页式存储管理

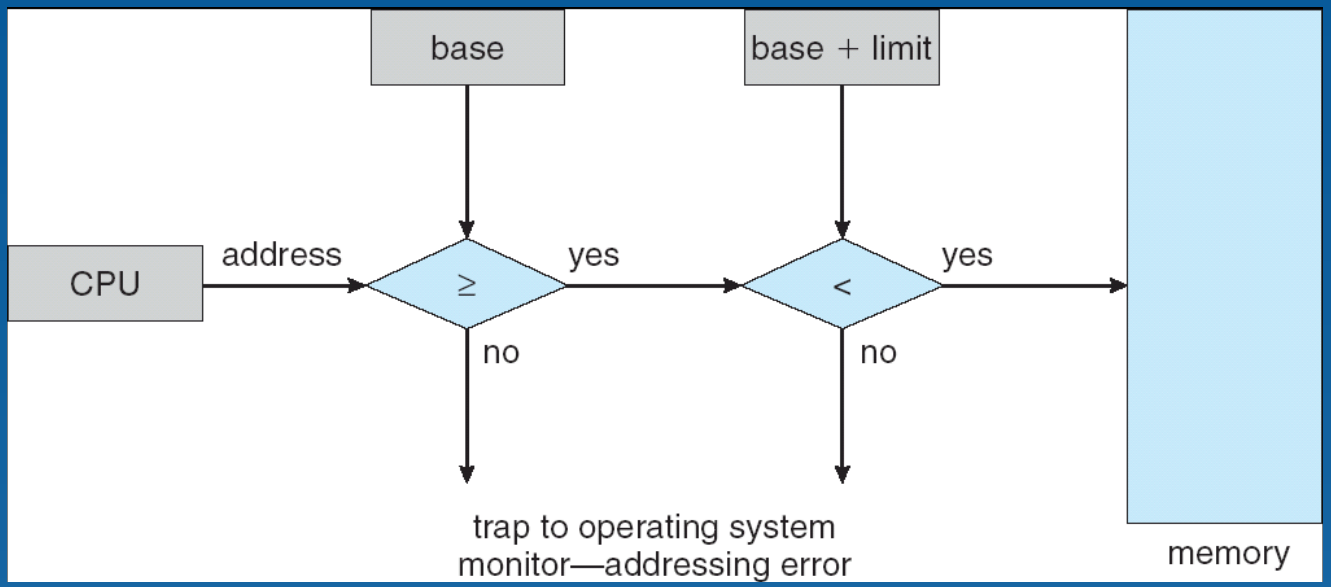
连续区内存分配

- 把主存划分成 2 个分区 (partitions)
  - 操作系统占 1 个分区。通常驻留主存的低端。中断矢量也在低端
  - 用户进程占另 1 个分区，通常在主存的高端

连续区内存分配（续）

- 运用重定位寄存器 (Relocation registers) 防止用户进程访问其它进程的空间，篡改操作系统的代码、数据
  - 基地址寄存器 (Base register) 保存了进程物理地址的首地址
  - 界限寄存器 (Limit register) 保存了逻辑地址的地址范围。任一个逻辑地址必须小于界限寄存器的值
  - MMU 能够动态地映射每一个地址

