

内容提要

- 存储管理基础
- 页式内存管理
 - 基本原理
 - 基本概念：页表、地址变换、多级页表、快表
 - 页表类型：哈希页表、反置页表
 - 页共享
- 段式内存管理
- 虚拟存储管理
- 存储管理实例

程序、进程和作业

- 程序程序是静止的，是存放在磁盘上的可执行文件
- 进程是动态的。进程包括程序和程序处理对象（数据集），是一个程序对某个数据集的执行过程，是分配资源的基本单位。通常把进程分为系统进程和用户进程两大类：
 - 完成操作系统功能的进程称为系统进程；
 - 完成用户功能的进程则称为用户进程。
- 作业是用户需要计算机完成的某项任务，是要求计算机所做工作的集合。

程序与进程之间的区别

1. 进程是竞争计算机系统有限资源的基本单位。进程更能真实地描述并发，而程序不能。
2. 程序是静态的概念；进程是程序在处理机上一次执行的过程，是动态的概念。
3. 进程有生存周期，有诞生有消亡。是短暂的；而程序是相对长久的。
4. 一个程序可以作为多个进程的运行政序；一个进程也可以运行多个程序。
5. 进程具有创建其他进程的功能；而程序没有。

作业与进程的区别

1. 一个作业的完成要经过作业提交、作业收容、作业执行和作业完成4个阶段。而进程是对已提交完毕的程序所执行过程的描述，是资源分配的基本单位。
2. 作业是用户向计算机提交任务的任务实体。在用户向计算机提交作业后，系统将它放入外存中的作业等待队列中等待执行。而进程则是完成用户任务的执行实体，是向系统申请分配资源的基本单位。任一进程，只要它被创建，总有相应的部分存在于内存中。
3. 一个作业可由多个进程组成，且必须至少由一个进程组成，反过来则不成立。
4. 作业的概念主要用在批处理系统中，而进程的概念则用在几乎所有的多道程序系统中。

作业、进程和程序之间的联系

- 一个作业通常包括程序、数据和操作说明书3部分。每一个进程由进程控制块PCB、程序和数据集合组成。这说明程序是进程的一部分，是进程的实体。因此，一个作业可划分为若干个进程来完成，而每一个进程有其实体——程序和数据集合。

分页式存储管理的基本思想

- 进程需要**连续的空间**！
- 如果可以把一个逻辑地址连续的的程序分散存放到若干不连续的内存区域内，并保证程序的正确执行，则既可充分利用内存空间，又可减少移动带来的开销。这就是页式管理的基本思想。
- 页式管理首先由英国Manchester大学提出并使用。支持页式管理的硬件部件通常称为MMU（Memory Management Unit）



动态地址翻译：用户进程发出的虚拟地址由MMU翻译成物理地址

基本概念

- **页**：在分页存储管理系统中，把每个作业的地址空间分成一些**大小相等的片**，称之为**页面（Page）**或页，各页从0开始编号。
- **存储块**：在分页存储管理系统中，把主存的存储空间也分成与**页面相同大小的片**，这些片称为存储块，或称为**页框（Frame）**，同样从0开始编号。
- **页表**：为了便于在内存找到进程的每个页面所对应块，分页系统中为每个进程配置一张页表，进程逻辑地址空间中的每一页，在页表中都对应有一个页表项

Frame
Number

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

D2

纯分页系统 (Pure Paging System)

- 在分页存储管理方式中，如果不具备页面对换功能，不支持虚拟存储器功能，这种存储管理方式称为纯分页或基本分页存储管理方式。
- 在调度一个作业时，必须把它的所有页一次装到主存的页框内；如果当时页框数不足，则该作业必须等待，系统再调度另外作业。
- 优点：没有外碎片，每个内碎片不超过页大小；一个程序不必连续存放；便于改变程序占用空间的大小。
- 缺点：程序全部装入内存。

页面的大小

- 页大小（与块大小一样）是由硬件来决定的。通常为2的幂。选择页的大小为2的幂可以方便的将逻辑地址转换为页号和页偏移。如果逻辑地址空间为 2^m ，且页大小为 2^n 单元，那么逻辑地址的高 $m-n$ 位表示页号（页表的索引），而低 n 位表示页偏移。每页大小从512B到16MB不等。
- 现代操作系统中，最常用的页面大小为4KB。

页面的大小

若页面较小

- 减少页内碎片和总的内存碎片，有利于提高内存利用率。
- 每个进程页面数增多，使页表长度增加，占用内存较大。
- 页面换进换出速度将降低。

若页面较大

- 每个进程页面数减少，页表长度减少，占用内存较小。
- 页面换进换出速度将提高。
- 增加页内碎片，不利于提高内存利用率。

地址结构



例：地址长为 32 位，其中 0-11 位为页内地址，即每页的大小为 $2^{12}=4\text{KB}$ ；
12-31 位为页号，地址空间最多允许有 $2^{20}=1\text{M}$ 页。



例：地址长为 22 位，其中 0-11 位为块内地址，即每块的大小为 $2^{12}=4\text{KB}$ ，与页相等；
12-21 位为块号b，内存地址空间最多允许有 $2^{10}=1\text{K}$ 块。

地址结构

已知逻辑地址求页号和页内地址

- 给定一个逻辑地址空间中的地址为 A ，页面的大小为 L ，则页号 P 和页内地址 d （从 0 开始编号）可按下列式求得：

$$P = \text{INT} \left[\frac{A}{L} \right], d = [A] \bmod L$$

其中，INT 是整除函数，mod 是取余函数。

数据结构

- 进程页表：每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序；
 - 逻辑页号（本进程的地址空间） \rightarrow 物理页面号（实际内存空间）；
- 物理页面表：整个系统有一个物理页面表，描述物理内存空间的分配使用状况。
 - 数据结构：位示图，空闲页面链表；
- 请求表：整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换，也可以结合到各进程的PCB里；

关于页表

- 页表存放在内存中，属于进程的现场信息。
- 用途：
 1. 记录进程的内存分配情况
 2. 实现进程运行时的动态重定位。
- 访问一个数据需访问内存 2 次 (页表一次，内存一次)。
- 页表的基址及长度由页表寄存器给出。

页表始址	页表长度
------	------

页表(page table)

