计算机系统

虚拟存储器

概念

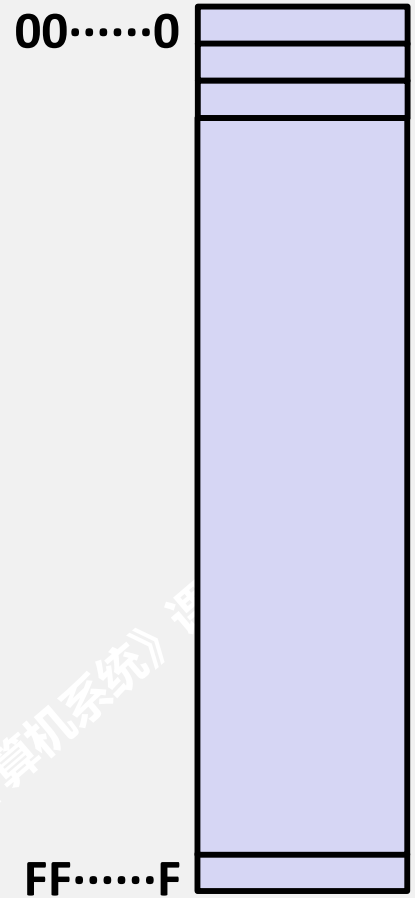
湖南大学
《计算机系统》课程教学组
肖雄仁

本讲学习内容

- **地址空间**
- 虚存作为缓存工具
- 虚存作为内存管理工具
- 虚存作为内存保护工具
- 地址转换

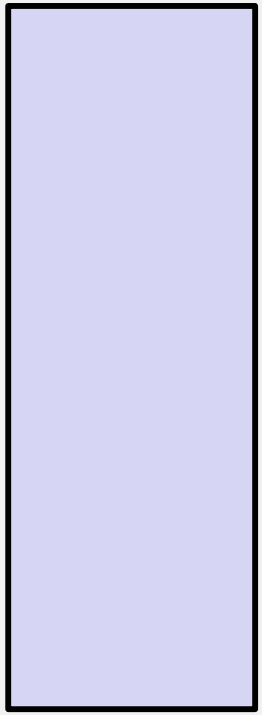
虚存抽象

- 程序实际引用的是虚拟存储器地址
  - movl (%ecx),%eax
  - 概念上是很大的字节数组
  - 每个字节有自己的地址
  - 以多种类型的存储层次共同实现
  - 系统为“进程”提供私有的地址空间
- 分配：编译器和运行时系统
  - 不同的程序对象应该存储在哪里
  - 在单一虚拟地址空间进行分配
- 为什么要使用虚拟内存，而不是直接使用物理内存？



Q1：满足所有进程需要?

64-bit 地址:
16 Exabyte



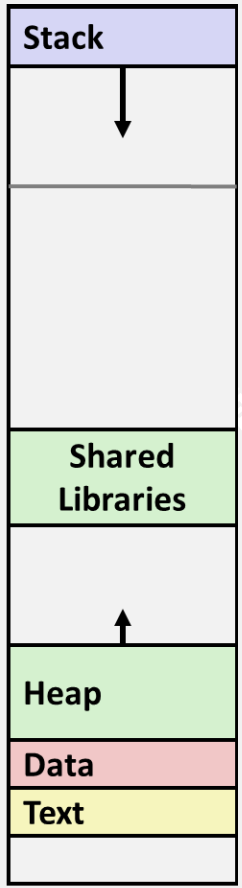
物理主存:
Few Gigabytes

有很多进程在运行...

Q2：内存如何管理?

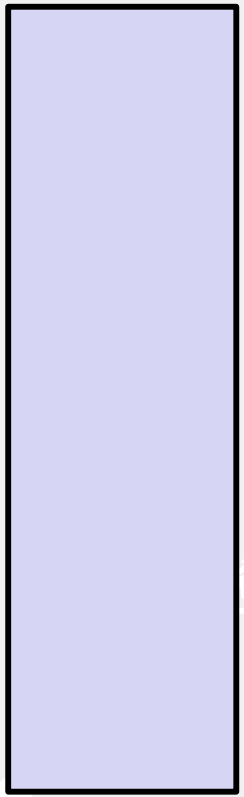
Process 1
Process 2
Process 3
…
Process n

X



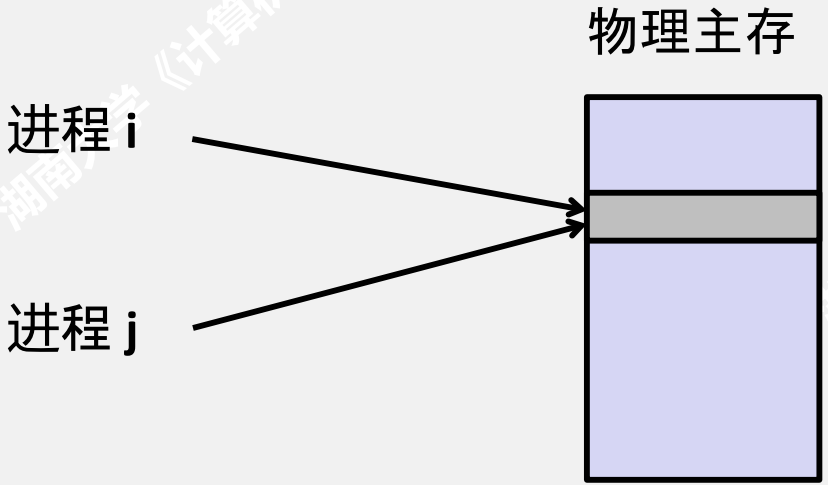
What goes where?

物理主存

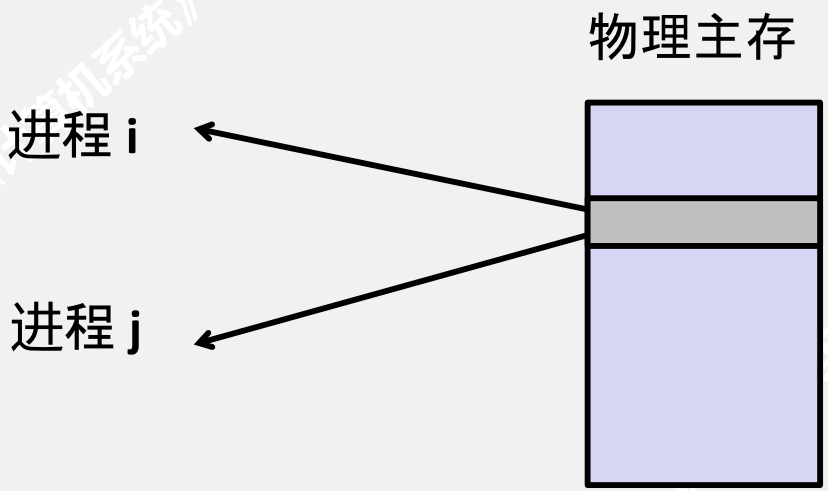


Q3：如何保护与共享?