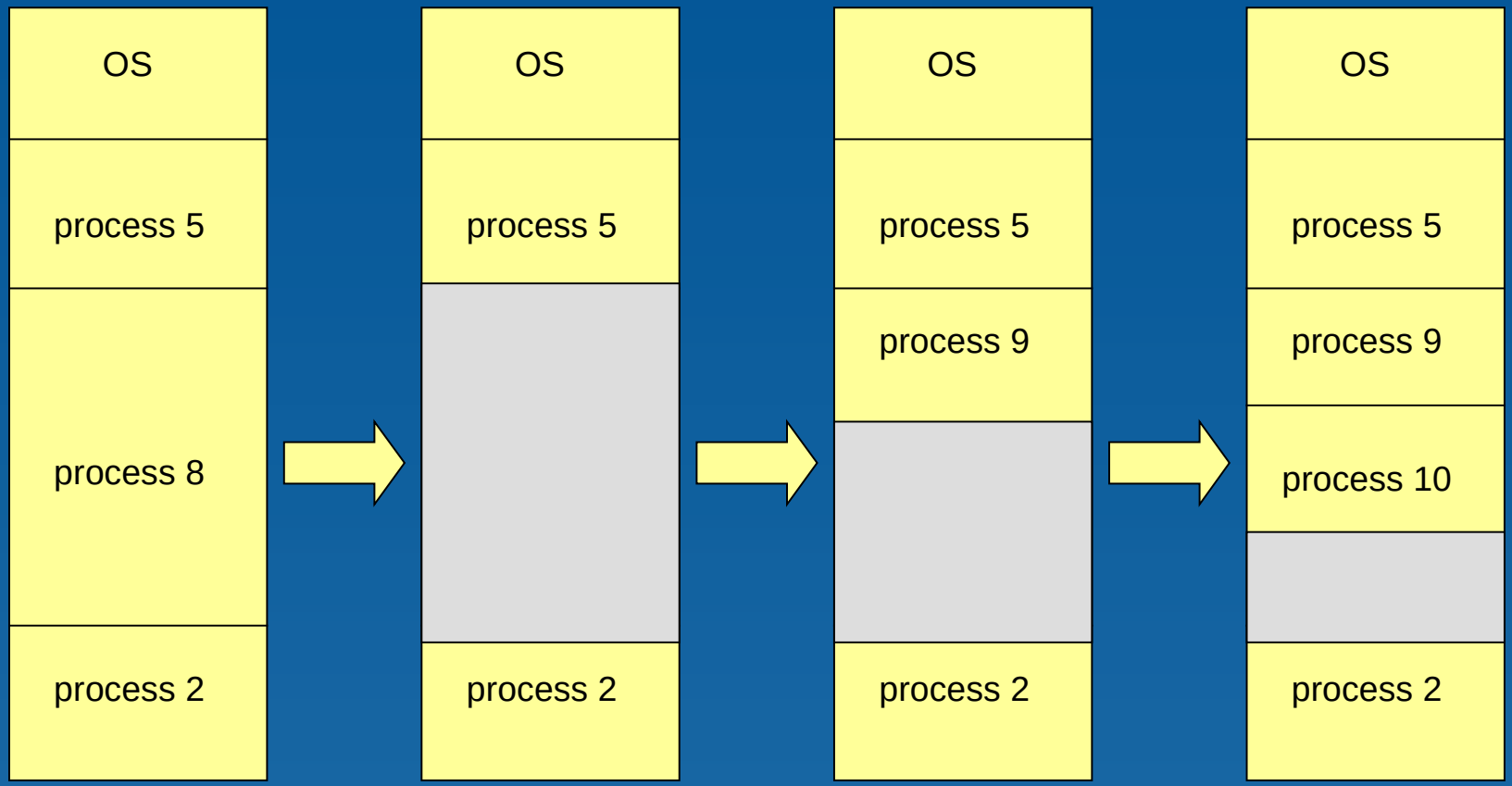
利用 base 和 limit 寄存器进行地址映射、地址保护



多重分区 (Multiple-partition) 连续区分配

- Hole – 有效可分配的内存块
- 多个长度不等的 holes 散布在内存各区域
- 当一个进程申请进入主存时， OS 选出一个 hole，其长度足够容纳进程的映像。它就是分配给该进程的分区 (partition)
- OS 维护一些管理信息，包括:
- 已经分配的分区
- 可分配的分区 (hole)

多重分区连续区分配（续）


OS
process 5
process 8
process 2
OS
process 5
process 2
OS
process 5
process 9
process 2
OS
process 5
process 9
process 10
process 2

如何找 Hole？

- 动态存储分配问题:
如何在一串 holes 中找出一个能存储 *n* 个单元
  - **First-fit**： 找到第 1 个足够大的 hole
  - **Best-fit**
  - 在所有足够大的 holes 里面，找出最小的一个 hole
  - 不得不寻找整个列表
  - 之前申请的分区归还后，留下一堆“最小” holes

如何找 Hole？

- **Worst-fit**
- 在所有足够大的 holes 里面，找出最小的一个 hole
- 不得不寻找整个列表
- 之前申请的分区归还后，留下一堆“最小” holes

碎片 (Fragmentation)