0	0
1	1
2	2
3	3

Process A

0	---
1	---
2	---

Process B

0	7
1	8
2	9
3	10

Process C

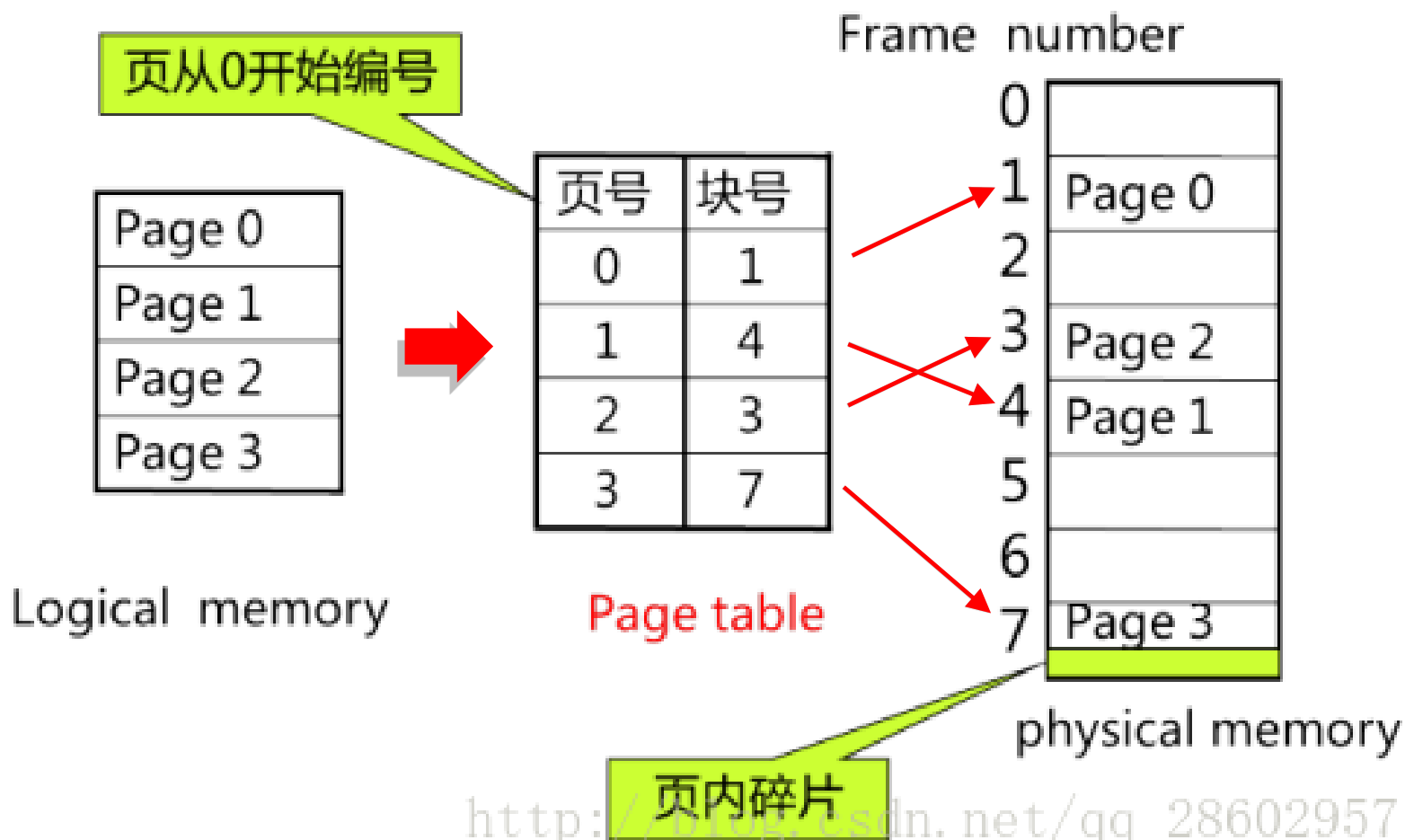
0	4
1	5
2	6
3	11
4	12

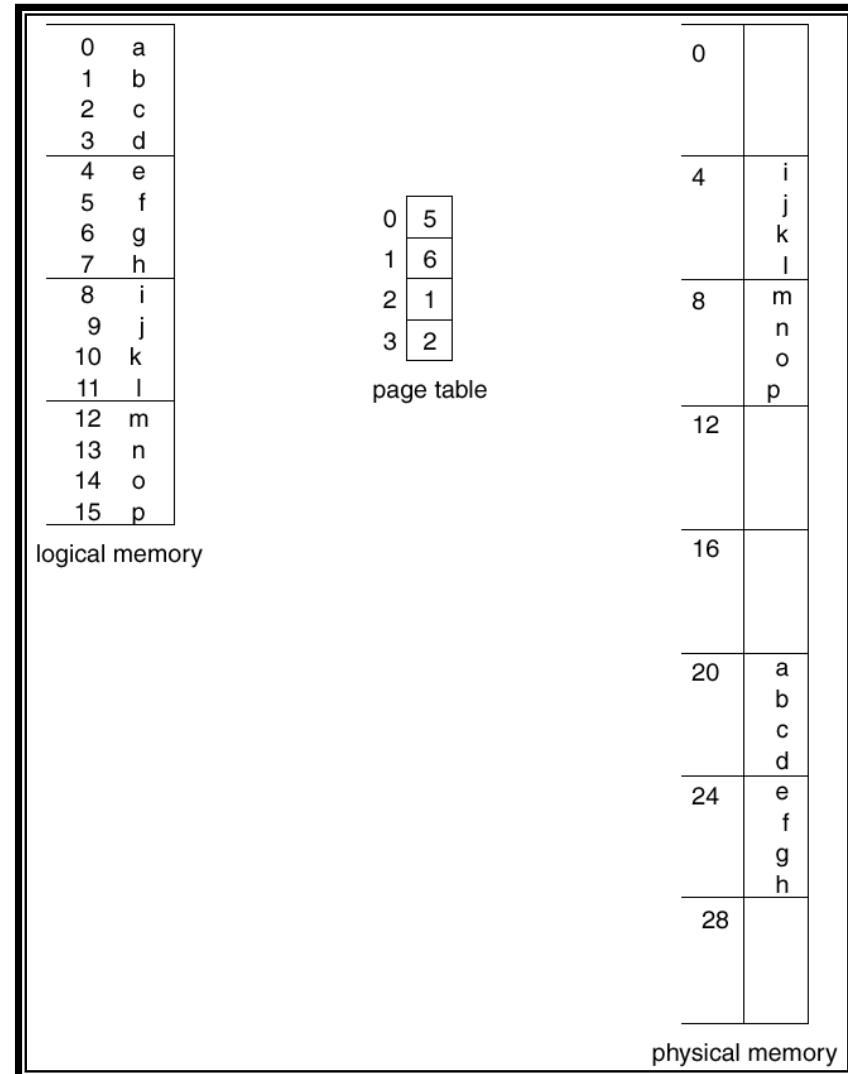
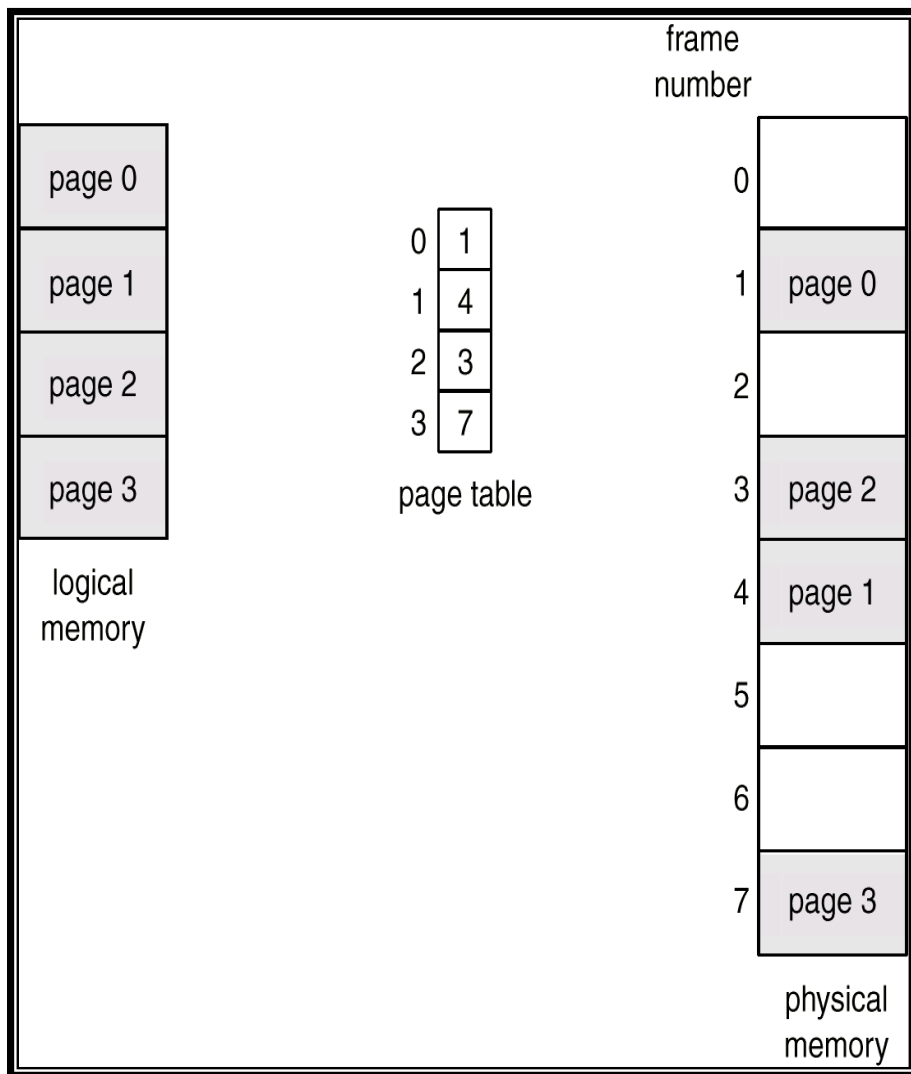
Process D

