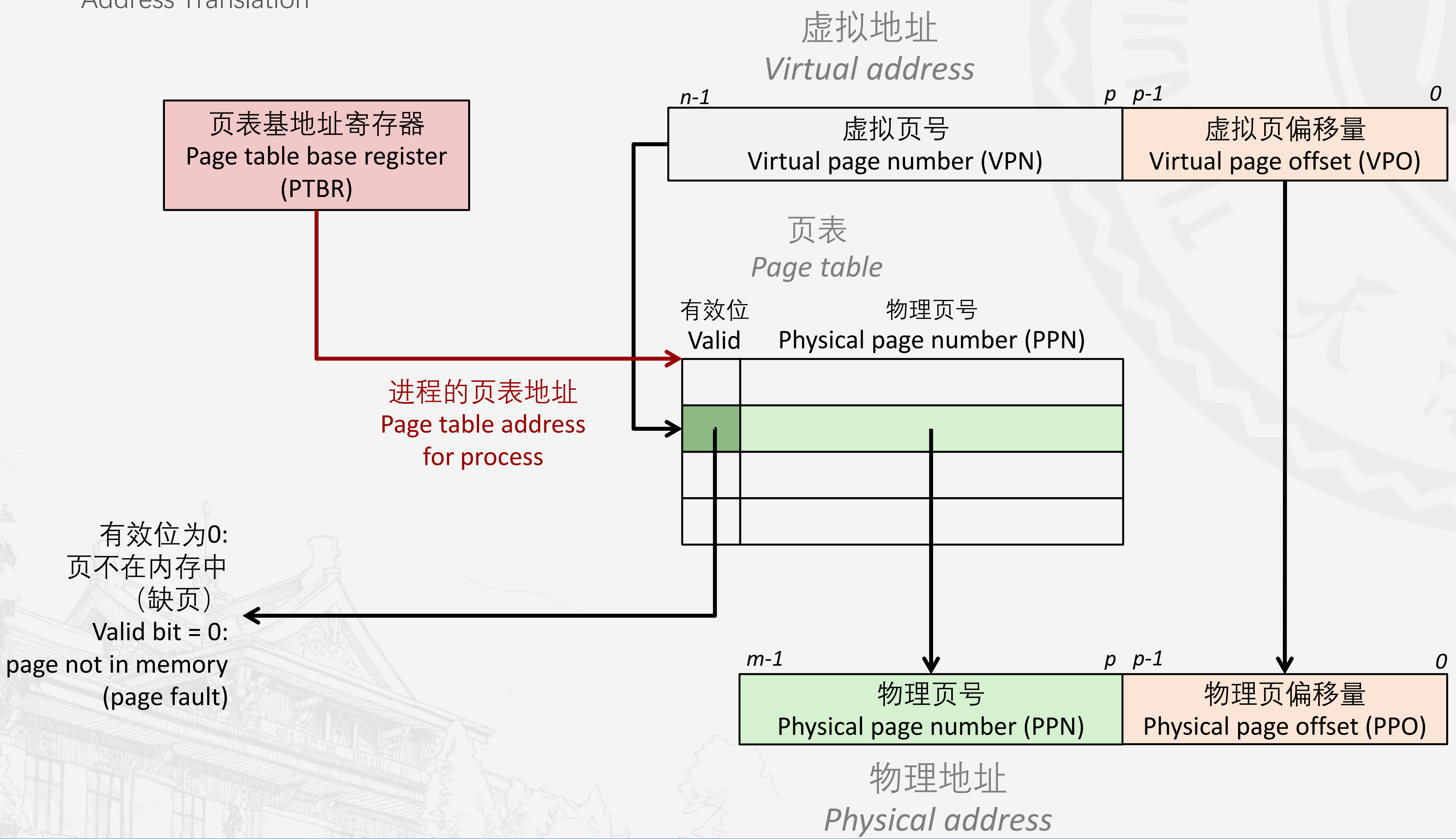
- 虚拟地址的组成
  Components of the virtual address (VA)
  - **VPO：** 虚拟页偏移量
    Virtual page offset
  - **VPN：** 虚拟页号
    Virtual page number
  - **TLBI：** TLB组索引
    TLB index
  - **TLBT：** TLB标记
    TLB tag

- 物理地址的组成
  Components of the physical address (PA)
  - **PPO：** 物理页偏移量（和VPO相同）
    Physical page offset (same as VPO)
  - **PPN：** 物理页号
    Physical page number
  - **CO：** 缓存块偏移量
    Byte offset within cache line
  - **CI：** 缓存组索引
    Cache index
  - **CT：** 缓存标识
    Cache tag

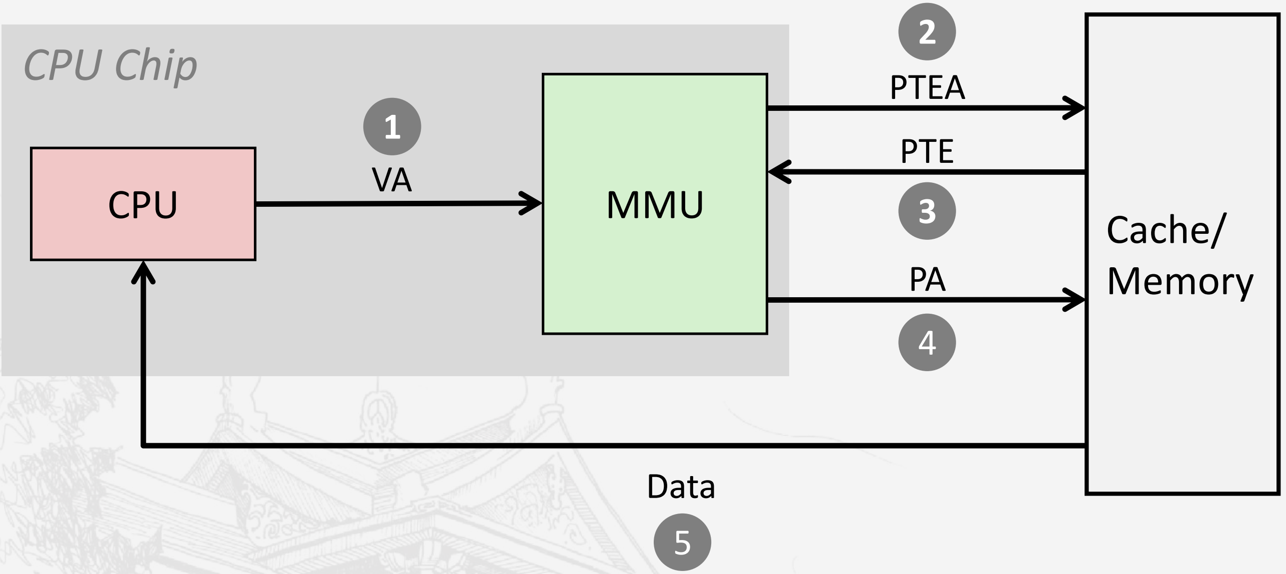
地址翻译

Address Translation

使用页表翻译地址
Address Translation With a Page Table



地址翻译：页命中
Address Translation: Page Hit



1. 处理器发送虚拟地址至MMU
   Processor sends virtual address to MMU

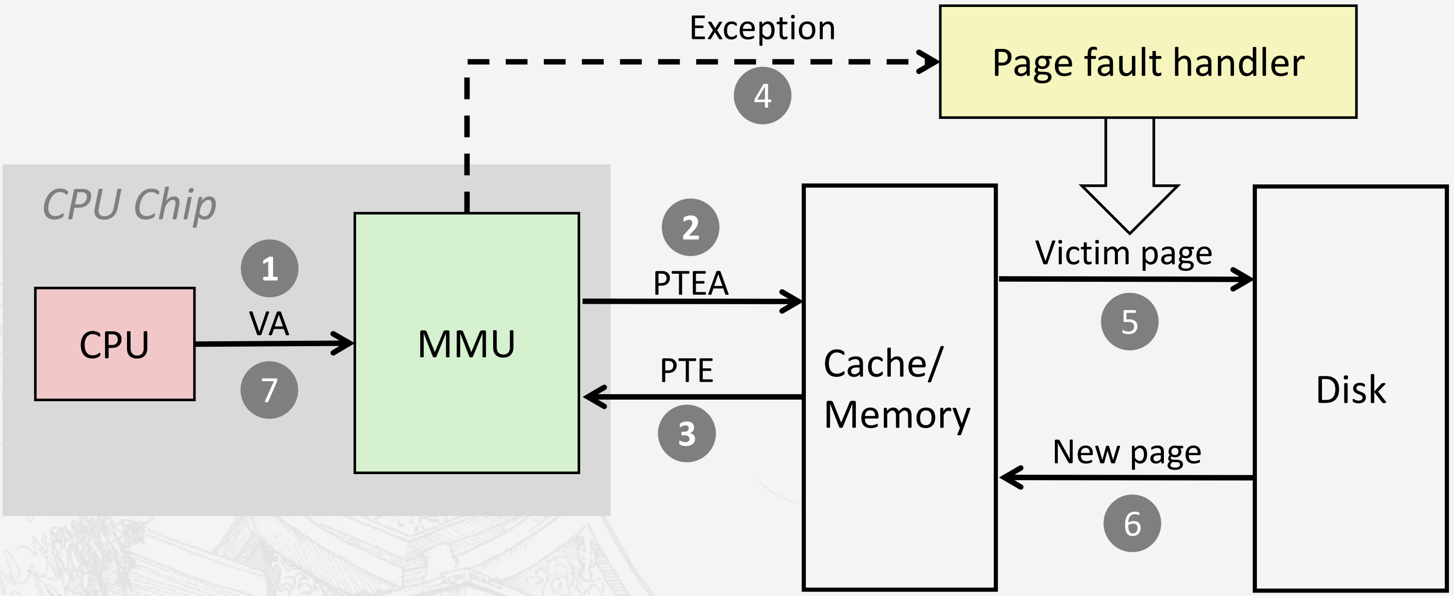
2-3. MMU从内存中取出页表项（PTE）
   MMU fetches PTE from page table in memory

4. MMU向高速缓存/内存发送物理地址
   MMU sends physical address to cache/memory

5. 高速缓存/内存发送数据至处理器
   Cache/memory sends data word to processor

VA: virtual address, PA: physical address, PTE: page table entry, PTEA: PTE address