- **External Fragmentation** – 这些内存块加起来能够满足一个请求，但是由于不连续（中间有断层），不能用来连续区分配
- **Internal Fragmentation** – 分出去的分区略大于请求的内存长度。这个余下的小内存块属于该分区，但是无法利用

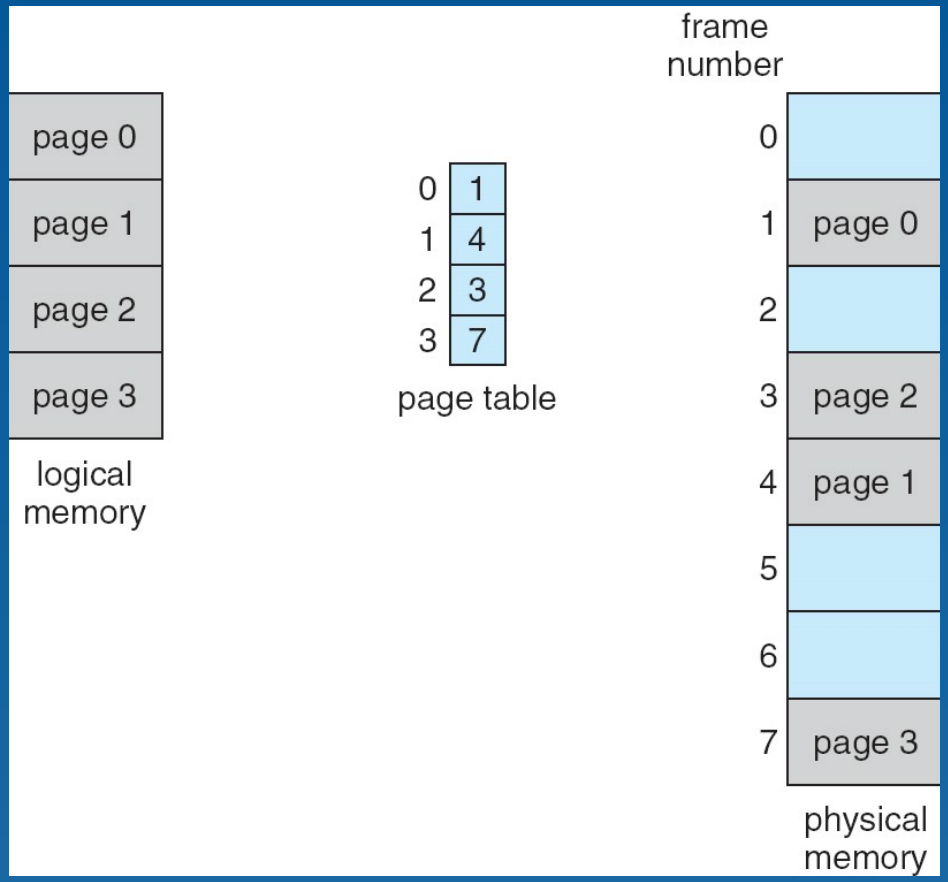
碎片（续）

- 紧缩 (**compaction**) 操作减少 external fragmentation
  - 重排内存块，使所有空闲内存连续排列，合并成一块大的内存块
  - 前提条件
    - 代码、数据可重定位；
    - 重定位可以在运行时操作

页式内存管理

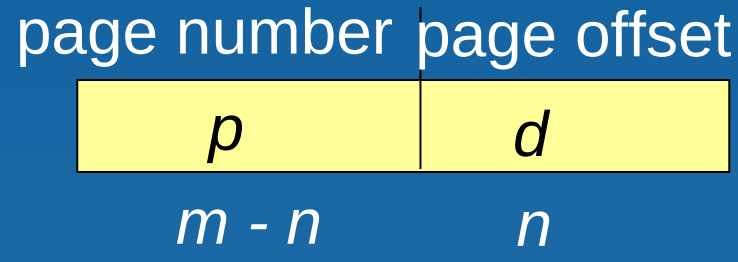
- 注意一个事实：进程并不要求逻辑地址必须连续的
- 把物理空间等分成长度一致的数据块，称作“页帧”（**frames**）
- 把逻辑空间等分成长度一致的数据块，称作“页”（**pages**），并且与页帧长度相等
- 通常，页长（也就是页帧长度）是 2 的次幂，取 512 字节与 8192 字节之间的数值