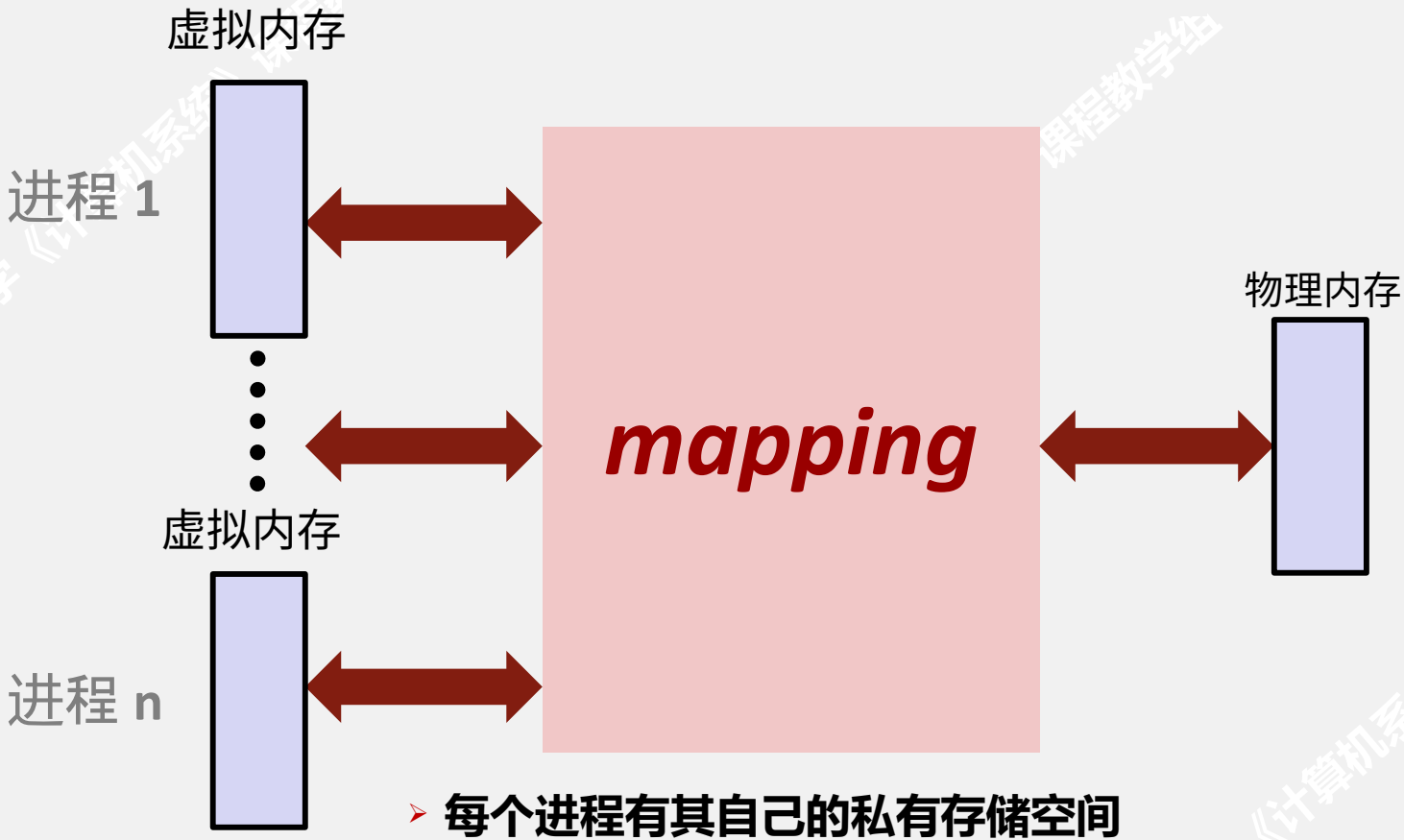
- 如何保护



- 如何共享



解决方案：增加一个间接层

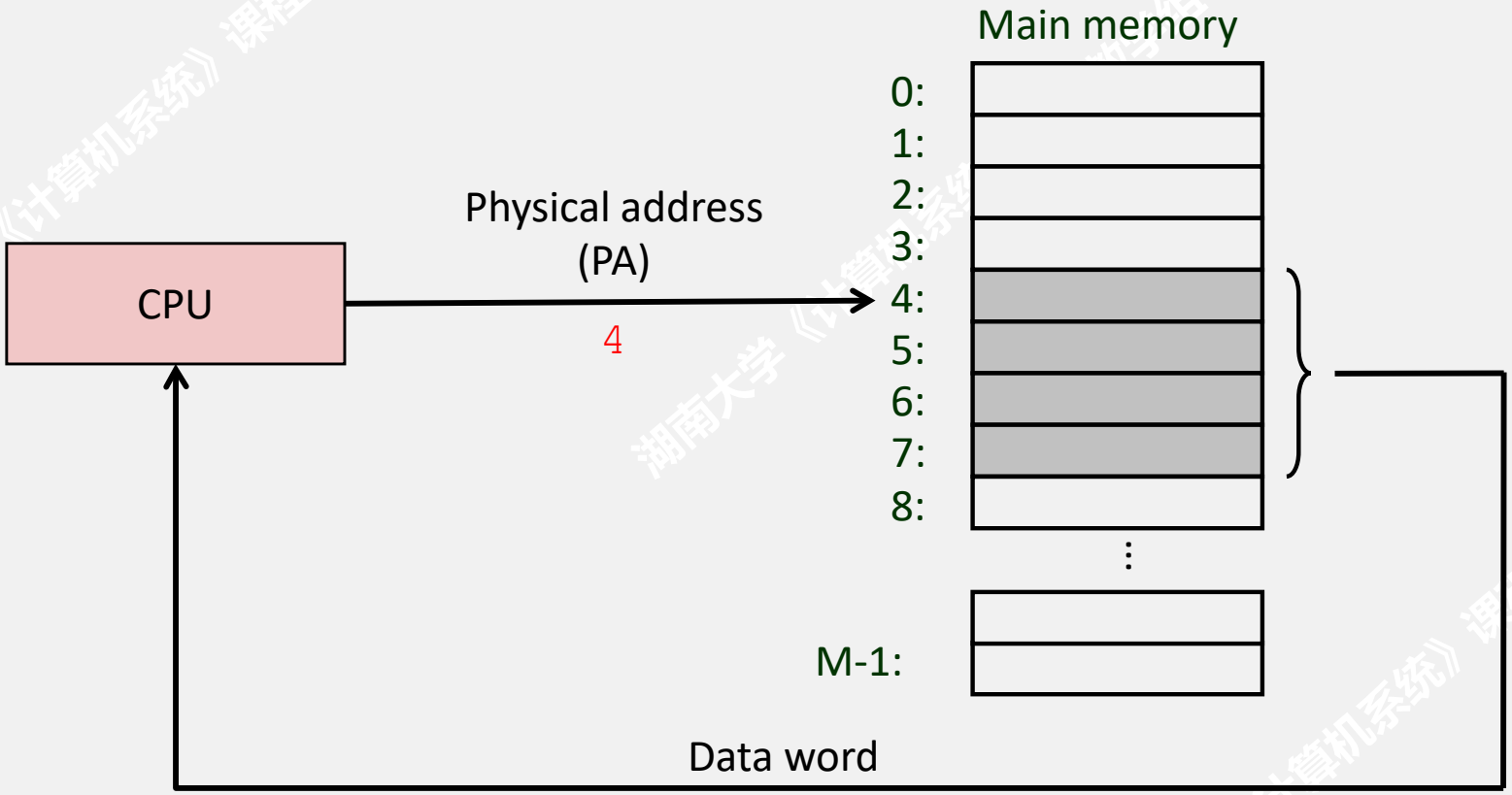


- 每个进程有其自己的私有存储空间
- 之前提及的问题均得以解决……

一个简单的映射解决所有三个问题

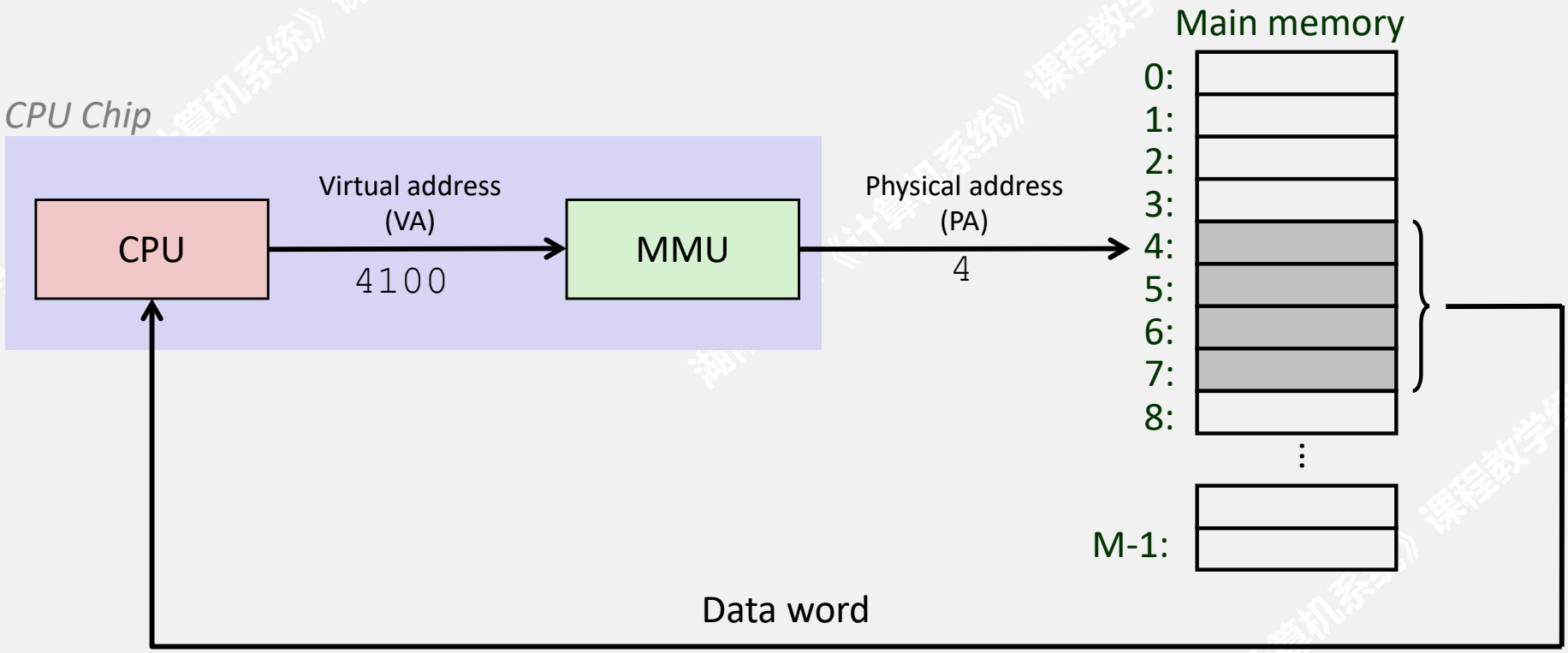
- 每个进程都有自己的私有内存映像
  - 似乎是一个完整大小的私有内存空间
- 这解决了
  - “如何选择存储位置”
  - “其他进程之间的存储空间不互相影响”
  - 使得每个进程都按需获得所需存储资源
- 实现：透明地转换地址
  - 添加映射函数，将私有地址映射到物理地址
  - 在每次加载或存储时进行映射
- 这个映射技巧是虚拟内存的核心
  - 程序实际引用的是虚存地址

使用物理寻址的系统



- 例如 汽车、电梯、数字相框等“简单”系统中的嵌入式微控制器

使用虚拟寻址的系统



- 例如：现代服务器、桌上电脑、笔记本电脑

地址空间

- 线性地址空间 : 连续的非负整数地址空间

  $\{0, 1, 2, 3 \ldots\}$

- 虚拟地址空间 : $N = 2^n$ 虚拟地址

  $\{0, 1, 2, 3, \ldots, N\text{-}1\}$

- 物理地址空间 : $M = 2^m$ 物理地址

  $\{0, 1, 2, 3, \ldots, M\text{-}1\}$

- 明确区分**数据 (字节)** 与 其**属性 (地址)**
- 每个对象有多重地址
- 内存的每个字节 具有:

  一个物理地址，一个或多个虚拟地址

为什么使用虚拟内存（总结）

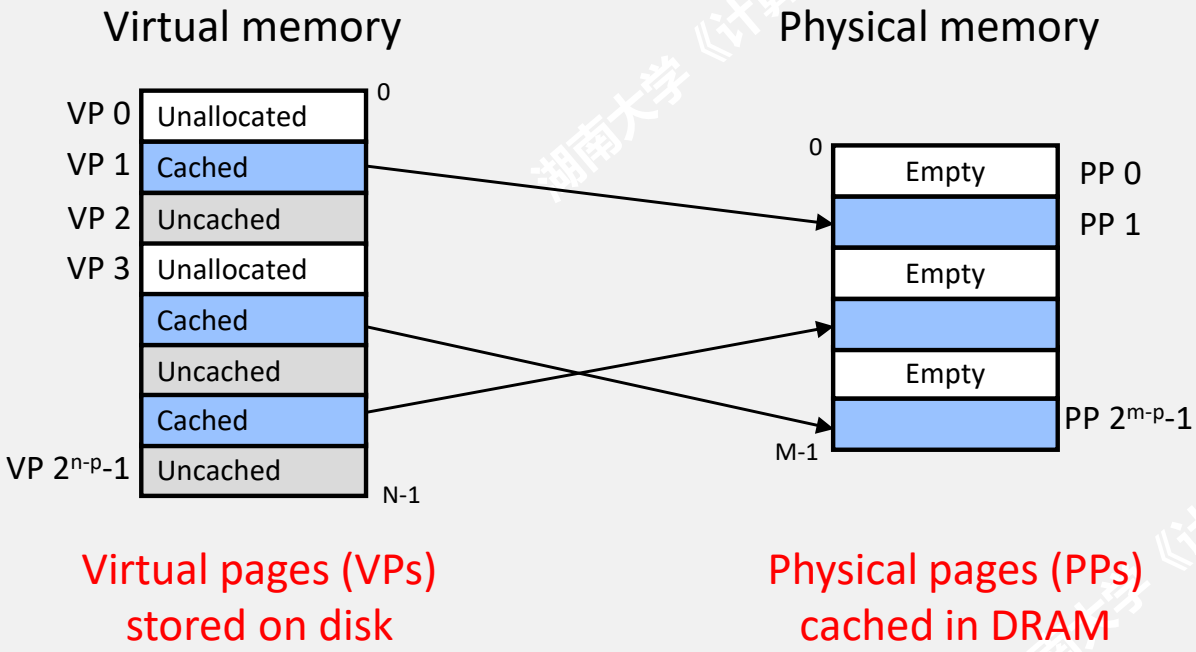
- **高效地使用主内存（效率）**
  - 使用DRAM作为虚拟地址空间部分的缓存
- **简化内存管理（简化）**
  - 每个进程获得相同的统一线性地址空间
- **隔离地址空间（隔离）**
  - 一个进程不能干涉另一个进程的存储器
  - 用户程序无法访问特权内核信息