13
14

Free Frame List

地址变换——页表查找

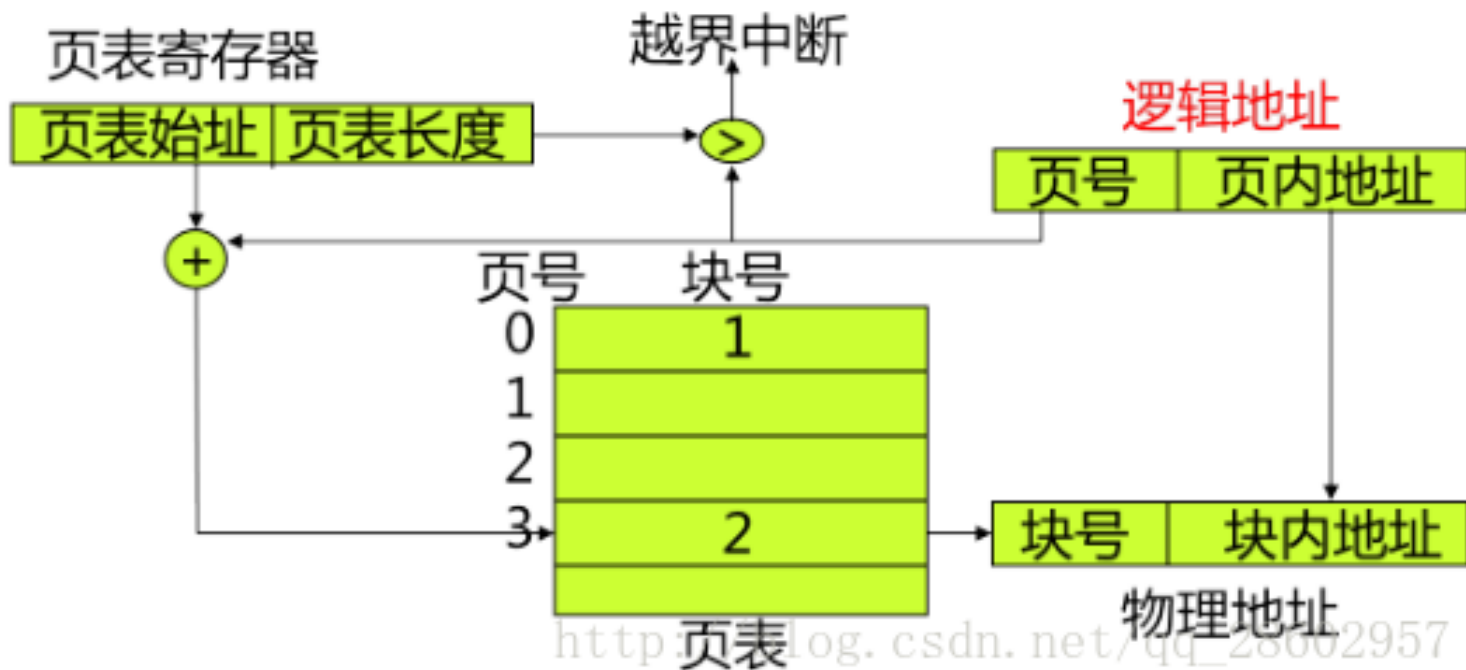




地址变换机构

- 当进程要访问某个逻辑地址中的数据时，分页地址变换机构会自动地将有效地址（相对地址）分为页号和页内地址两部分。
- 将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间，产生地址越界中断。
- 将页表始址与页号和页表项长度的乘积相加，得到该表项在页表中的位置，于是可从中得到该页的物理块号，将之装入物理地址寄存器中。
- 将有效地址寄存器中的页内地址送入物理地址寄存器的块内地址字段中。