逻辑空间至物理空间的函数模型

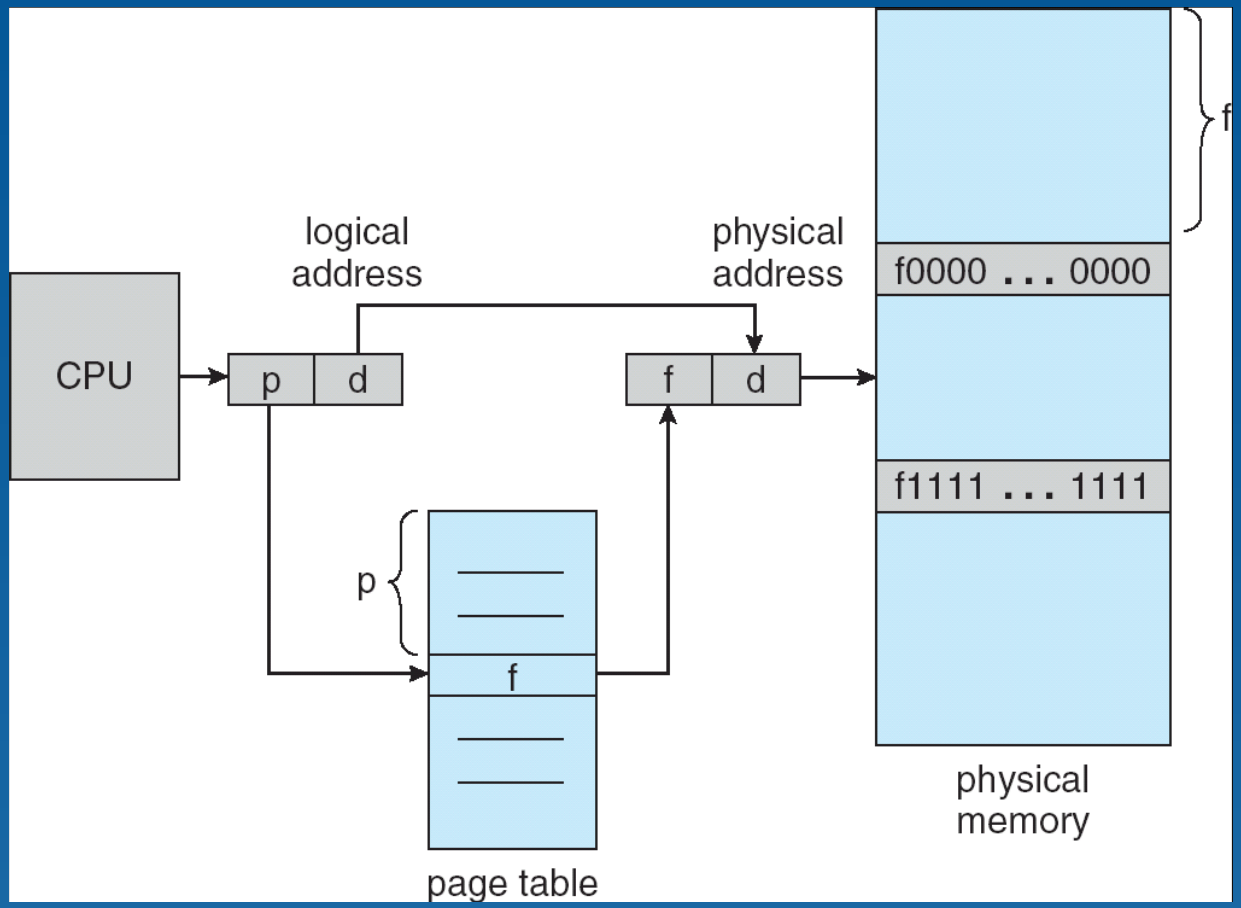


逻辑地址至物理地址的地址翻译

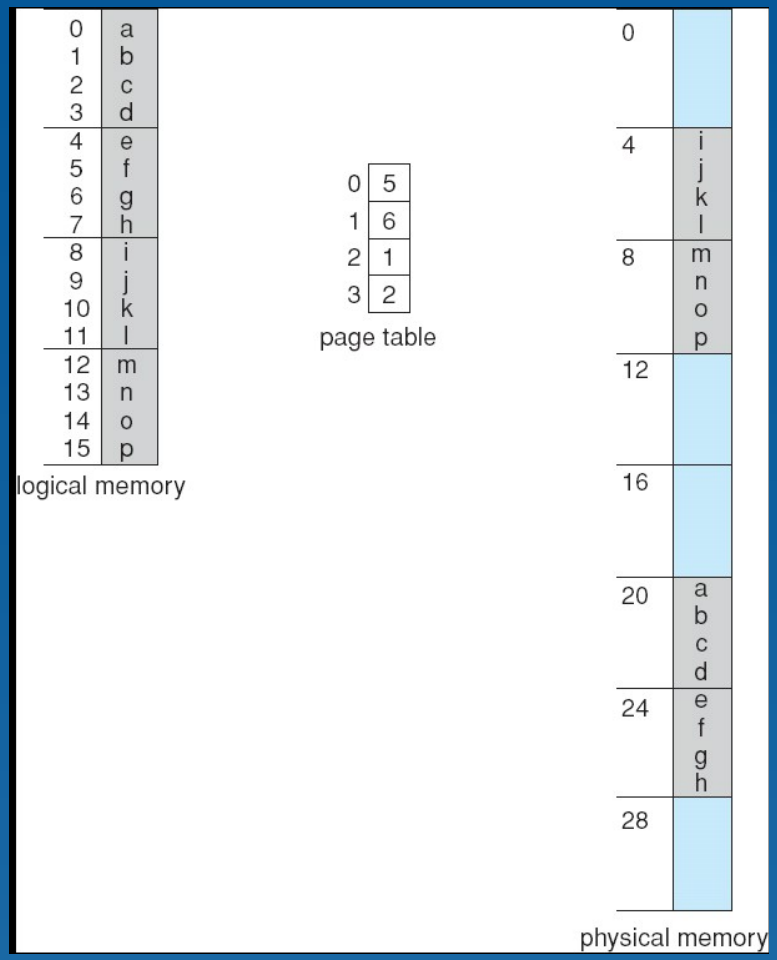
- 假设逻辑地址空间 2^m ，页长 P 为 2^n
- CPU 提供的逻辑地址 addr 区分为两个部分
  - **页号 (*p*)** – *p*=addr/P 。作为下标查询页表 (*page table*) 中目标单元，该单元包含对应于物理空间的页帧的基地址
  - **页内偏移量 (d)** – *d*=addr%P 。该地址在对应页帧的偏移位置



地址翻译


CPU
logical address
p
d
physical address
f
d
f
p
page table
f0000 . . . 0000
f1111 . . . 1111
physical memory