本讲学习内容

- 地址空间
- **虚存作为缓存工具**
- 虚存作为内存管理工具
- 虚存作为内存保护工具
- 地址转换

虚存作为缓存工具

- 虚拟内存是存储在磁盘上的N个连续字节的数组
- 磁盘上的内容缓存在物理内存（DRAM缓存）中
  - 这些缓存块称为页面（大小为 $P = 2^p$ 字节）

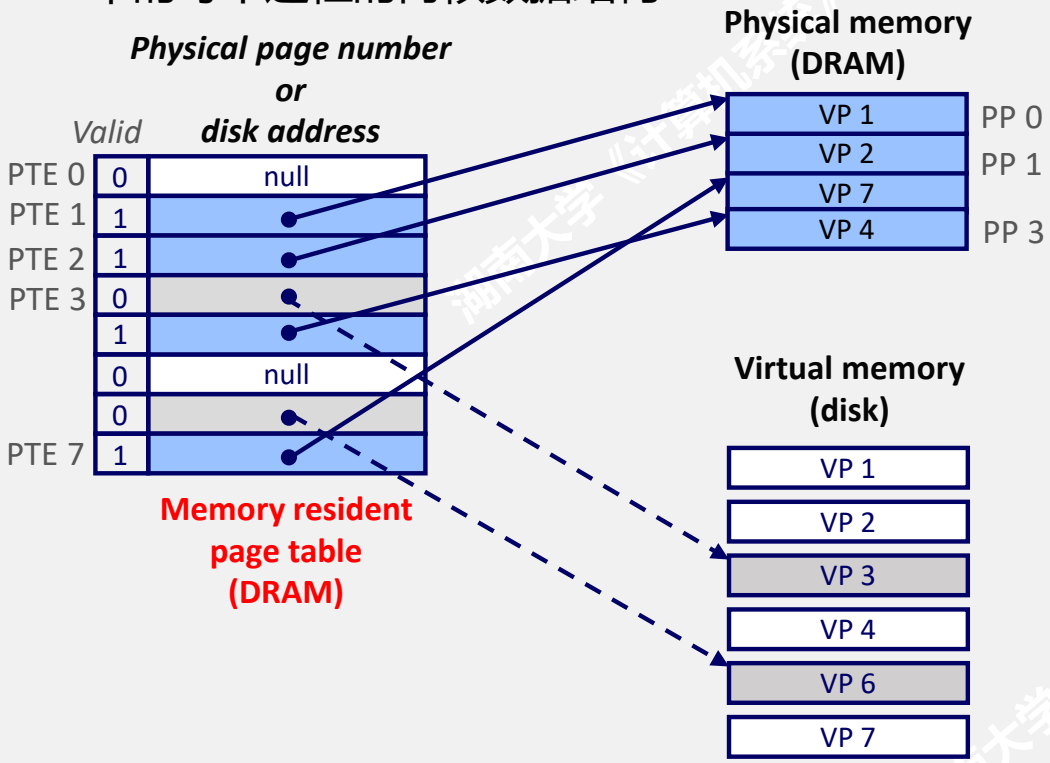


DRAM缓存的组织结构

- DRAM缓存的组织结构源于未命中引起的巨大代价
  - DRAM比SRAM慢大约10倍
  - 磁盘比DRAM慢大约10,000倍
- 结论
  - 大页面（块）大小：通常为4-8 KB，甚至达4 MB
  - 全相联
    - 任何虚存页VP都可以放在任何物理页PP中
    - 需要“大”映射函数 - 与CPU缓存不同
  - 高度复杂，昂贵的替换算法
    - 更为复杂精密，超出我们的讨论范畴
  - 写回 而非 直写

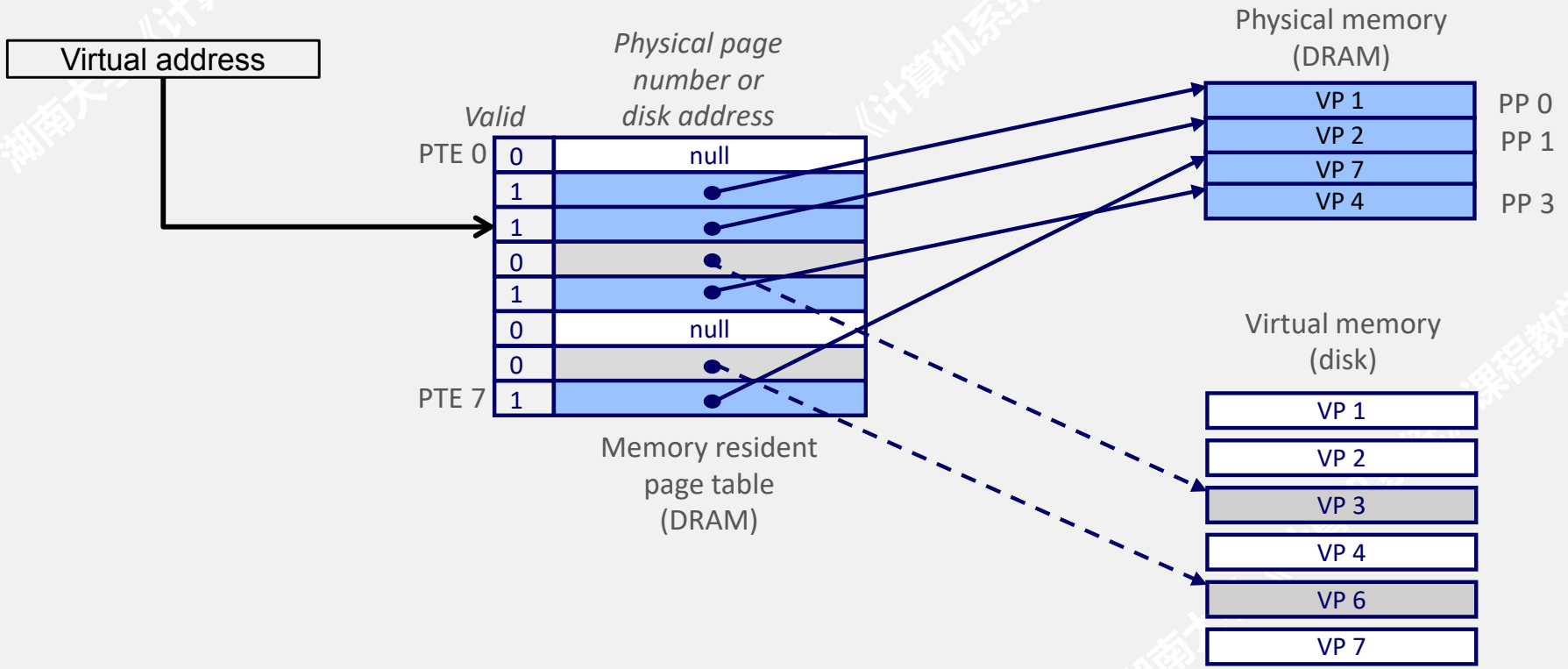
页表(Page Tables)

- **页表是将虚拟页映射到物理页的页表项(page table entries,PTEs)的数组。**
  - 存放在DRAM中的每个进程的内核数据结构



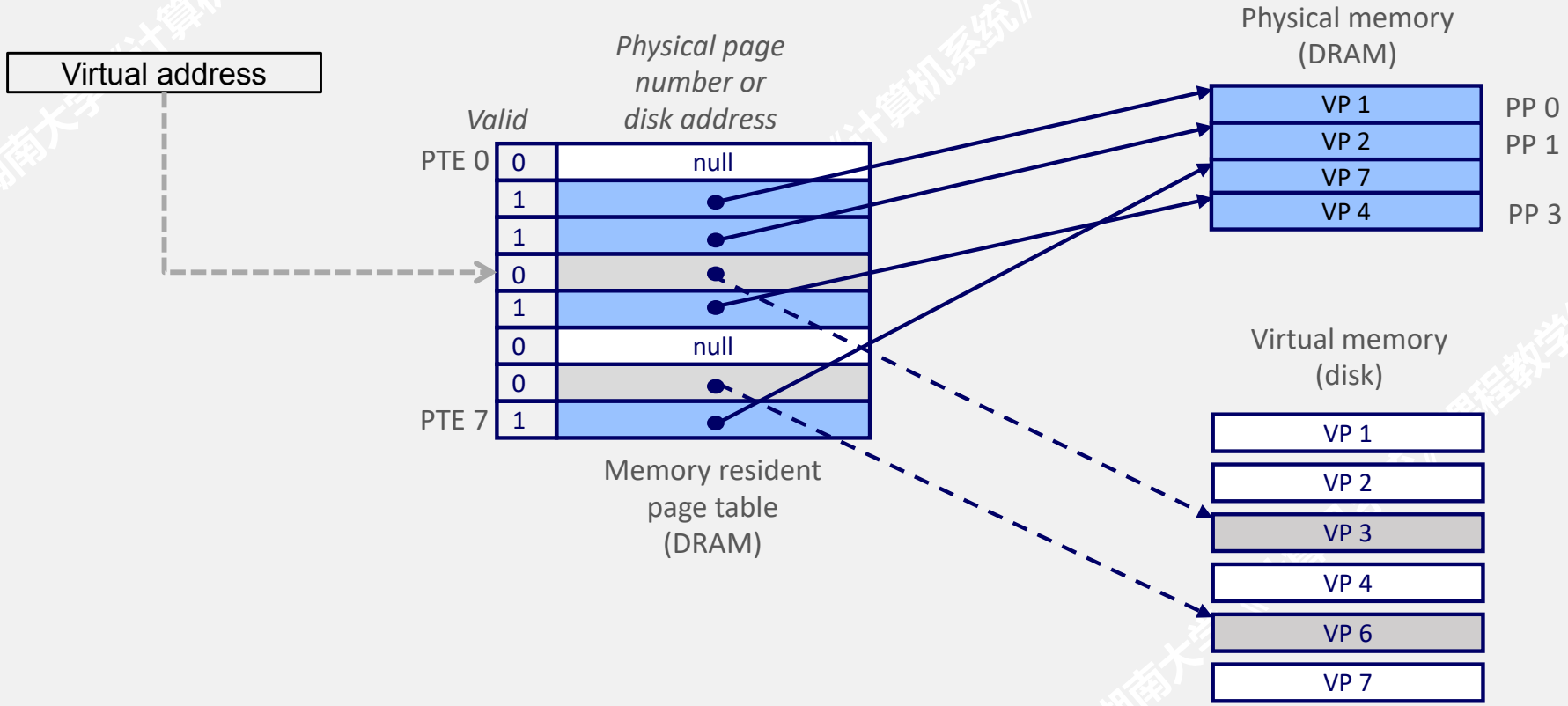
页命中(Page Hit)