地址转换机构



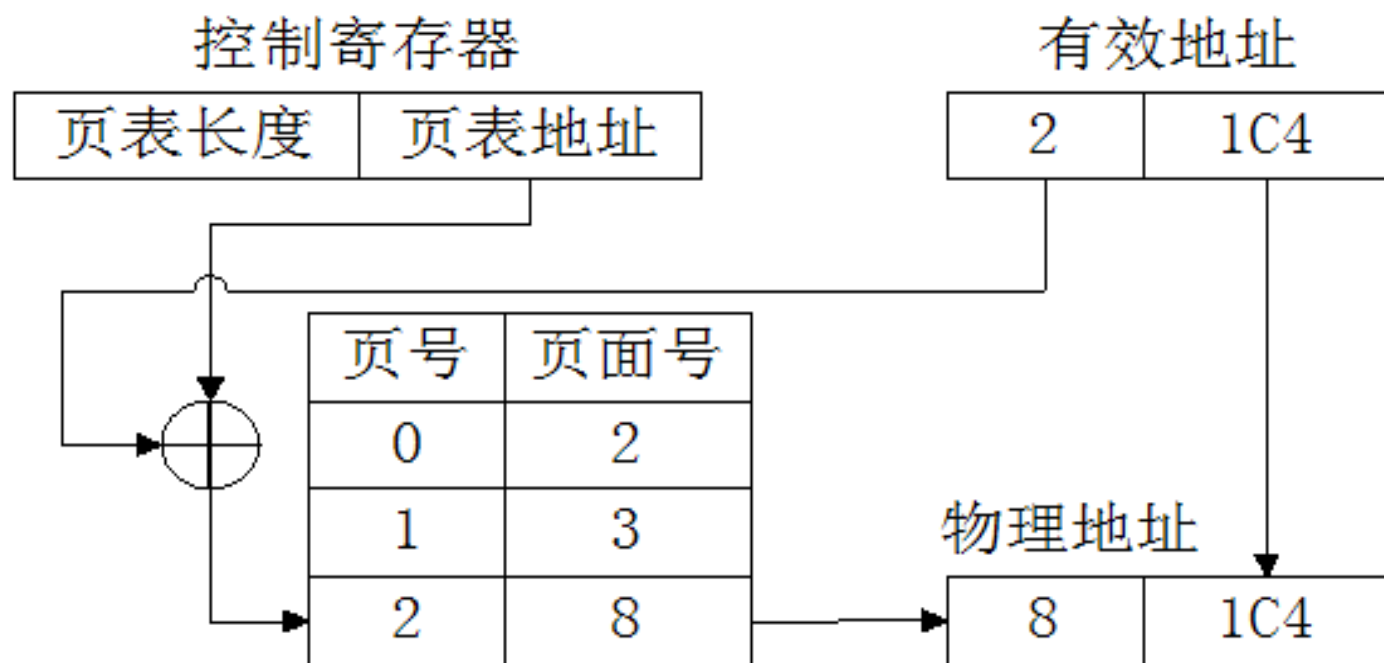
逻辑地址：把相对地址分为页号和页内地址两部分。

页表定位：页表始址 + 页号 × 页表项长度。

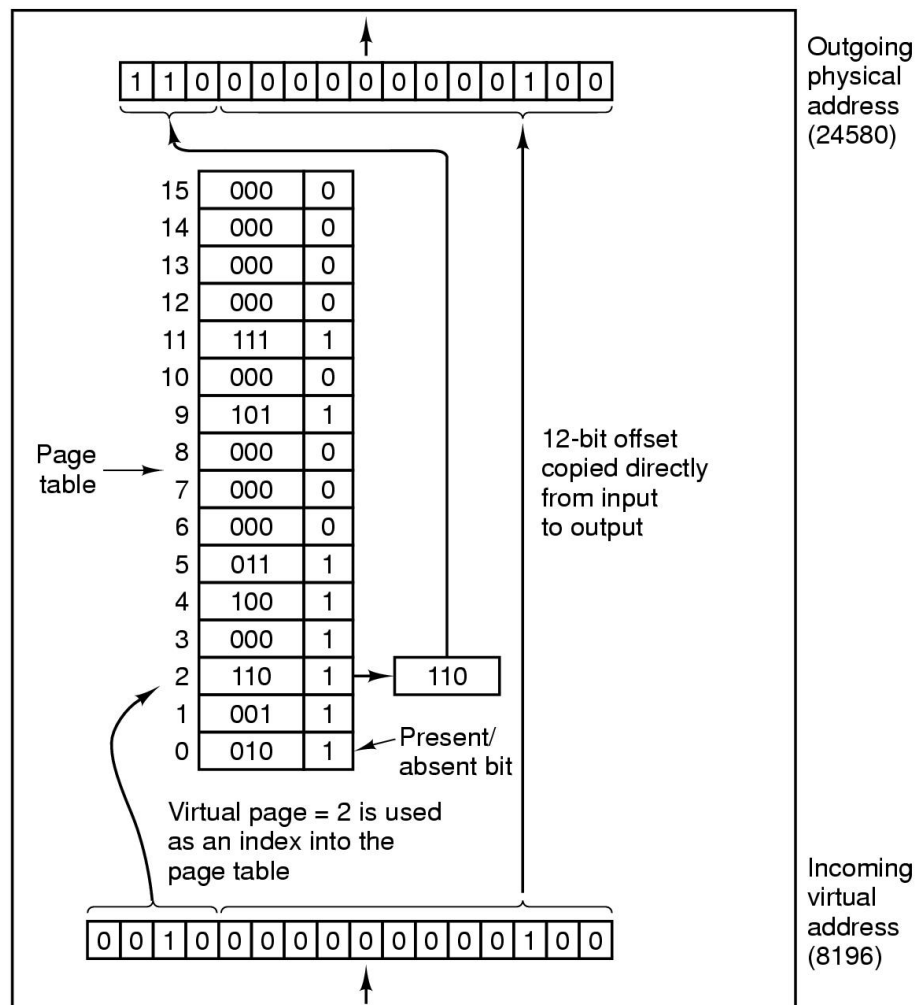
查询页表：读出块号。

物理地址：块号 + 块内地址。（块内地址 = 页内地址）

地址转换例 (21C4→81C4)



- ① 逻辑地址21C4分为页号2和页内位移1C4
- ② 根据寄存器所指页表地址与页号2找到对应的页面号8
- ③ 将8与页内位移1C4合并成物理地址81C4



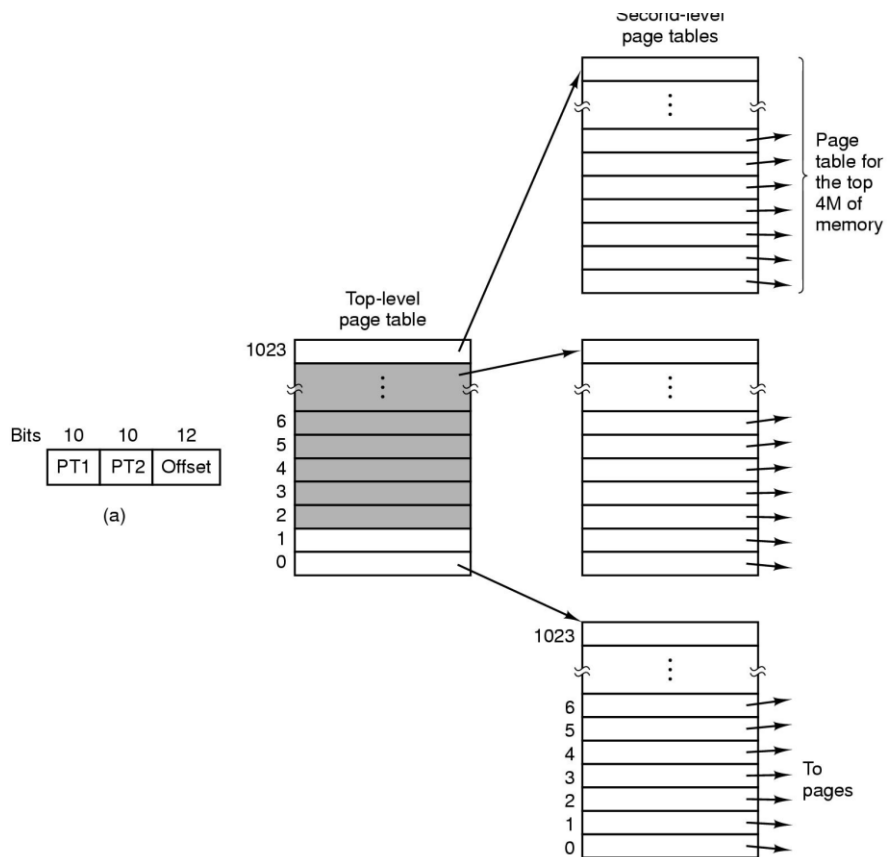
Internal operation of MMU with 16 4 KB pages

一级页表的问题

- 若逻辑地址空间很大 ($2^{32} \sim 2^{64}$)，则划分的页比较多，页表就很大，占用的存储空间大（要求连续），实现较困难。
- 例如，对于 32 位逻辑地址空间的分页系统，如果规定页面大小为 4 KB 即 2^{12} B，则在每个进程页表就由高达 2^{20} 页组成。设每个页表项占用 4 个字节，每个进程仅仅页表就要占用 4 MB 的内存空间。
- 解决问题的方法
 - 动态调入页表: 只将当前需用的部分页表项调入内存，其余的需用时再调入。
 - 多级页表

