计算机系统

虚拟存储器系统

湖南大学
《计算机系统》课程教学组
肖雄仁

本讲学习内容

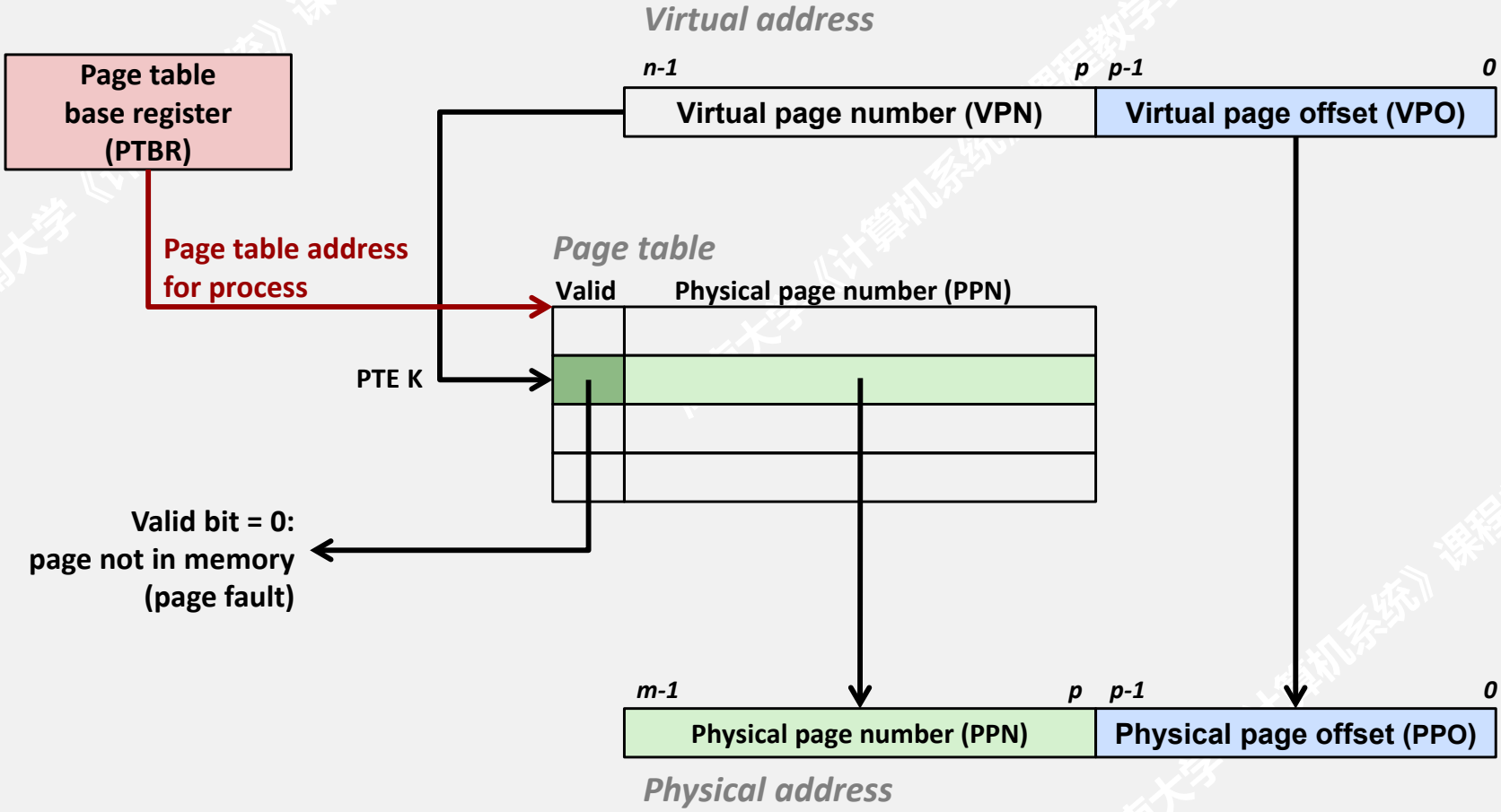
- **虚拟内存问题与解答**
- 简单存储系统示例
- 案例研究：Core i7 内存系统
- 内存映射

虚拟内存回顾

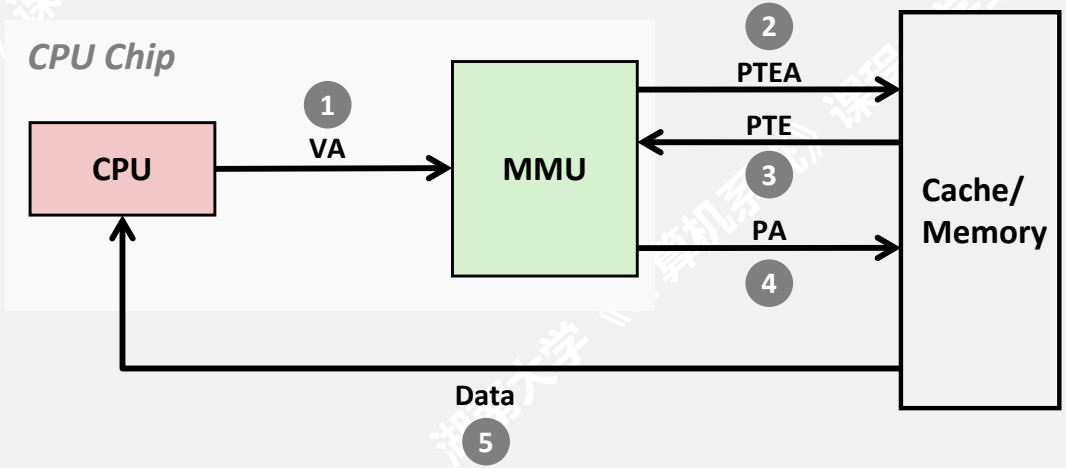
- 虚拟内存的程序员视角
  - 每个进程都有自己的专用线性地址空间
  - 不能被其他进程破坏

- 虚拟内存的系统视角
  - 通过缓存虚拟内存页面高效地使用内存
    - 仅因为局部性而高效
  - 简化内存管理和编程
  - 通过提供相应的标志位来检查权限，从而简化保护

回顾：使用页表的地址转换



回顾：地址转换——页命中



1) 处理器将虚拟地址发送到MMU

2-3) MMU从内存中的页表中获取PTE

4) MMU将物理地址发送到高速缓存/内存

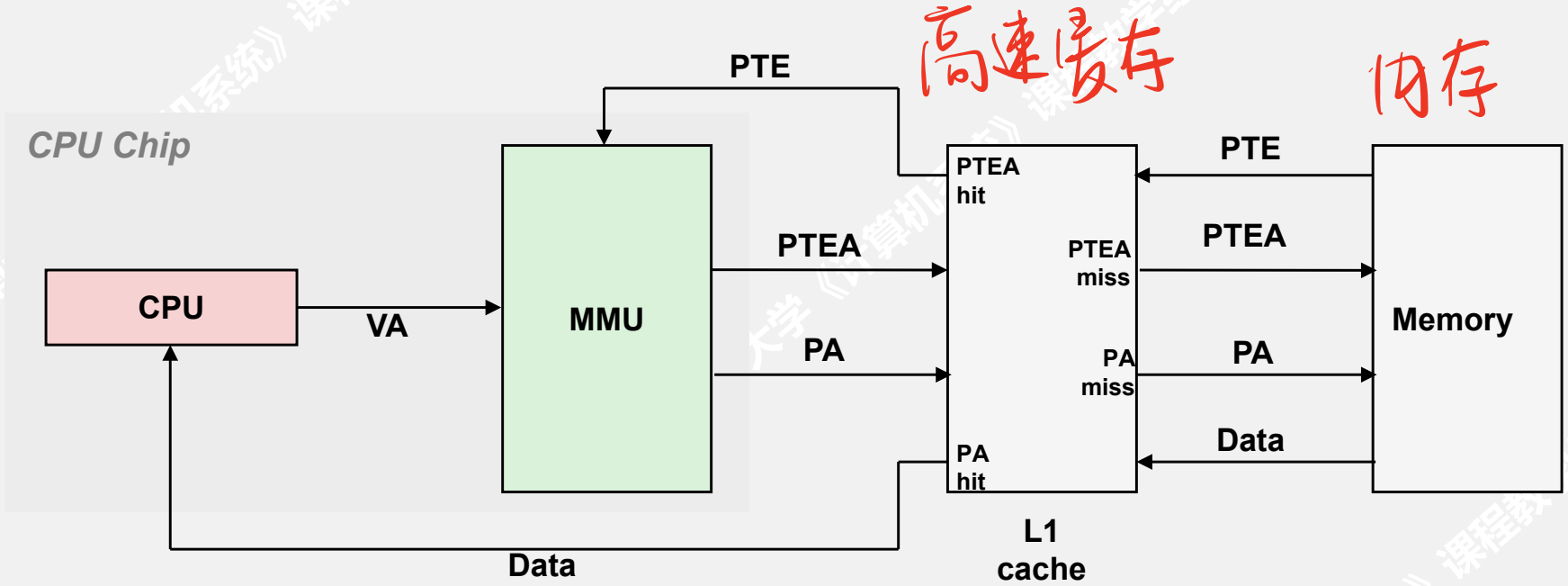
5) 高速缓存/内存将数据字发送到处理器

问题一

- PTE和其它内存数据一样保存吗?