地址翻译：缺页
Address Translation: Page Fault



1. 处理器发送虚拟地址至MMU
   Processor sends virtual address to MMU

2-3. MMU从内存中取出页表项（PTE）
   MMU fetches PTE from page table in memory

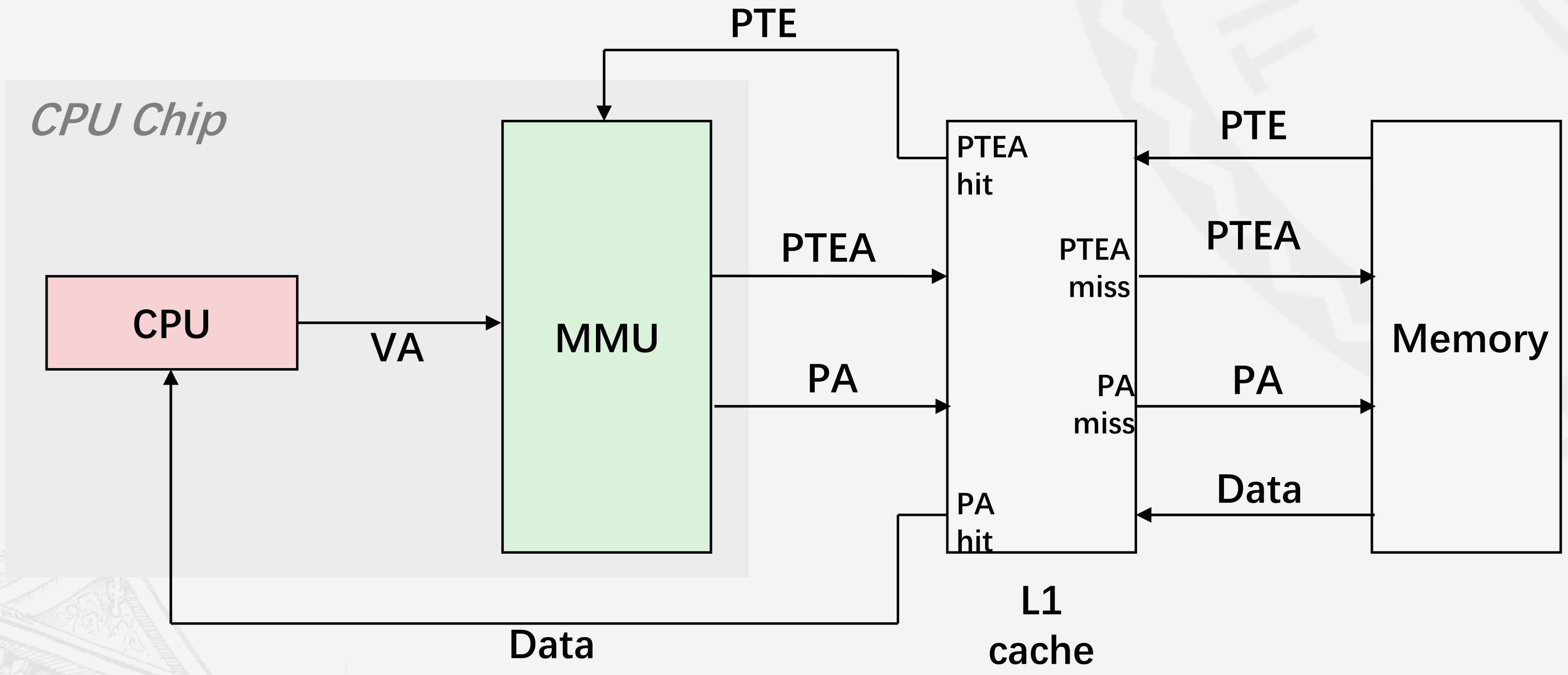
4. 有效位为0，MMU触发缺页故障异常
   Valid bit is zero, so MMU triggers page fault exception

5. 异常处理程序选择一个页换出（如果脏标志位置位，页将写回磁盘）
   Handler identifies victim (and, if dirty, pages it out to disk)

6. 异常处理程序将新的页加载至内存，并更新PTE
   Handler pages in new page and updates PTE in memory

7. 异常处理程序返回至原进程，重新执行引起故障异常的指令
   Handler returns to original process, restarting faulting instruction

把虚拟内存和高速缓存结合起来
Integrating VM and Cache



VA: virtual address, PA: physical address, PTE: page table entry, PTEA: PTE address

利用TLB加速地址翻译
Speeding up Translation with a TLB

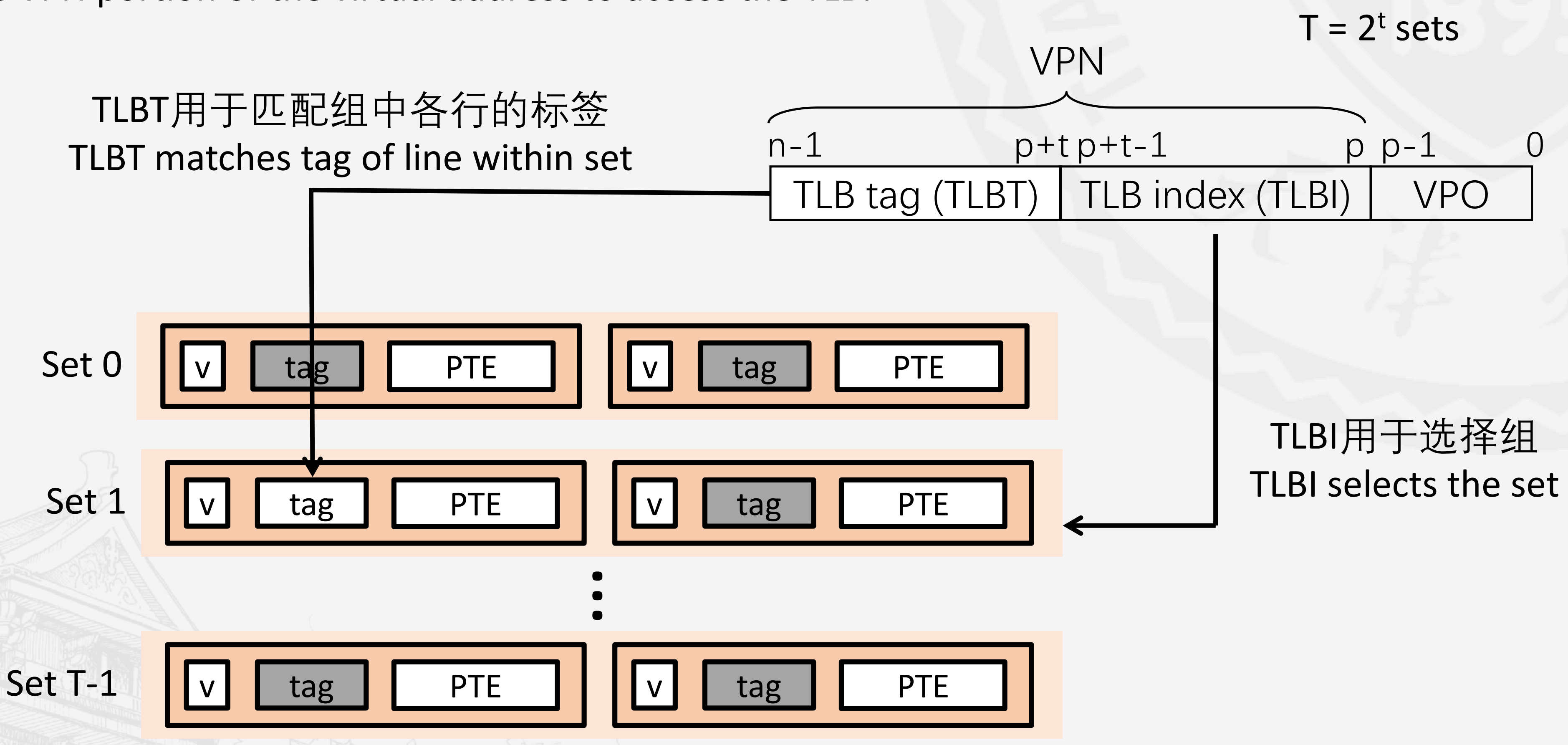
- 页表项和其他内存中的数据一样，被缓存在一级高速缓存内
  Page table entries (PTEs) are cached in L1 like any other memory word
  - 在其他数据页被引用时，页表项所在页可能会被换出内存
    PTEs may be evicted by other data references
  - PTE命中至少需要一个一级缓存的延迟时间（虽然很小）
    PTE hit still requires a small L1 delay

- 解决方案：**翻译后备缓冲器**（又译为：旁视缓冲器、快表）（TLB）
  Solution: **Translation Lookaside Buffer** (TLB)
  - 位于MMU中的一个小的硬件缓存结构
    Small hardware cache in MMU
  - 内部包含少量完整的页表项
    Contains complete page table entries for small number of pages
  - 记录虚拟页号与物理页号间的映射关系
    Maps virtual page numbers to physical page numbers

访问TLB
Accessing the TLB

MMU使用虚拟地址的VPN部分访问TLB：

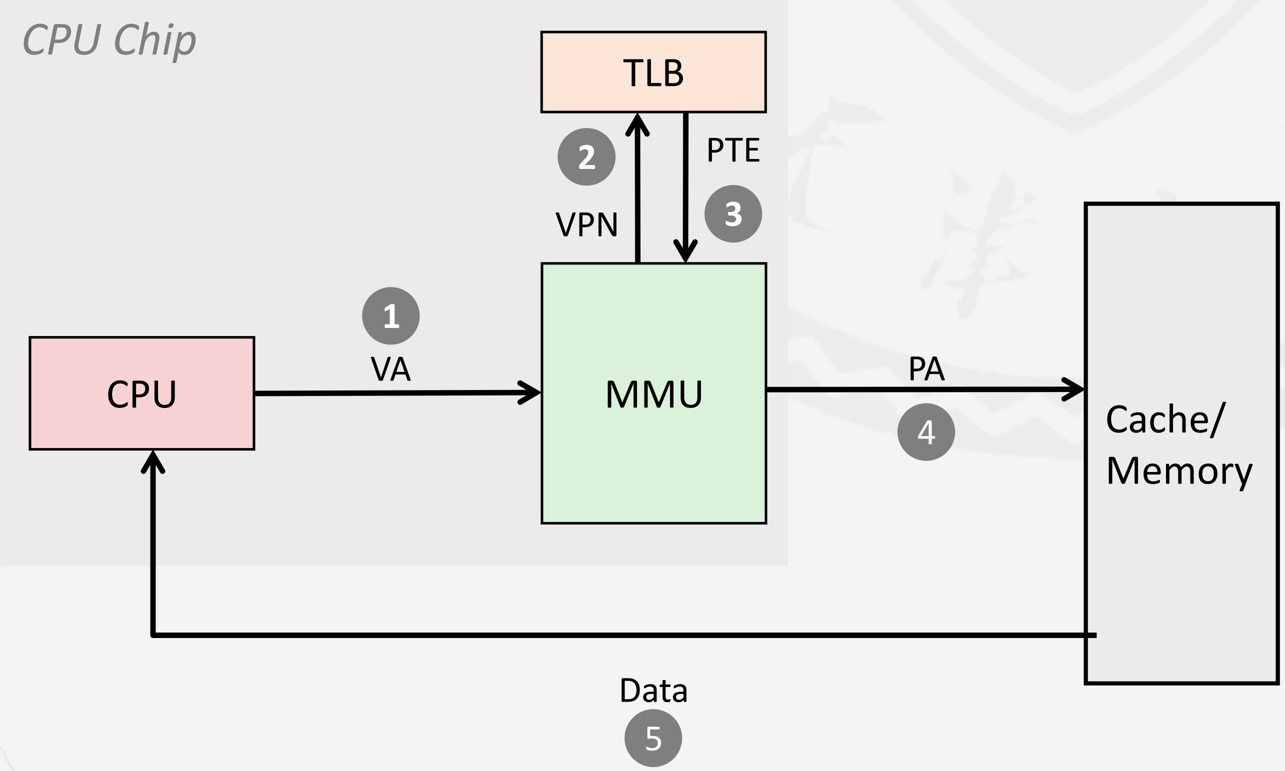
MMU uses the VPN portion of the virtual address to access the TLB:



TLB命中
TLB Hit

TLB命中会减少一次内存访问

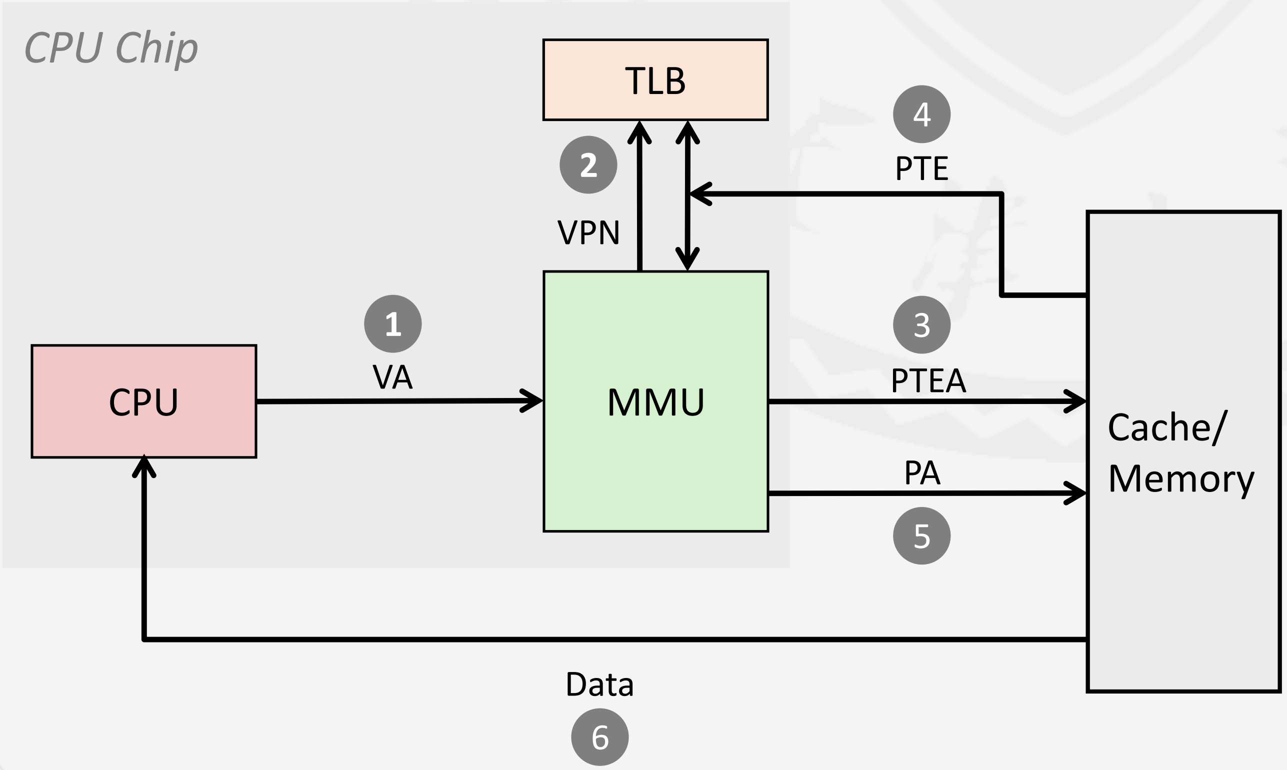
A TLB hit eliminates a memory access



TLB未命中
TLB Miss

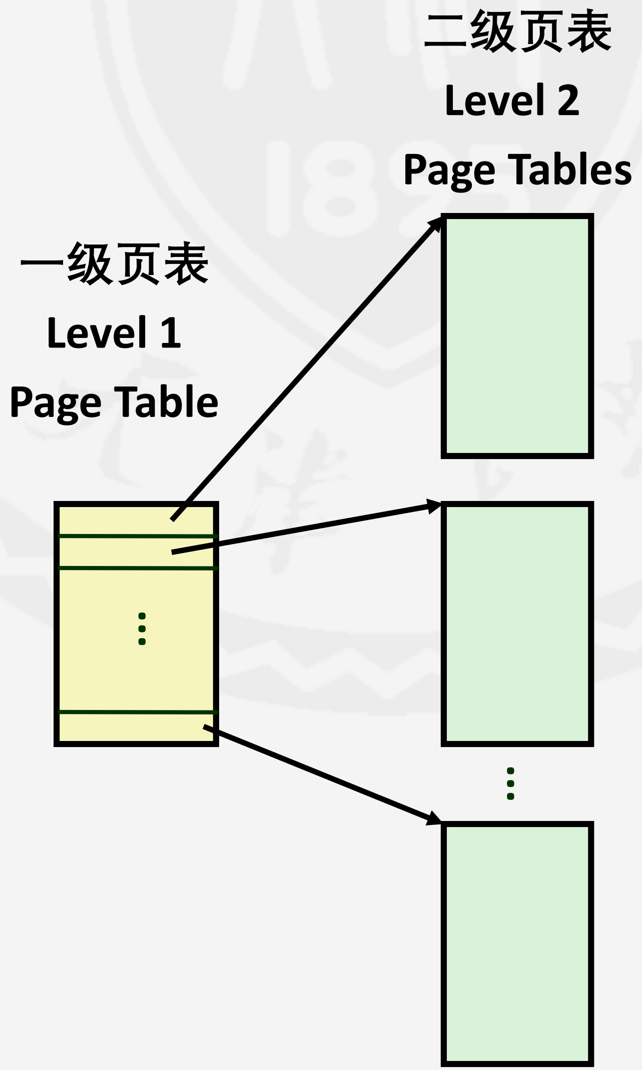
TLB未命中会导致一次额外的内存访问（页表项）
A TLB miss incurs an additional memory access (the PTE)

幸运的是，TLB未命中这种情况非常罕见，为什么？
Fortunately, TLB misses are rare. Why?



多级页表
Multi-Level Page Tables

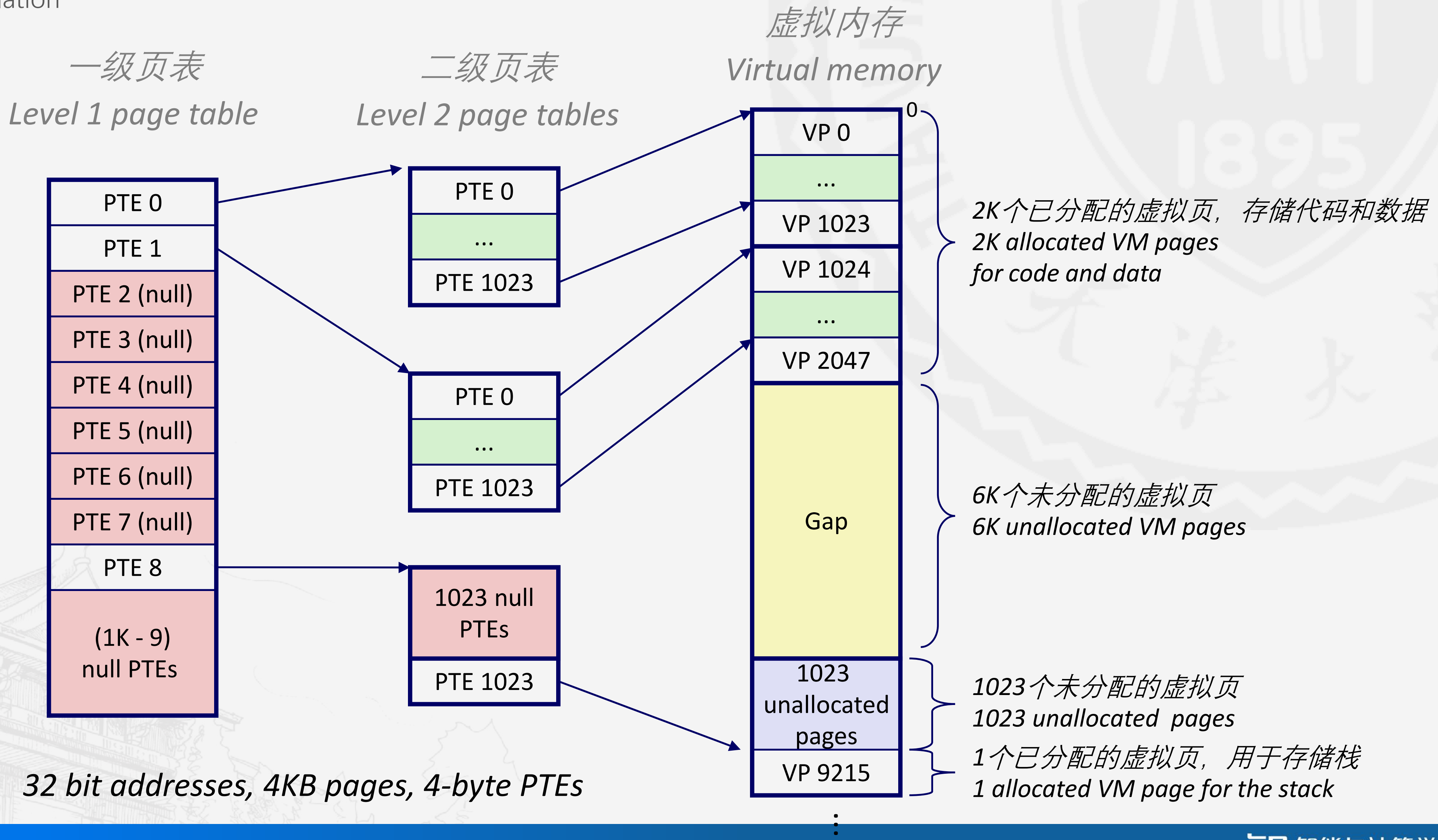
- 假设：
  Suppose:
  - 页大小：4KB；48位地址空间；页表项大小：8字节
    4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- 问题：每个页表将需要512GB！
  Problem: Would need a 512 GB page table!
  - 2^{48} / 2^{12} * 2^{3} = 2^{39} bytes
- 常见的解决方案：多级页表
  Common solution: Multi-level page table
- 举例：两级页表
  Example: 2-level page table
  - 一级页表：每个页表项指向一个页表（在内存中常驻）
    Level 1 table: each PTE points to a page table (always memory resident)
  - 二级页表：每个页表项指向一个页（向其他数据一样可以在内存中换入换出）
    Level 2 table: each PTE points to a page (paged in and out like any other data)