示例：地址翻译

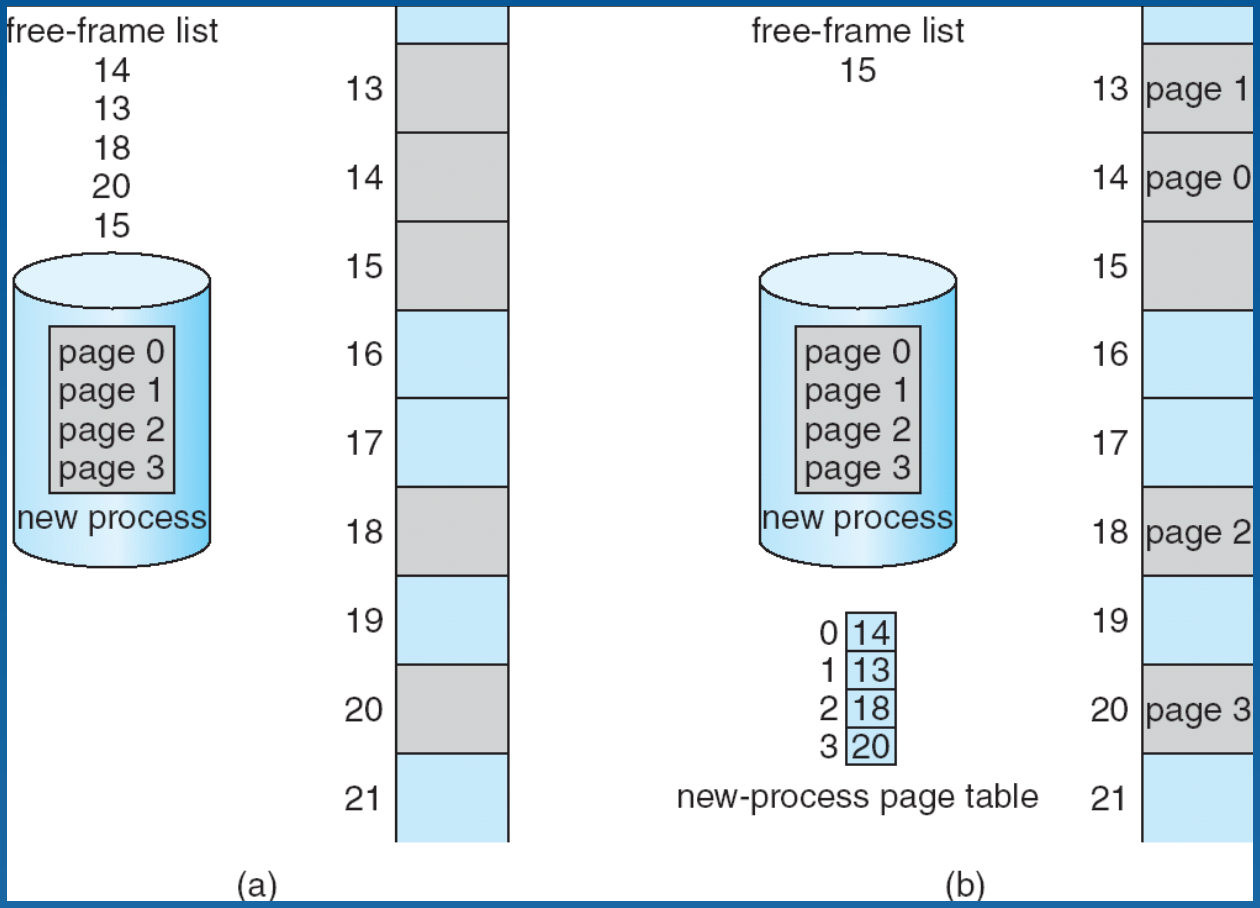


物理空间 32 字节，页长 4 字节

页式内存管理（续）

- OS 负责监控所有空闲页帧
- 若进程需要 ***n*** 页逻辑空间， OS 分配 *n* 个空闲页帧给它，装入代码和数据
- OS 分配页表需要的物理空间，布置好页表（就是 f 函数）
- 页式内存管理存在 Internal fragmentation 问题

空闲页帧


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

分配前

分配后

如何实现页表

- 页表必须驻留内存。为什么？

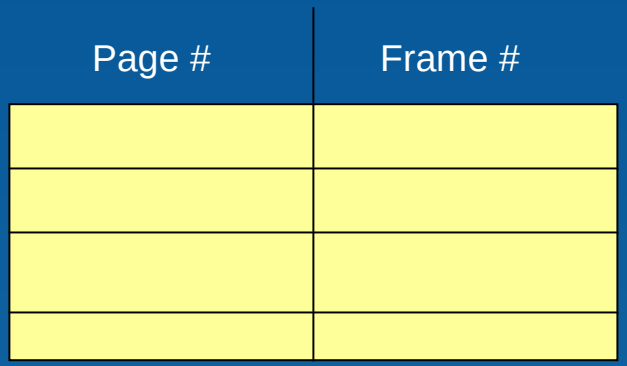
- 页表基地址寄存器 **Page-table base register (PTBR)** 指向页表的首地址
- 页表长度寄存器 **Page-table length register (PRLR)** 表示页表占用的空间长度

如何实现页表(续)

- 访问一个数据 / 地址，需 2 次内存访问！
  - 1 次访问页表
  - 1 次访问数据 / 地址本身
- 解决 2 次访问问题，借助 **translation look-aside buffers (TLBs)**