- 是的

内存中的页表 与 其他内存数据一样



页表条目就是数据

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

问题二

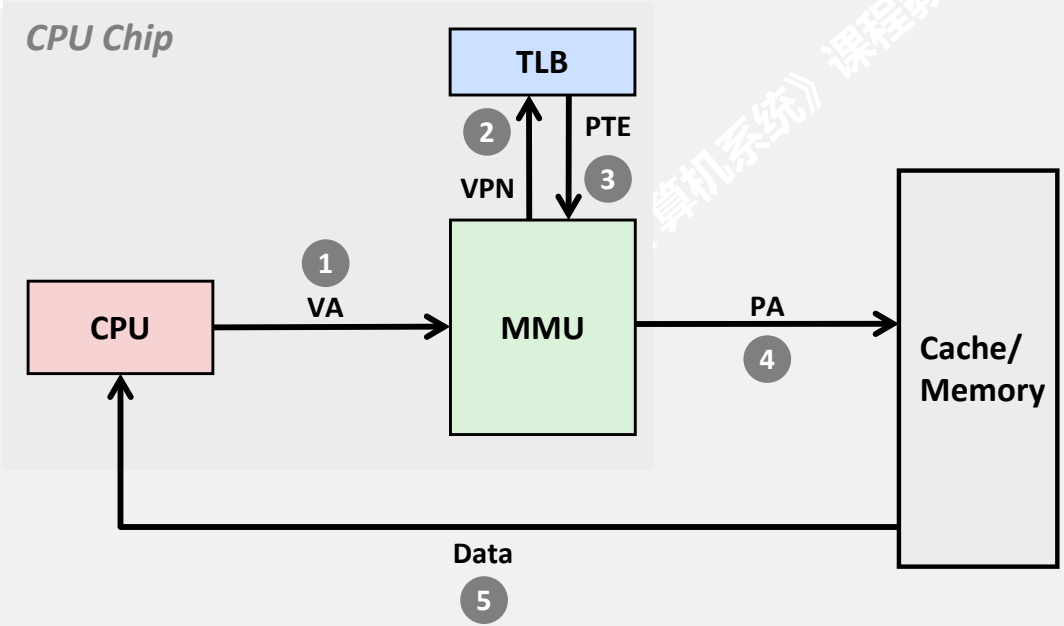
- 每次需要数据都需两次访内会不会很慢?
- 是的，但是实际上MMU并不会这样做
  - TLB出现了

使用TLB 加速地址转换

- **页表条目（PTE）缓存在L1高速缓存中**
  - PTE可能被其他数据驱逐
  - PTE命中仍需要较小的L1延迟

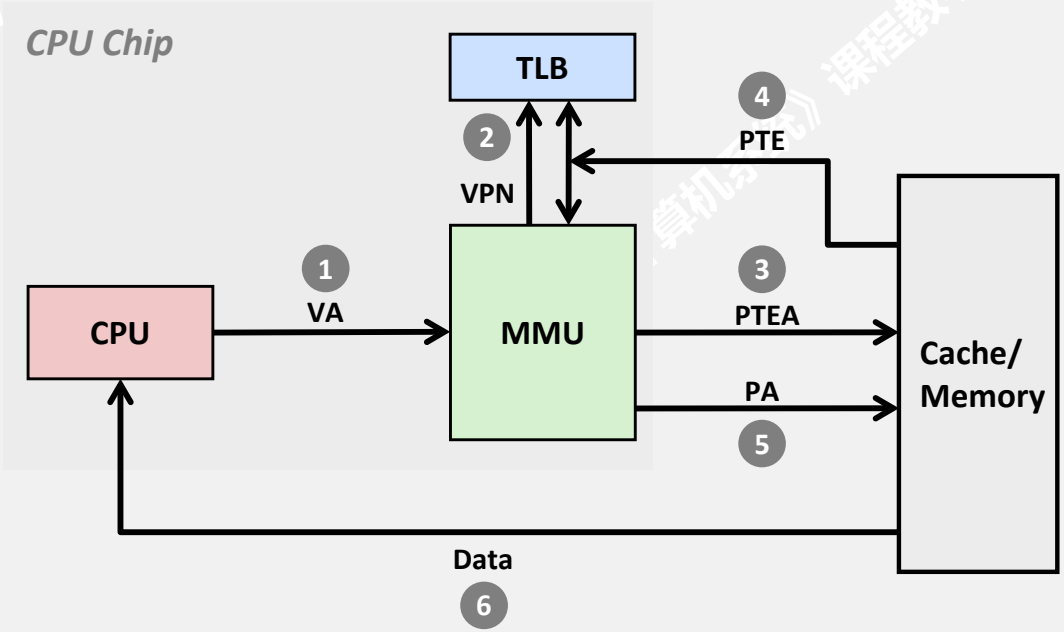
- **解决方案：Translation Lookaside Buffer（TLB）**
  - MMU中的小硬件缓存
  - 将虚拟页码映射到物理页码
  - 包含少量页的完整页表条目

TLB命中



一次 TLB 命中则避免了一次内存访问

TLB不命中



一次TLB不命中引发额外的内存访问 (获取PTE)

幸运的是：TLB 不命中很少见， Why?

