分页（page）

邰颖
南京大学
智能科学与技术学院

分页概念

•将**虚拟地址空间**分割成**固定大小**的单元，这种思想称为分**页**

- 不同于分段，分段中每个逻辑段/部分（如代码、栈、堆等）的大小是可变的

•将**物理内存**分割成**固定大小**的单元，称为页帧（page frames）

•每个进程需要一个**页表**来将虚拟地址转换为物理地址

分页的优点

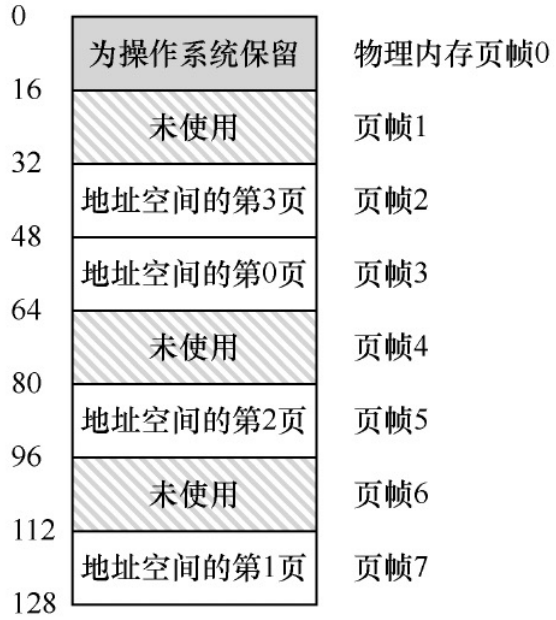
- **灵活性**：有效支持地址空间的抽象
  - 不需要假设堆和栈如何增长以及如何使用
- **简便性**：易于管理空闲空间
  - 虚拟地址空间中的分页与物理内存中的页帧大小相同
  - 易于分配和维护空闲列表

