Virtual Memory: Systems
虚拟内存:系统

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. Bryant and David R. O’Hallaron

内容提纲/Today

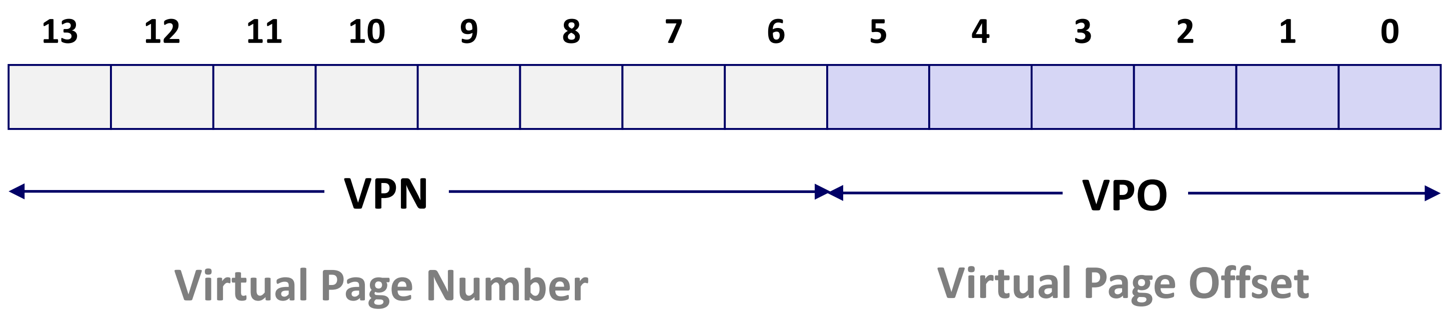
- **简单内存系统示例/Simple memory system example**
- 实例：Core i7/Linux内存系统/Case study: Core i7/Linux memory system
- 内存映射/Memory mapping

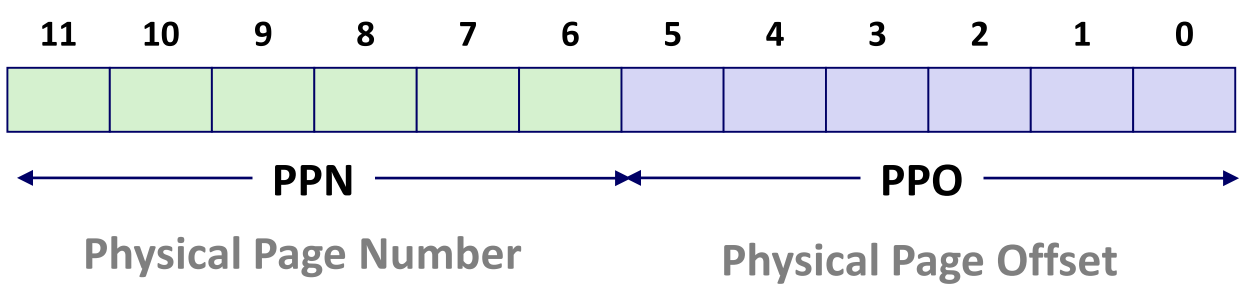
符号回顾/Review of Symbols

- 基本参数/Basic Parameters
  - **$N = 2^n$** : Number of addresses in virtual address space/虚拟内存空间的地址数量
  - **$M = 2^m$** : Number of addresses in physical address space/物理内存空间的地址数量
  - **$P = 2^p$** : Page size (bytes)/页大小（字节）
- 虚拟页划分/Components of the virtual address (VA)
  - **TLBI**: TLB index/TLB索引
  - **TLBT**: TLB tag/TLB标记
  - **VPO**: Virtual page offset/虚拟页偏移量
  - **VPN**: Virtual page number /虚拟页编号
- 物理页划分/Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)/物理页偏移量
  - **PPN:** Physical page number/物理页号
  - **CO**: Byte offset within cache line/Cache行中的偏移量
  - **CI:** Cache index/Cache索引
  - **CT**: Cache tag/Cache标记

简单的内存系统示例/Simple Memory System Example

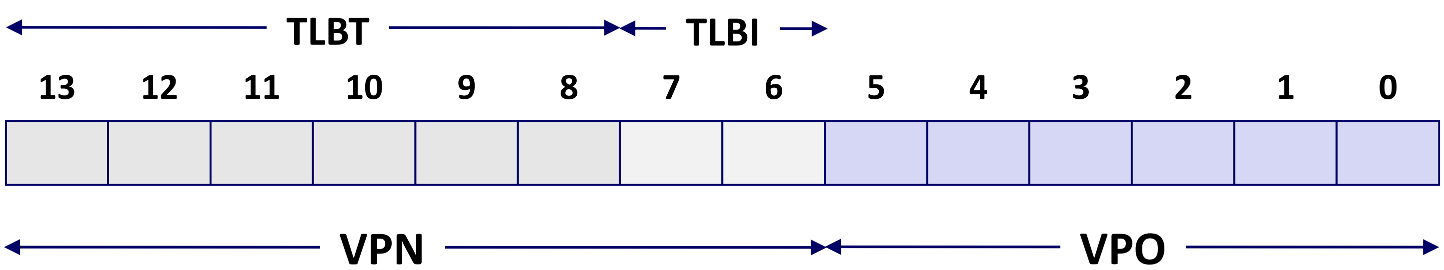
- 寻址/Addressing
  - 14位虚拟地址/14-bit virtual addresses
  - 12位物理地址/12-bit physical address
  - 页大小为64字节/Page size = 64 bytes





1. 简单内存系统TLB/Simple Memory System TLB

- 16条记录/16 entries
- 4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

2. 简单内存系统页表/Simple Memory System Page Table

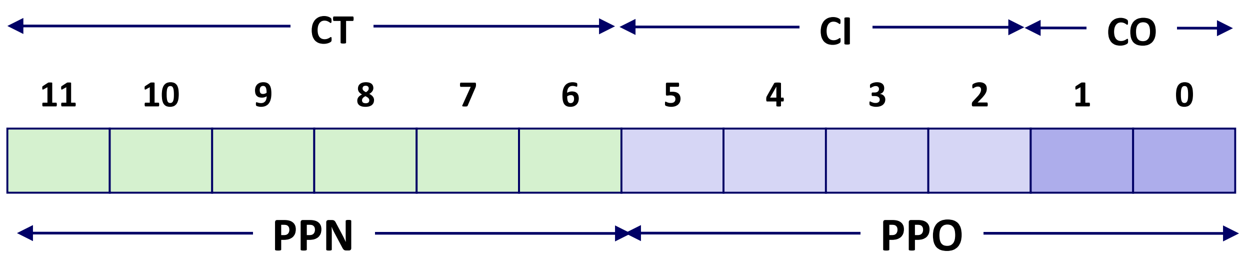
只显示了256的前16个/Only show first 16 entries (out of 256)

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

3. 简单内存系统Cache/Simple Memory System Cache

- 16行，4字节Cache块大小/16 lines, 4-byte block size
- 物理寻址/Physically addressed
- 直接映射/Direct mapped



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	–	–	–	–

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

虚拟地址/Virtual Address: 0x03D4

TLBT | TLBI

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 | VPO

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

物理地址/Physical Address

CT | CI | CO

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 | PPO

CO 0　CI 0x5　CT 0x0D　Hit? Y　Byte: 0x36

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

虚拟地址/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**

物理地址/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**

地址翻译示例#3/Address Translation Example #3

虚拟地址/Virtual Address: 0x0020

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	0	0	0	0	1	0	0	0	0	0

VPN: bits 13–6; VPO: bits 5–0

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

物理地址/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
1	0	1	0	0	0	1	0	0	0	0	0