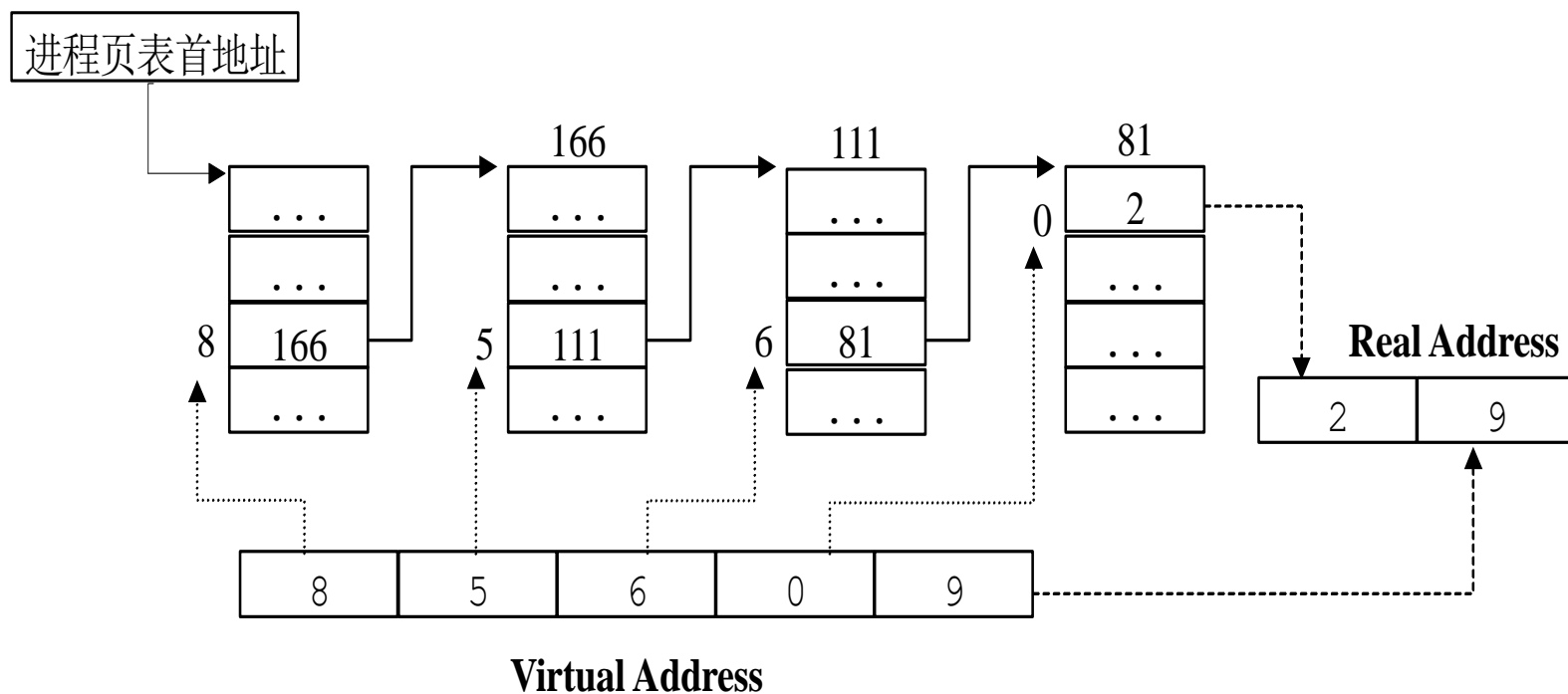
两级页表

- 32位
- 4K页面大小
- 4M页表
- 10 10 12



多级页表

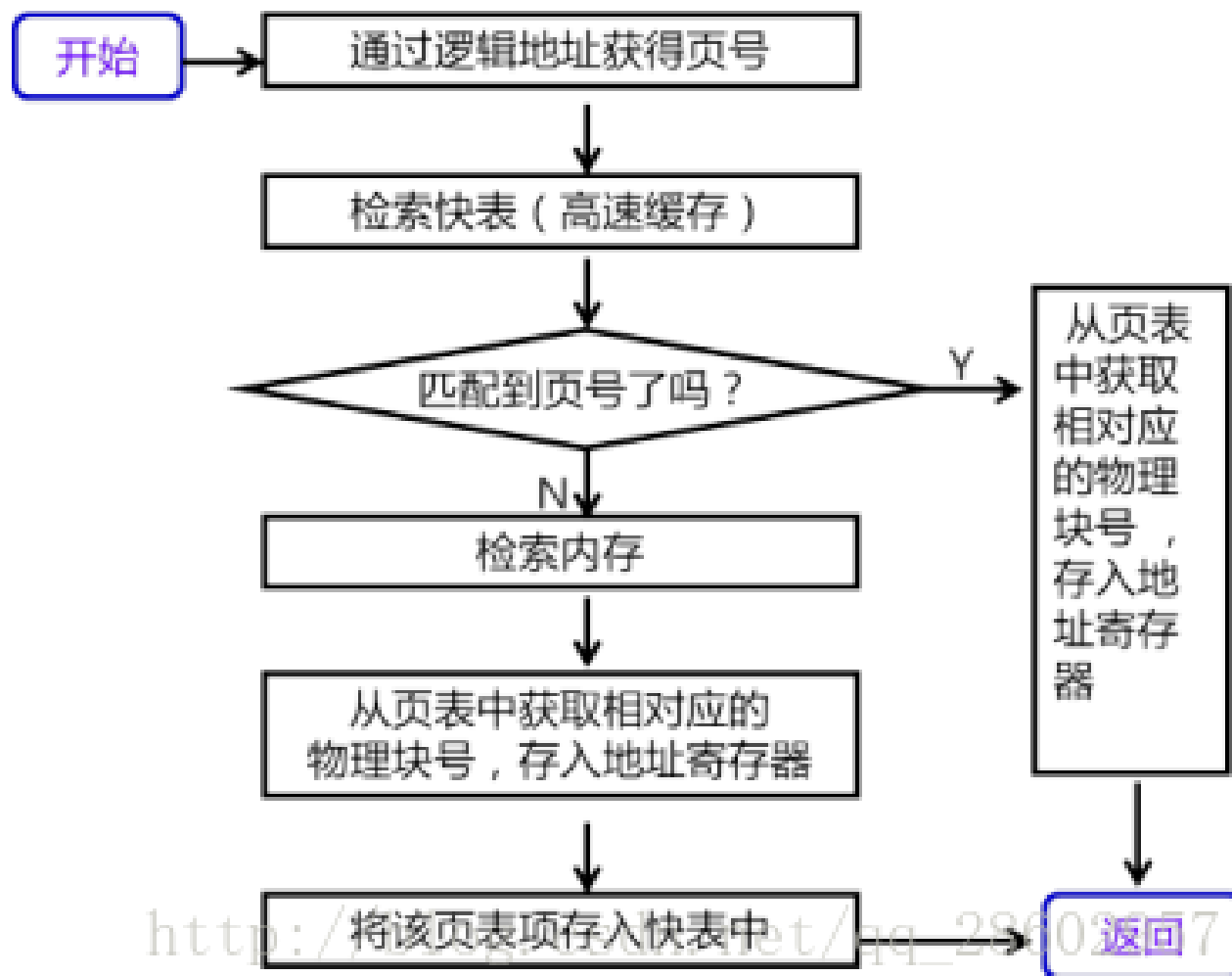
- 多级页表结构中，指令所给出的地址除偏移地址之外的各部分全是各级页表的页表号或页号，而各级页表中记录的全是物理页号，指向下级页表或真正的被访问页。