示例：简单分页

- 64字节的虚拟地址空间，采用16字节的页面大小
- 128字节的物理内存，采用16字节的页帧大小

虚拟页0	虚拟页1	虚拟页2	虚拟页3
物理帧3	物理帧7	物理帧5	物理帧2

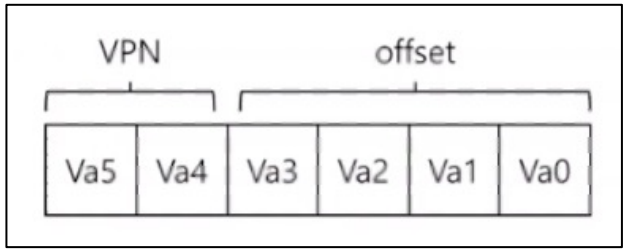
一个简单的64字节地址空间



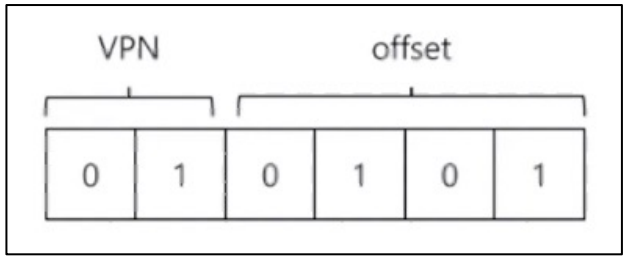
64 字节的地址空间在128 字节的物理内存中

地址转换

- 虚拟地址中有两个组成部分：
  - **VPN**：虚拟页号(Virtual page number)
  - **偏移量**：页内偏移量

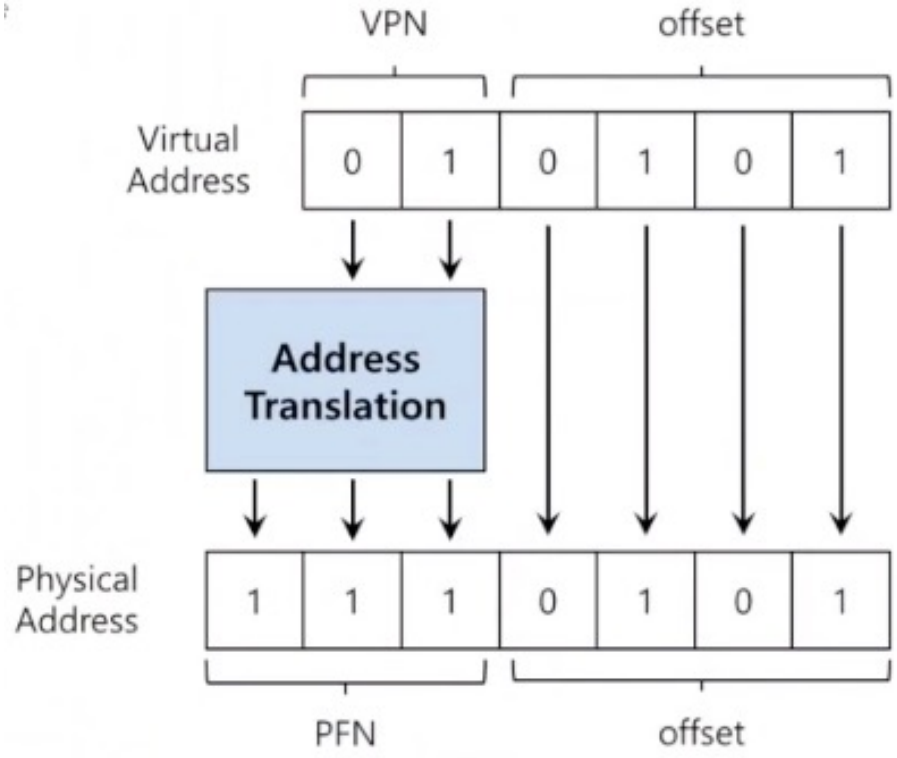


- 示例：64字节地址空间中的虚拟地址21

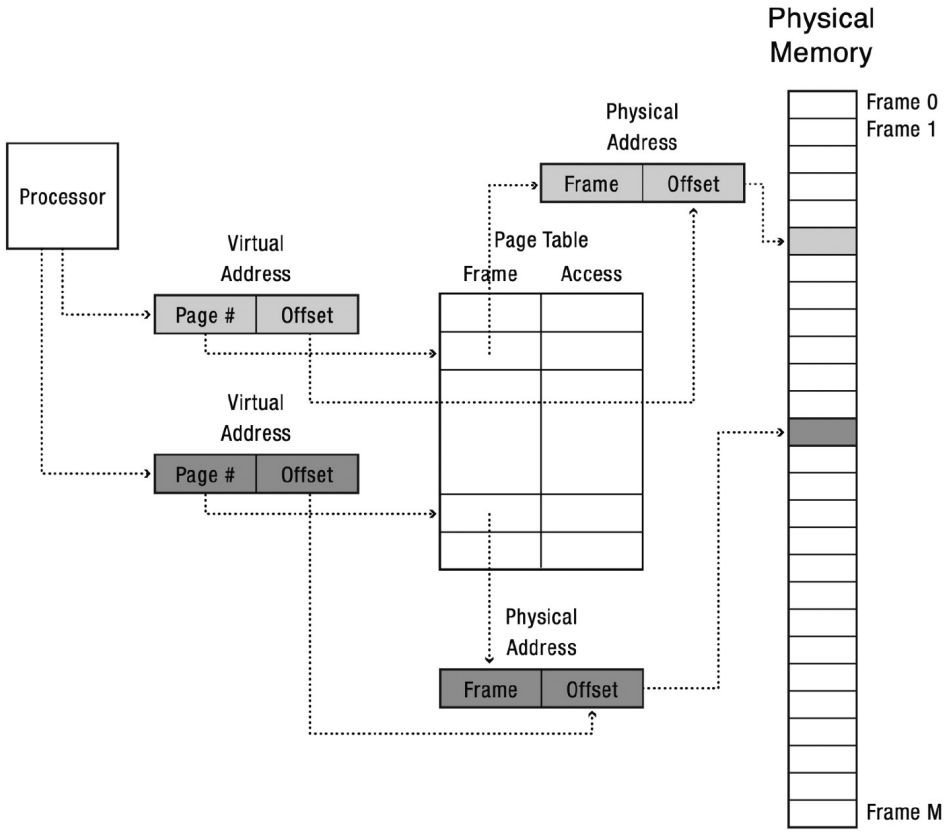


示例：地址转换

- 虚拟地址21在64字节地址空间中的转换



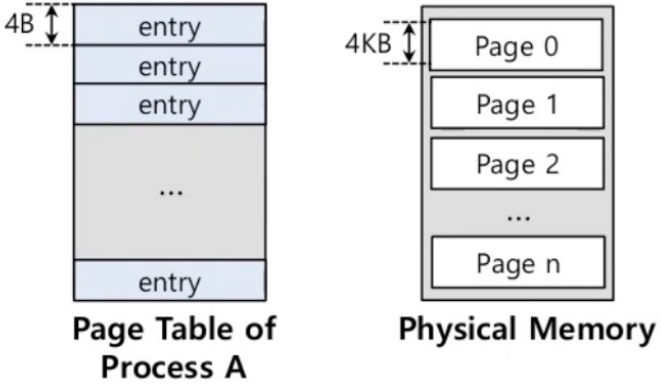
示例：从处理器到物理内存



页表存在哪里？