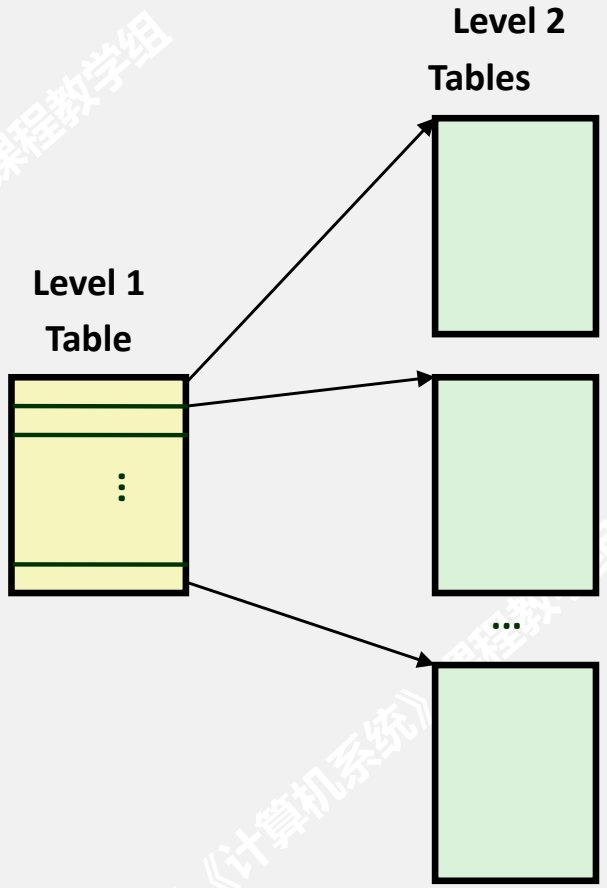
问题三

- 页面表不是很大吗？ 如何将其存储在RAM中？

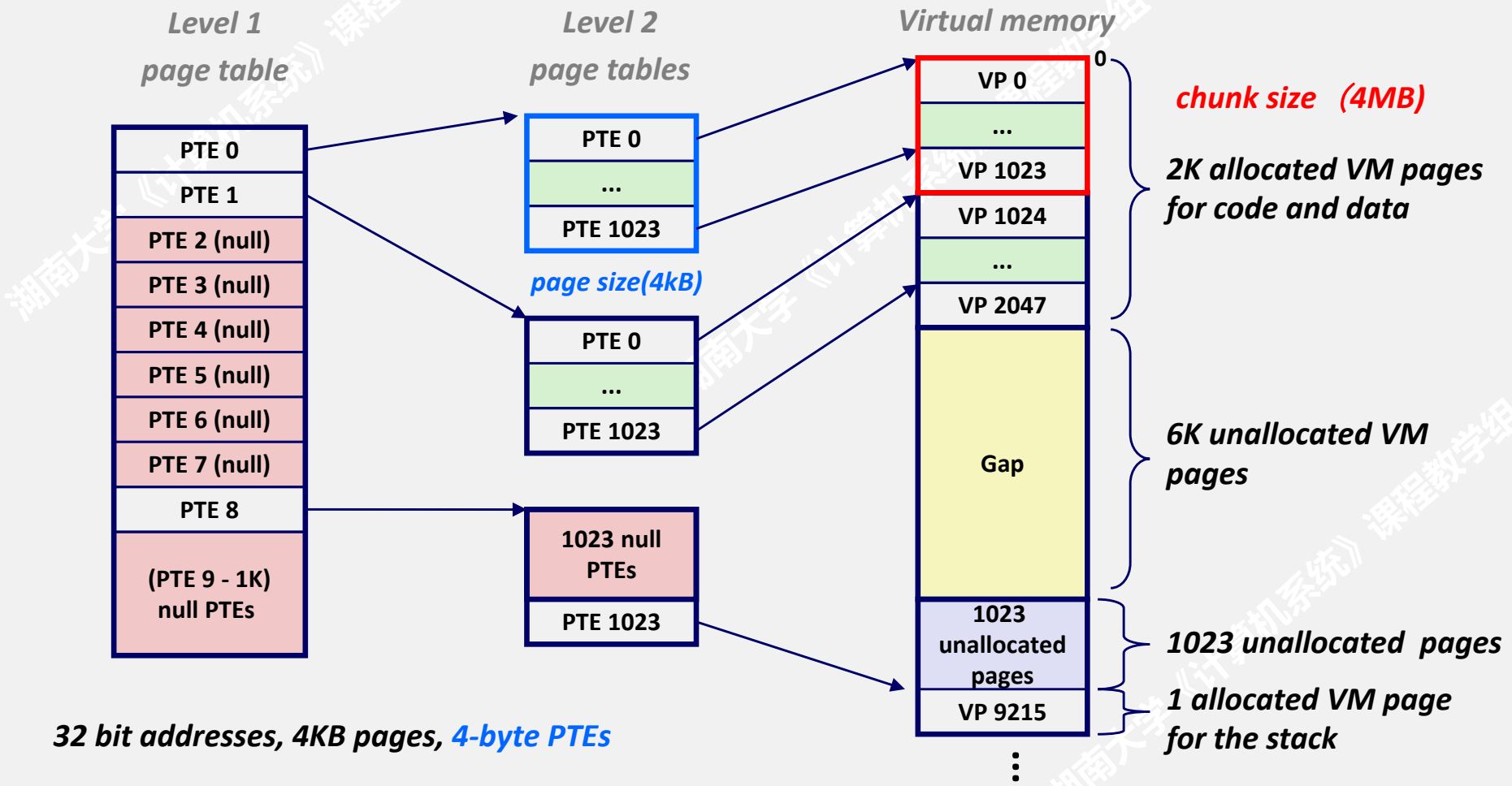
- 是的，应该是
  - 所以，实际的页表不是简单的数组

多级页表

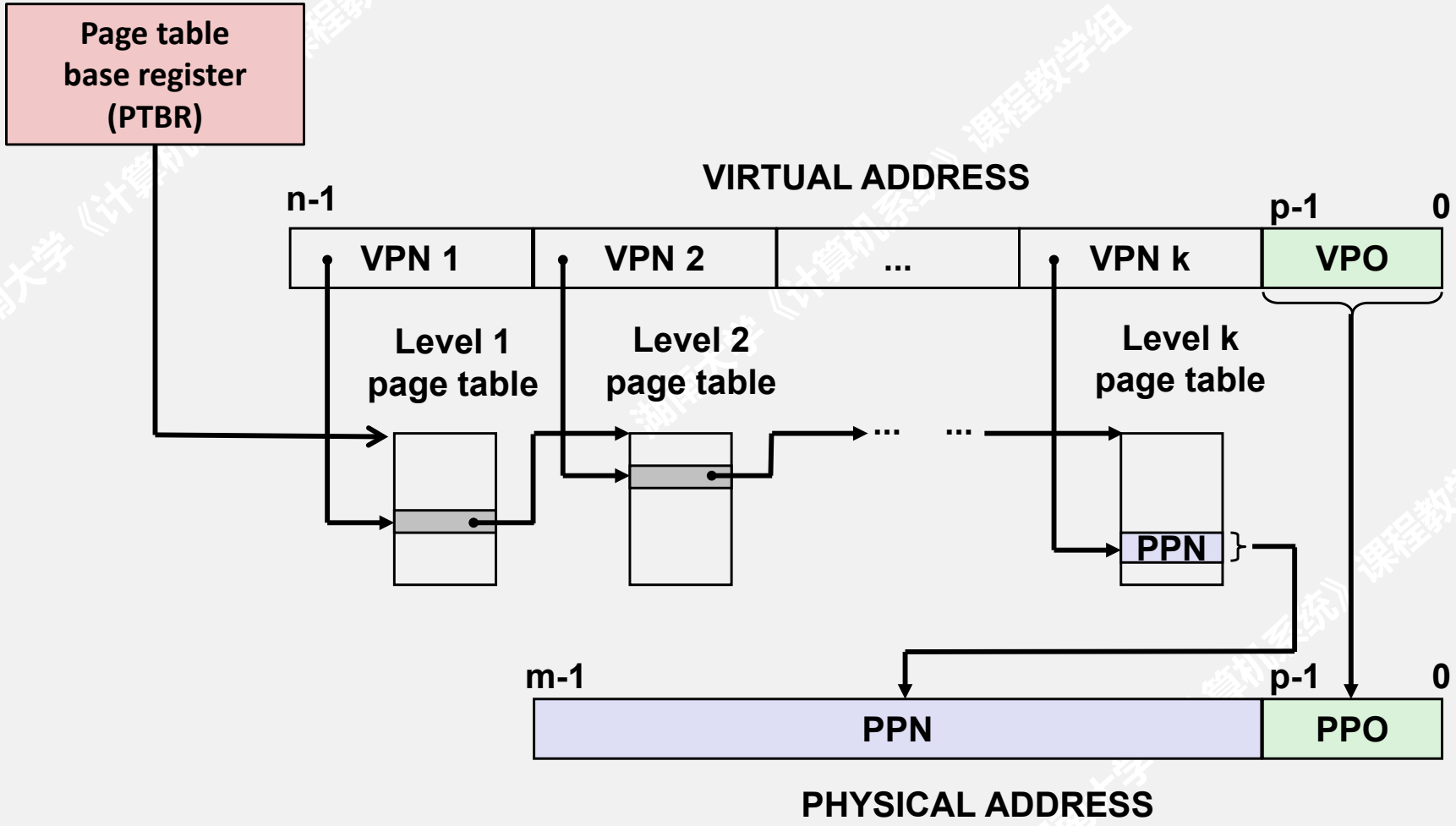
- **设若:**
  - 页大小为4KB (2^{12}), 48位 地址空间, 8字节 PTE
- **问题:**
  - 需要大小为512 GB 页表!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- **常用解决方案:多级页表**
- 例如:两级页表
  - 一级页表: 每个PTE指向一个页表。 (**总是驻留内存**)
  - 二级页表: 每个PTE指向一页 (像其他任何数据一样换页: **page in&page out**)



二级页表层次结构


Level 1
page table
Level 2
page tables
Virtual memory
PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(PTE 9 - 1K)
null PTEs
PTE 0
...
PTE 1023
page size(4kB)
PTE 0
...
PTE 1023
1023 null
PTEs
PTE 1023
0
VP 0
...
VP 1023
VP 1024
...
VP 2047
Gap
1023
unallocated
pages
VP 9215
chunk size (4MB)
2K allocated VM pages
for code and data
6K unallocated VM
pages
1023 unallocated pages
1 allocated VM page
for the stack
32 bit addresses, 4KB pages, 4-byte PTEs

多级页表地址转换


Page table
base register
(PTBR)
VIRTUAL ADDRESS
n-1
p-1
0
VPN 1
VPN 2
...
VPN k
VPO
Level 1
page table
Level 2
page table
Level k
page table
PPN
m-1
p-1
0
PPN
PPO
PHYSICAL ADDRESS

问题四

- 那么之前学习过的 fork() 执行是不是很慢，因为子进程需要完全拷贝一份父进程的地址空间?

- 看起来似乎是这样……所以实际上fork()并不会真的这样处理……

本讲学习内容

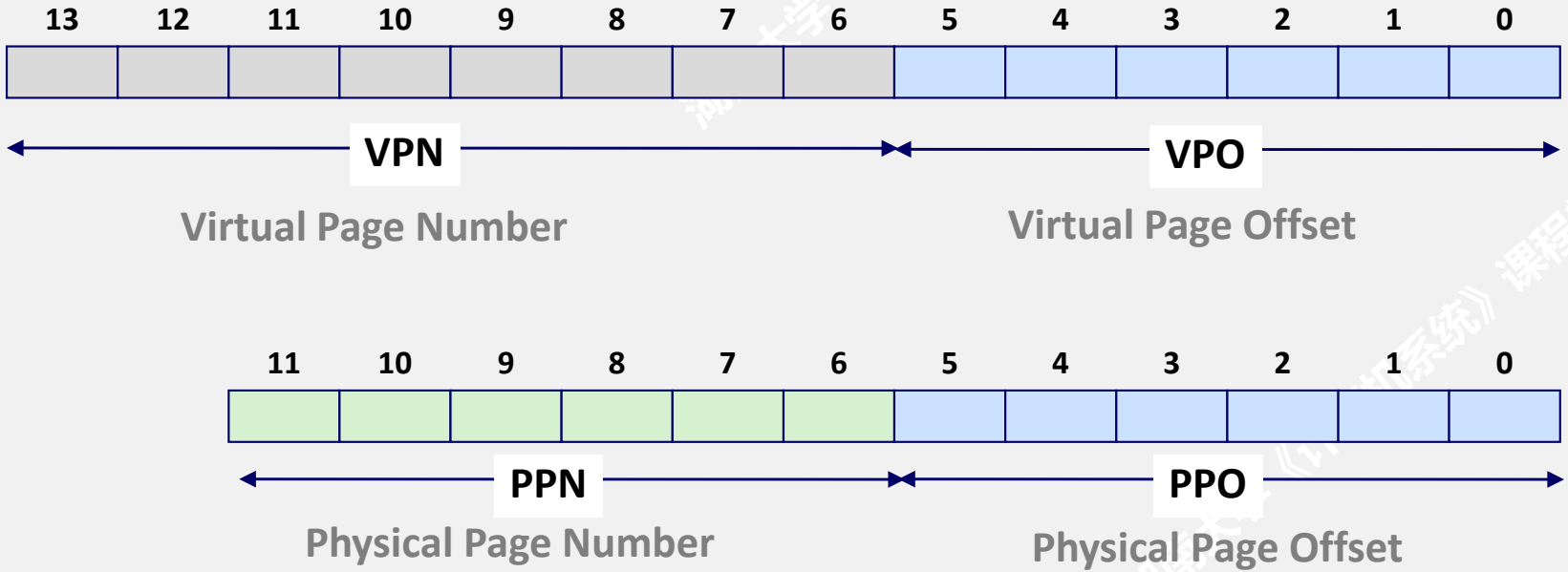
- 虚拟内存问题与解答
- **简单存储系统示例**
- 案例研究：Core i7 内存系统
- 内存映射