- Page hit: 引用的虚存字就在物理主存中（DRAM缓存命中）



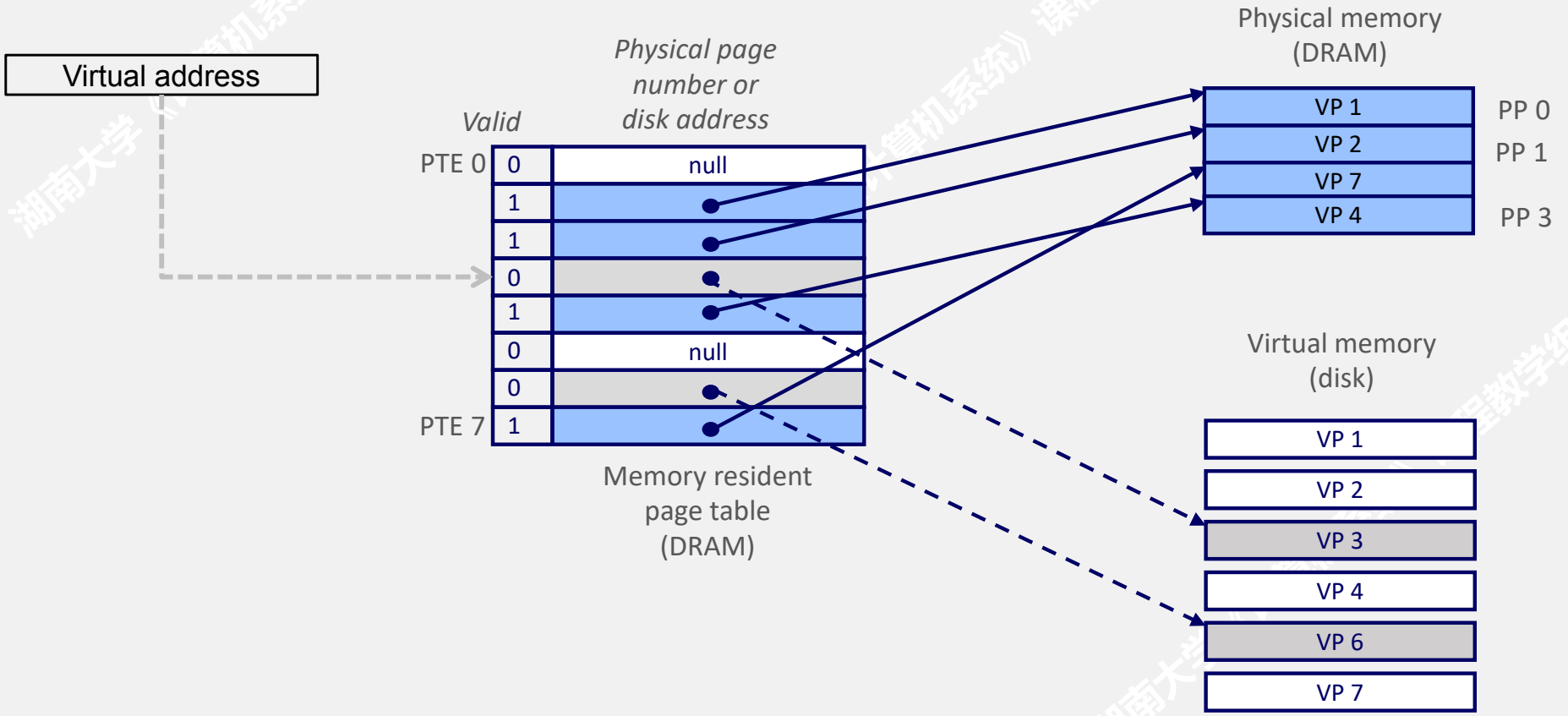
页不命中(Page Fault)

- Page fault: 引用的虚存字不在物理存储器中 (DRAM缓存不命中)



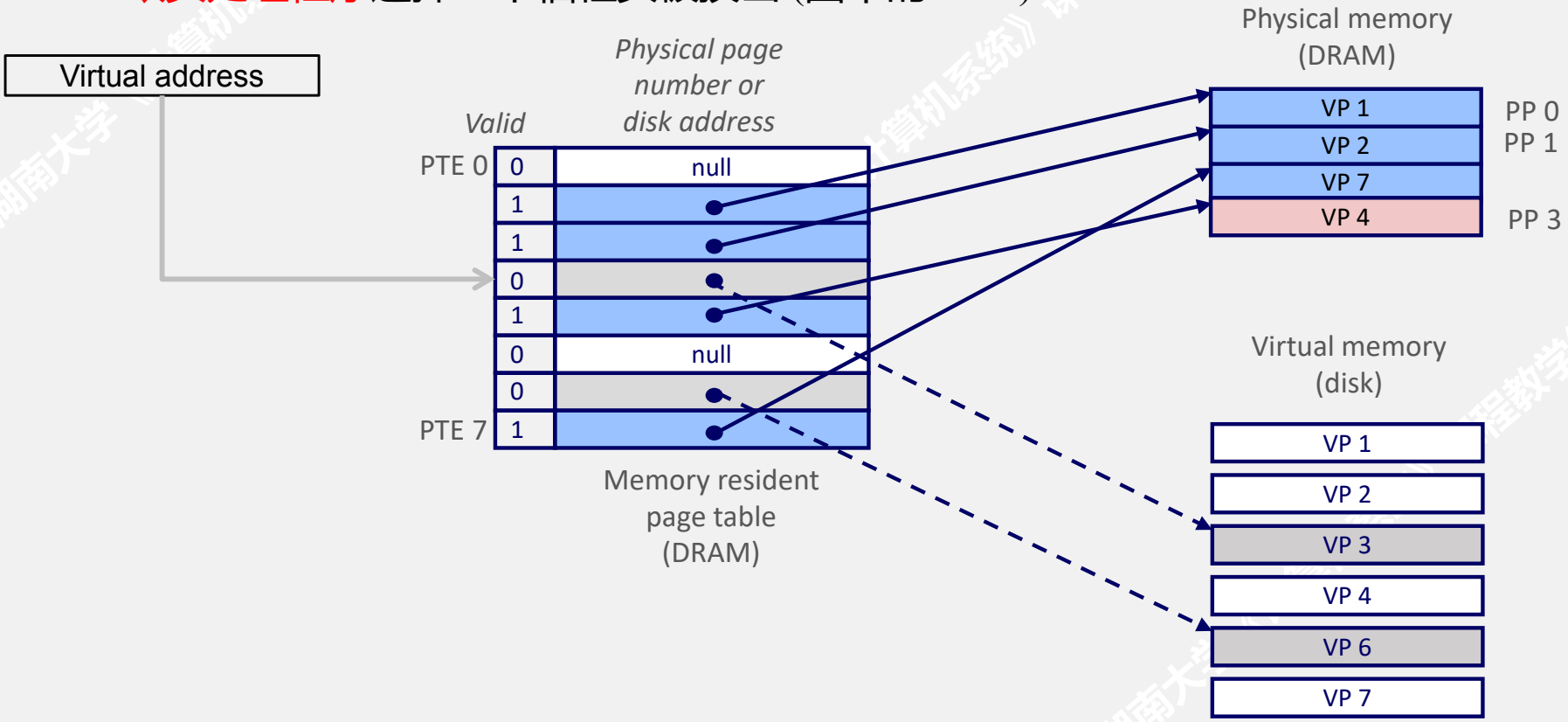
页故障的处理

- 页不命中引发页故障（一种异常），硬件MMU触发异常



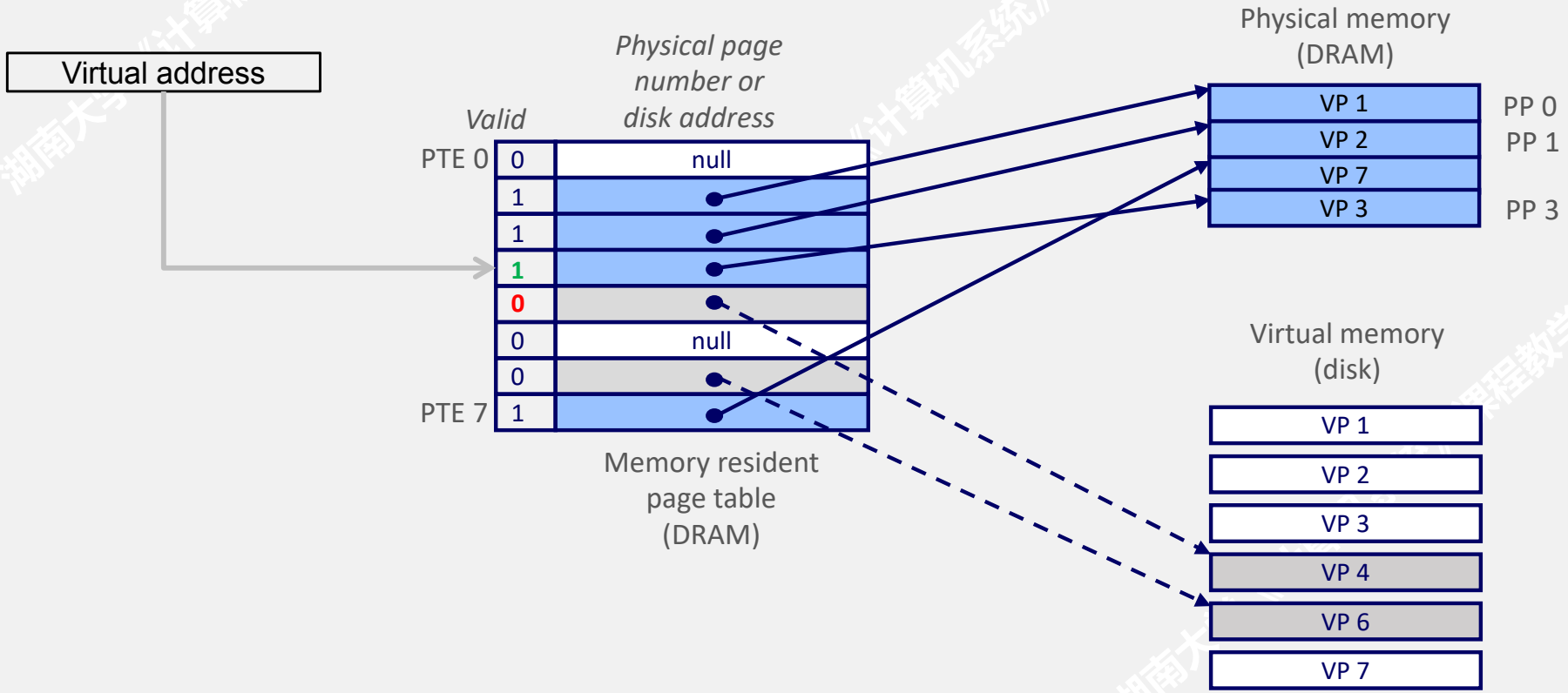
页故障的处理

- 页不命中引发页故障（一种异常）
- 缺页处理程序选择一个牺牲页被换出 (图中的 VP 4)



页故障的处理

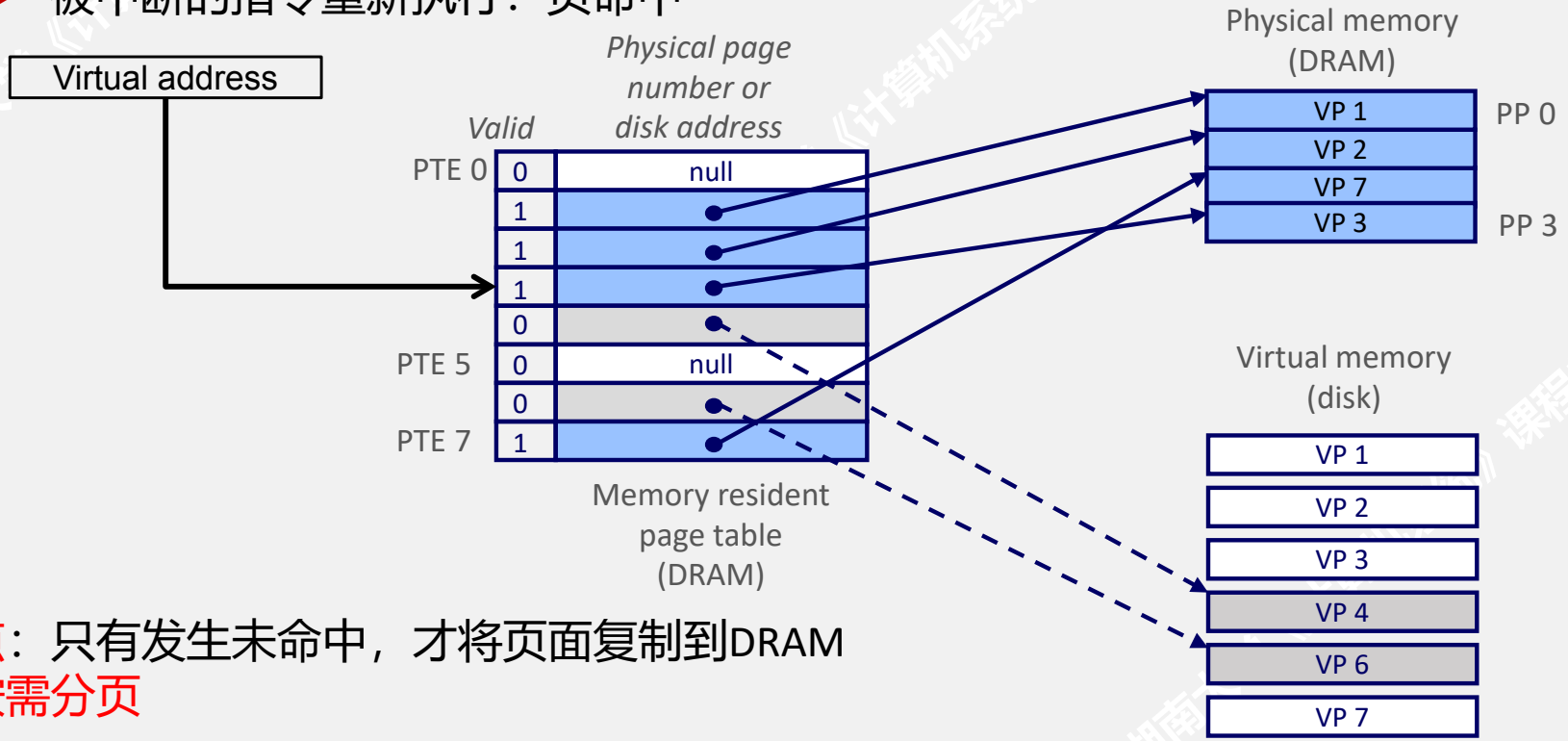
- 页不命中引发页故障（一种异常）
- 缺页处理程序选择一个牺牲页被换出 (图中的 VP 4)