- 页表将虚拟页号（VPN）映射到物理帧号（PFN）

- 页表可能变得非常大：**直接取决于虚拟地址空间的大小、页面大小以及页表项的大小**
  - 32位地址空间，假设4KB页面大小，12位用于偏移量，20位用于虚拟页号（VPN）
  - 页表大小：4MB = 2^20页数 * 4字节（每个页表项）



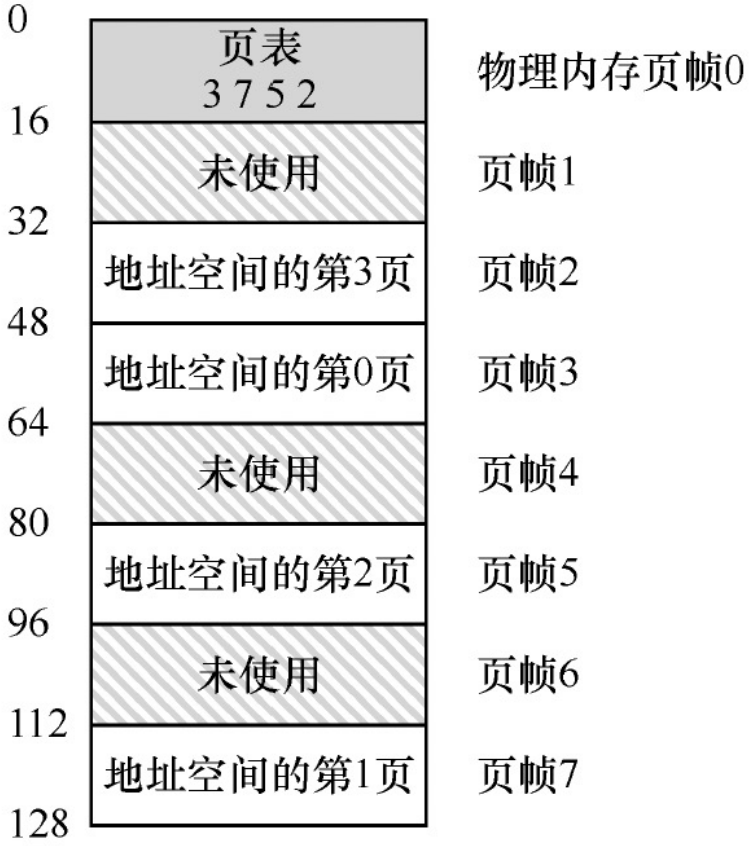
$$\text{页面数} = \frac{\text{虚拟地址空间大小}}{\text{页面大小}} = \frac{2^{32}\text{字节}}{2^{12}\text{字节}} = 2^{20}\text{页}$$

$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

页表存在哪里？（续）

- **每个进程**的页表存储在**内存**中（可能非常大）
  - **注意：每个进程需要一个页表！**
  - 示例：假设一个32位地址空间，4KB的页
    - 20位用于VPN，12位用于偏移量
    - 假设每个页表项需要4字节存储 -> 每个进程需要4MB的页表