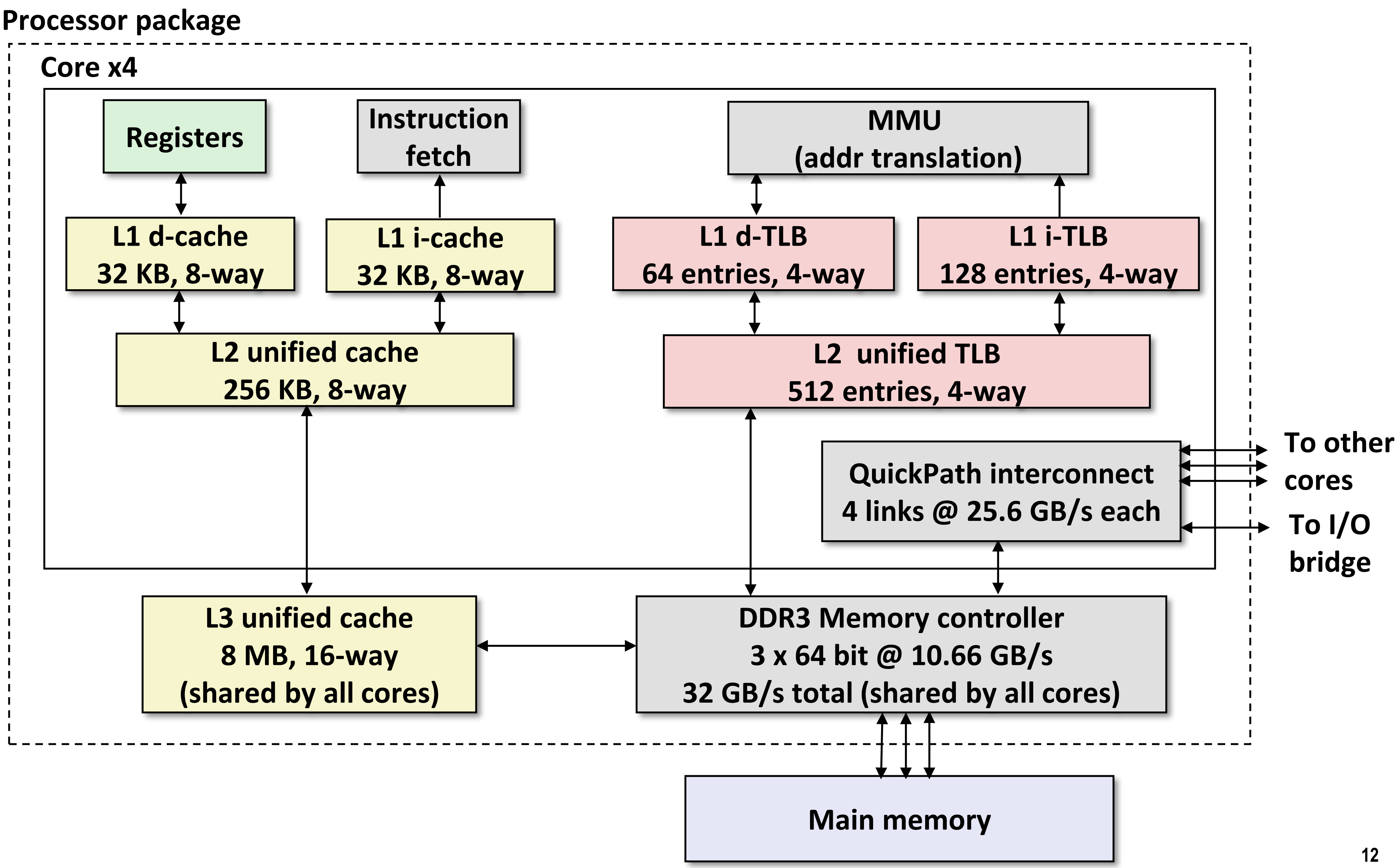
PPN: bits 11–6; PPO: bits 5–0

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

内容提纲/Today

- 简单内存系统示例/Simple memory system example
- 实例：Core i7/Linux内存系统/Case study: Core i7/Linux memory system
- 内存映射/Memory mapping

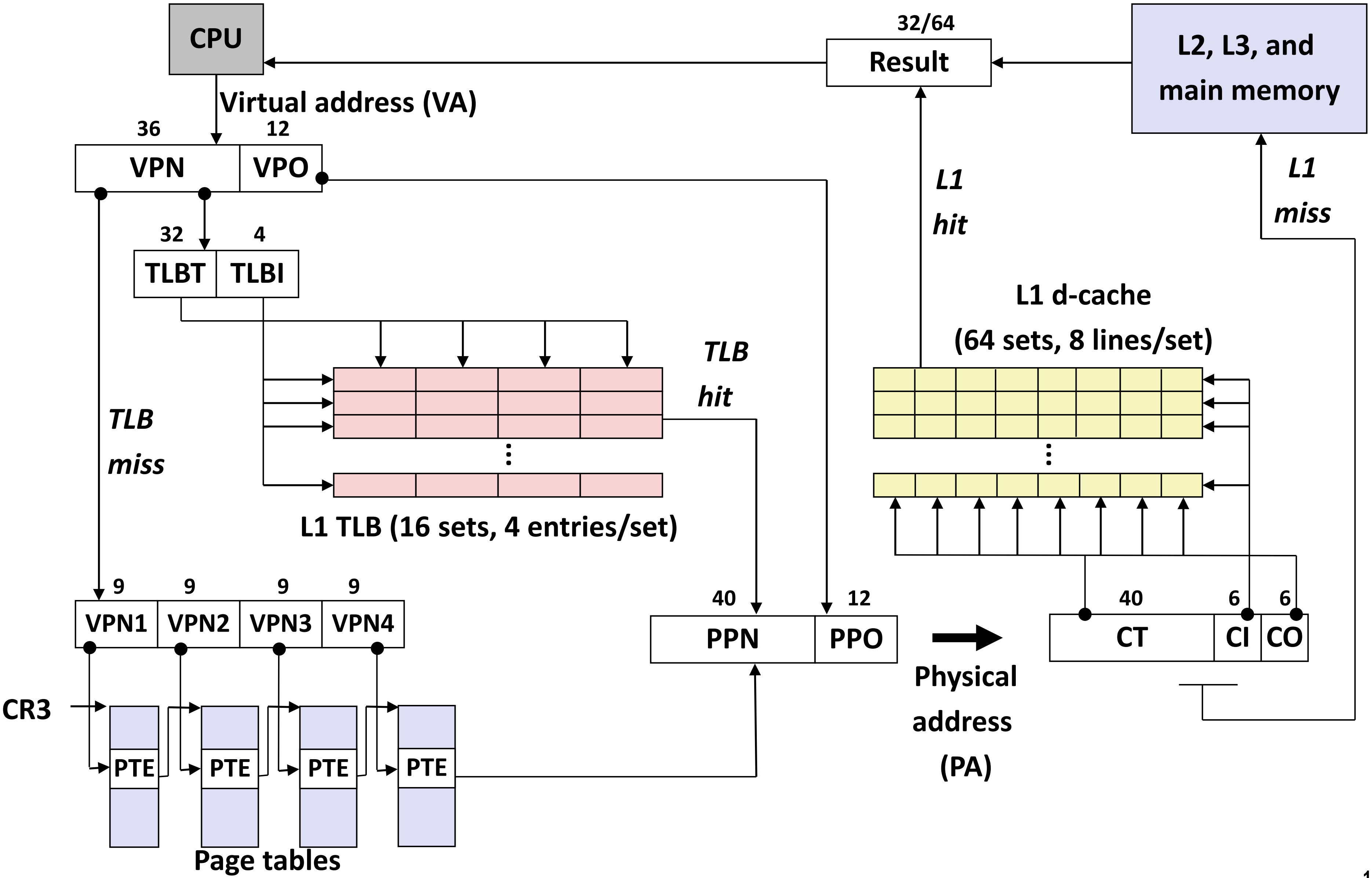
Intel Core i7存储系统/Intel Core i7 Memory System



符号回顾/Review of Symbols

- 基本参数/Basic Parameters
  - **$N = 2^n$** : Number of addresses in virtual address space/虚拟内存空间的地址数量
  - **$M = 2^m$** : Number of addresses in physical address space/物理内存空间的地址数量
  - **$P = 2^p$** : Page size (bytes)/页大小（字节）
- 虚拟页划分/Components of the virtual address (VA)
  - **TLBI**: TLB index/TLB索引
  - **TLBT**: TLB tag/TLB标记
  - **VPO**: Virtual page offset/虚拟页偏移量
  - **VPN**: Virtual page number /虚拟页编号
- 物理页划分/Components of the physical address (PA)
  - **PPO**: Physical page offset (same as VPO)/物理页偏移量
  - **PPN:** Physical page number/物理页号
  - **CO**: Byte offset within cache line/Cache行中的偏移量
  - **CI:** Cache index/Cache索引
  - **CT**: Cache tag/Cache标记

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



Core i7 1-3级页表记录/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	页表物理及地址/ Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

操作系统可用（页表位于磁盘上）Available for OS (page table location on disk)	P=0

每个条目对应一个4k子页表，主要的域包括/Each entry references a 4K child page table. Significant fields:

P:子页表是否在物理内存/Child page table present in physical memory (1) or not (0).