页故障的处理

- 页不命中引发页故障（一种异常）
- 缺页处理程序选择一个牺牲页被换出 (图中的 VP 4)
- 被中断的指令重新执行：页命中

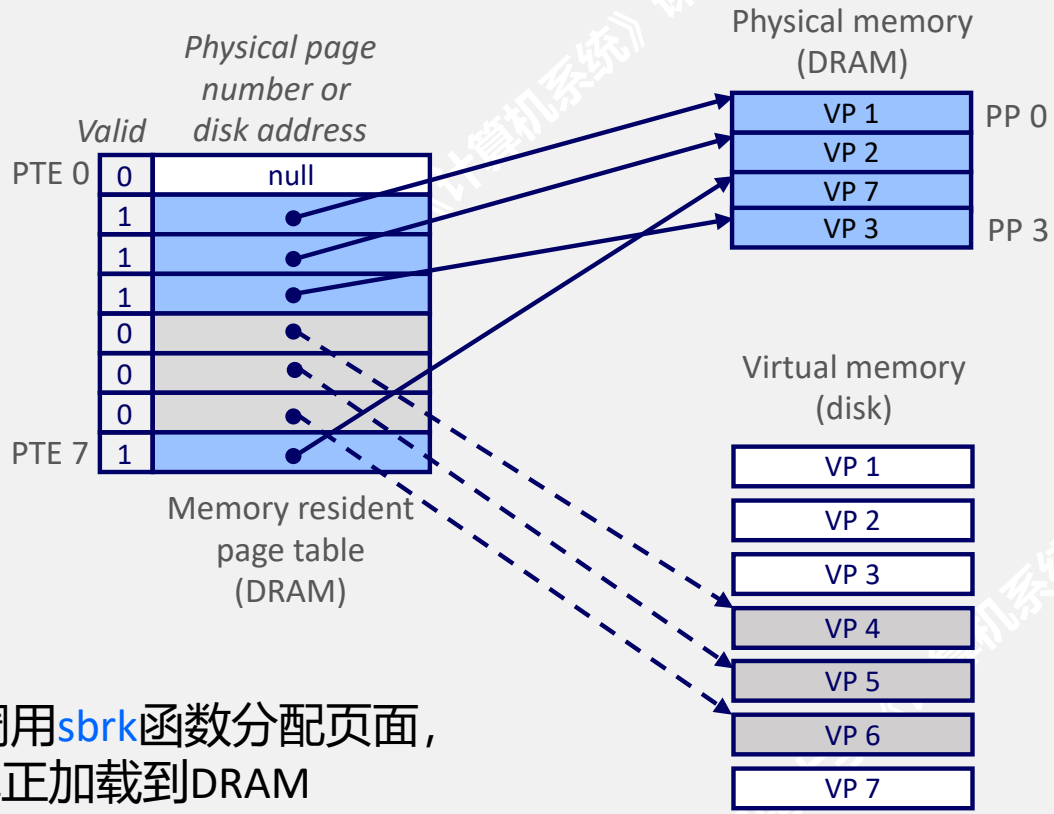
对页表操作



关键点：只有发生未命中，才将页面复制到DRAM
称为按需分页

分配页面

- 分配虚拟内存的新页面（VP 5）



malloc分配虚拟地址空间：调用sbrk函数分配页面，只记录位置，修改PTE，不真正加载到DRAM

局部性再次救援！

- **虚拟内存似乎效率极低，但是由于局部原因，它可以工作。**
- **在任何时间点，程序都倾向于访问称为工作集的一组活动虚拟页面。**
  - ▷ 时间局部性更好的程序将具有较小的工作集
- **如果（工作集大小 < 主内存大小）**
  - ▷ 在强制不命中后，进程展示良好性能
- **如果（SUM（工作集大小）> 主内存大小）**
  - ▷ 颠簸（Thrashing）：页被不断换进换出，性能极差

回忆：异常 中的“页故障”

- “页故障”何时发现？如何发现？谁来发现？
  - 执行每条指令都要访存（取指令、取操作数、存结果）
  - （在保护模式下）每次访存都要进行逻辑地址向物理地址转换
  - 在地址转换过程中会发现是否发生了“页故障”！
  - 逻辑地址向物理地址的转换由硬件（MMU）实现，故“页故障”事件由硬件发现
  - 所有异常和中断事件都由硬件检测发现！

异常 中的“页故障”

- **以下几种情况都会发生“页故障”**
  - 缺页：页表项有效位为0 ← **可通过读硬盘恢复故障**
  - 地址越界：地址大于最大界限
  - 访问越级或越权（保护违例）：

  } **不可恢复，称为“段故障（segmentation fault）”**

    - 越级：用户进程访问内核数据（CPL=3 / DPL=0）
    - 越权：读写权限不相符（如对只读段进行了写操作）

一个例子

假设在IA-32/linux系统中一个C语言源程序P 如下:

```
1 int a[1000];
2 int x;
3 main( )
4 {
5    a[10]=1;
6    a[1000]=3;
7    a[10000]=4;
8 }
```

正常的控制流为

…、0x8048300、 0x804830a、 0x8048314、…

可能的异常控制流是什么?

假设编译、汇编和链接后，第5、 6和7行源代码对应的指令序列如下:

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   804830a: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048314: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则:

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的? 哪些问题是不可恢复的?

一个例子

假设在IA-32/linux系统中一个C语言源程序P 如下:

```
1 int a[1000];
2 int x;
3 main( )
4 {
5    a[10]=1;
6    a[1000]=3;
7    a[10000]=4;
8 }
```

三条指令在取指令时都不会发生缺页，Why?

它们都位于起始地址为0x08048000（是一个4KB页面的起始位置）的同一个页面，执行这三条指令之前，该页已经调入内存。因为没有其他进程在系统中运行，所以不会因为执行其他进程而使得调入主存的页面被调出到磁盘。因而都不会在取指令时发生页故障。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下:

```
5   8048300: c7 05 28 90 04 08 01 00 00 00  movl   $0x1, 0x8049028
6   804830a: c7 05 a0 9f 04 08 03 00 00 00  movl   $0x3, 0x8049fa0
7   8048314: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则:

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的? 哪些问题是不可恢复的?

一个例子

假设在IA-32/linux系统中一个C语言源程序P 如下:

```
1 int a[1000];
2 int x;
3 main( )
4 {
5    a[10]=1;
6    a[1000]=3;
7    a[10000]=4;
8 }
```

第5行指令取数据时是否发生页故障，Why?

对a[10]（地址0x8049028）的访问是对所在页面（首址为0x08049000）的第一次访问，故不在主存，缺页处理结束后，再回到这条movl指令重新执行，再访问数据就没有问题了。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下:

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   804830a: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048314: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则:

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的? 哪些问题是不可恢复的?

一个例子

假设在IA-32/linux系统中一个C语言源程序P 如下:

```
1 int a[1000];
2 int x;
3 main( )
4 {
5    a[10]=1;
6    a[1000]=3;
7    a[10000]=4;
8 }
```

第6行指令取数据时是否发生页故障，Why?

对a[1000]（地址0x8049fa0）的访问是对所在页面（首址为0x08049000）的第2次访问，故在主存，不会发生缺页。但a[1000]实际不存在，只不过编译器未检查数组边界，0x8049fa0处可能是x的地址，故该指令执行结果是x被赋值为3

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下:

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   804830a: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048314: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则:

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的? 哪些问题是不可恢复的?

一个例子

假设在IA-32/linux系统中一个C语言源程序P 如下:

```
1 int a[1000];
2 int x;
3 main( )
4 {
5    a[10]=1;
6    a[1000]=3;
7    a[10000]=4;
8 }
```

第7行指令取数据时是否发生页故障，Why?

地址0x8052c40偏离数组首址0x8049000已达4×10000=40000个单元，即偏离了9个页面，很可能超出可读写区范围，故执行该指令时可能会发生保护违例。页故障处理程序发送一个“段错误”信号（SIGSEGV）给用户进程，用户进程接受到该信号后就调出一个信号处理程序执行，该信号处理程序根据信号类型，在屏幕上显示“段故障（segmentation fault）”信息，并终止用户进程。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下:

```
5   8048300: c7 05 28 90 04 08 01 00 00 00  movl   $0x1, 0x8049028
6   804830a: c7 05 a0 9f 04 08 03 00 00 00  movl   $0x3, 0x8049fa0
7   8048314: c7 05 40 2c 05 08 04 00 00 00  movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则:

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

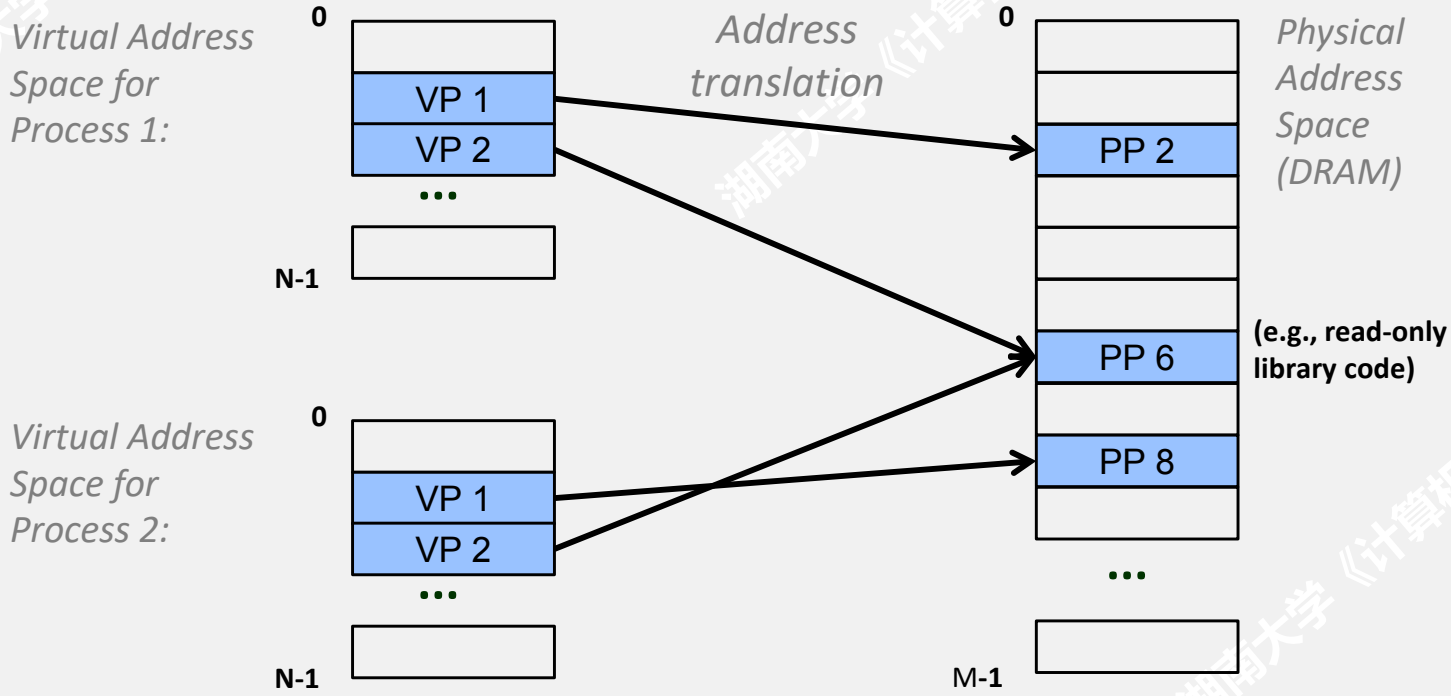
(3) 哪些问题是可恢复的? 哪些问题是不可恢复的?