地址翻译

Address Translation

一个两级页表的结构
A Two-Level Page Table Hierarchy

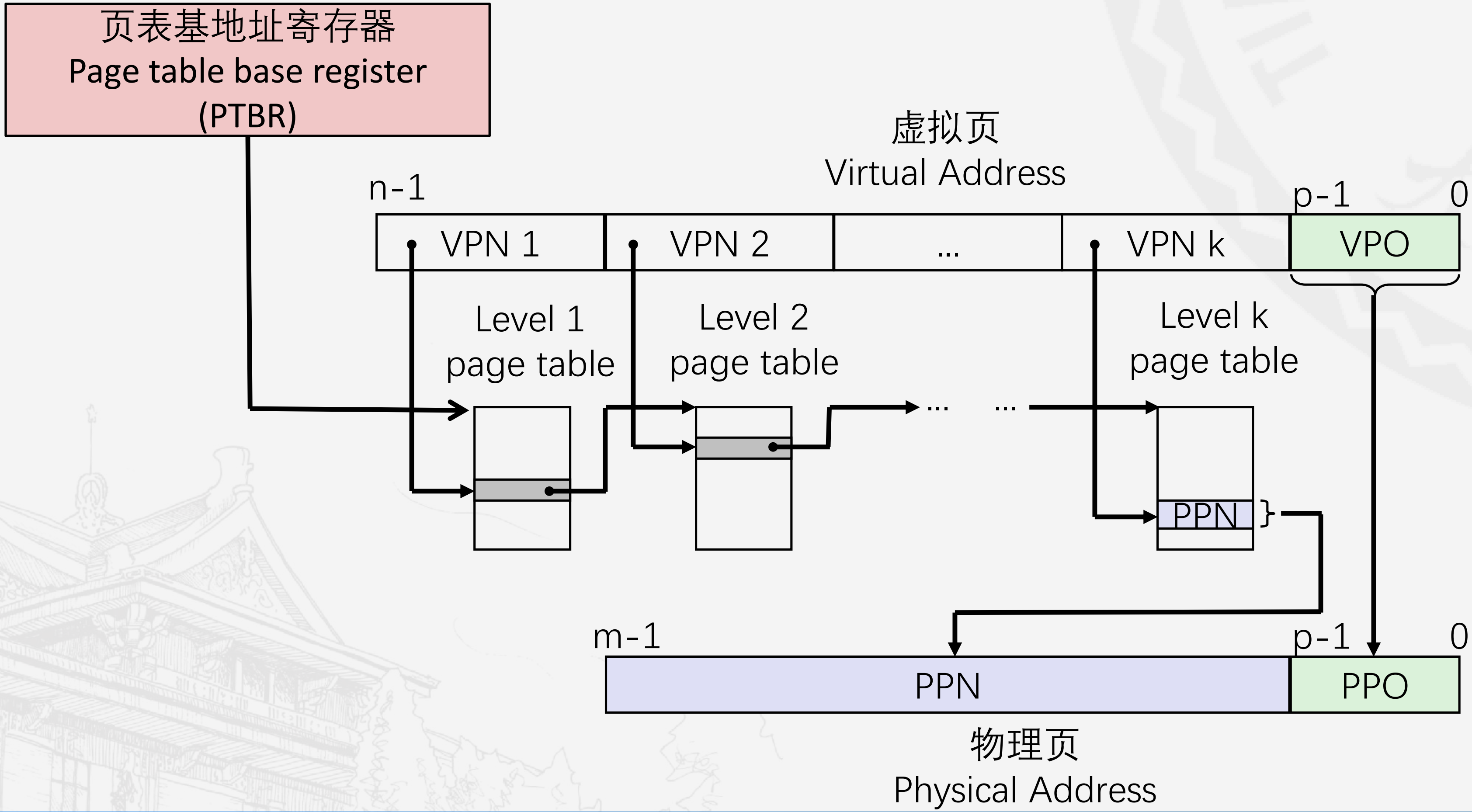


32 bit addresses, 4KB pages, 4-byte PTEs

地址翻译

Address Translation

使用K级页表进行地址翻译
Translating with a k-level Page Table



本章内容

Topic

- 地址翻译

  Address Translation

- 举例：简单的内存系统

  Simple Memory System example

- 案例研究：Core i7 / Linux 内存系统

  Case study: Core i7/Linux Memory System

- 内存映射

  Memory mapping

举例：简单的内存系统

Simple Memory System example

地址翻译中用到的符号
Address Translation Symbols

- 基本参数
  Basic Parameters
  - $N = 2^n$：虚拟地址空间的地址数量
    Number of addresses in virtual address space
  - $M = 2^m$：物理地址空间的地址数量
    Number of addresses in physical address space
  - $P = 2^p$：页大小（字节）
    Page size (bytes)

- 虚拟地址的组成
  Components of the virtual address (VA)
  - **VPO：** 虚拟页偏移量
    Virtual page offset
  - **VPN：** 虚拟页号
    Virtual page number
  - **TLBI：** TLB组索引
    TLB index
  - **TLBT：** TLB标记
    TLB tag

- 物理地址的组成
  Components of the physical address (PA)
  - **PPO：** 物理页偏移量（和VPO相同）
    Physical page offset (same as VPO)
  - **PPN：** 物理页号
    Physical page number
  - **CO：** 缓存块偏移量
    Byte offset within cache line
  - **CI：** 缓存组索引
    Cache index
  - **CT：** 缓存标识
    Cache tag

举例：简单的内存系统

Simple Memory System example

- 寻址
  Address
  - 14位虚拟地址
    14-bit virtual addresses
  - 12位物理地址
    12-bit physical address
  - 页大小：64字节
    Page size = 64 bytes

13	12	11	10	9	8	7	6	5	4	3	2	1	0

VPN 虚拟页号 Virtual Page Number (bits 13–6)

VPO 虚拟页偏移量 Virtual Page Offset (bits 5–0)

11	10	9	8	7	6	5	4	3	2	1	0

PPN 物理页号 Physical Page Number (bits 11–6)

PPO 物理页偏移量 Physical Page Offset (bits 5–0)

举例：简单的内存系统

Simple Memory System example

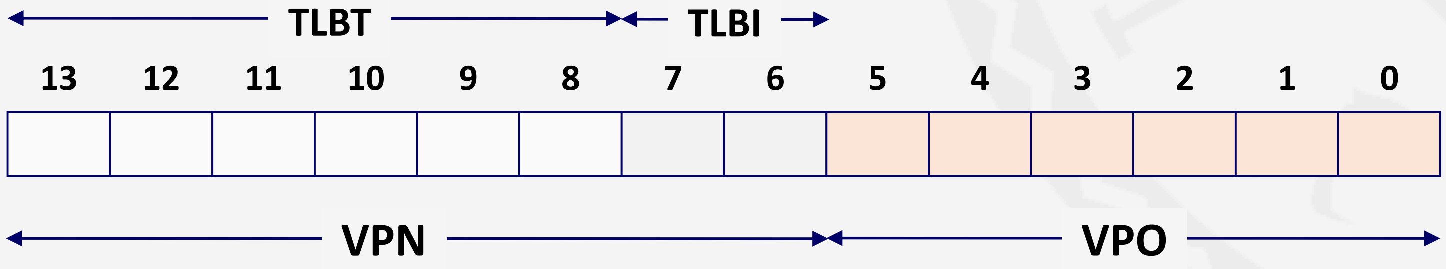
1. TLB

16个项

16 entries

四路组相联

4-way associative



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

举例：简单的内存系统

Simple Memory System example

2. Page Table

只给出了前16个项，共256项

Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

举例：简单的内存系统

Simple Memory System example

3. Cache

共16行（块），块大小：4 字节

16 lines, 4-byte block size

物理寻址

Physically addressed

直接映射

Direct mapped

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

举例：简单的内存系统

Simple Memory System example

地址翻译示例 #1
Address Translation Example #1

Virtual Address: **0x03D4**

TLBT (bits 13–8) | TLBI (bits 7–6)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN (bits 13–6) | VPO (bits 5–0)

VPN **0x0F** TLBI **0x3** TLBT **0x03** TLB Hit? **Y** Page Fault? **N** PPN: **0x0D**

Physical Address

CT (bits 11–6) | CI (bits 5–2) | CO (bits 1–0)

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN (bits 11–6) | PPO (bits 5–0)