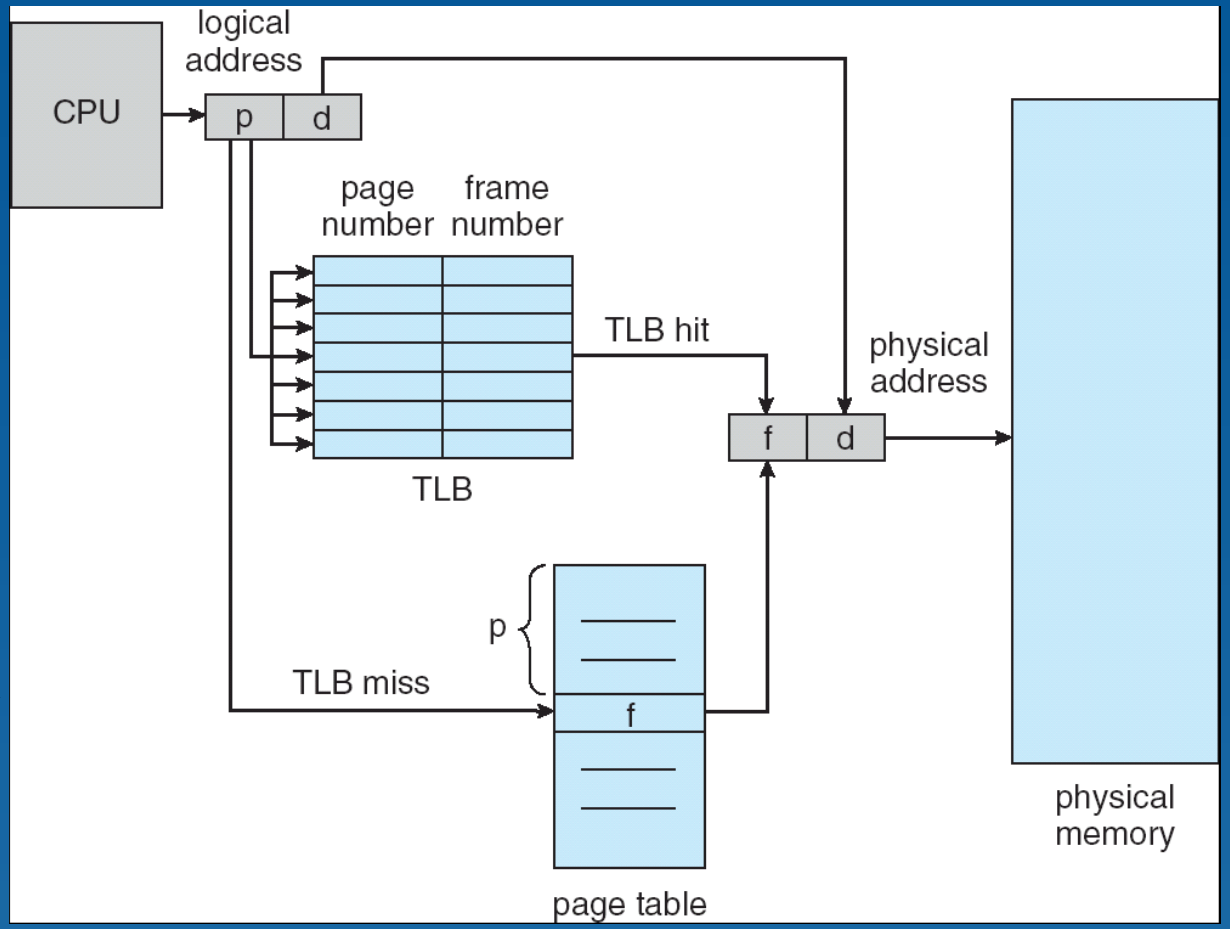
TLB

- 也称关联存储器 Associative memory
- 其特征 – 并行搜索



- TLB 用于对 (p, d) 的地址翻译
  - 如果 p 恰好在 TLB(hit)，直接从 TLB 得到页帧号
  - 否则 (fail)，从内存页表中取得页帧号

有 TLB 参与的地址翻译



有效访问时间 (Effective Access Time)

- 设 TLB 的查询时间 = ε 单位时间
- 假设内存访问周期为 1 微秒
- 命中率 (Hit ratio)
- 成功地在 TLB 取得页号的百分率;
- 命中率与 TLB 的单元总数有关
- 设命中率 = α
- 有效访问时间 (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