本讲学习内容

- 地址空间
- 虚存作为缓存工具
- **虚存作为内存管理工具**
- 虚存作为内存保护工具
- 地址翻译

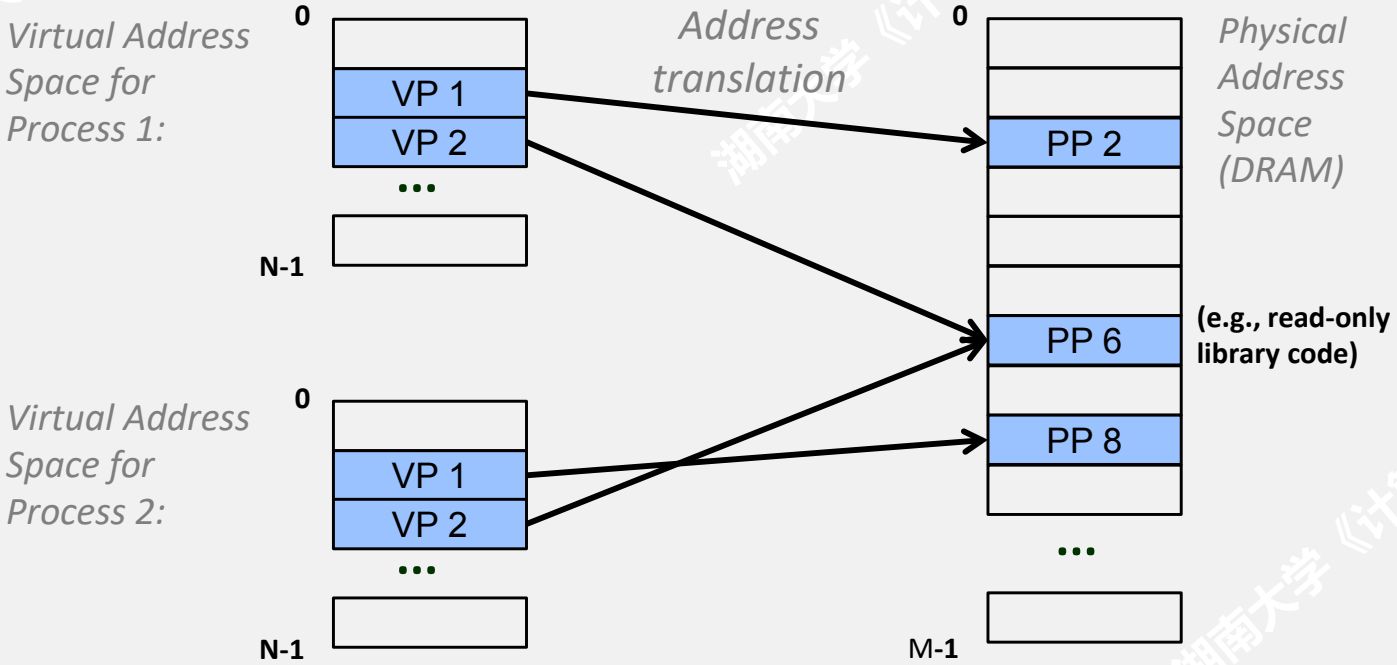
虚存作为内存管理工具

- 关键思想：每个进程都有自己的虚拟地址空间
  - 可以将内存视为简单的线性数组
  - 映射函数通过物理内存分散地址
    - 精心选择的映射可以改善局部性



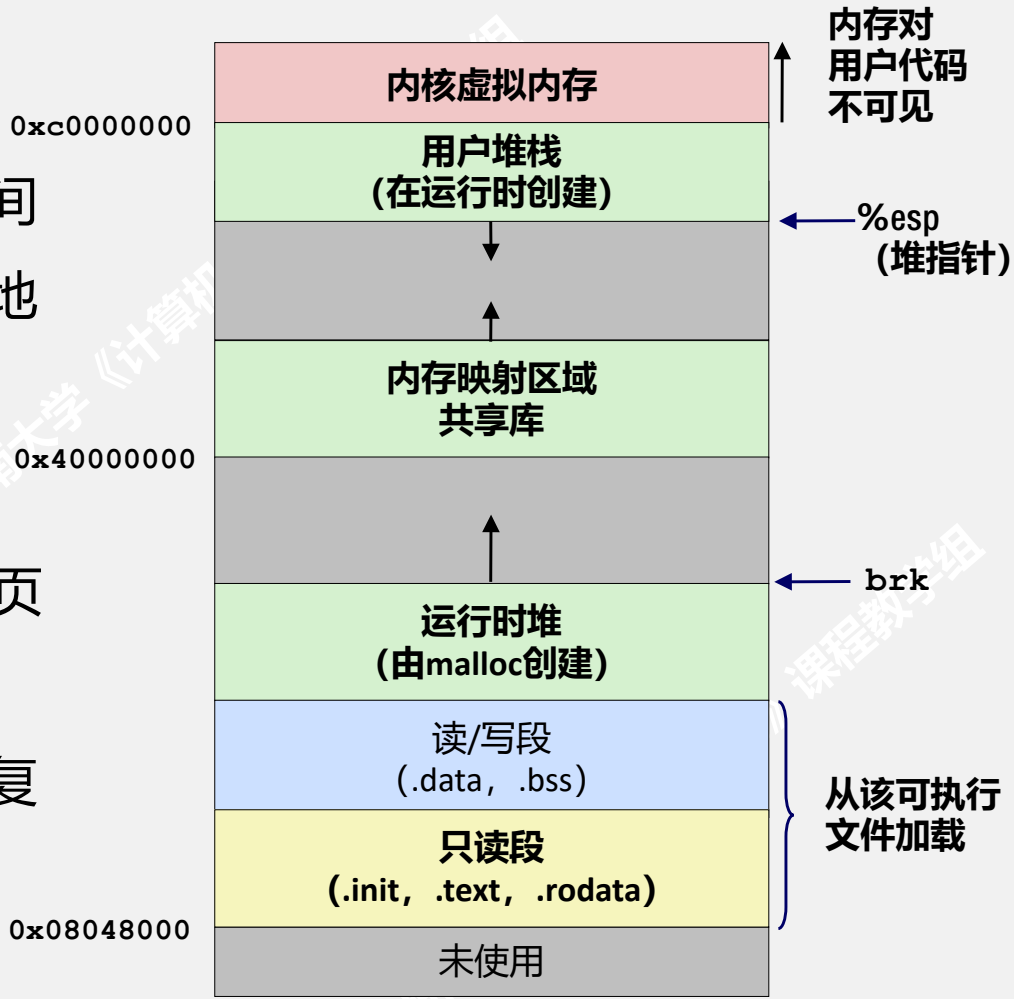
虚存作为内存管理工具

- 简化主存分配
  - 每一个虚存页映射到任意物理页面
  - 一个虚页在不同时刻存储于不同物理页
- 进程之间共享代码和数据
  - 映射多个虚拟页到相同物理页 (如下图示: PP 6)



简化链接与加载

- 链接
  - 每个程序都有类似的虚拟地址空间
  - 代码，堆栈和共享库始终从相同地址开始
- 加载
  - execve()为.text和.data段分配虚拟页面（创建标记为无效的PTE）
  - 根据虚拟内存系统的需求，逐页复制.text和.data段

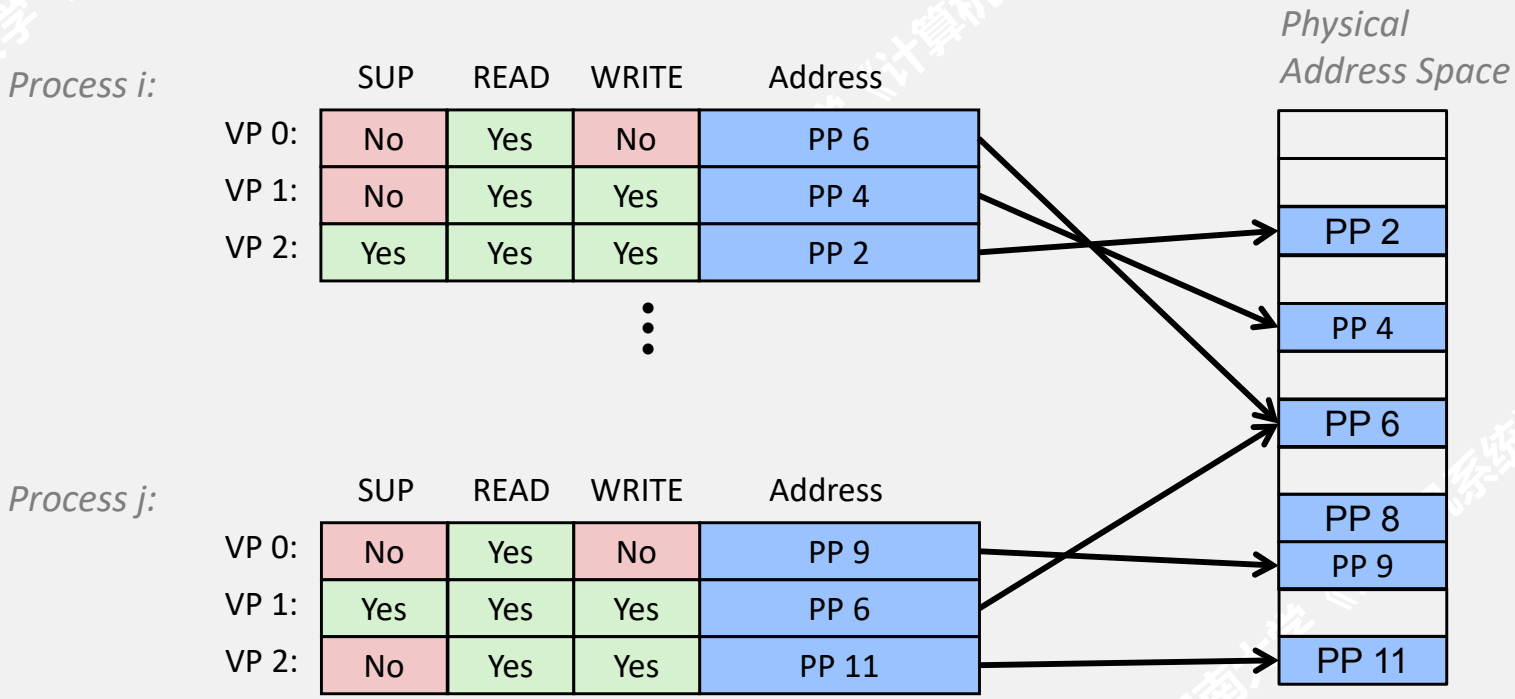


本讲学习内容

- 地址空间
- 虚存作为缓存工具
- 虚存作为内存管理工具
- **虚存作为内存保护工具**
- 地址翻译

虚存作为内存保护工具

- 以许可位（permission bits）扩展页表条目（PTE）
- 页故障处理程序在再映射前重新检查
  - 若违反许可条件向进程发送信号SIGSEGV (段故障)