R/W: 只读或者读写权限标记位/Read-only or read-write access permission for all reachable pages.

U/S: 用户或特权（内核）模式标记位/user or supervisor (kernel) mode access permission for all reachable pages.

WT: 子页表的写透或者写回Cache策略/Write-through or write-back cache policy for the child page table.

A: 引用标记/Reference bit (由MMU读写时设置，软件清除/set by MMU on reads and writes, cleared by software).

PS: 页面大小，4KB或者4MB/Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

Page table physical base address: 物理页表地址的高40位(强制页表按照4KB对齐)/40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

XD: 禁止或允许取指操作/Disable or enable instruction fetches from all pages reachable from this PTE.

Core i7 4级页表条目/Core i7 Level 4 Page Table Entries

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

操作系统可见（内存页位于磁盘）Available for OS (page location on disk)	P=0

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

P:子页表是否在物理内存/Child page table present in physical memory (1) or not (0).

R/W: 只读或者读写权限标记位**/**Read-only or read-write access permission for all reachable pages.

U/S: 用户或特权（内核）模式标记位**/**user or supervisor (kernel) mode access permission for all reachable pages.

WT: 子页表的写透或者写回Cache策略**/**Write-through or write-back cache policy for the child page table.

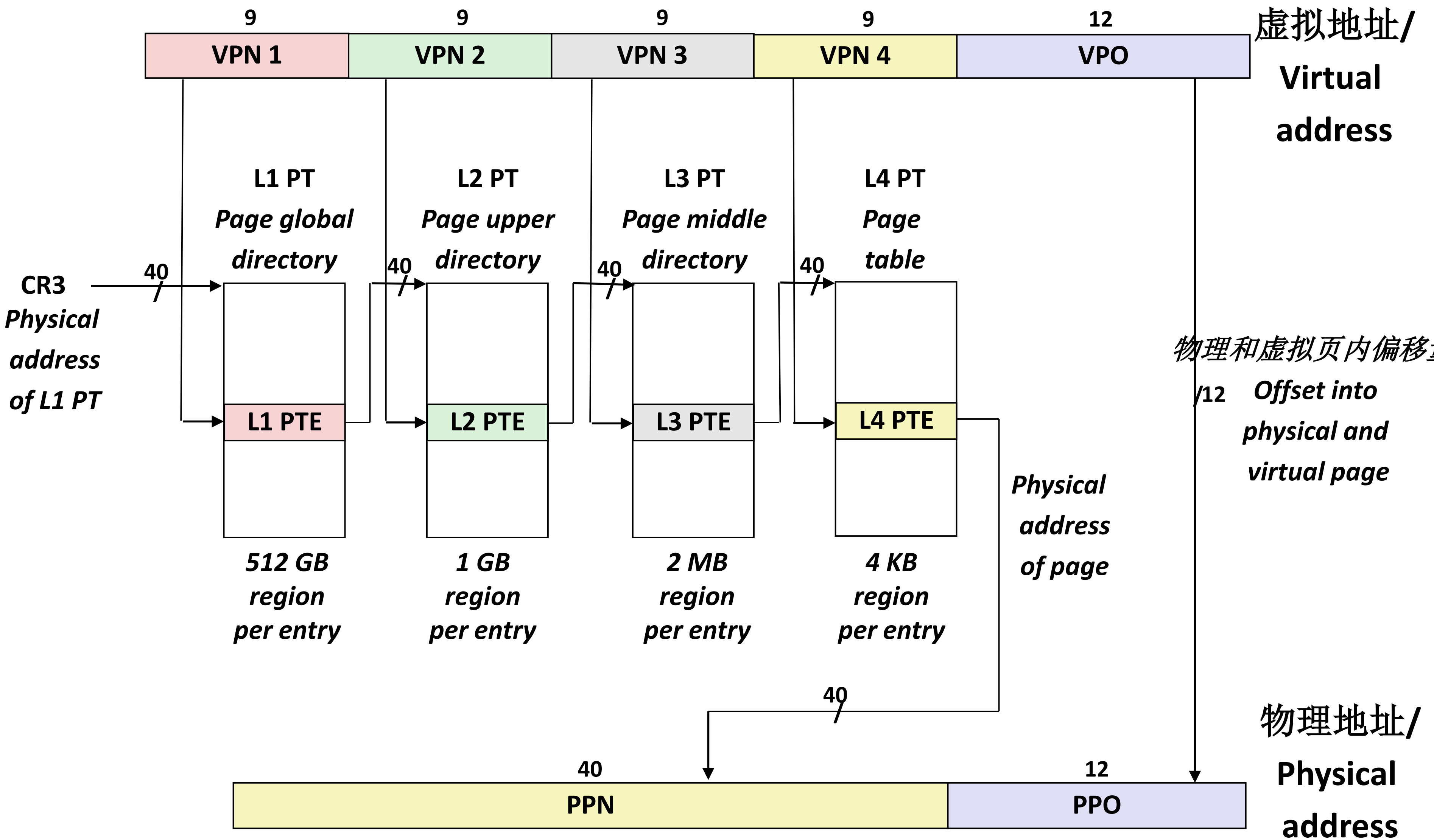
A: 引用标记/Reference bit (由MMU读写时设置，软件清除/set by MMU on reads and writes, cleared by software).

D: 脏位**/**Dirty bit (写操作时由MMU设置，软件清除/set by MMU on writes, cleared by software)

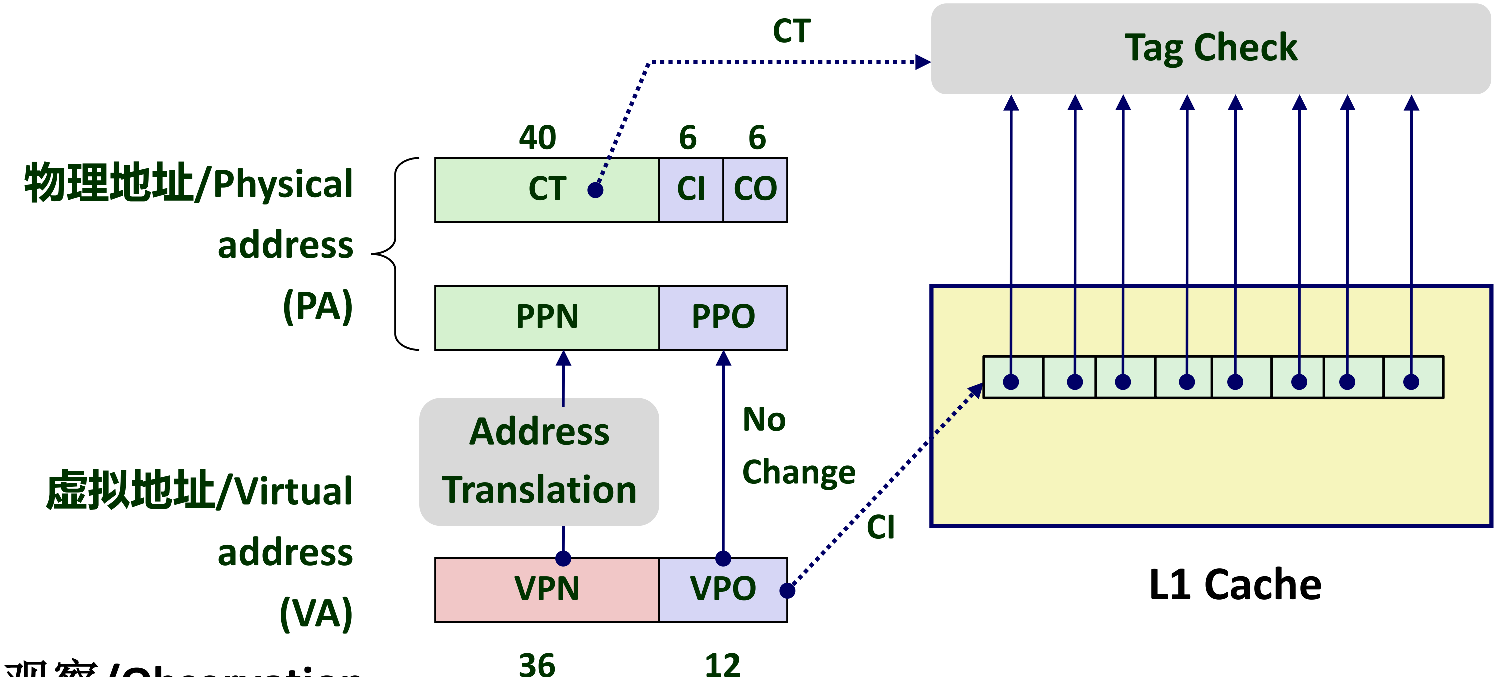
Page physical base address:物理页表地址的高40位(强制页表按照4KB对齐)/ 40 most significant bits of physical page address (forces pages to be 4KB aligned)