地址转换符号回顾

- **基本参数**
  - **N = 2^n :** 虚拟地址空间中的地址数
  - **M = 2^m :**物理地址空间中的地址数
  - **P = 2^p :** 页面大小（字节）
- **虚拟地址的组成部分（VA）**
  - **TLBI**: TLB 索引
  - **TLBT**: TLB 标记 Tag
  - **VPO**: 虚拟页面偏移
  - **VPN**: 虚拟页号
- **物理地址的组成部分（PA）**
  - **PPO**: 物理页面偏移（与VPO相同）
  - **PPN**: 物理页码
  - **CO**: 缓存行中的字节偏移量
  - **CI**: 缓存索引
  - **CT**: 缓存标记

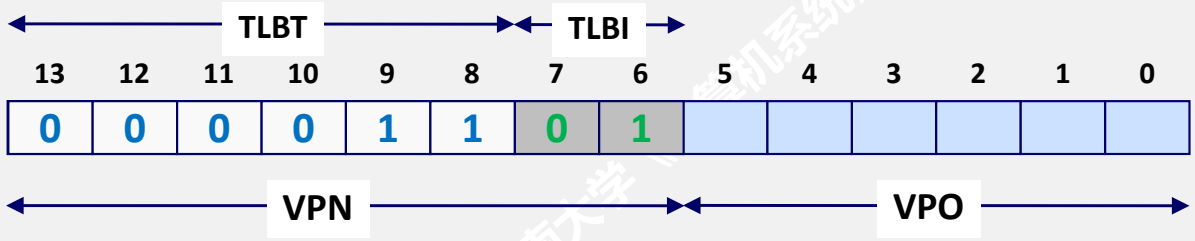
简单内存系统示例

- **地址**
  - 14位虚地址
  - 12位物理地址
  - 页大小 = 64 字节



1.简单内存系统的TLB

- 16 个条目
- 4路组相连



VPN = 0b1101 = 0x0D

Translation Lookaside Buffer (TLB)

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

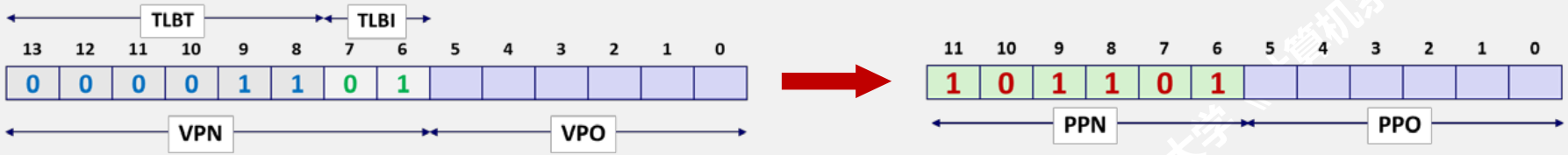
2.简单内存系统的页表

仅显示最开始的16条 (共256条)

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

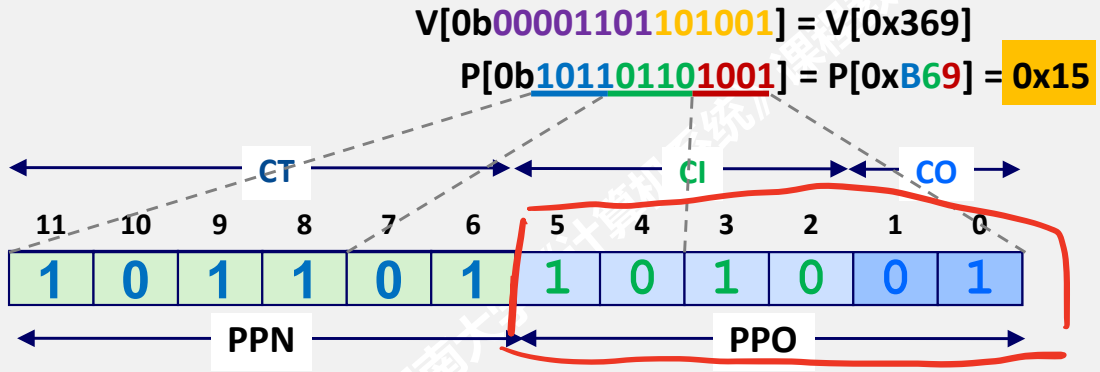
VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D



3.简单内存系统的高速缓存

- 16 行, 每个高速缓存行4字节
- 物理寻址
- 直接映射

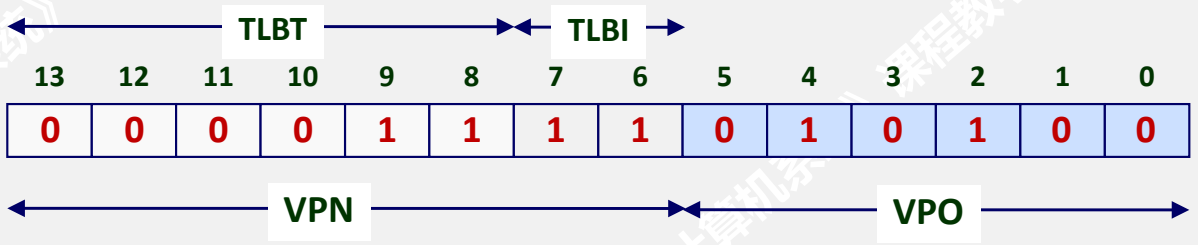


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

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	–	–	–	–

地址转换示例一

Virtual Address: 0x03D4

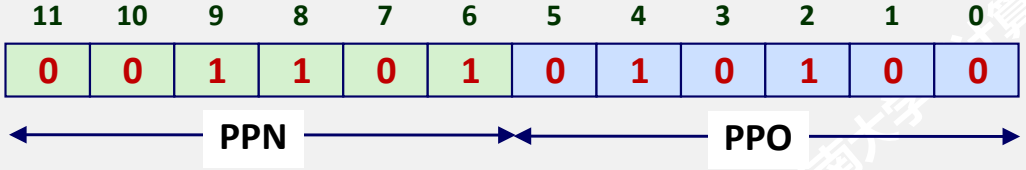


VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

TLB

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

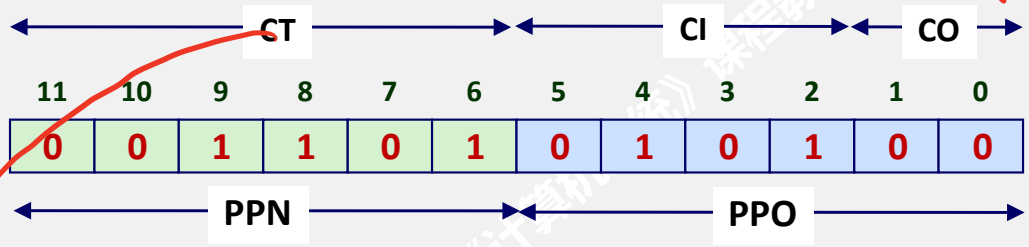
Physical Address: 0x0354



地址转换示例一

Physical Address: 0x0354

有物理地址后取高速缓存找



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

Cache

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

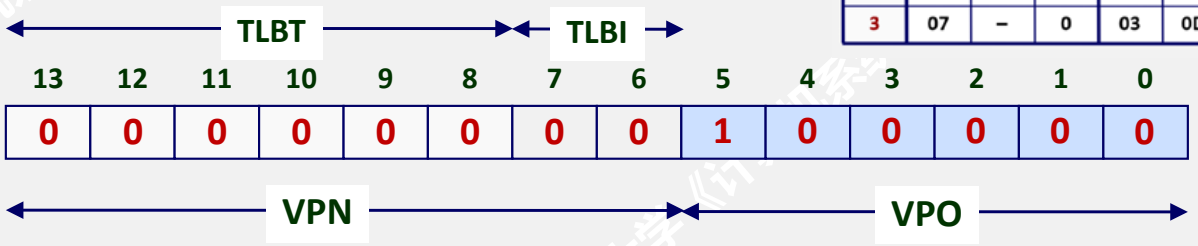
Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

地址转换示例二: TLB/Cache Miss

Translation Lookaside Buffer (TLB)

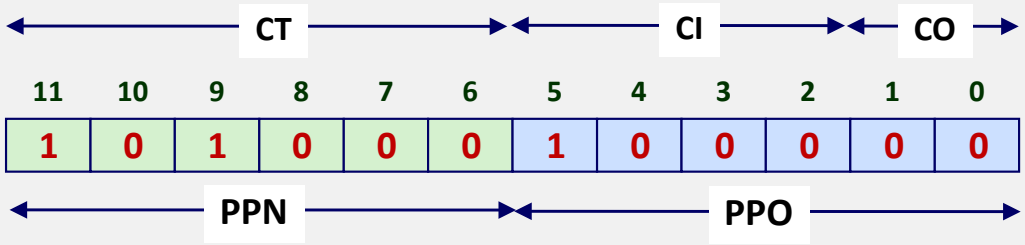
Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Virtual Address: 0x0020