本讲学习内容

- 地址空间
- 虚存作为缓存工具
- 虚存作为内存管理工具
- 虚存作为内存保护工具
- **地址翻译**

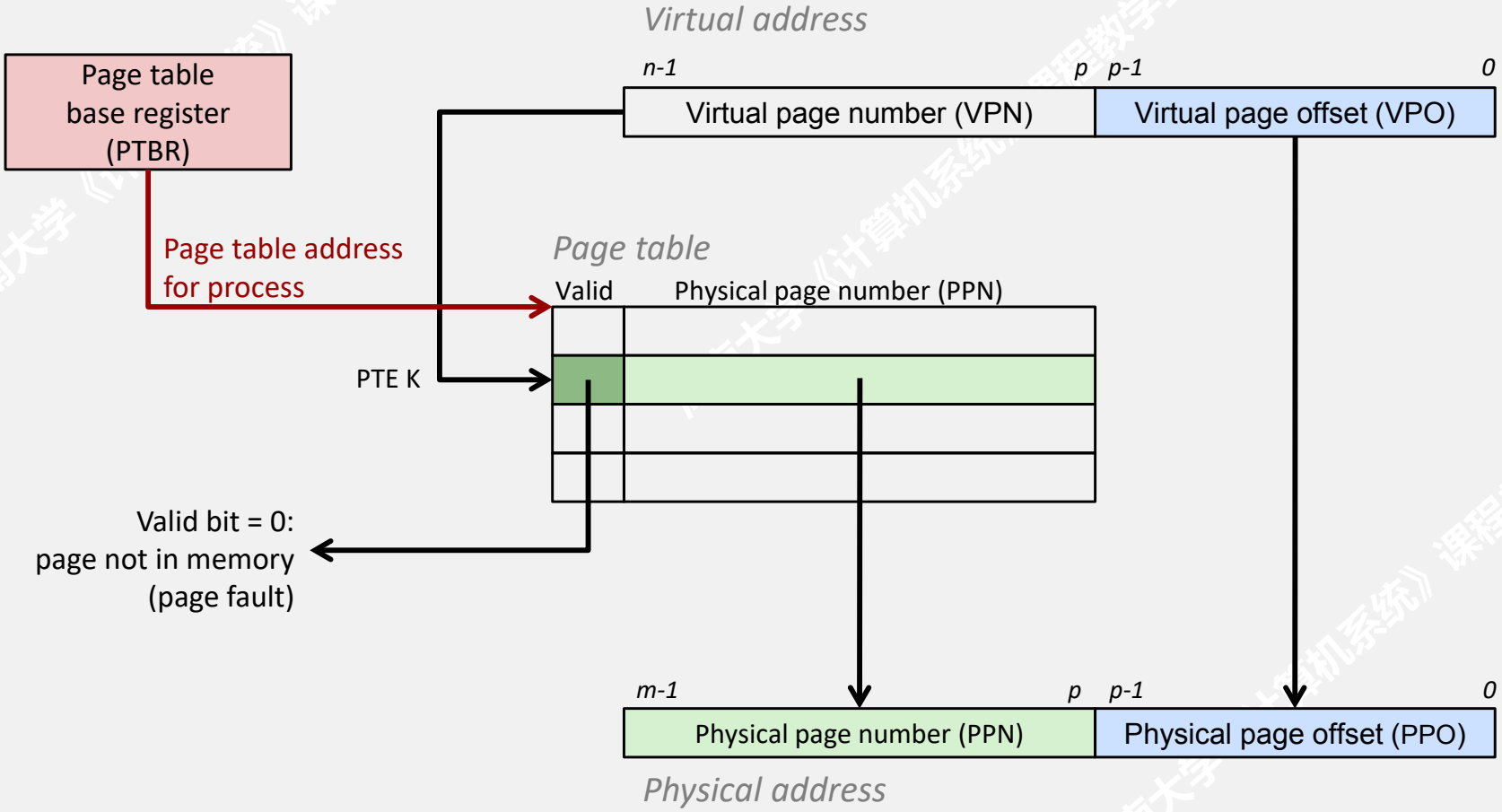
虚存地址翻译

- **虚拟地址空间**
  - $V = \{0, 1, \ldots, N-1\}$
- **物理地址空间**
  - $P = \{0, 1, \ldots, M-1\}$
- **地址翻译**
  - $MAP: V \rightarrow P \cup \{\varnothing\}$
  - 对于虚拟地址 a：
    - $MAP(a) = a'$ 如果虚拟地址 a 处的数据在 P 中的物理地址 a' 处
    - $MAP(a) = \varnothing$ *如果虚拟地址 a 处的数据不在物理内存中*
      - 无效或存储在磁盘上

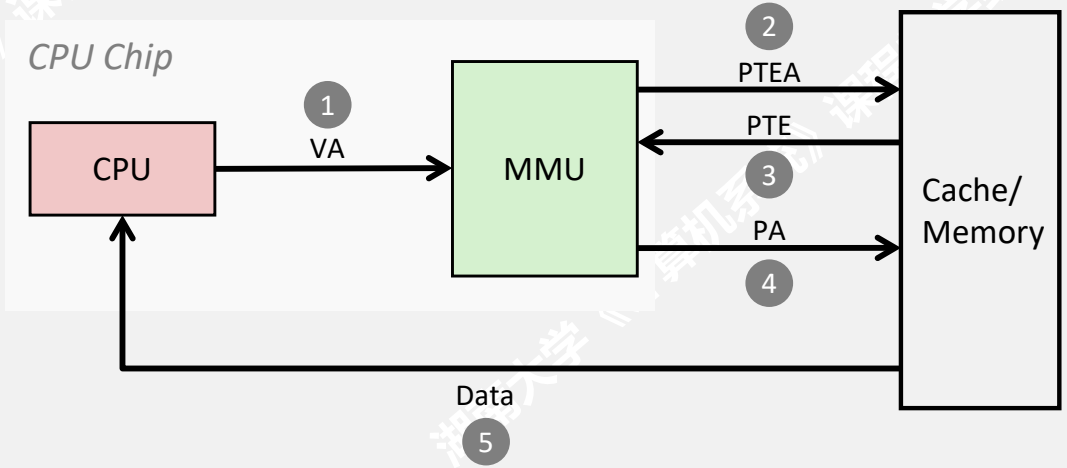
地址翻译符号汇总

- **基本参数**
  - **N = 2^n**：虚拟地址空间中的地址数量
  - **M = 2^m**：物理地址空间中的地址数量
  - **P = 2^p**：页面大小（字节）
- **虚拟地址（VA）的组件**
  - **VPO**：虚拟页偏移量（字节）
  - **VPN**：虚拟页号
- **物理地址（PA）的组件**
  - **PPO**：物理页面偏移量（与VPO相同）
  - **PPN**：物理页号

使用页表的地址翻译


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR)
Page table address for process
Page table
Valid
Physical page number (PPN)
PTE K
Valid bit = 0: page not in memory (page fault)
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