CO **0** CI **0x5** CT **0x0D** Hit? **Y** Byte: **0x36**

举例：简单的内存系统

Simple Memory System example

地址翻译示例 #2
Address Translation Example #2

Virtual Address: **0x0020**

← TLBT →← TLBI →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

← VPN →← VPO →

VPN **0x00** TLBI 0 TLBT **0x00** TLB Hit? N Page Fault? N PPN: **0x28**

Physical Address

← CT →← CI →← CO →

11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

← PPN →← PPO →

CO **0** CI **0x8** CT **0x28** Hit? N Byte: Mem

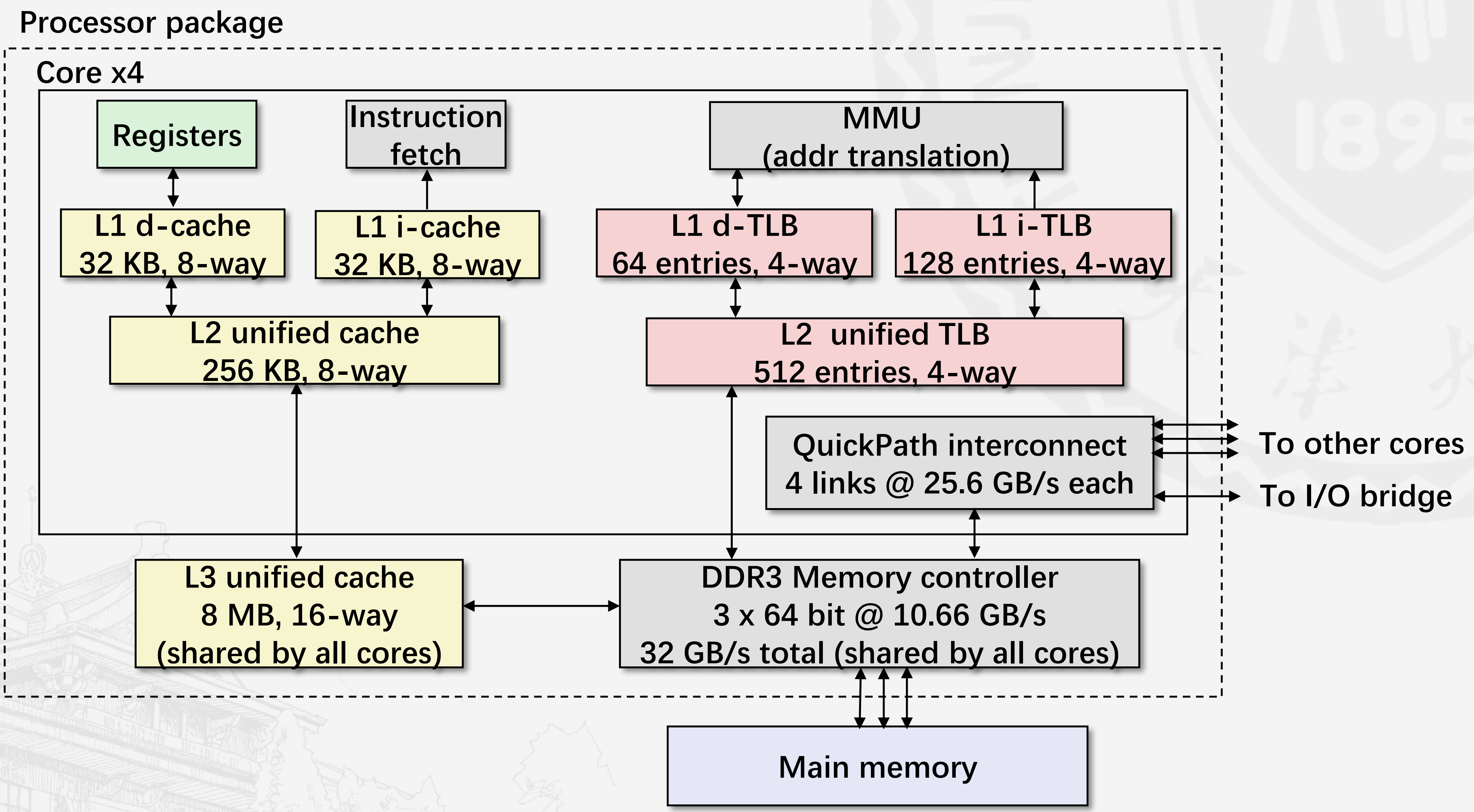
本章内容

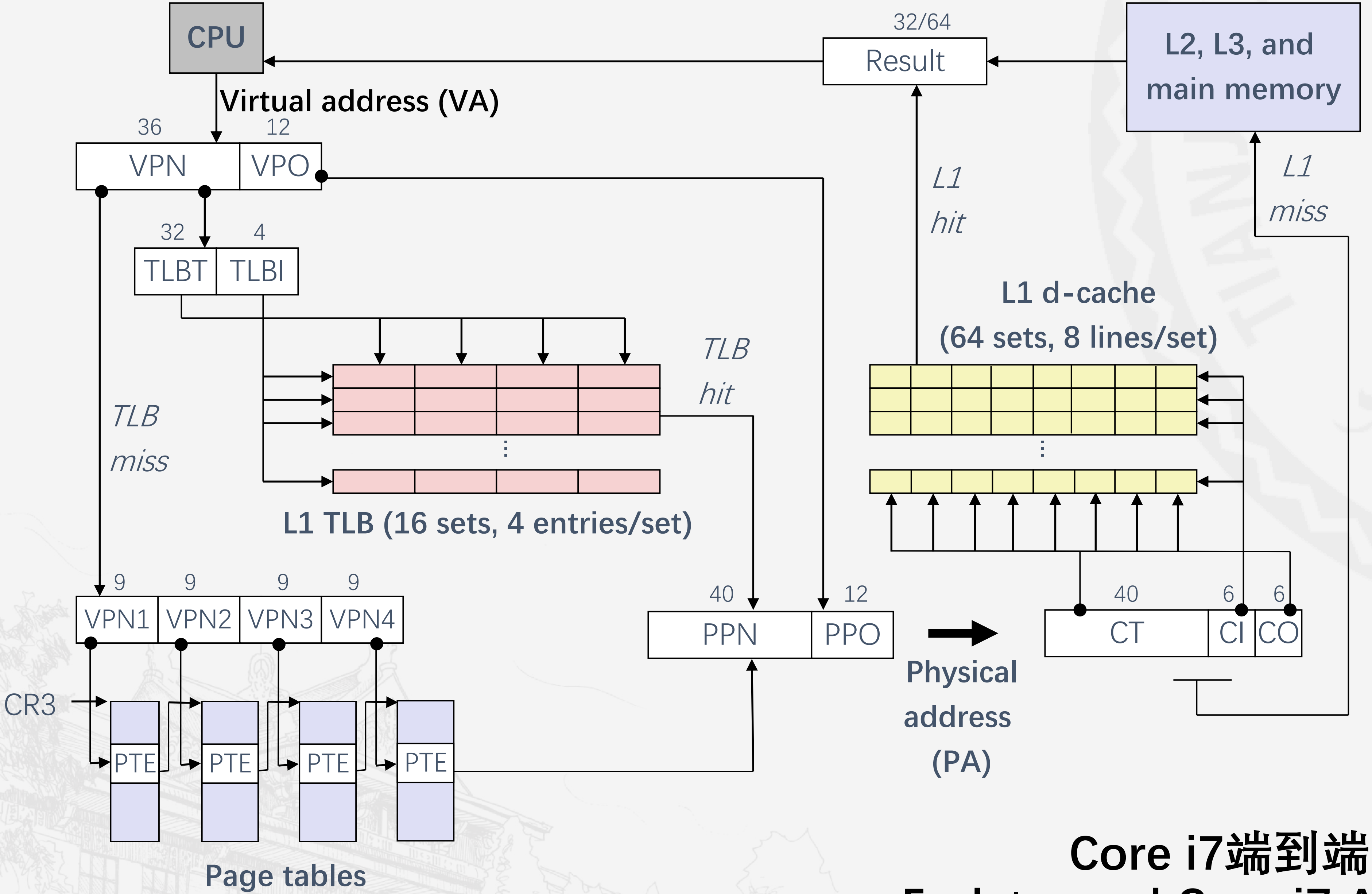
Topic

- 地址翻译
  Address Translation
- 举例：简单的内存系统
  Simple Memory System example
- 案例研究：Core i7 / Linux 内存系统
  Case study: Core i7/Linux Memory System
- 内存映射
  Memory mapping

案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System





Core i7端到端的地址翻译
End-to-end Core i7 Address Translation

Core i7 第1-3级页表项
Core i7 Level 1-3 Page Table Entries

63	62 … 52	51 … 12	11 … 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS (page table location on disk)	P=0

Each entry references a 4K child page table. Significant fields:

有效位

P: Child page table present in physical memory (1) or not (0).

读写权限

R/W: Read-only or read-write access access permission for all reachable pages.

超级用户（内核模式）

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

直写/回写

WT: Write-through or write-back cache policy for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

下一级页表的物理地址

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

可执行权限

XD: Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 第4级页表项
Core i7 Level 1-3 Page Table Entries

63	62 … 52	51 … 12	11 … 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page physical base address	Unused	G		D	A	CD	WT	U/S	R/W	P=1
Available for OS (page location on disk)												P=0

Each entry references a 4K child page. Significant fields:

有效位

P: Child page is present in memory (1) or not (0)

读写权限

R/W: Read-only or read-write access permission for child page

超级用户（内核模式）

U/S: User or supervisor mode access

直写/回写

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

页物理地址

Page physical base address: 40 most significant bits of physical page address (forces pages to be 4KB aligned)

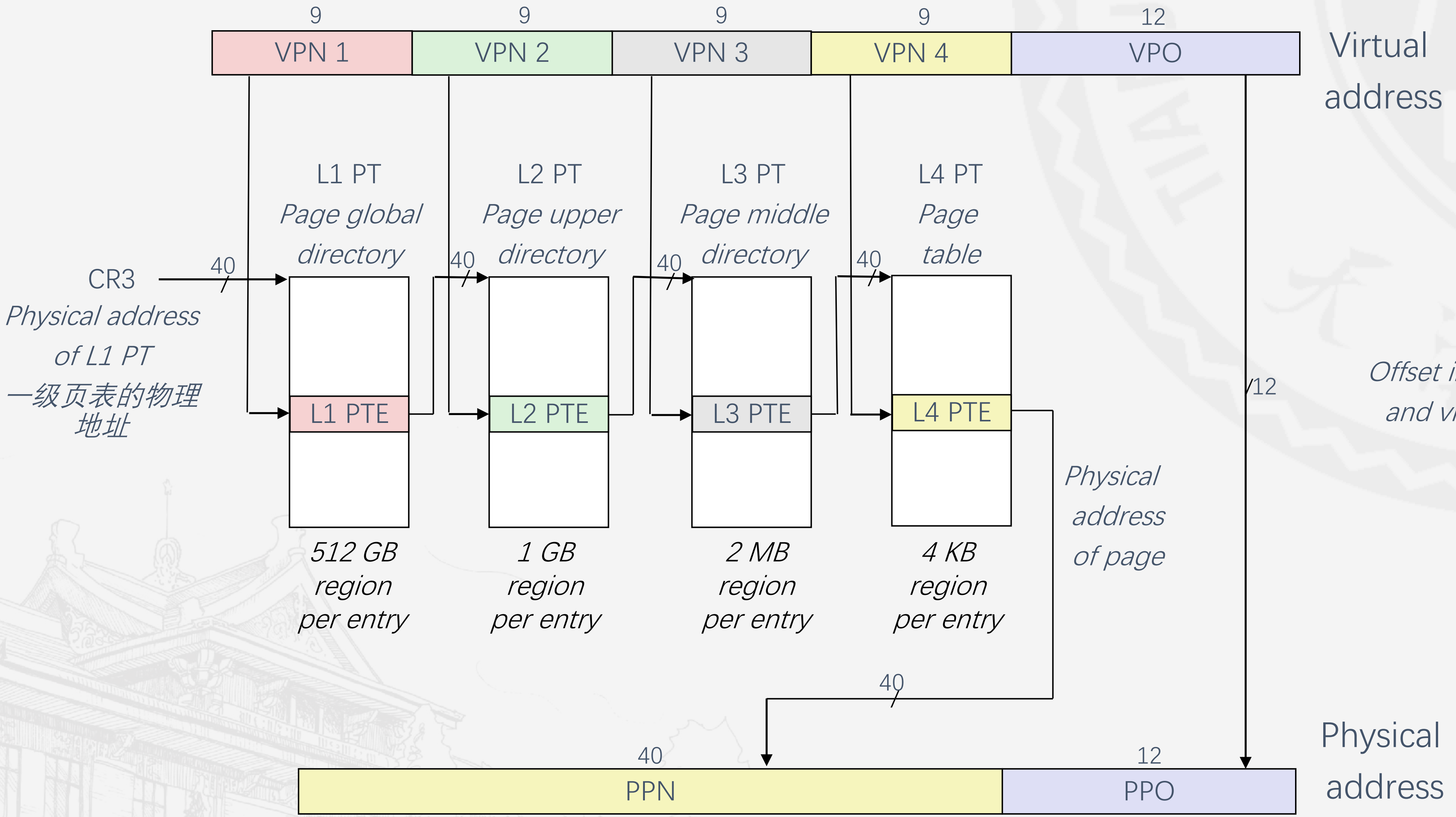
可执行权限

XD: Disable or enable instruction fetches from this page.

案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System

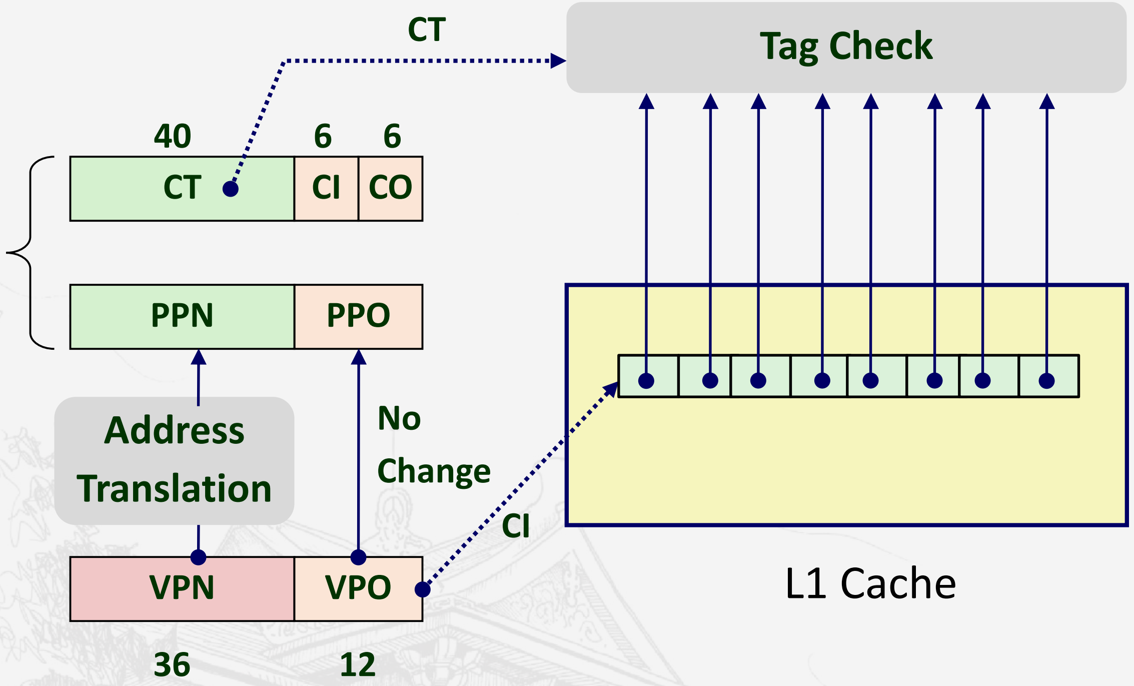
i7 页表翻译
Core i7 Table Translation



案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System

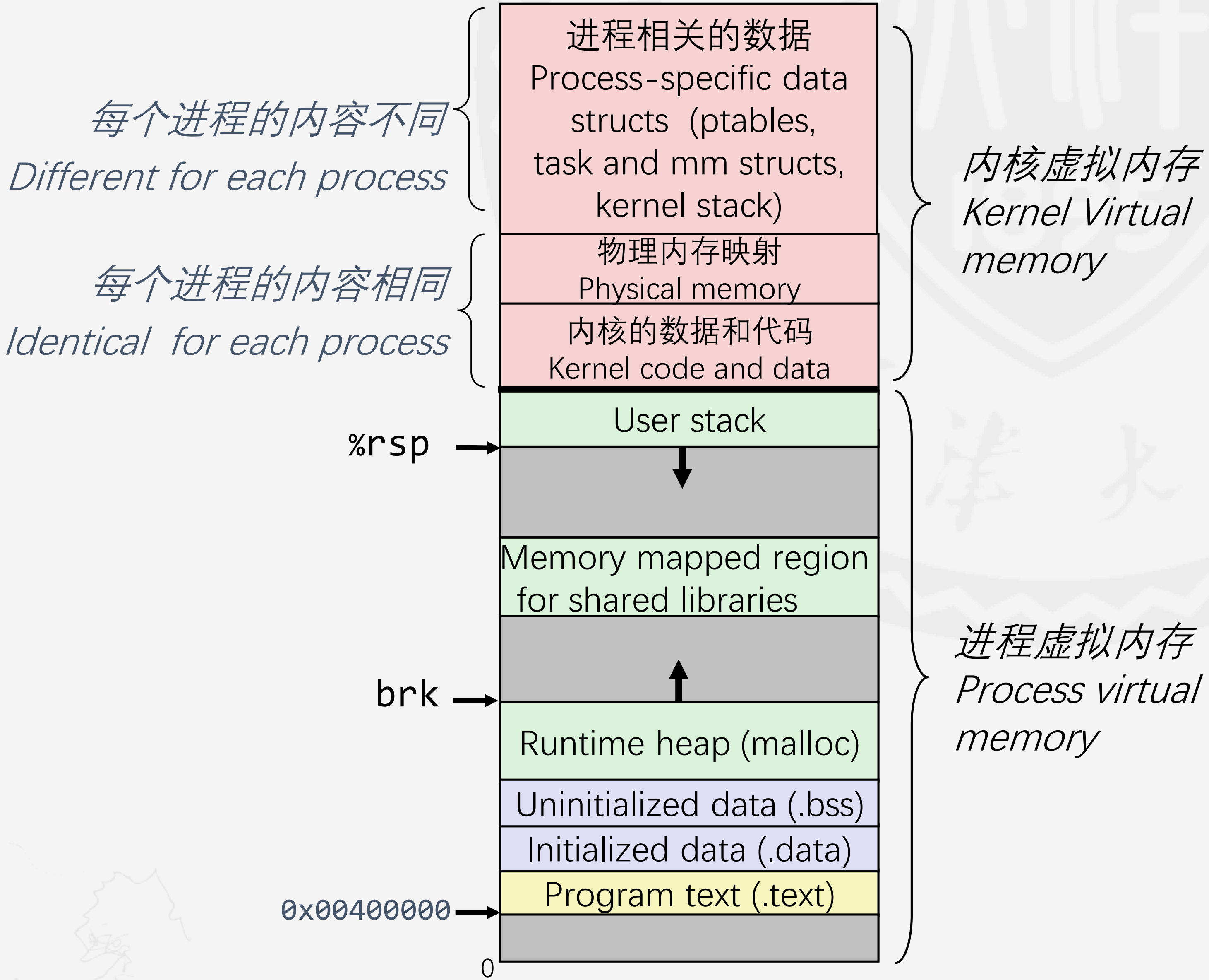
加速一级缓存访问速度的技巧
Cute Trick for Speeding Up L1 Access