示例：内核物理内存中的页表



虚拟页0	虚拟页1	虚拟页2	虚拟页3
物理帧3	物理帧7	物理帧5	物理帧2

- 物理页帧0一般被预留给操作系统的基本控制结构，如内核代码、页表管理
- 页表条目记录了每个虚拟页面与物理页帧的映射。
- 示例中，页表的每个条目按顺序排列，其中“地址空间的第X页”表示虚拟地址空间中的不同页。

页表中包含什么？

- 页表只是一个用于将虚拟地址映射到物理地址的数据结构
  - 最简单的形式：线性页表（一个数组）

- 操作系统通过虚拟页号（VPN）和索引数组（页表），然后查找对应的页表项（Page Table Entry, PTE）
  - 页表项的作用是将虚拟地址空间中的虚拟页号（VPN）映射到物理内存中的物理页帧号（PFN），并附带一些控制信息

页表项的常见标志

- **物理页帧号（PFN, Physical Frame Number）：** 映射到的物理页的地址。
- **有效位（Valid Bit）：** 表明该页表项是否有效
- **保护位（Protection Bit）：** 表明该页面是否可以读取、写入或执行
- **存在位（Present Bit）：** 表明该页面是否在物理内存中，或者是否被交换到磁盘上（交换出去）
- **脏位（Dirty Bit）：** 表明该页面自从被加载到内存后是否被修改过
- **参考位（Reference Bit，或者叫访问位）：** 表明该页面是否被访问过

示例：x86页表项

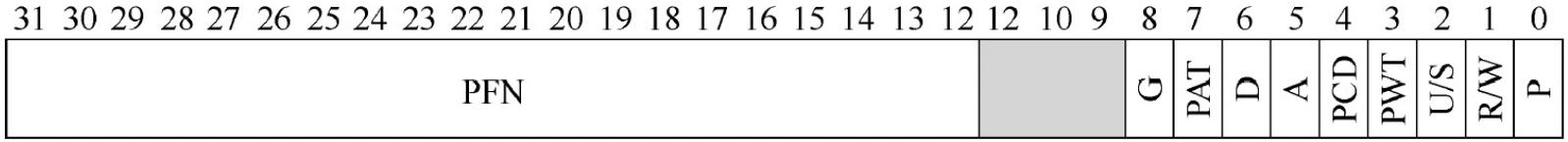


图 18.5　一个 x86 页表项（PTE）

- **P**：存在位（Present）
- **R/W**：读/写位（Read/Write bit）
- **U/S**：用户/超级用户位（User/Supervisor）
- **A**：访问位（Accessed bit）
- **D**：脏位（Dirty bit）
- **PFN**：页面框架号（Page Frame Number）

分页：太慢了

- 为了找到所需页表项（PTE）的位置，需要知道页表的**起始位置**。
  - 页表起始地址存储在**页表基址寄存器（PTBR）**中

- 虚拟地址到物理地址的转换过程中涉及多个内存访问步骤：
  - 操作系统需要根据虚拟地址查找对应的页表项（PTE），然后获取物理页号（PFN）和页内偏移（Page Offset）

  - **虚拟页号与页表项查找**：系统需要使用虚拟地址中的页号（VPN，Virtual Page Number）查找页表中对应的页表项（PTE）。页表项包含物理页框号（PFN），以及其他控制位（如访问权限、脏位等）。

  - **两次内存访问**：在最基本的分页机制中，每次虚拟地址到物理地址的转换都需要访问至少两次内存（一次查找页表项，另一次访问数据）

通过分页访问内存

- **分页涉及两次内存访问：**一次查找页表项（PTE），另一次访问数据

```
1       // Extract the VPN from the virtual address
2       VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4       // Form the address of the page-table entry (PTE)
5       PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7       // Fetch the PTE
8       PTE = AccessMemory(PTEAddr)
9
10      // Check if process can access the page
11      if (PTE.Valid == False)
12              RaiseException(SEGMENTATION_FAULT)
13      else if (CanAccess(PTE.ProtectBits) == False)
14              RaiseException(PROTECTION_FAULT)
15      else
16              // Access is OK: form physical address and fetch it
17              offset = VirtualAddress & OFFSET_MASK
18              PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19              Register = AccessMemory(PhysAddr)
```