具有快表的地址变换机构

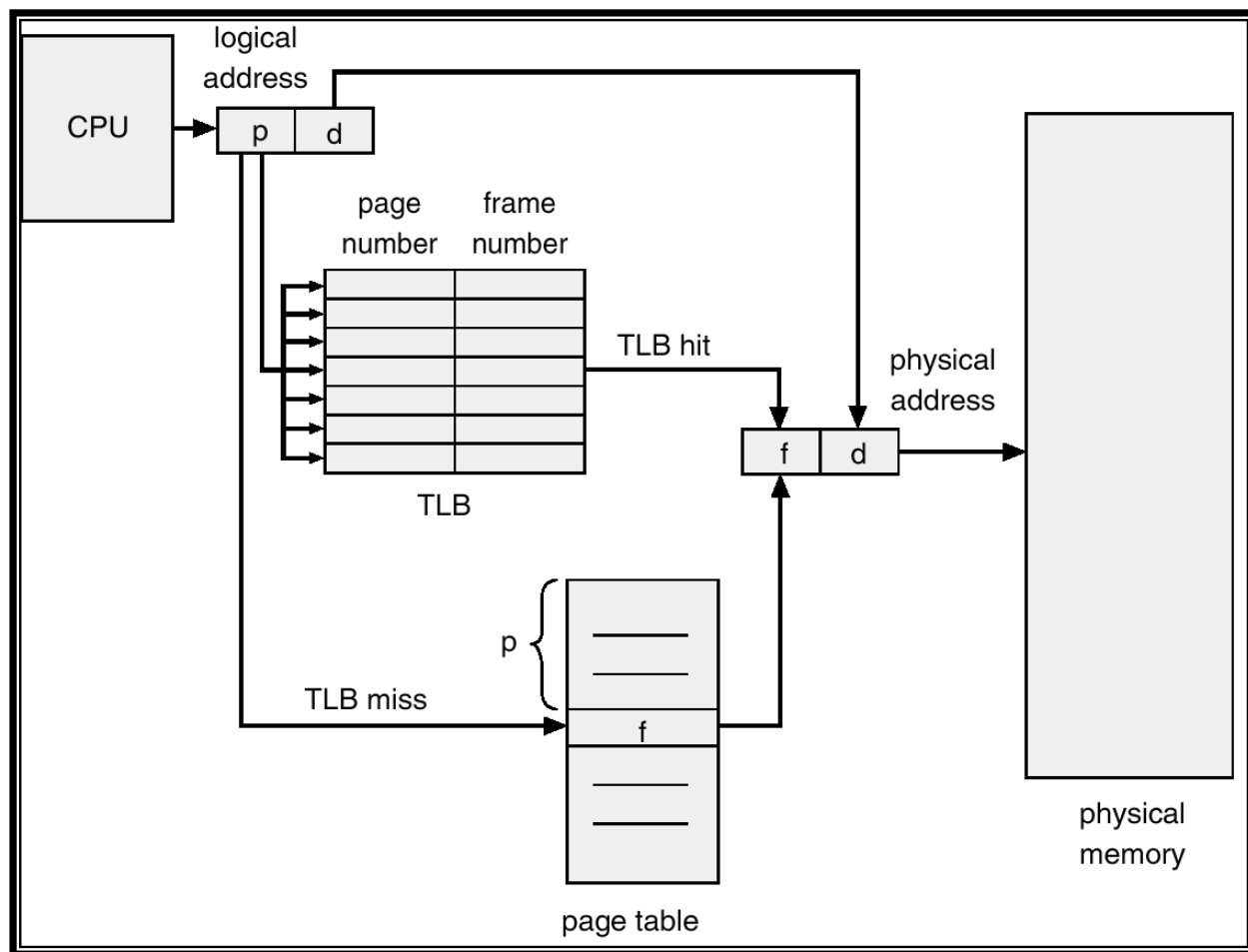
- 快表又称联想存储器 (Associative Memory)、TLB (Translation Lookaside Buffer) 转换表查找缓冲区, IBM最早采用TLB。
- 快表是一种特殊的高速缓冲存储器 (Cache) , 内容是页表中的一部分或全部内容。
- CPU 产生逻辑地址的页号, 首先在快表中寻找, 若命中就找出其对应的物理块; 若未命中, 再到页表中找其对应的物理块, 并将之复制到快表。若快表中内容满, 则按某种算法淘汰某些页。
- 通常, TLB中的条目数并不多, 在64~1024之间。

快表 (TLB)



快表 (TLB)

- 页表项的高速缓存



快表 (TLB) 的结构

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

快表 (TLB)

TLB的性质和使用方法与Cache相同：

- TLB只包括也表中的一小部分条目。当CPU产生逻辑地址后，其页号提交给TLB。如果页码不在TLB中（称为TLB失效），那么就需要访问页表。将页号和帧号增加到TLB中。
- 如果TLB中的条目已满，那么系统会选择一个来替换。替换策略有很多，从最近最少使用替换（LRU）到随机替换等。
- 另外，有的TLB允许有些条目固定下来。通常内核代码的条目是固定下来的。

快表 (TLB)

TLB的其它特性

- 有的TLB在每个TLB条目中还保存地址空间标识码 (address-space identifier, ASID)。ASID可用来唯一标识进程，并为进程提供地址空间保护。当TLB试图解析虚拟页号时，它确保当前运行进程的ASID与虚拟页相关的ASID相匹配。如果不匹配，那么就作为TLB失效。
- 除了提供地址空间保护外，ASID允许TLB同时包含多个进程的条目。如果TLB不支持独立的ASID，每次选择一个页表时（例如，上下文切换时），TLB就必须被冲刷 (flushed) 或删除，以确保下一个进程不会使用错误的地址转换。

Size	2-way		4-way		8-way	
	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

有效内存访问时间（一级页表）

- EAT-Effective Access Time
- TLB查询时间 = ε 时间单位
- 单次内存访问时间 = τ 时间单位
- TLB的命中率是 α
- EAT的计算：
 - $$\text{EAT} = (\tau + \varepsilon) * \alpha + (2\tau + \varepsilon) * (1 - \alpha) = 2\tau + \varepsilon - \tau\alpha$$

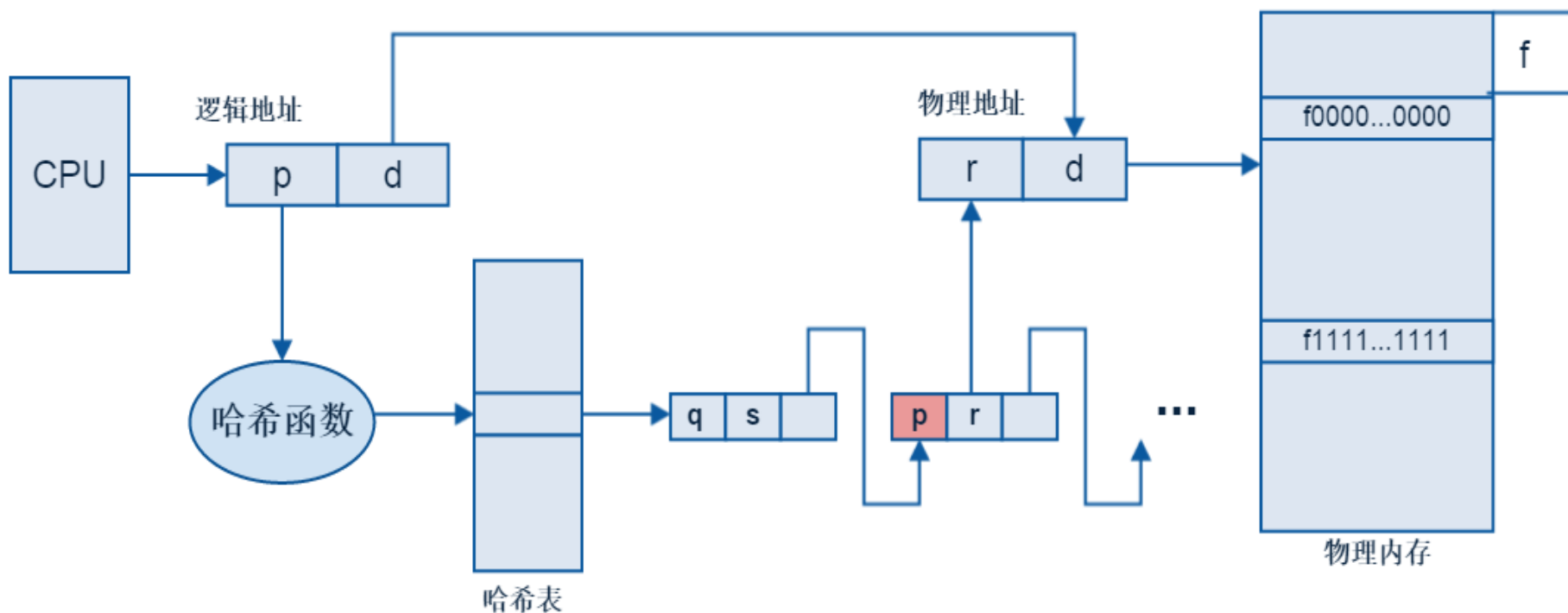
有效内存访问时间：举例

- 假如查找TLB需要20ns，访问内存需要100ns，TLB命中率为80%
 - 有效内存访问时间 = $2 \times 100 + 20 - 80\% \times 100 = 140 \text{ ns}$
- 过程分析：如果访问位于TLB中的页号，那么需要20+100=120ns。如果不能在TLB中找到（20ns），那么必须先访问位于内存中的页表得到帧号（100ns），并进而访问内存中所需字节（100ns），这总共需要220ns。
 - 有效内存访问时间 = $0.80 \times 120 + 0.2 \times 220 = 140 \text{ (ns)}$
 - 比不用页表的内存访问速度要慢40%（100ns v.s. 140ns）
- 如果TLB命中率为98%，那么有效内存访问时间 = $2 \times 100 + 20 - 0.98 \times 100 = 122 \text{ (ns)}$
 - 由于提高了命中率，内存访问时间只慢了22%
- 有没有其他办法解决多级页表访问多次访问内存？