XD:禁止或允许取指操作/ Disable or enable instruction fetches from this page.

Core i7页表翻译/Core i7 Page Table Translation

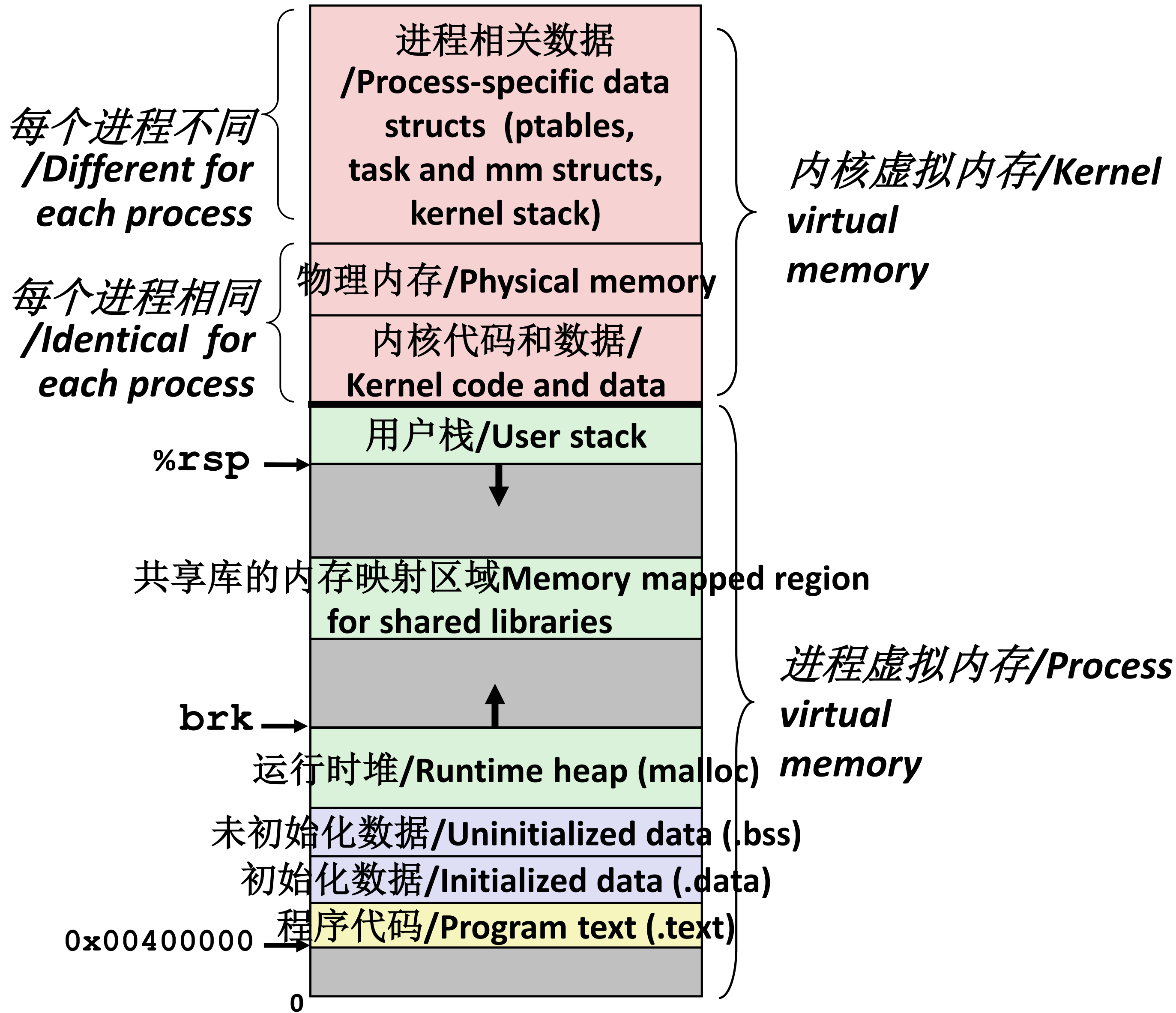


L1访问加速小技巧/Cute Trick for Speeding Up L1 Access

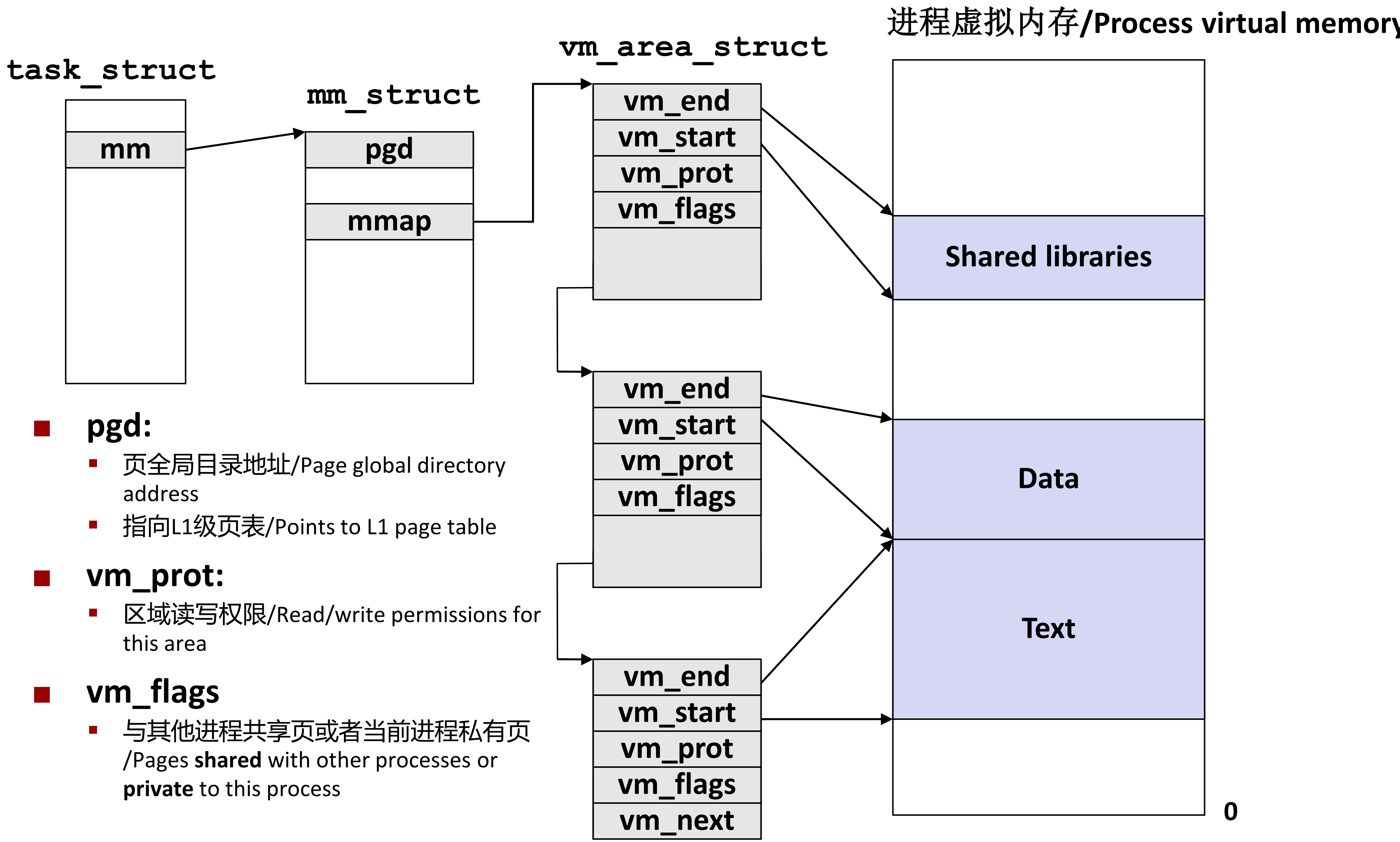


- 观察/Observation
  - 虚拟地址和物理地址中用于Cache索引的位是相同的/Bits that determine CI identical in virtual and physical address
  - 地址翻译的同时可以进行Cache索引/Can index into cache while address translation taking place
  - 通常情况下TLB会命中，PPN（Cache标记）接下来会可用/Generally we hit in TLB, so PPN bits (CT bits) available next
  - 虚拟索引，物理标记/“Virtually indexed, physically tagged”
  - Cache大小设计需要注意才能这样并行做/Cache carefully sized to make this possible