- CI所在的位置，在虚拟地址和物理地址中都不会变化（CI完全被包含在偏移量中）
  Bits that determine CI identical in virtual and physical address
- 当进行地址翻译时， 可以同时对缓存进行索引
  Can index into cache while address translation taking place
- 通常情况下TLB都会命中，接下来PPN可用（包含了CT）
  Generally we hit in TLB, so PPN bits (CT bits) available next
- 使用虚拟地址进行索引，使用物理地址进行标记匹配（将部分对缓存的操作，与地址翻译并行）
  “Virtually indexed, physically tagged”
- 通过将缓存设计成适当的大小，可以实现这个目标
  Cache carefully sized to make this possible

案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System

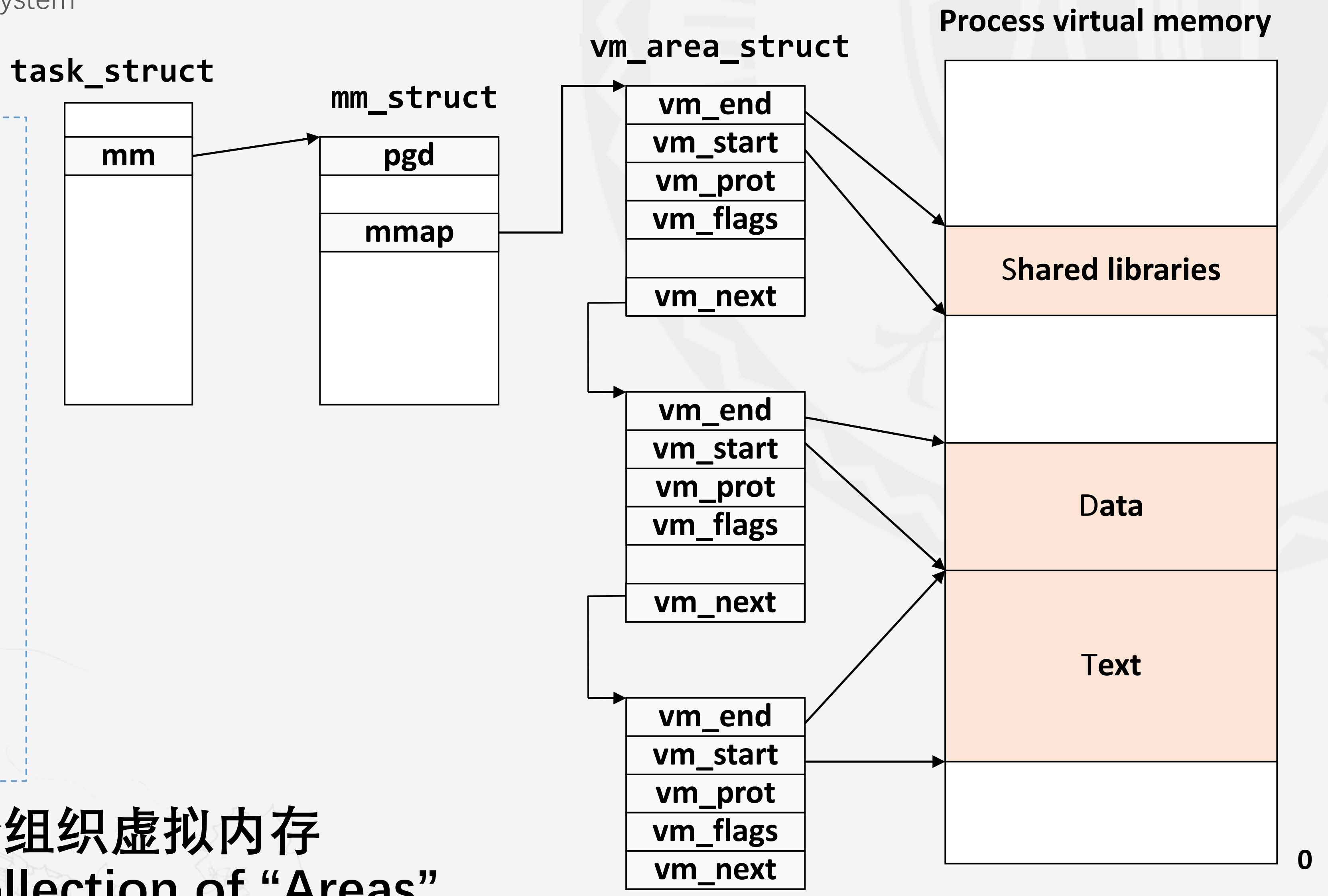


Linux进程的虚拟地址空间
Virtual Address Space of a Linux Process

案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System

- pgd: 指向第一级页表的基地址
  Points to L1 page table
- mmap：指向一个vm_area_struct结构的链表
  Points to a linked list of "area" structure
- vm_prot：这片区域的读写权限
  Read/write permissions for this area
- vm_flags：共享页/私有页标识
  Pages shared with other processes or private to this process

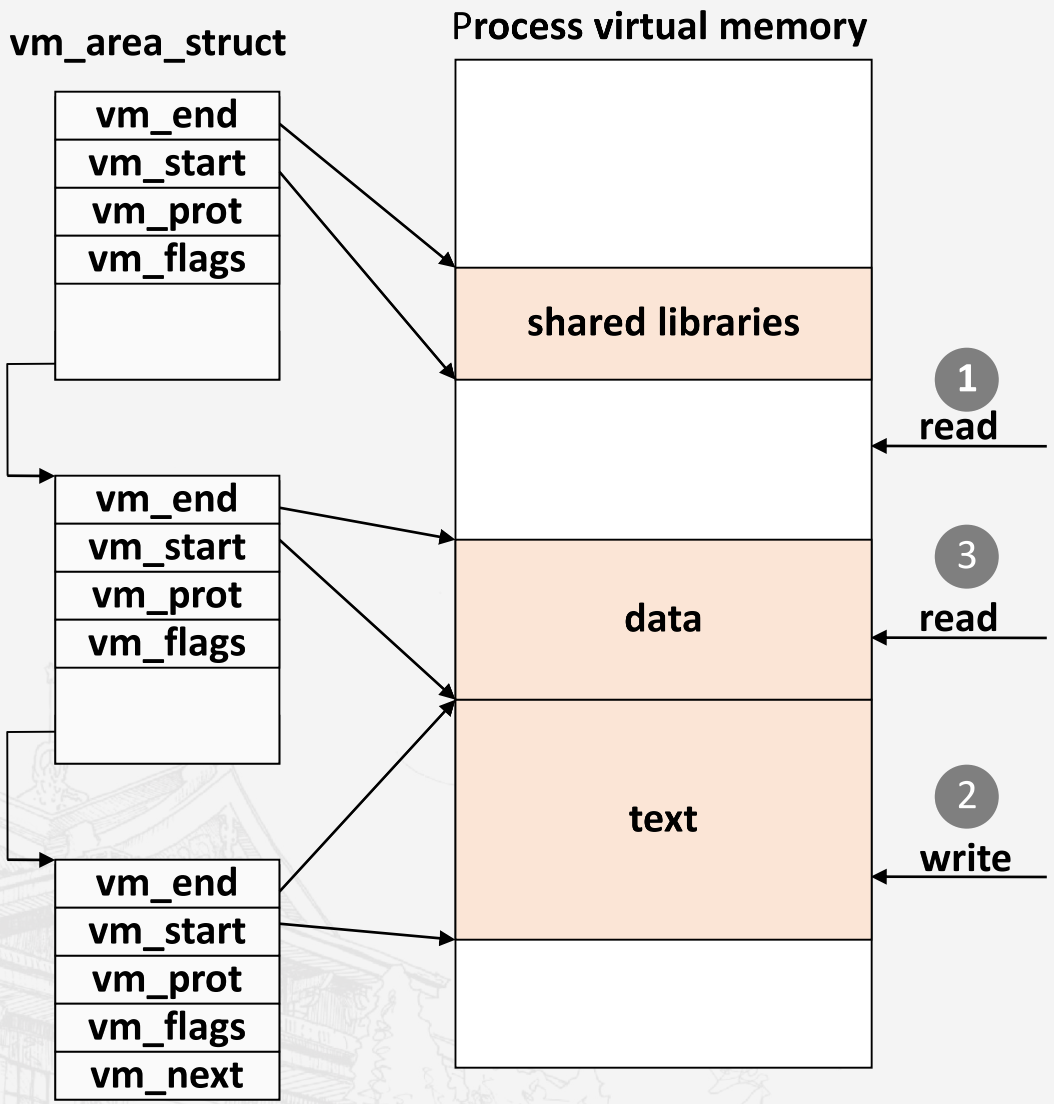


Linux使用"区域" 集合组织虚拟内存

Linux Organizes VM as Collection of "Areas"

案例研究：Core i7/Linux 内存系统

Case study: Core i7/Linux Memory System



Linux中的缺页故障处理
Linux Page Fault Handling

段错误

Segmentation fault:

访问一个不存在的页

accessing a non-existing page

正常的缺页故障

Normal page fault

保护故障异常

Protection exception:

例如：违反权限，向一个只读的页中写入数据（Linux也会报告为段错误）

e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

本章内容

Topic

- 地址翻译
  Address Translation
- 举例：简单的内存系统
  Simple Memory System example
- 案例研究：Core i7 / Linux 内存系统
  Case study: Core i7/Linux Memory System
- 内存映射
  Memory mapping