哈希页表 (hashed page table)

- 处理超过32位地址空间的常用方法是使用哈希页表 (hashed page table)，并以虚拟页码作为哈希值。哈希页表的每一条目都包括一个链表的元素，这些元素哈希成同一位置（要处理器碰撞）。每个元素有3个域：
 1. 虚拟页码
 2. 所映射的帧号
 3. 指向链表中下一个元素的指针。
- 该算法按照如下方式工作：虚拟地址中的虚拟页号转换为哈希表号，用虚拟页号与链表中的每一个元素的第一个域相比较。如果匹配，那么相应的帧号（第二个域）就用来形成物理地址，如果不匹配，那么就对链表中的下一个节点进行比较，以寻找一个匹配的页号。

哈希页表 (Hashed Page Table)



ML与SYS

- SYS支持ML
- ML优化SYS

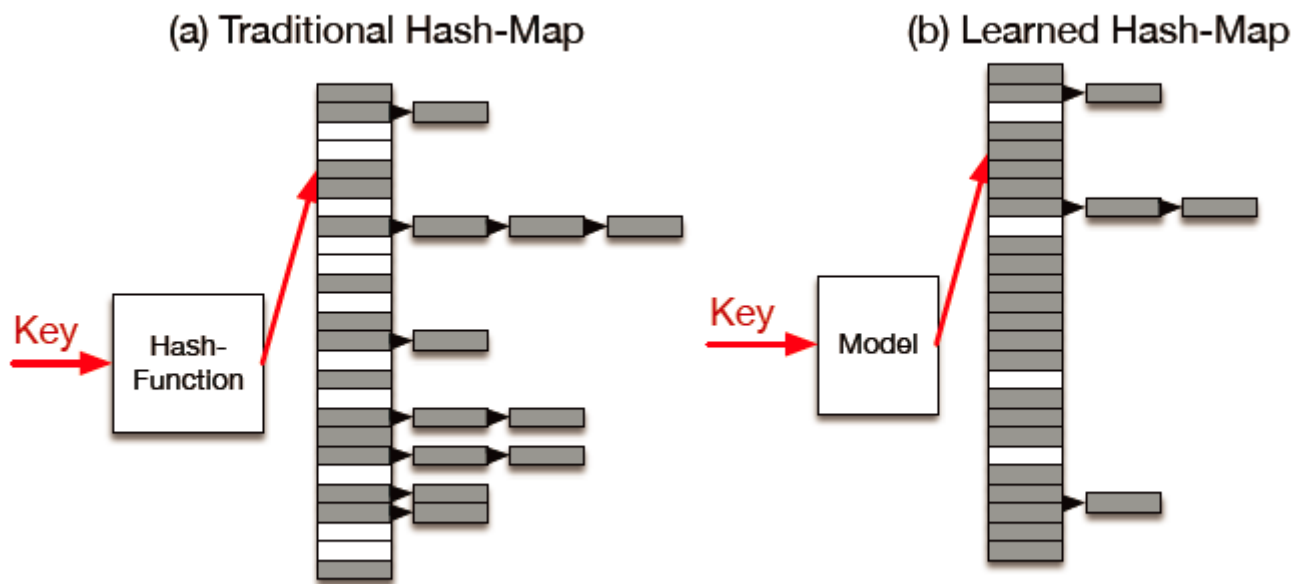


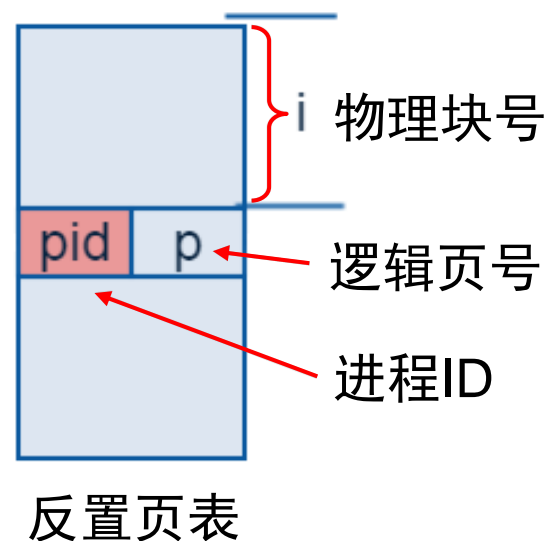
Figure 9: Traditional Hash-map vs Learned Hash-map

反置页表(Inverted page table)

- 一般意义上，每个进程都有一个相关页表。该进程所使用的每个页都在页表中有一项。这种页的表示方式比较自然，这是因为进程是通过页的虚拟地址来引用页的。操作系统必须将这种引用转换成物理内存地址。
- 这种方法的缺点之一是每个页表可能有很多项。这些表可能消耗大量物理内存，却仅用来跟踪物理内存是如何使用的。如每个使用32位逻辑地址的进程其页表长度均为4MB。
- 为了解决这个问题，可以使用反置页表（inverted pagetable）。

反置页表(Inverted page table)

- 反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（即：**按物理页面号排列**），其表项的内容是逻辑页号 P 及隶属进程标志符 pid 。
- **反置页表的大小**只与物理内存的大小相关，与逻辑空间大小和进程数无关。如：64M主存，若页面大小为 4K，则反向页表只需 64KB。
- 如64位的PowerPC, UltraSparc等处理器。