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

Physical Address



CO 0 CI 0x8 CT 0x28 Hit? __ Byte: ____

Page table

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

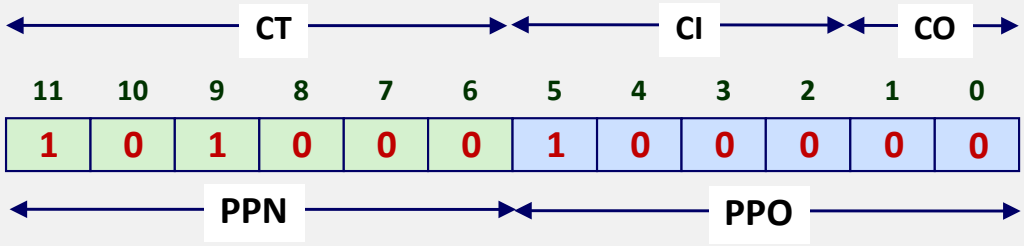
地址转换示例二: TLB/Cache Miss

Cache

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	–	–	–	–

Physical Address

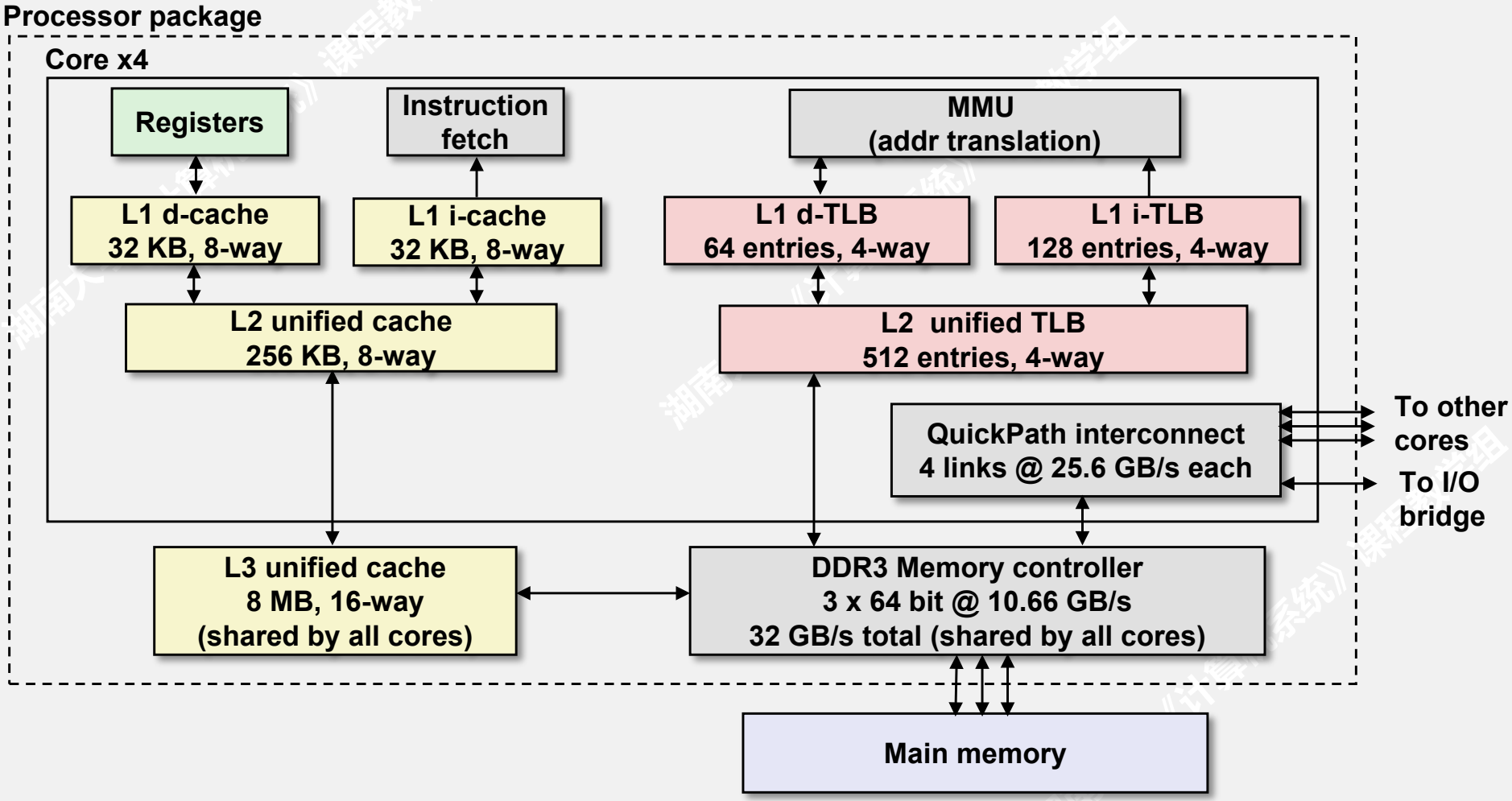


CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

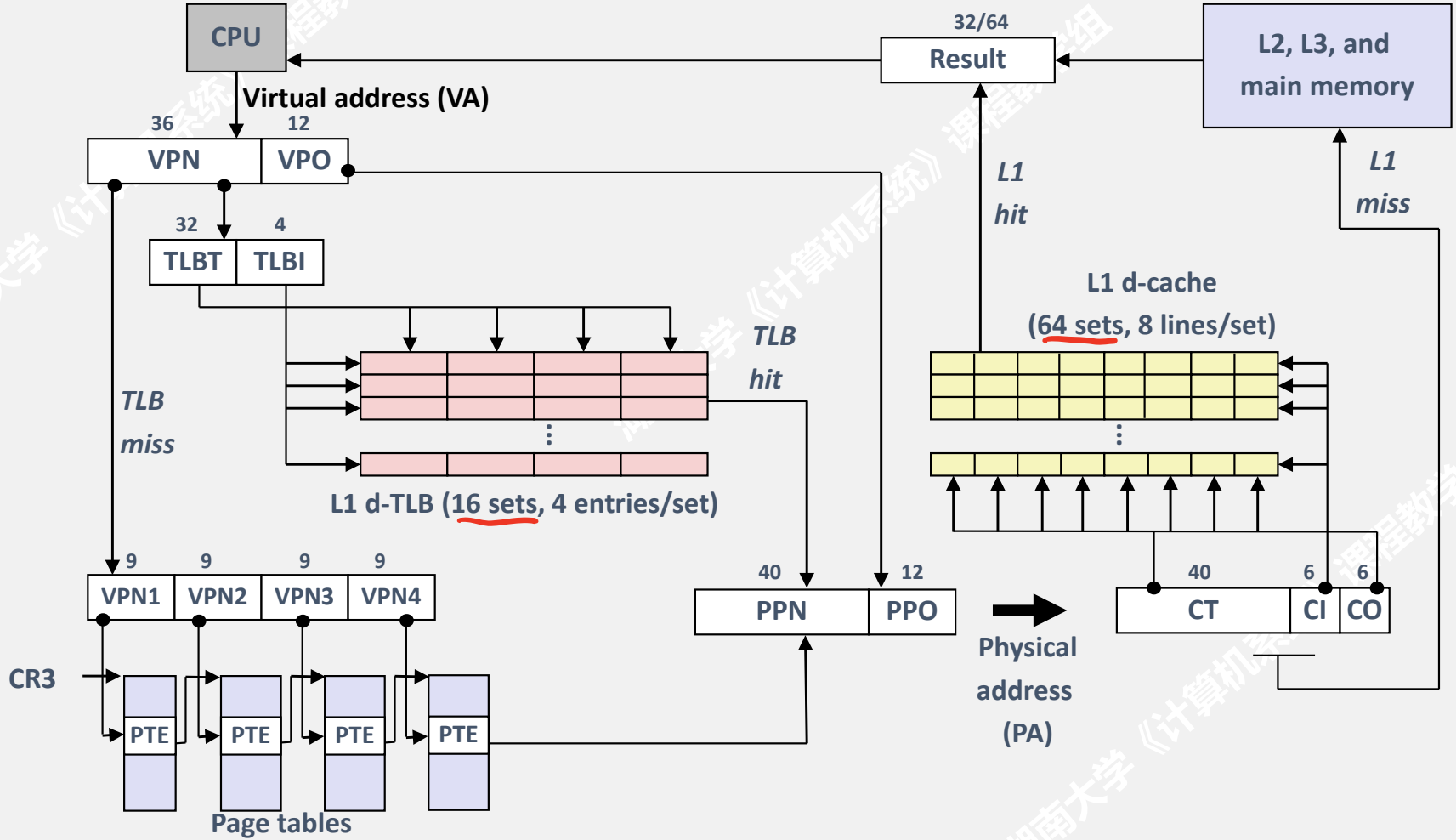
本讲学习内容

- 虚拟内存问题与解答
- 简单存储系统示例
- **案例研究：Core i7 内存系统**
- 内存映射

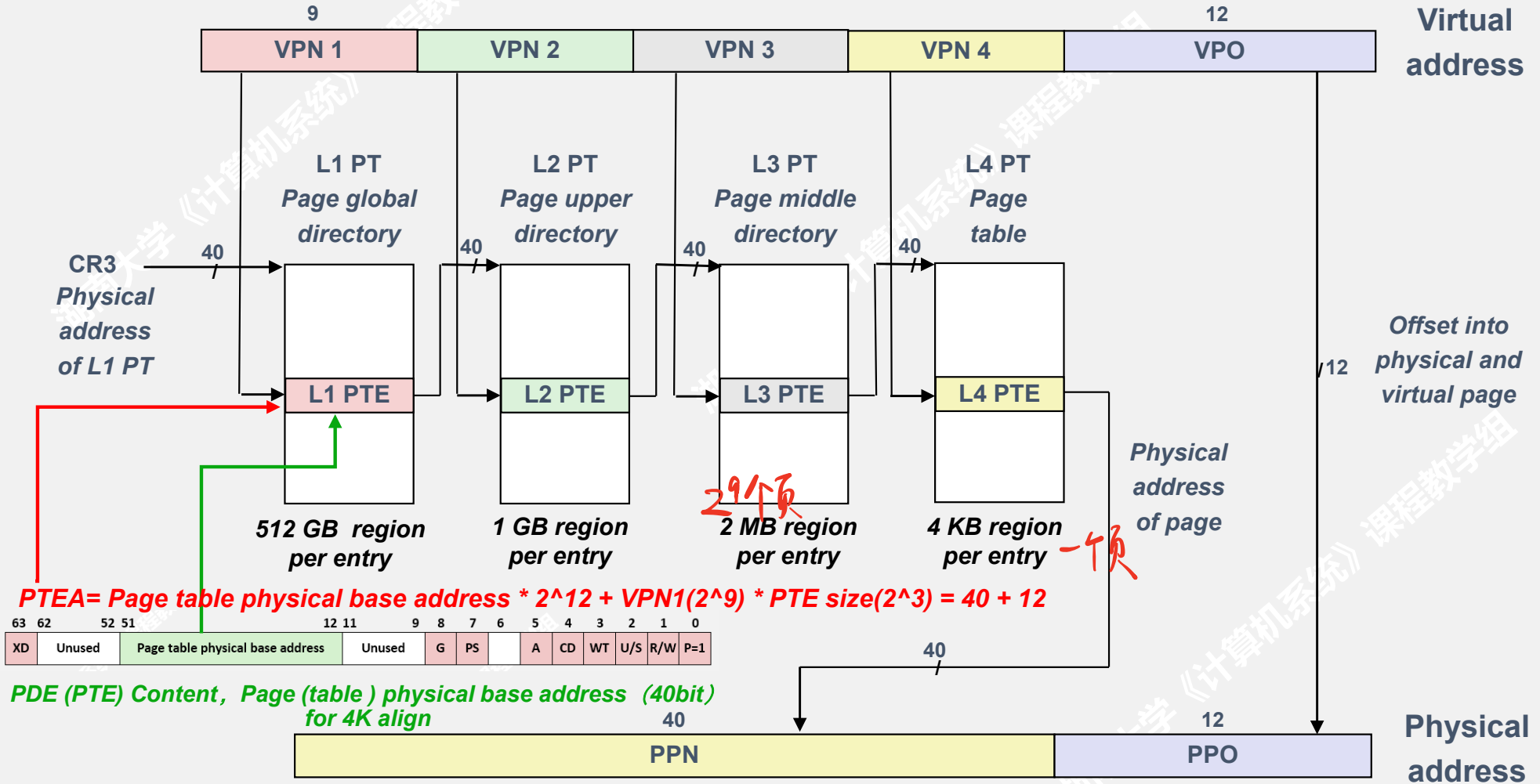
Intel Core i7 存储系统



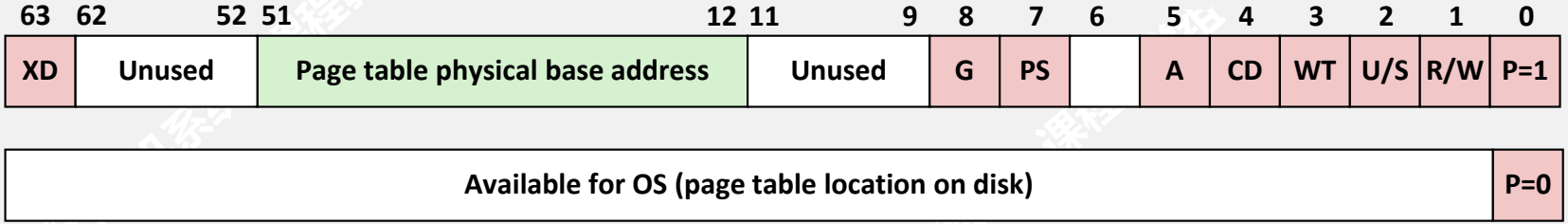
端到端Core i7地址转换



Intel Core i7 页表结构



Core i7 Level 1-3 PTE(PDE)



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

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

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

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

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