内存映射

Memorey Mapping

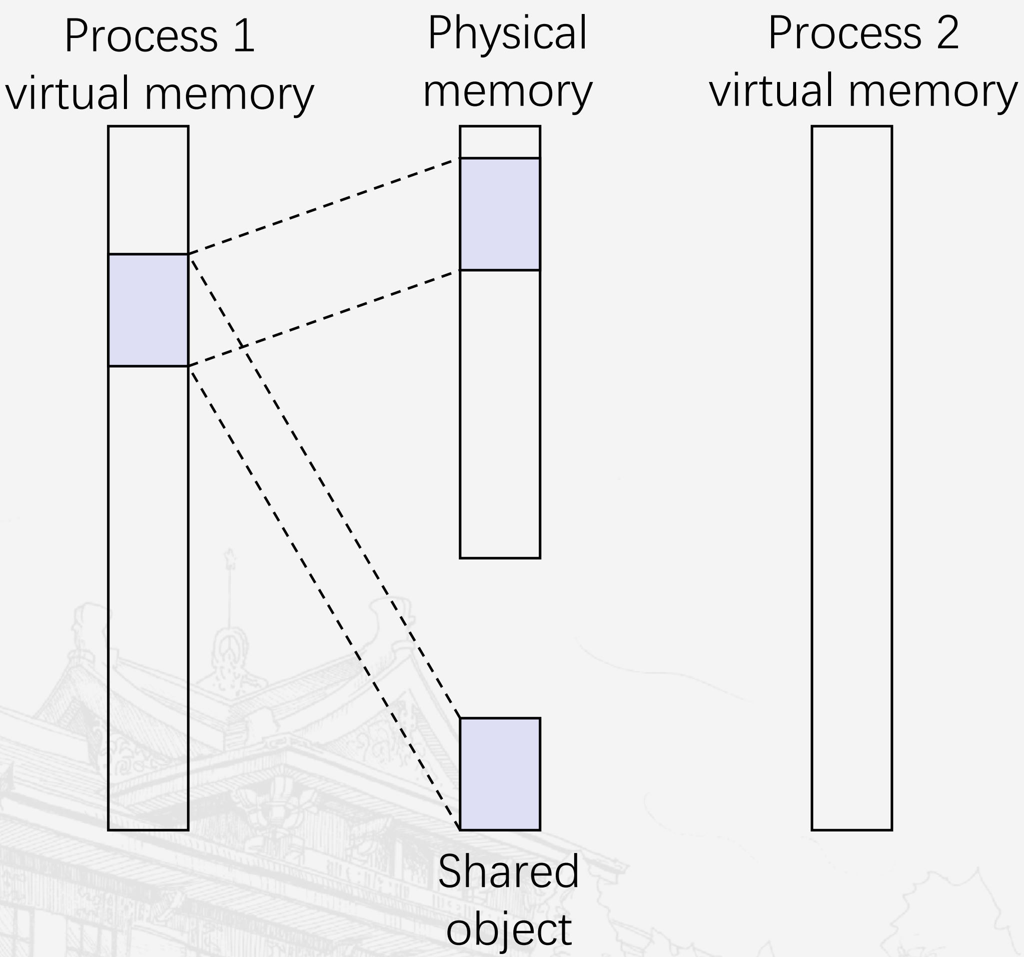
- 虚拟内存区域通过与磁盘对象关联进行初始化
  - 称为内存映射

- 虚拟内存区域可以由下列两种方式进行映射（从中获取其初始值）：
  - 磁盘上的常规文件（例如，可执行对象文件）
    - 初始页面字节来自文件的一个部分
  - 匿名文件（例如，空文件）
    - 首次缺页将分配一个由0填充的物理页面（需求零页面）
    - 一旦页面被写入（脏页），它就像任何其他页面一样。
- 脏页在内存和交换空间之间来回复制
  - 交换空间由内核维护