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

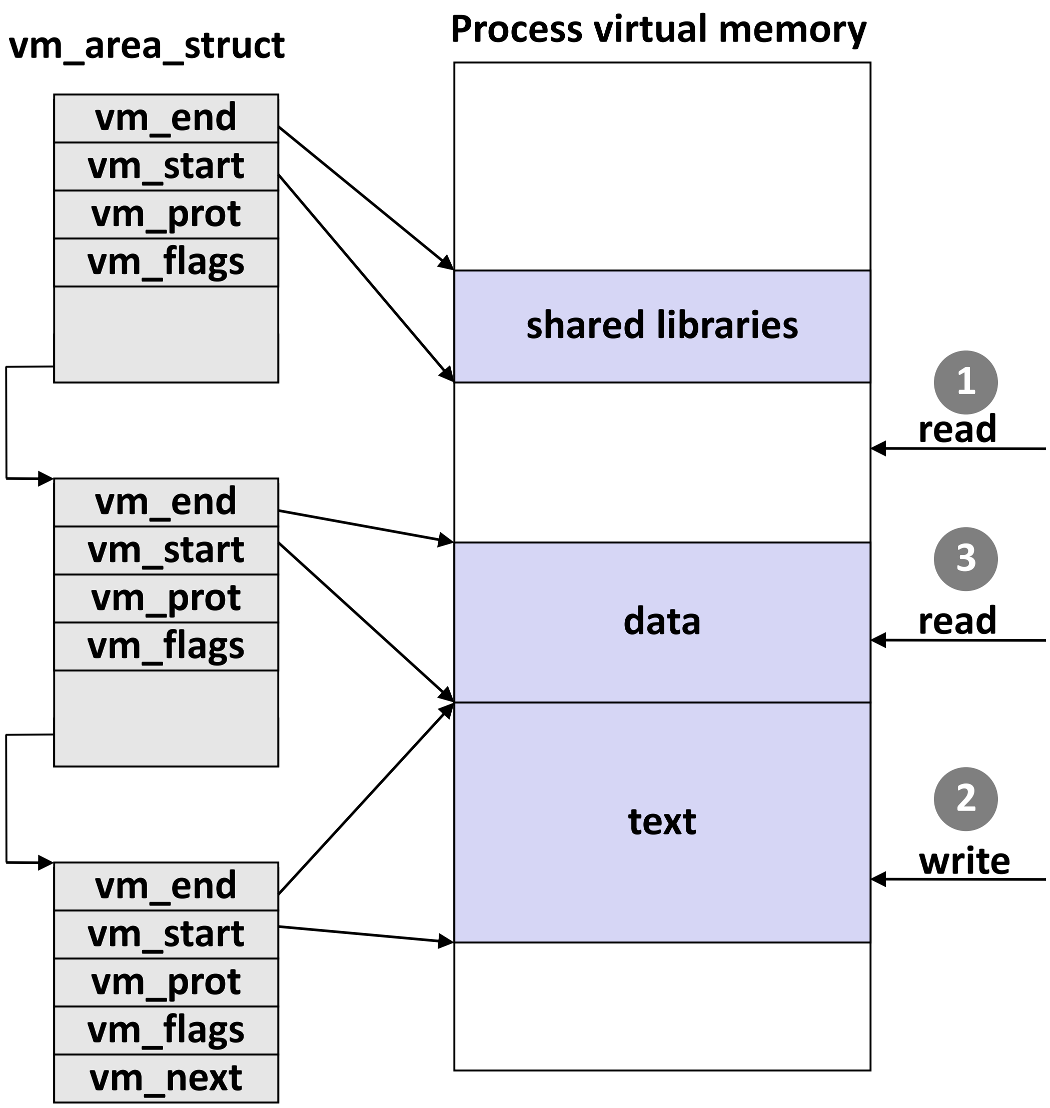


Linux将虚拟内存组织为一些区域的集合/Linux Organizes VM as Collection of “Areas”



- **pgd:**
  - 页全局目录地址/Page global directory address
  - 指向L1级页表/Points to L1 page table
- **vm_prot:**
  - 区域读写权限/Read/write permissions for this area
- **vm_flags**
  - 与其他进程共享页或者当前进程私有页/Pages **shared** with other processes or **private** to this process

Linux中的缺页中断处理/Linux Page Fault Handling



段错误/Segmentation fault:
访问不存在的页/
accessing a non-existing page

普通缺页中断/
Normal page fault

保护异常/Protection exception:
例如，对只读页进行写操作
/e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