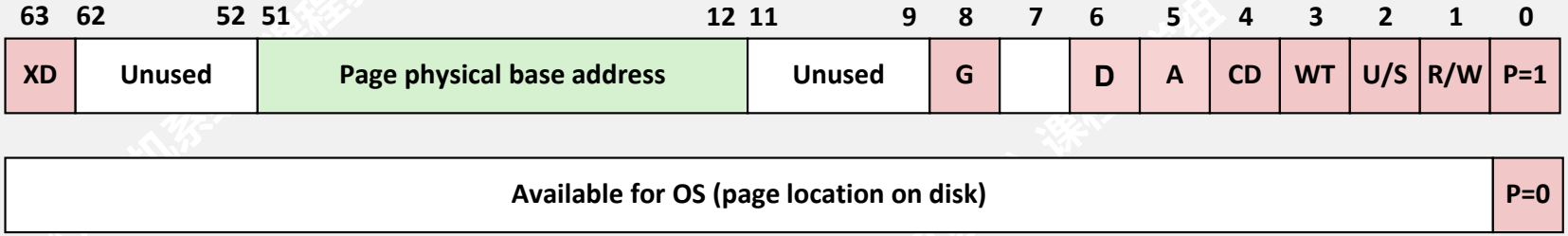
A: Reference bit (set by MMU on reads and writes, cleared by software).

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

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

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

Core i7 Level 4 PTE



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

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

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

U/S: User or supervisor mode access

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)

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

XD: Disable or enable instruction fetches from this page.

本讲学习内容

- 虚拟内存问题与解答
- 简单存储系统示例
- 案例研究：Core i7 内存系统
- **内存映射**

内存映射

- **通过将虚存区域与磁盘对象相关联来初始化虚存区域**
  - 所谓的内存映射

- **虚存区域可以由（即从其获取其初始值）支持：**
  - 磁盘上的常规文件（例如，可执行目标文件）
    - 初始页字节来自文件的一部分
  - 匿名文件（例如，什么都没有）
    - 第一个故障将分配一个全0的物理页面（请求零页面）
    - 一旦页面被写入（脏），它就像任何其他页面一样