内存映射

Memorey Mapping

再看共享对象

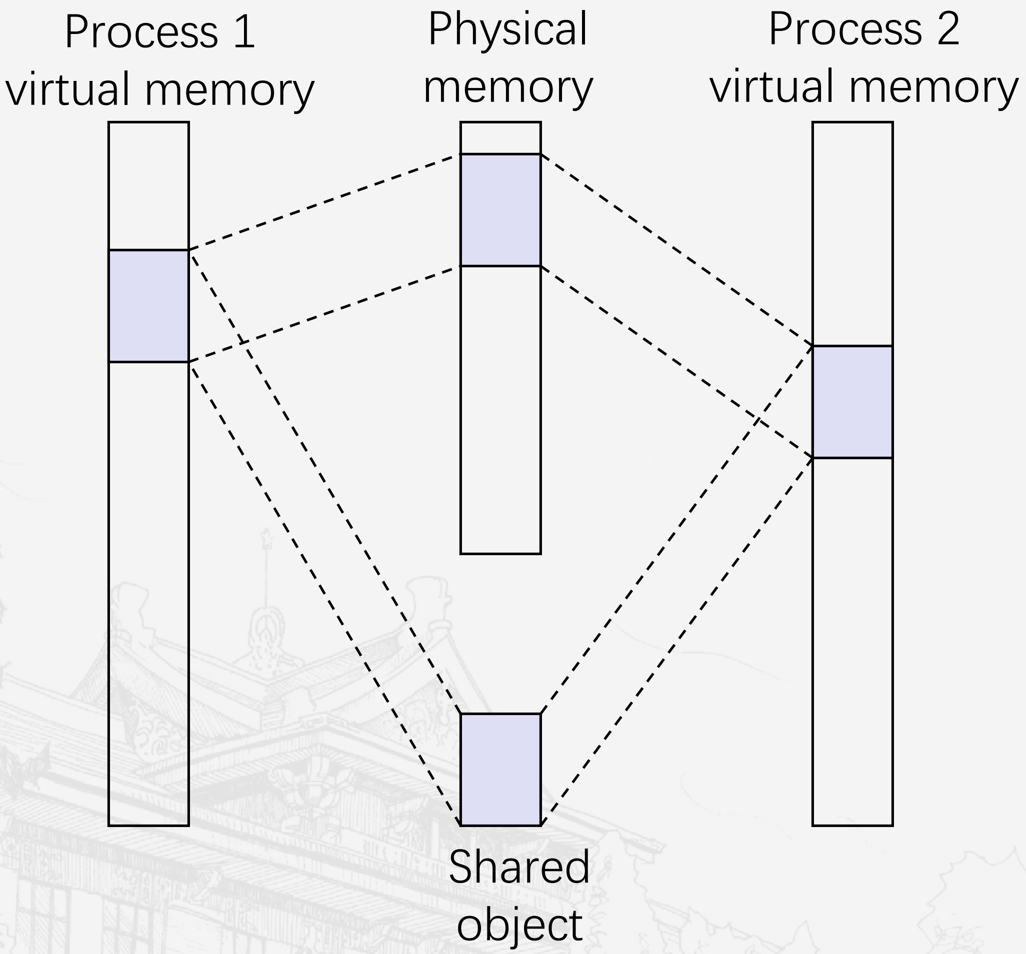


- 进程1映射了共享对象

内存映射

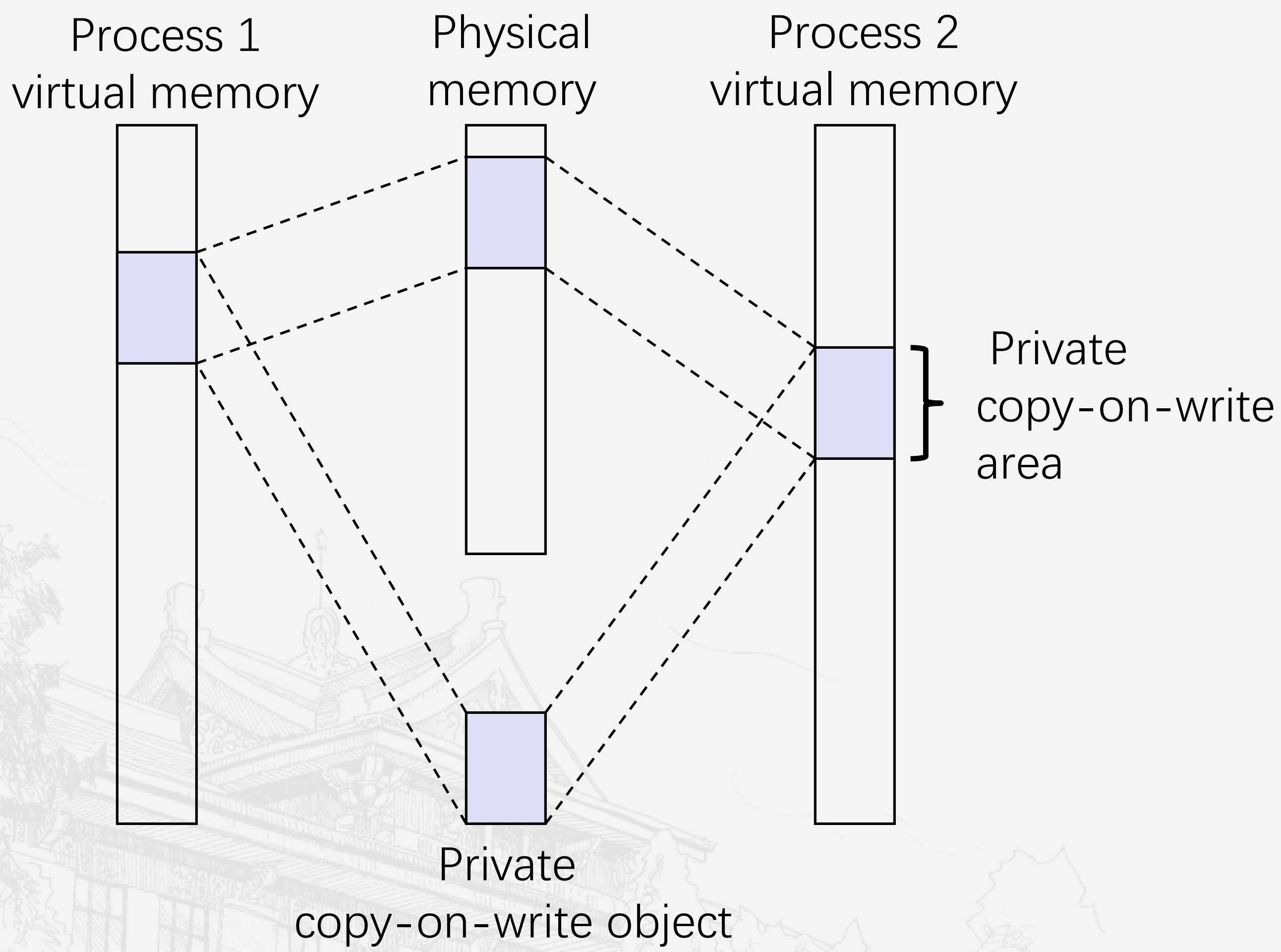
Memorey Mapping

再看共享对象



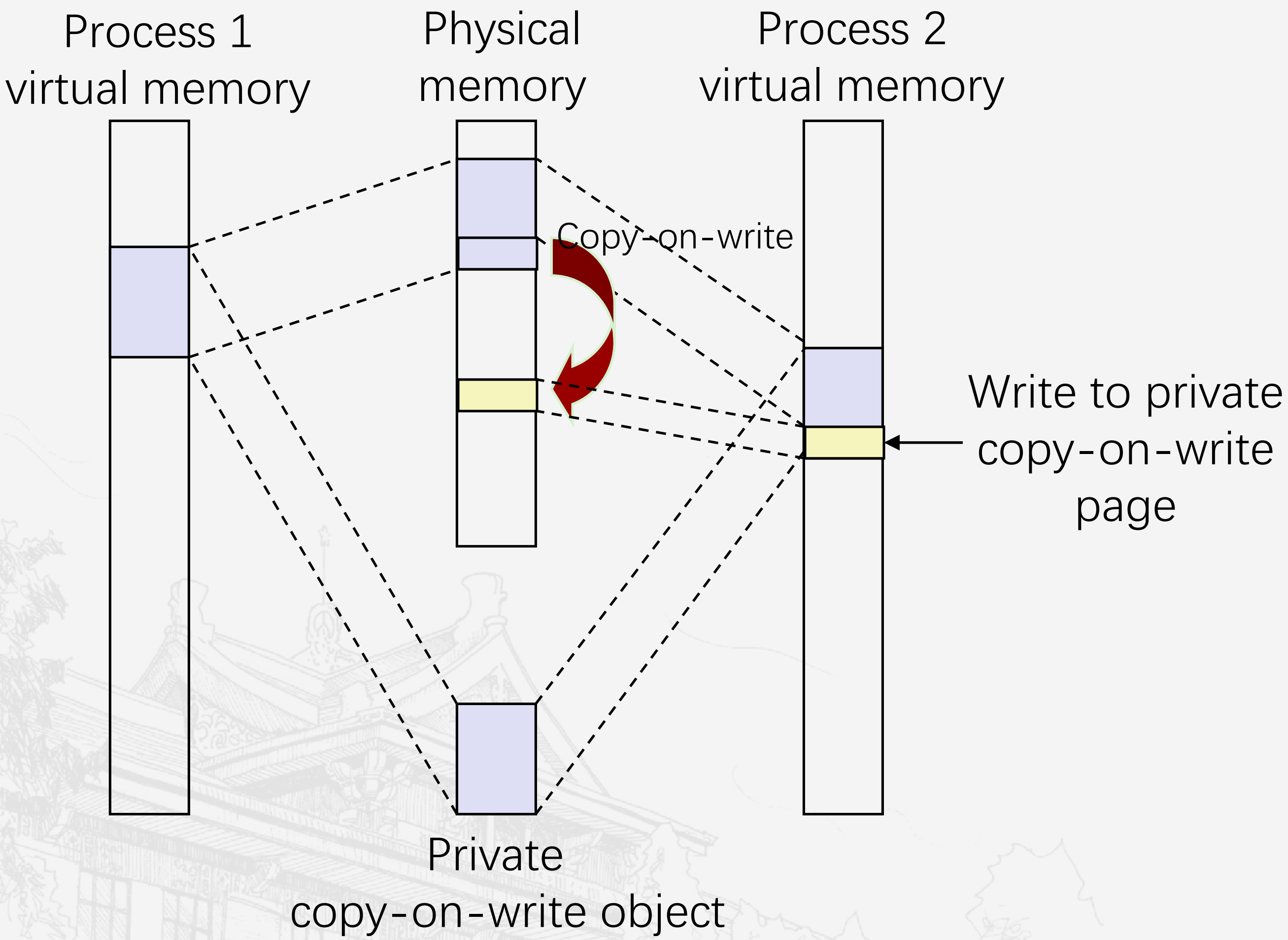
- 进程2映射了共享对象。
- 请注意：虚拟地址可以是不同的。

私有对象：写时复制
Copy-on-write (COW)



- 两个进程映射一个私有写时复制（Copy-on-write，COW）对象。
- 区域标记为私有写时复制。
- 私有区域的页表项（PTEs）被标记为只读。

私有对象：写时复制
Copy-on-write (COW)



- 指令写入私有页面引发保护故障异常
- 故障处理程序创建新的读写页面
- 指令在处理程序返回时重新启动
- 以尽可能推迟复制的时机

内存映射

Memorey Mapping

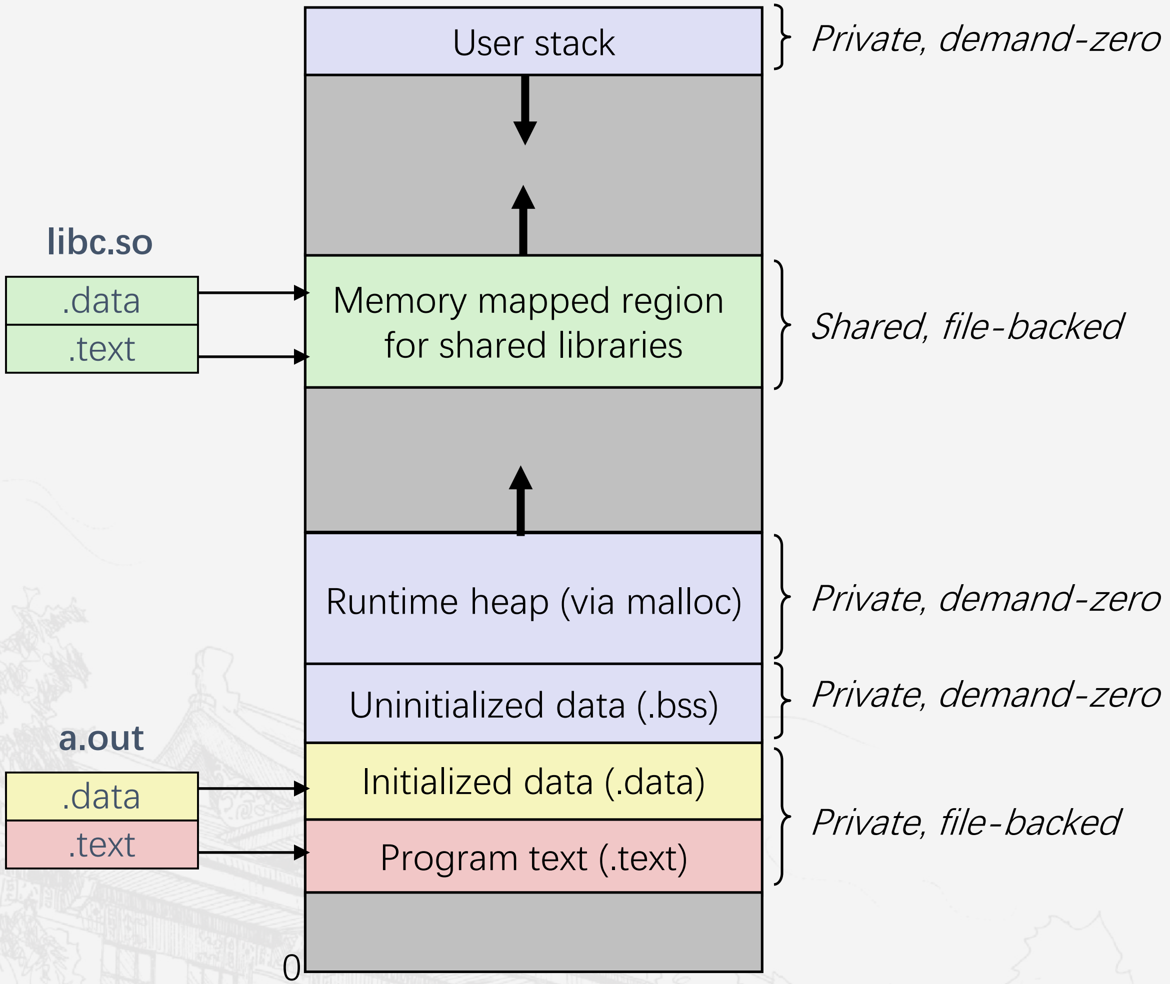
- 虚拟内存和内存映射解释了fork如何为每个进程提供私有地址空间。

- 为新进程创建虚拟地址的步骤：
  - 创建当前mm_struct、vm_area_struct和页表的精确副本。
  - 将两个进程中的每个页面标记为只读。
  - 将两个进程中的每个vm_area_struct标记为私有写时复制（COW）。

- fork返回时，每个进程都具有虚拟内存的精确副本。
- 随后的写入使用写时复制（COW）机制创建新页面。

内存映射

Memorey Mapping



再看execve系统调用

- 使用execve加载和运行当前进程中的新程序（a.out）
  - 释放旧区域的vm_area_struct和页表。
  - 为新区域创建vm_area_struct和页表。
    - 程序和初始化数据由目标文件支持。
    - .bss和堆栈由匿名文件支持。
  - 将PC设置为.text中的入口点。
    - 后续根据需要引发代码和数据页面缺页故障

用户级内存映射

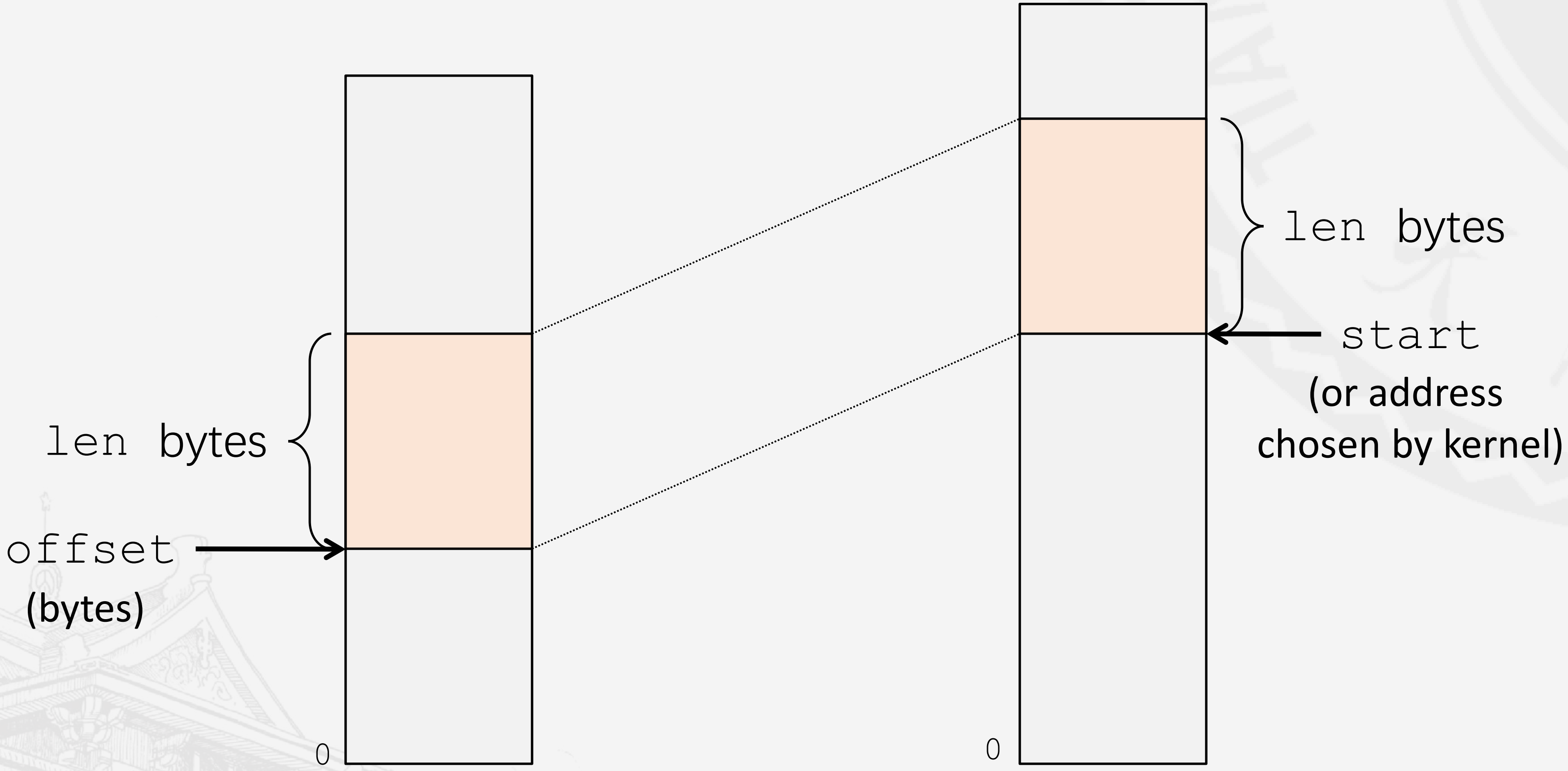
- `void *mmap(void *start, int len, int prot, int flags, int fd, int offset)`
- 映射文件描述符fd指定的文件中，从偏移量offset开始的len字节，建议映射到地址start
  - start：可以是0，表示映射的地址由mmap决定。
  - prot：页面访问权限 PROT_READ，PROT_WRITE等
  - flags：MAP_ANONYMOUS，MAP_PRIVATE，MAP_SHARED等。
- 返回指向映射区域开头的指针（可能不是start）

内存映射

Memorey Mapping

用户级内存映射

```
void *mmap(void *start, int len, int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor `fd`

Process virtual memory

使用mmap复制文件

将文件复制到标准输出stdout，不需要任何额外的用户空间

```c
#include "csapp.h"

void mmapcopy(int fd, int size)
{

    /* Ptr to memory mapped area */
    char *bufp;

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

mmapcopy.c

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

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

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

mmapcopy.c