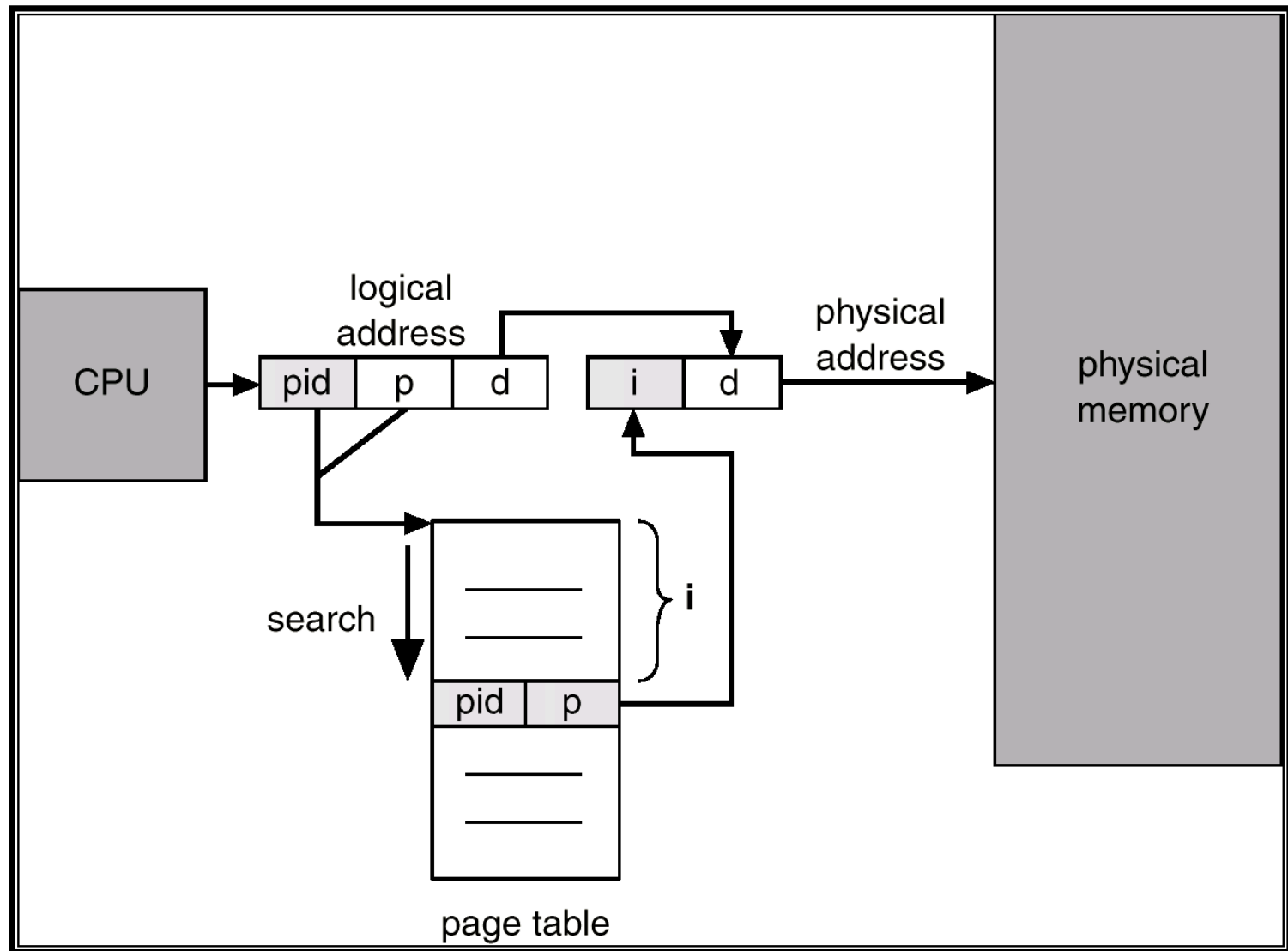


反置页表

利用反置页表进行地址变换：

- 用进程标志符和页号去检索反置页表。
- 如果检索完整个页表未找到与之匹配的页表项，表明此页此时尚未调入内存，对于具有请求调页功能的存储器系统产生请求调页中断，若无此功能则表示地址出错。
- 如果检索到与之匹配的表项，则表项的序号 i 便是该页的物理块号，将该块号与页内地址一起构成物理地址。

反置页表(Inverted page table)



反置页表(Inverted page table)

- 反向页表按照物理地址排序，而查找依据虚拟地址，所以可能需要查找整个表来寻找匹配。
- 可以使用 哈希页表 限制页表条目或加入 TLB 来改善。
- 通过**哈希表**(hash table)查找可由逻辑页号得到物理页面号。虚拟地址中的逻辑页号通过哈希表指向反置页表中的表项链头（因为哈希表可能指向多个表项），得到物理页面号。
- 采用反向页表的系统很难共享内存，因为每个物理帧只对应一个虚拟页条目。

Virtual Address

Page #	Offset
--------	--------

(Hash)

Hash Table

Page Table

Page #	Entry	Chain
--------	-------	-------

Inverted Page Table

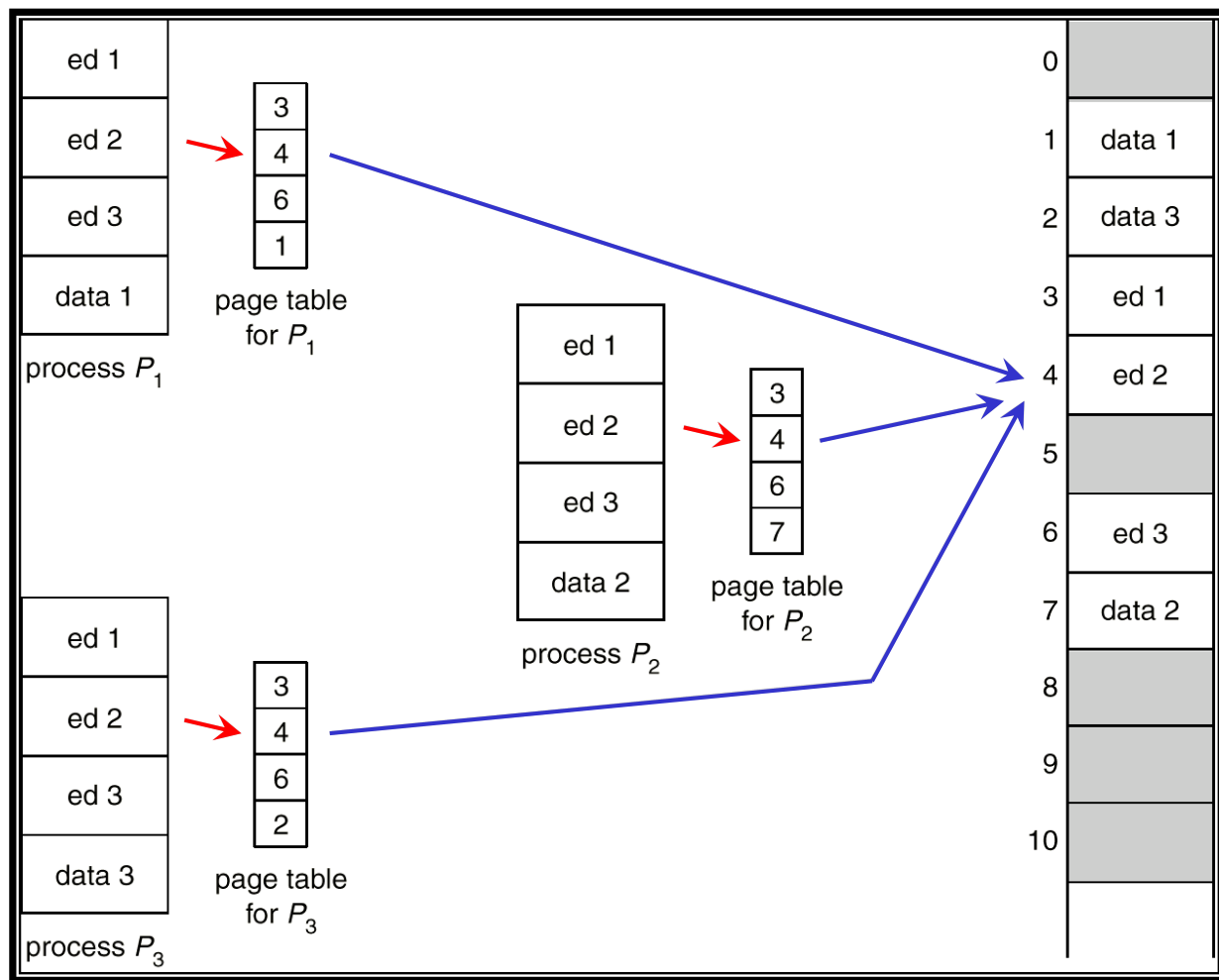
Frame #

Frame #	Offset
---------	--------

Real Address

页共享与保护

- 各进程把需要共享的数据/程序的相应页指向相同物理块。



页共享与保护

Virtual Address
(Process A):

Virtual Page #	Offset
----------------	--------

PageTablePtrA

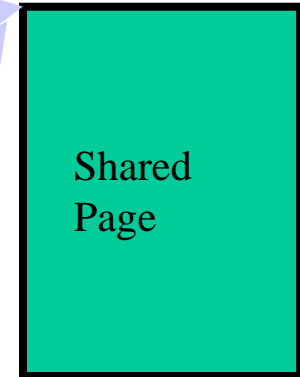
page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

PageTablePtrB

page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

Virtual Address:
Process B

Virtual Page #	Offset
----------------	--------



This physical page
appears in address
space of both processes

页共享与保护

页的保护

- 页式存储管理系统提供了两种方式：
 - 地址越界保护
 - 在页表中设置保护位（定义操作权限：只读，读写，执行等）

共享带来的问题

- 若共享数据与不共享数据划在同一块中，则：
 - 有些不共享的数据也被共享，不易保密。
- 实现数据共享的最好方法：分段存储管理。

内容小结

- 基本原理
 - 非连续的内存分配
 - 固定的分配单位
 - 虚拟页面与物理页框的分离、映射
- 基本概念：页表、地址变换、多级页表、快表
- 其他页表类型：哈希页表、反置页表
- 页共享与保护
- 思考：页式管理有什么不足？