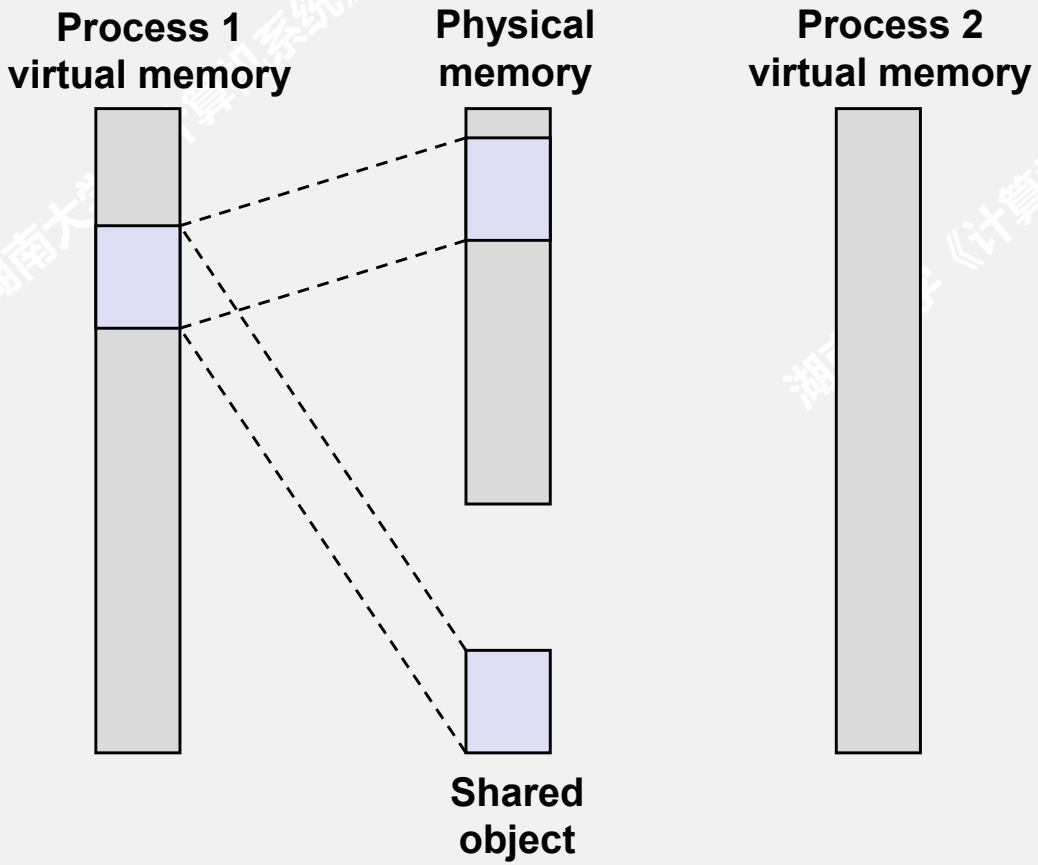
- **脏页在内存和特殊交换文件（swap file）之间来回复制**

按需调度页

- **关键点：在引用虚拟页面之前，不会将虚拟页面复制到物理内存中！**
  - 按需调度页 demand paging

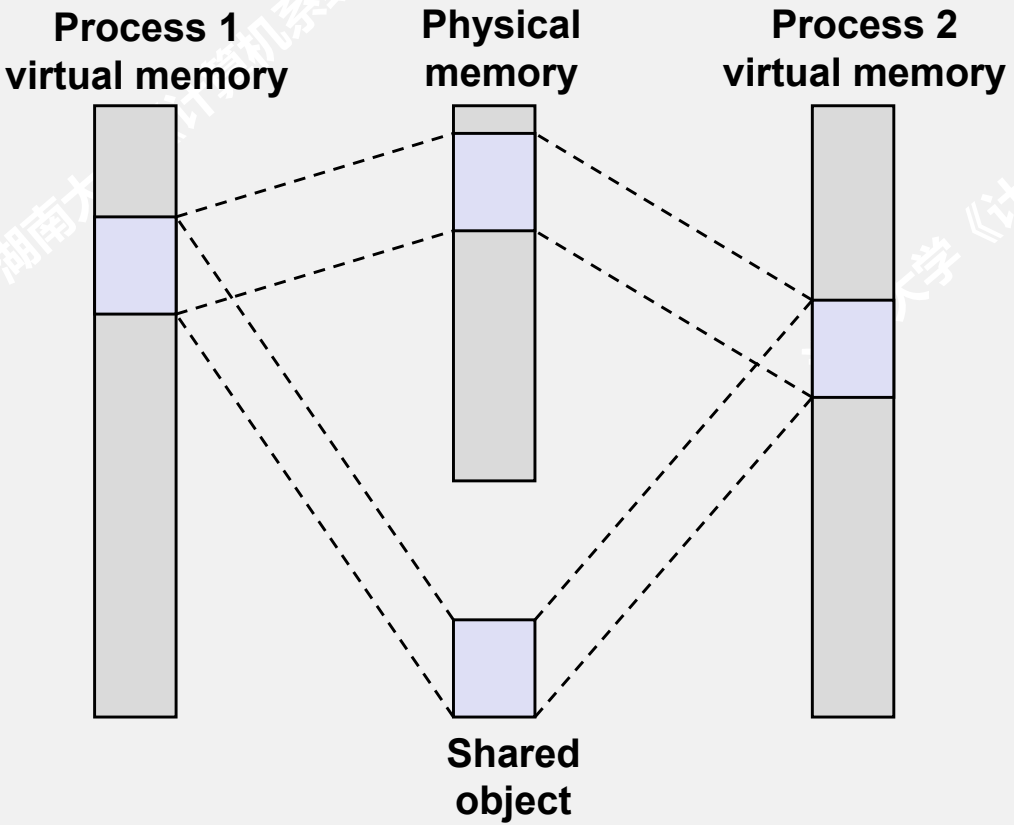
- **时间和空间效率至关重要**

再谈共享对象



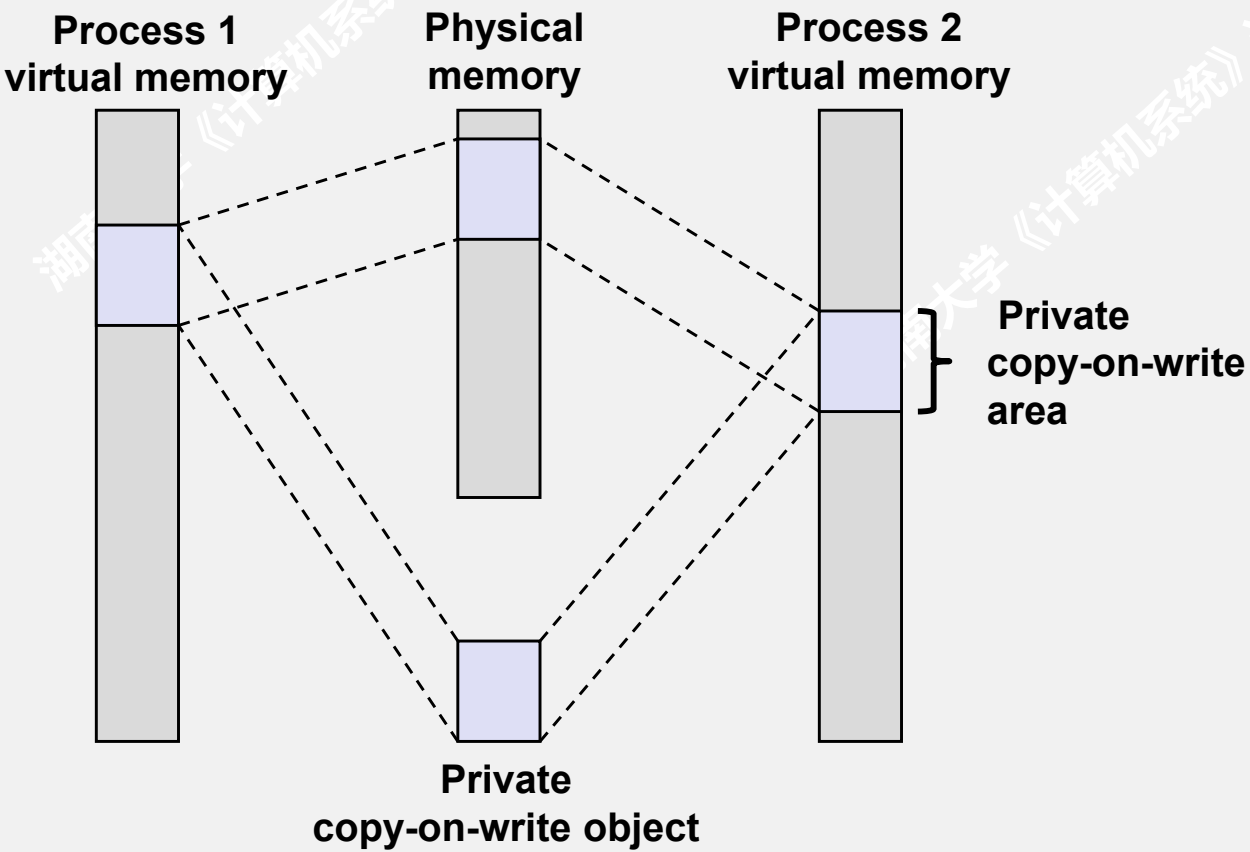
- 进程1映射到共享对象

共享对象



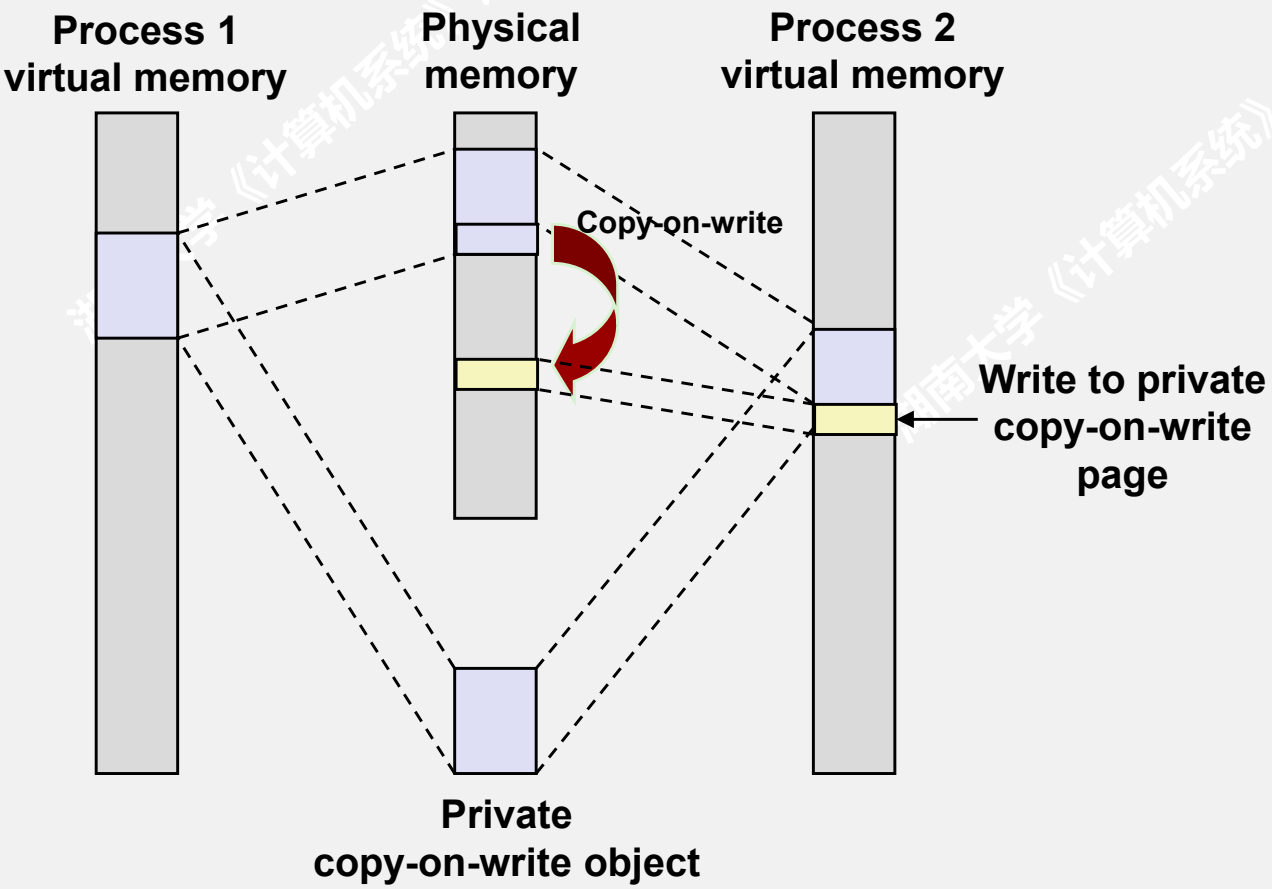
- 进程 2 映射到共享对象
- 注意虚拟地址是不同的

私有写时复制（COW）对象



- 两个进程映射一个私有的写时复制（COW）对象
- 区域被标记为私有写时复制
- 私有区域中的PTE被标记为只读

私有的COW对象

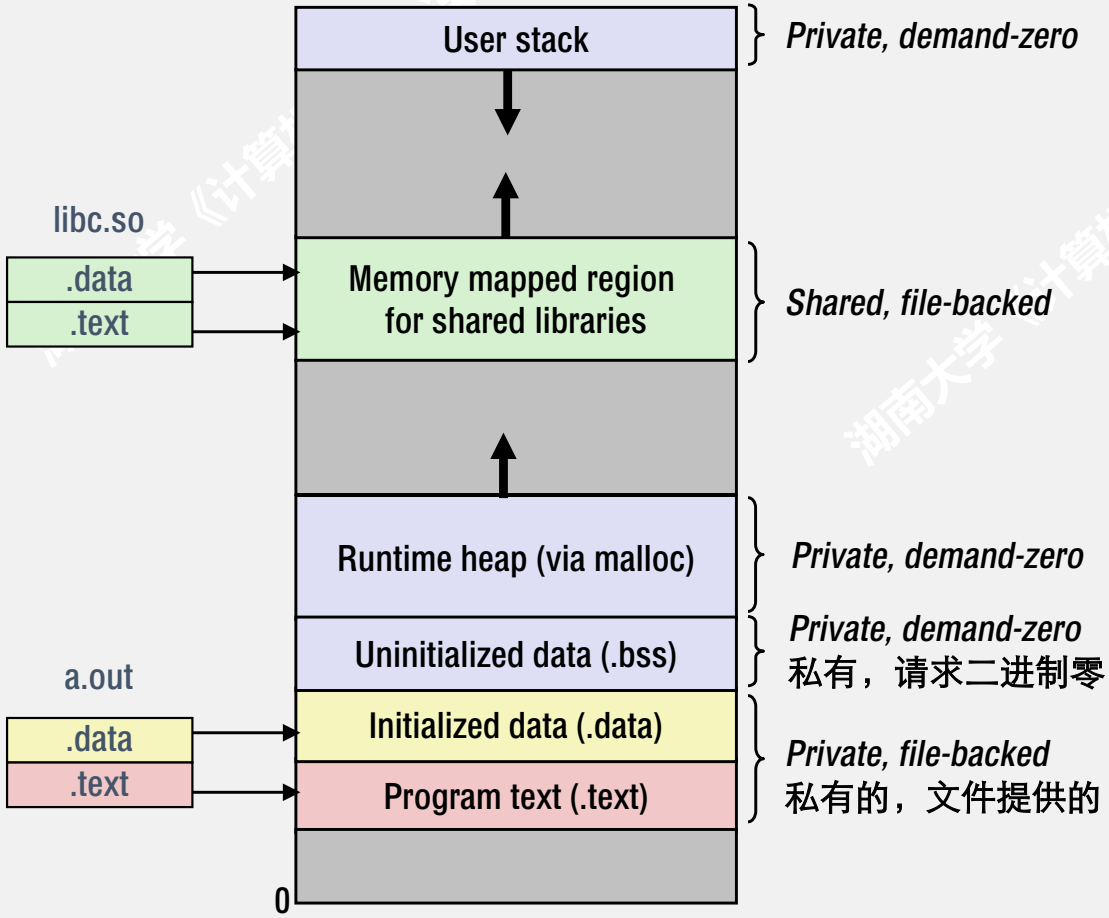


- 指令写入私有页面会触发保护故障。
- 处理程序创建新的R/W页面。
- 指令在处理程序返回时重新执行
- 尽量推迟复制动作！

再谈fork函数

- **虚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**
  - 创建父页表的精确副本（只复制父进程的mm_struct和area_struct以及页表）
    - 将两个进程（父进程和子进程）中的每个页面标记为只读
    - 将两个进程中的可写区域(area_struct)标记为私有COW
- **返回时，每个进程都具有虚拟内存的确定副本**
- **后续写入使用COW机制创建新的物理页面**

再谈execve函数



- **使用execve在当前进程中加载和运行新程序a.out：**