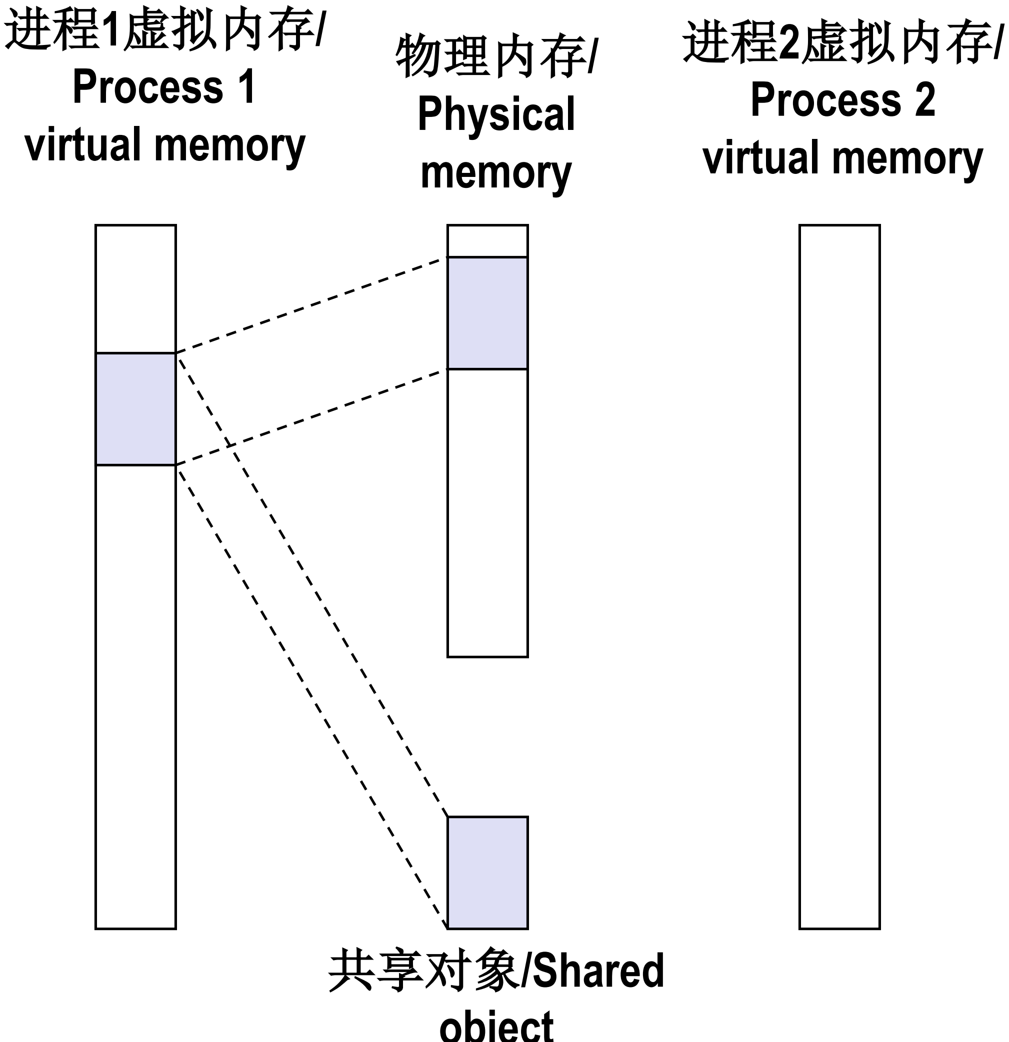
内容提纲/Today

- 简单内存系统示例/Simple memory system example
- 实例：Core i7/Linux内存系统/Case study: Core i7/Linux memory system
- 内存映射/Memory mapping

内存映射/Memory Mapping

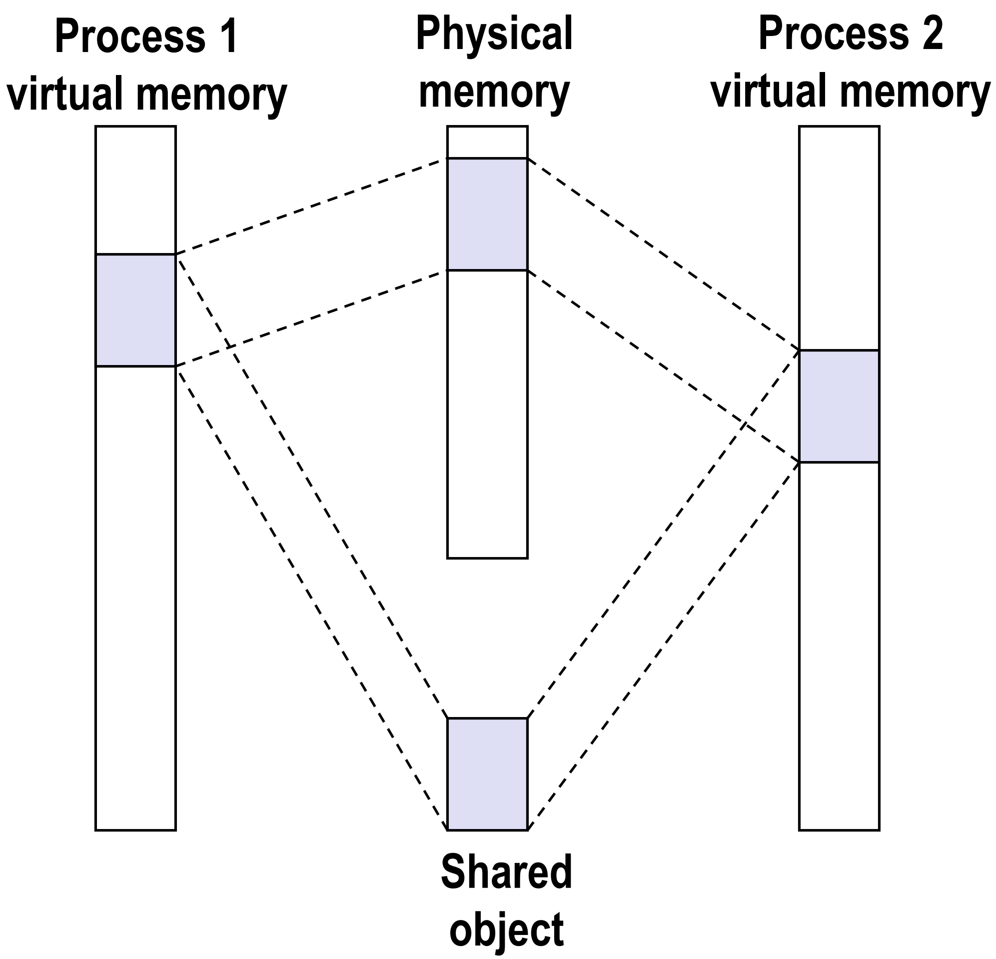
- **VM区域初始化是将其与磁盘对象相关联/VM areas initialized by associating them with disk objects.**
  - 这一过程称为内存映射/Process is known as ***memory mapping.***
- **区域可以备份/Area can be *backed by* (例如从以下获得初始值/i.e., get its initial values from) :**
  - 磁盘上的***常规文件/Regular file*** on disk (例如一个可执行目标文件/e.g., an executable object file)
    - 通过文件的节初始化页中数据/Initial page bytes come from a section of a file
  - ***匿名文件/Anonymous file*** (e.g., nothing)
    - 第一次缺页时分配一个填充为0的物理页/First fault will allocate a physical page full of 0's (***demand-zero page***)
    - 页面被写之后就和其他页一样/Once the page is written to (***dirtied***), it is like any other page
- **脏页会在内存和一个特殊的交换文件之间来回拷贝/Dirty pages are copied back and forth between memory and a special *swap file*.**