内存跟踪

- 示例：简单的内存访问

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

- 编译并执行

```
prompt> gcc -o array array.c -Wall    prompt> ./array
```

- 生成的汇编代码（32位x86系统）

```
0x1024 movl $0x0, (%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8, %eax
0x1030 jne 0x1024
```

赋值

递增

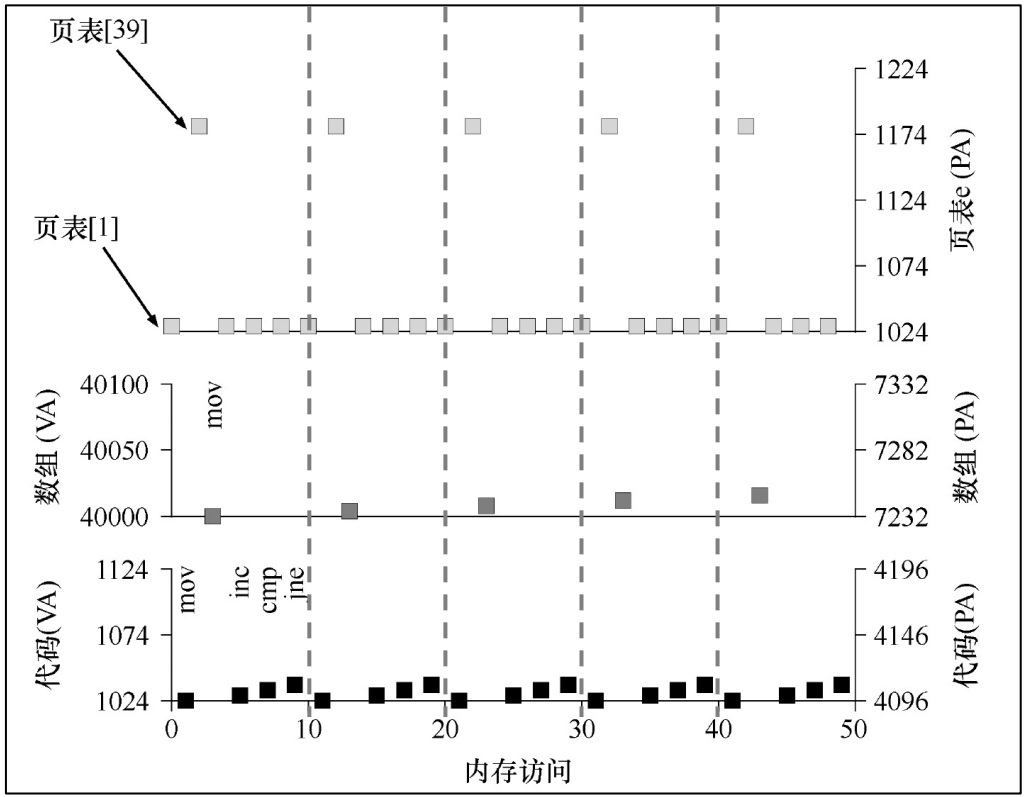
比较

跳转

内存跟踪：一些假设

- 虚拟地址空间大小：64KB（相比实际情况偏小）
- 假定页面大小：1 KB
- 线性页表（基于数组）位于物理地址1KB（1024）
- 页表内容：
  - 代码：假设VPN 1 -> PFN 4
  - 数组：
    - 4000字节（1000 * sizeof(int)）
    - 假设虚拟地址从40000到44000，涉及VPN39~VPN42
    - VPN 39 -> PFN 7
    - VPN 40 -> PFN 8
    - VPN 41 -> PFN 9
    - VPN 42 -> PFN 10

虚拟（和物理）内存跟踪：前5次迭代



- **每条指令获取会生成两个内存引用：**
  - 一个指向页表
  - 另一个指向实际的指令位置
- **mov指令**也需要两个内存引用：
  - 一个指向页表
  - 另一个指向数组元素本身

✓ 左边VA代表虚拟地址
✓ 右边PA代表物理地址

小结

我们介绍了分页（paging）、页表的基本原理

我们介绍了分页比较慢的问题

我们介绍了分页中的内存访问和跟踪