- **释放旧区域所有的vm_area_struct和页表**
- **为新区域创建vm_area_struct和页表**
  - 程序和初始化数据映射到目标文件
  - .bss、堆和栈映射到匿名文件
- **将PC设置为.text中的入口点**
  - Linux将根据需要在代码和数据页中出现页故障（缺页）

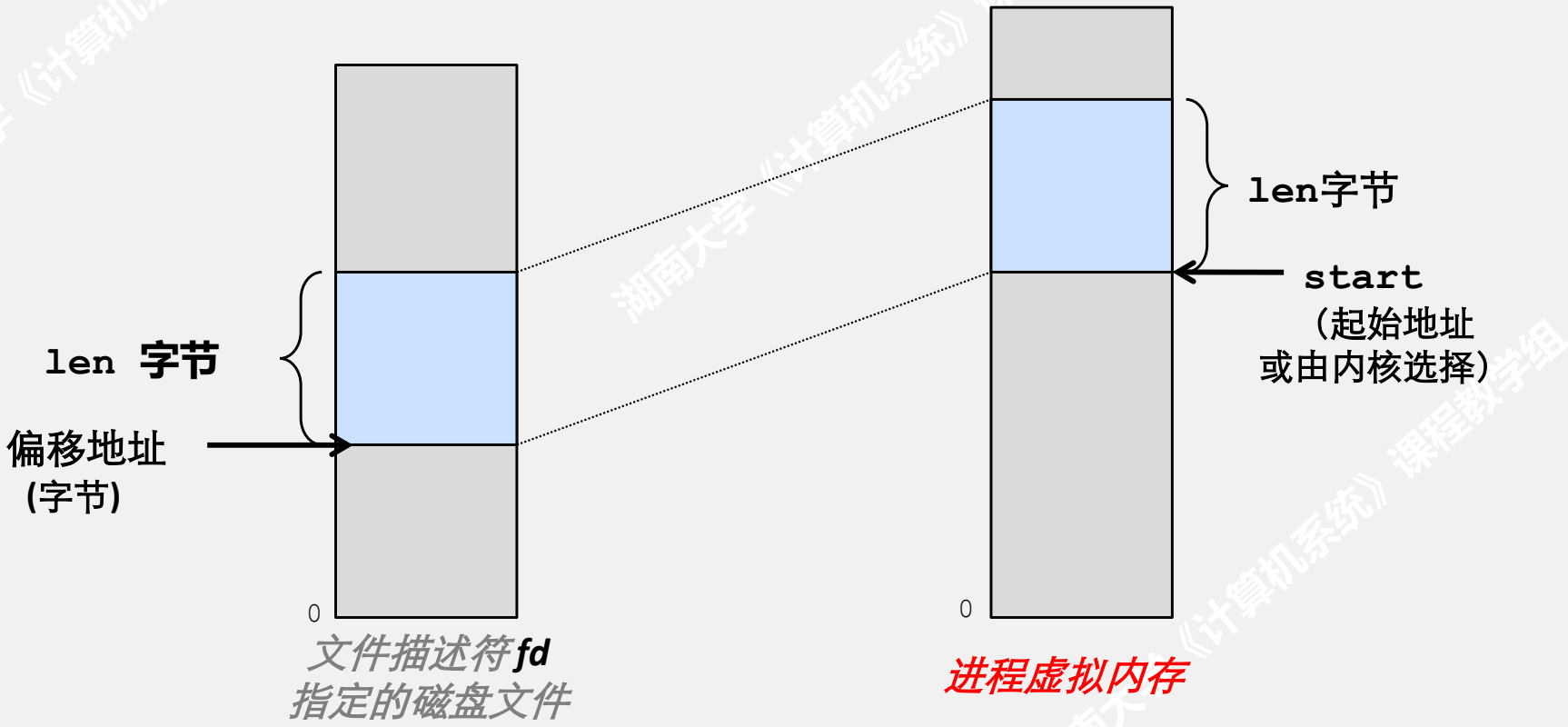
用户级内存映射

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

- **映射len个字节，从文件描述fd指定的文件的offset偏移处开始，最好是在地址start处（如果该处空间未使用）**
  - start:对于“选择地址”可能为0，通常定义为NULL
  - prot: PROT_READ, PROT_WRITE, ...
  - flags: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- **返回一个指向映射区域开始的指针（可能不是起始位置）**

用户级内存映射

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



使用mmap复制文件

将文件**复制到标准输出**而无需将数据传输到用户空间。

```c
#include "csapp.h"
/* mmapcopy - uses mmap to copy file fd
 *            to stdout
 */
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

本讲小结

- 为什么需要虚拟存储器
- 虚存带来的地址空间与地址转换问题
- 一个重要例子
- 内存映射

计算机系统

下一讲 预告

动态存储器分配

湖南大学

《计算机系统》课程教学组