地址翻译：页命中



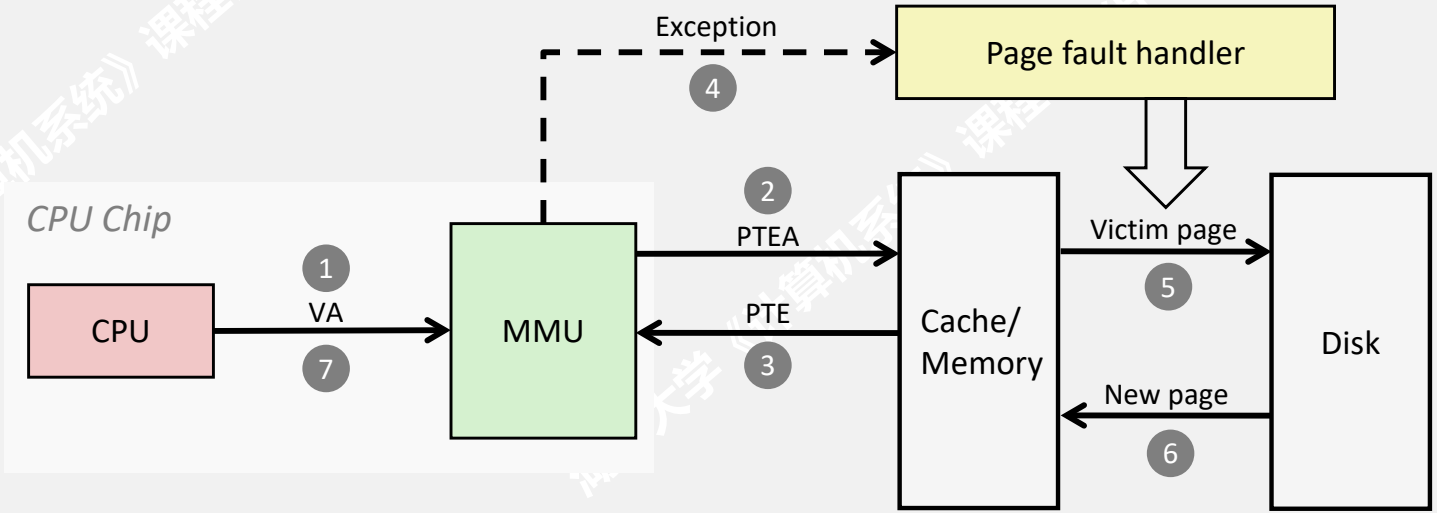
1) 处理器将虚拟地址发送到MMU

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

地址翻译：页故障



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

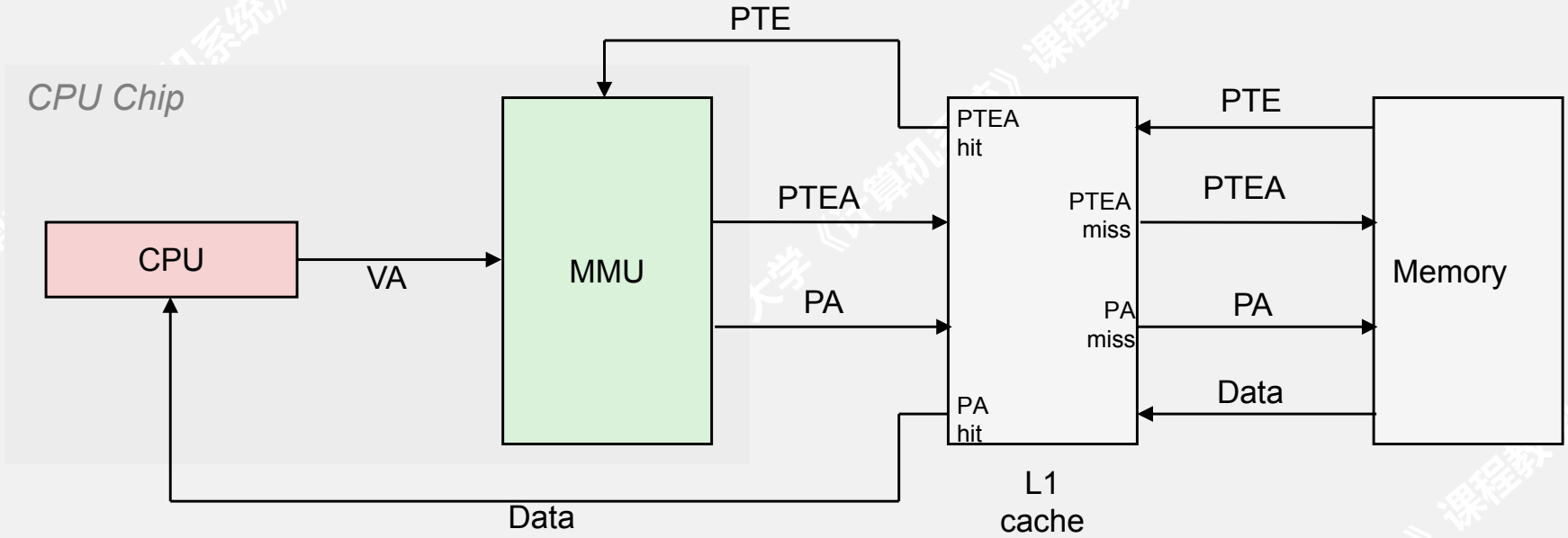
4) PTE有效位是零，所以MMU触发缺页异常

5) 缺页异常处理程序确定出物理存储页中的牺牲页（如果这个页面已经被修改，则把它页换出到磁盘）

6) 缺页异常处理程序调入新的页面，并更新内存中的PTE。

7) 缺页异常处理程序返回到原来的进程，再次执行导致缺页的指令

集成虚存与高速缓存



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

使用TLB 加速地址翻译

- 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

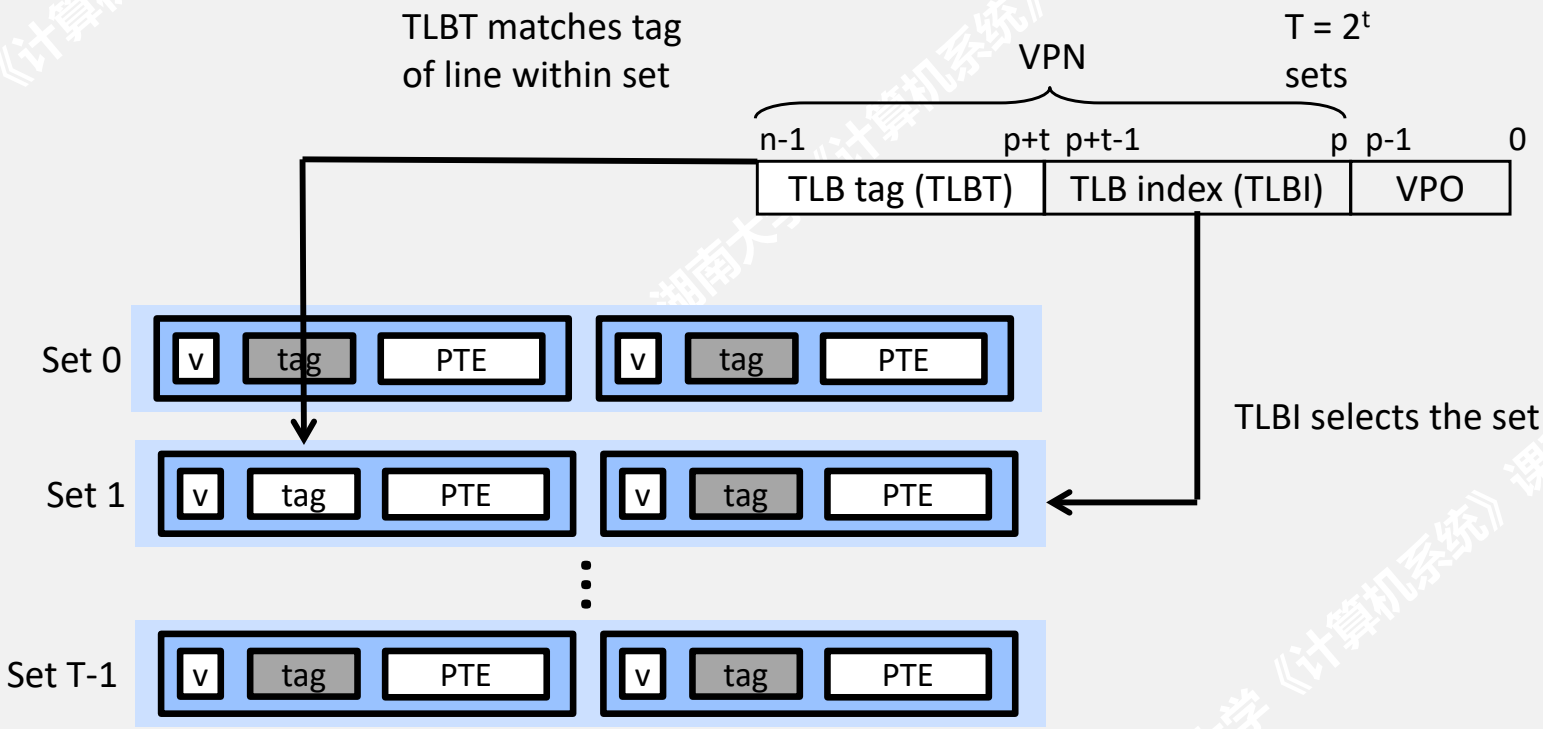
- 解决方案：Translation Lookaside Buffer（TLB）
  - MMU中的小硬件缓存
  - 将虚拟页码映射到物理页码
  - 包含少量页的完整页表条目

地址翻译符号汇总

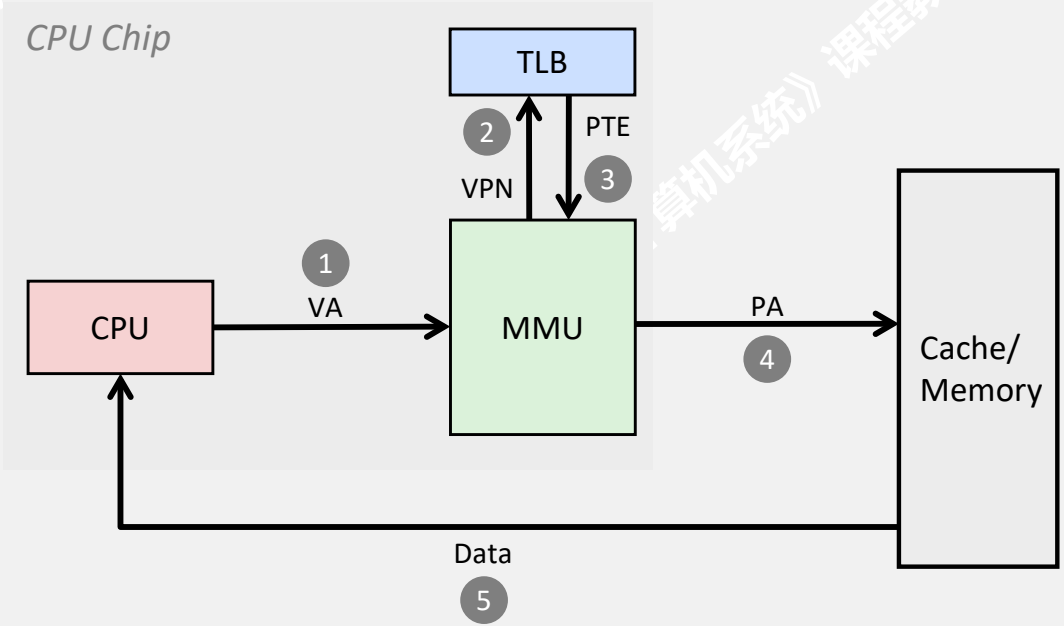
- **基本参数**
  - **N = 2^n**：虚拟地址空间中的地址数量
  - **M = 2^m**：物理地址空间中的地址数量
  - **P = 2^p**：页面大小（字节）
- **虚拟地址（VA）的组件**
  - **VPO**：虚拟页偏移量（字节）
  - **VPN**：虚拟页号
  - **TLBI**：TLB索引
  - **TLBT**：TLB标签
- **物理地址（PA）的组件**
  - **PPO**：物理页面偏移量（与VPO相同）
  - **PPN**：物理页号

访问TLB

- MMU使用虚拟地址的VPN部分访问TLB：

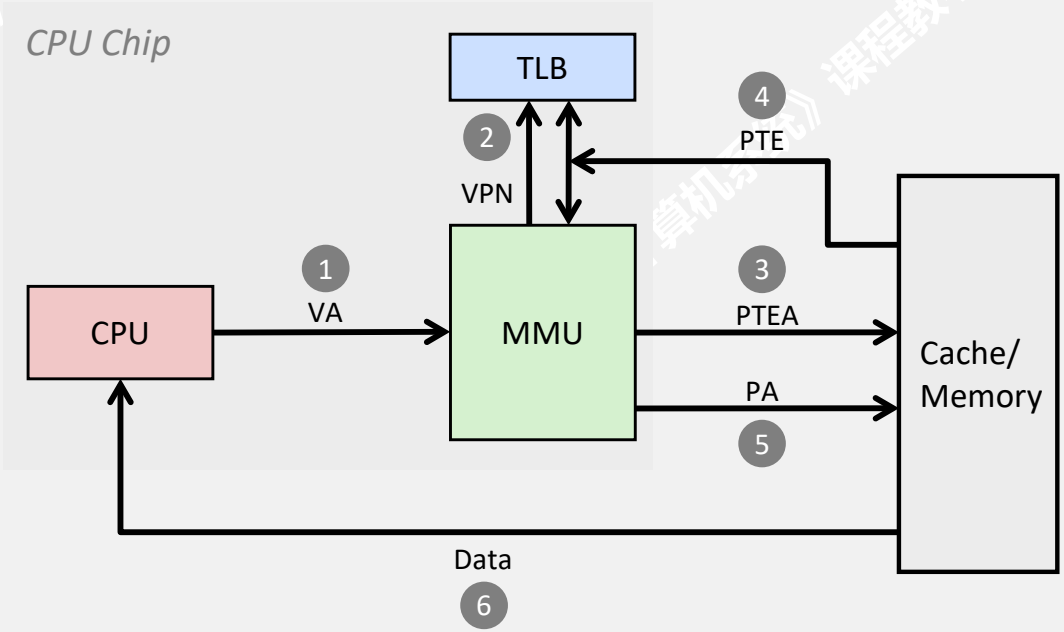


TLB命中



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?

小结

(1) 虚存可以有效使用有限的主存（RAM）

- 使用RAM作为虚拟地址空间部分的缓存
  - 一些非缓存部分存储在磁盘上
  - 一些（未分配的）未缓存的部分未存储
- 内存中只保留虚拟地址空间的活动区域
  - 根据需要来回传输数据

(2) 虚存简化了程序员的内存管理

- 每个进程都有一个完整的专用线性地址空间

(3) 虚存隔离地址空间

- 一个进程不能干扰另一个的内存
  - 因为它们在不同的地址空间中运行
- 用户进程无法访问特权信息
  - 地址空间的不同部分具有不同的权限

计算机系统

下一讲 预告

虚拟存储器-系统

湖南大学

《计算机系统》课程教学组