共享重回顾：共享对象/Sharing Revisited: Shared Objects



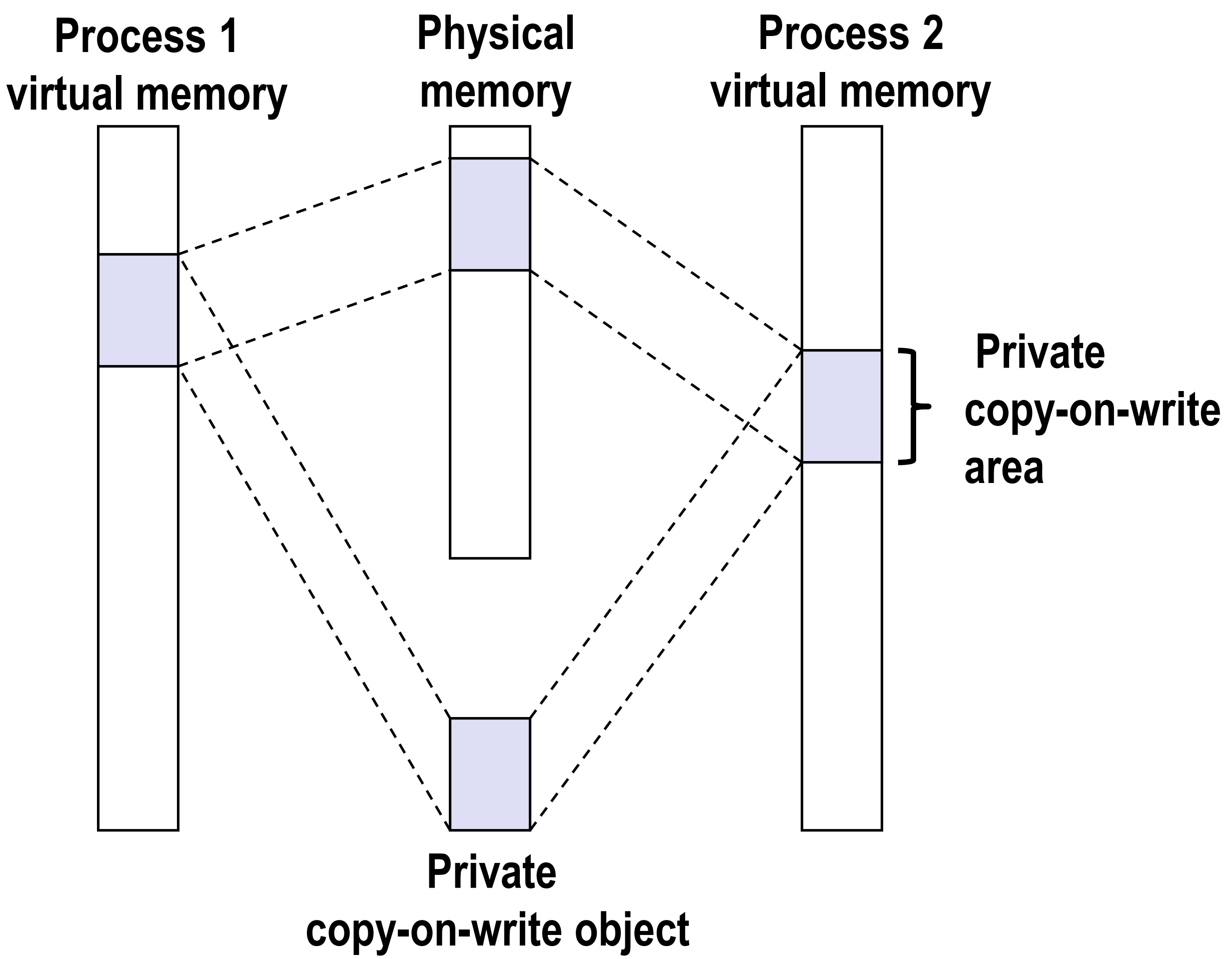
- 进程1映射共享对象/Process 1 maps the shared object.

共享重回顾：共享对象/Sharing Revisited: Shared Objects



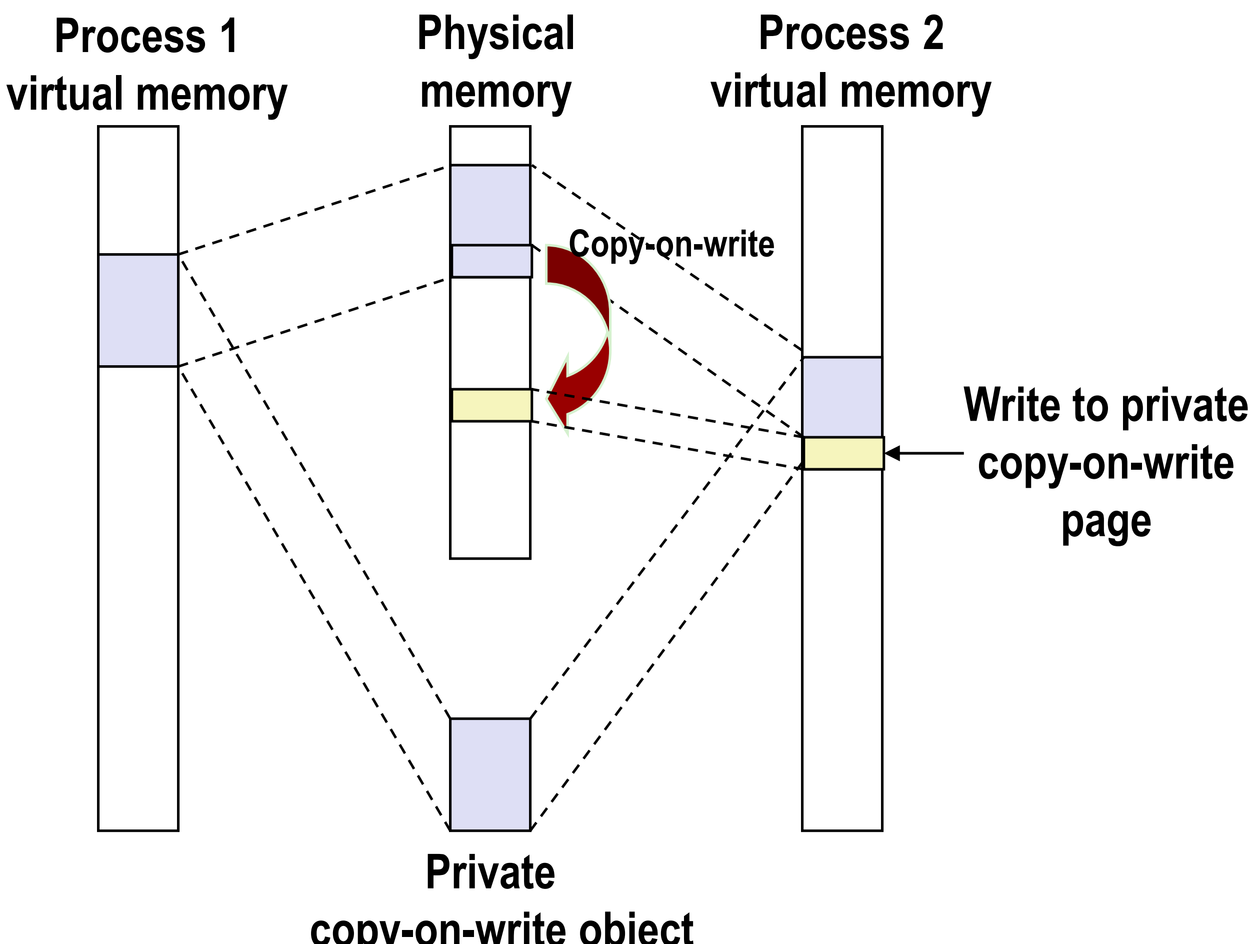
- 进程2映射共享对象/Process 2 maps the shared object.
- 注意虚拟地址如何不同/Notice how the virtual addresses can be different.

共享重回顾:私有写时拷贝对象/Sharing Revisited: Private Copy-on-write (COW) Objects



- 两个进程映射了一个私有写时拷贝对象/Two processes mapping a *private copy-on-write (COW)* object.
- 区域被标记为私有写时拷贝/Area flagged as private copy-on-write
- 私有区域的PTE被标记位只读/ PTEs in private areas are flagged as read-only

共享重回顾:私有写时拷贝对象/ Sharing Revisited: Private Copy-on-write (COW) Objects

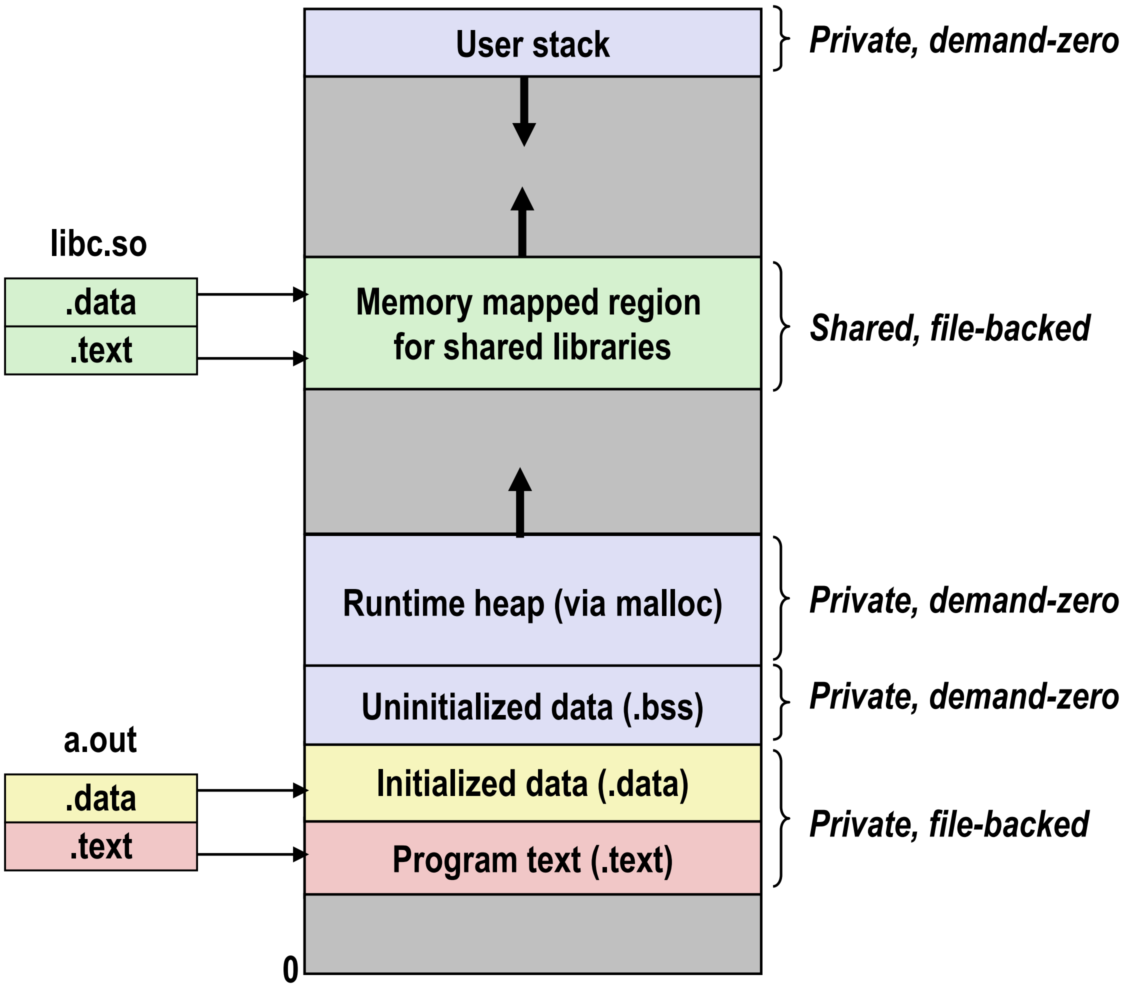


- 写私有页会触发保护异常/Instruction writing to private page triggers protection fault.
- 处理程序创建一个新的R/W页/Handler creates new R/W page.
- 处理程序返回后重新执行指令/Instruction restarts upon handler return.
- 尽可能延迟拷贝操作/Copying deferred as long as possible!

fork函数重回顾/The `fork` Function Revisited

- **VM和内存映射解释了fork如何为每个进程设置私有空间/VM and memory mapping explain how `fork` provides private address space for each process.**
- **为新进程创建虚拟地址/To create virtual address for new new process**
  - 创建完全与现有的完全相同的内存数据结构/Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - 每个进程都将其标记为只读/Flag each page in both processes as read-only
  - 在两个进程空间中的`vm_area_struct` 设置为`COW/`Flag each `vm_area_struct` in both processes as private COW
- **返回时，每个进程有完全相同的虚拟内存/On return, each process has exact copy of virtual memory**
- **后续写操作会因为COW创建新的页/Subsequent writes create new pages using COW mechanism.**

execve重回顾/The execve Function Revisited



- 在现有进程用execve加载并运行一个新的程序a.out/To load and run a new program `a.out` in the current process using `execve`:
- 释放旧区域的相关数据结构/Free `vm_area_struct`'s and page tables for old areas
- 创建新区域的相关数据结构/Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files .
- 设置PC到.text/Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.

用户级内存映射/User-Level Memory Mapping

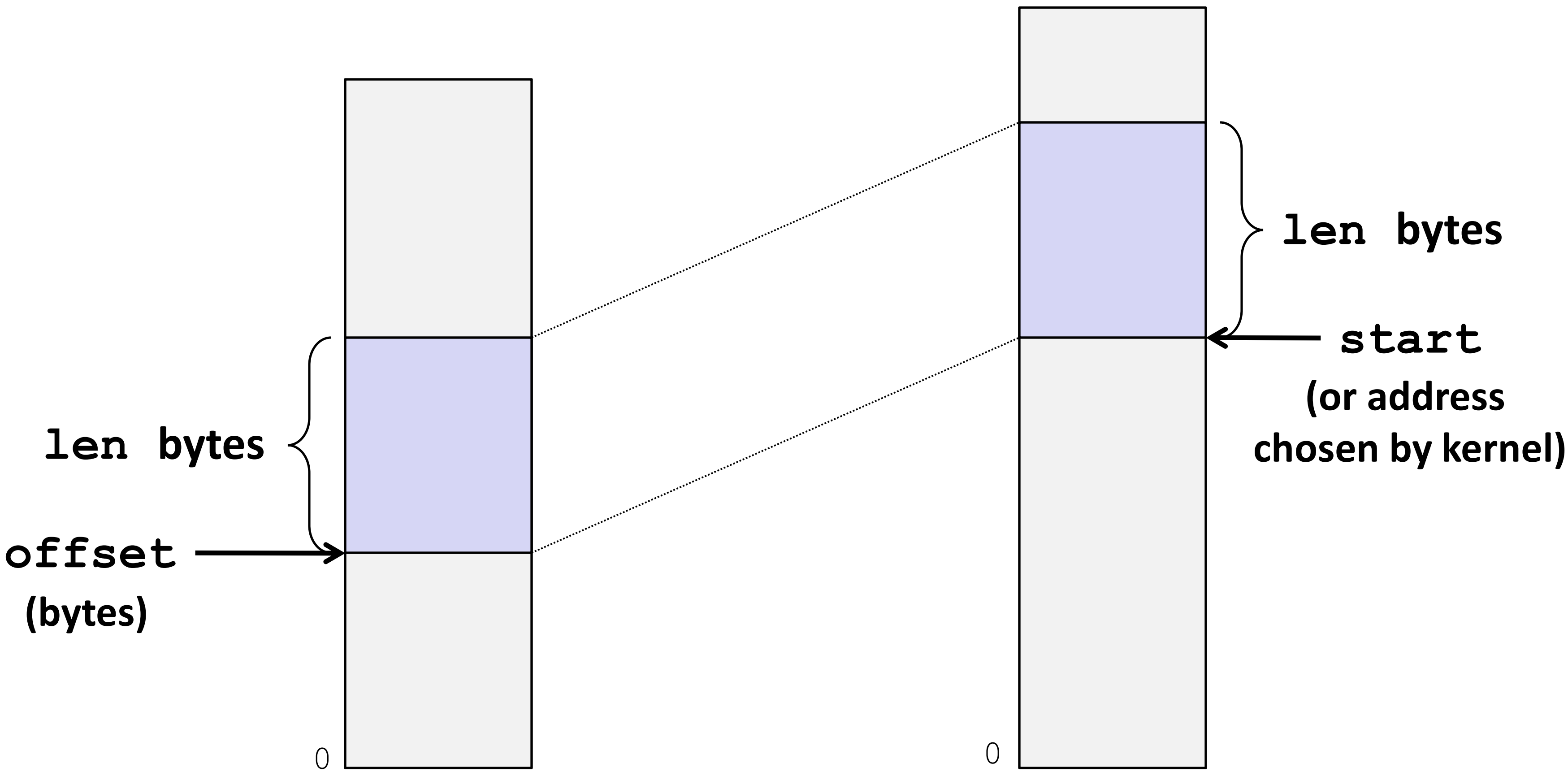
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **将文件描述符fd中偏移量`offset`开始的长度为`len`的字节映射到地址`start`/Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`**
  - `start`: may be 0 for “pick an address”/有可能是0
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- **返回一个映射区域的开始地址（有可能不是start）/Return a pointer to start of mapped area (may not be `start`)**

用户级内存映射/User-Level Memory Mapping

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



Disk file specified by file descriptor `fd`

进程虚拟内存/Process virtual memory

示例：使用mmap拷贝文件/Example: Using mmap to Copy Files

- 不用传输数据到用户空间来，就可以将一个文件拷贝到stdout/Copying a file to `stdout` without transferring data to user space .

```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