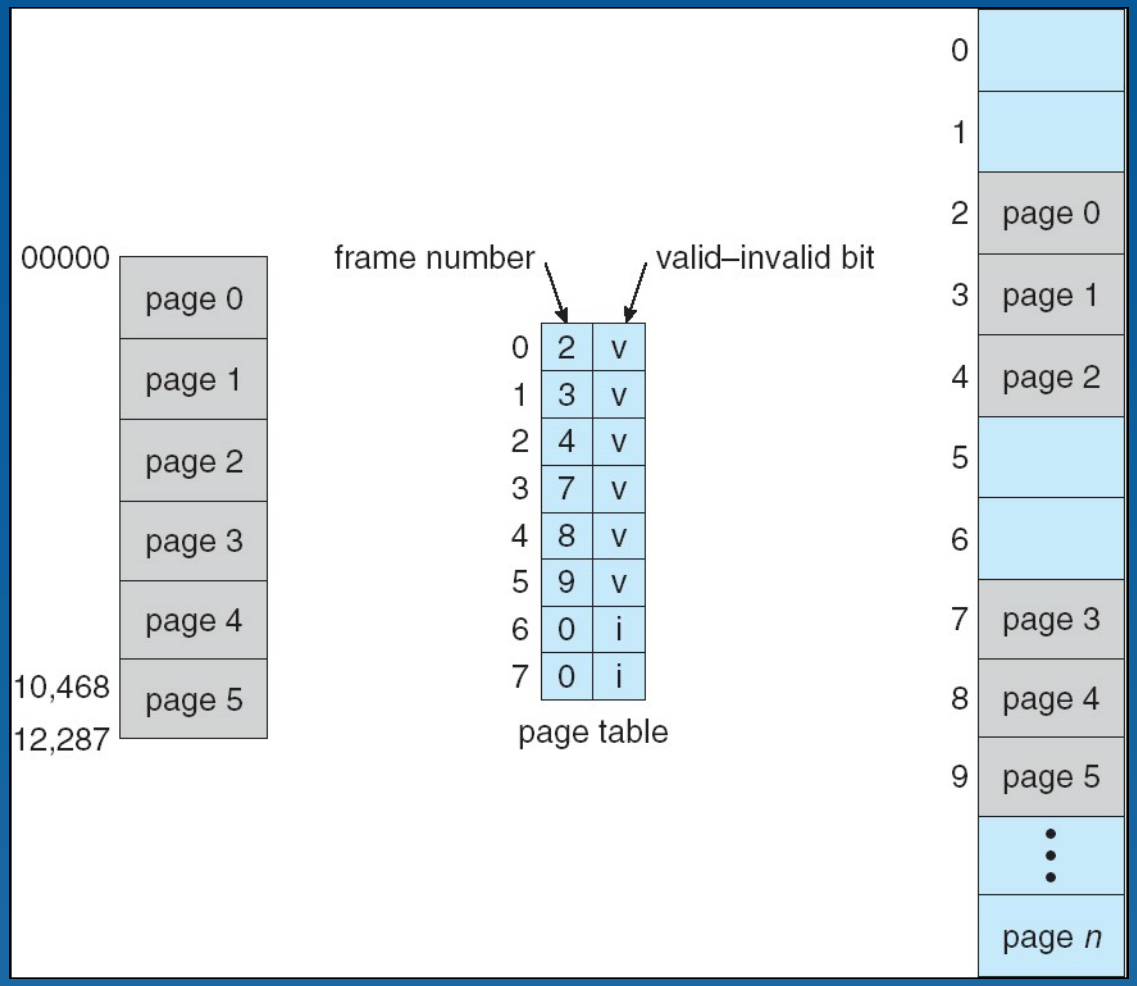
$$= 2 + \varepsilon - \alpha$$

内存保护

- 在进程页表的每个页表项中，为每个页设置一个保护位 **Valid-invalid** bit

- 页表项的“有效 - 无效”位
  - “有效”表示该页面在进程的逻辑地址空间范围内，因此是合法页面
  - “无效”表示该页面不在进程的逻辑地址空间范围内

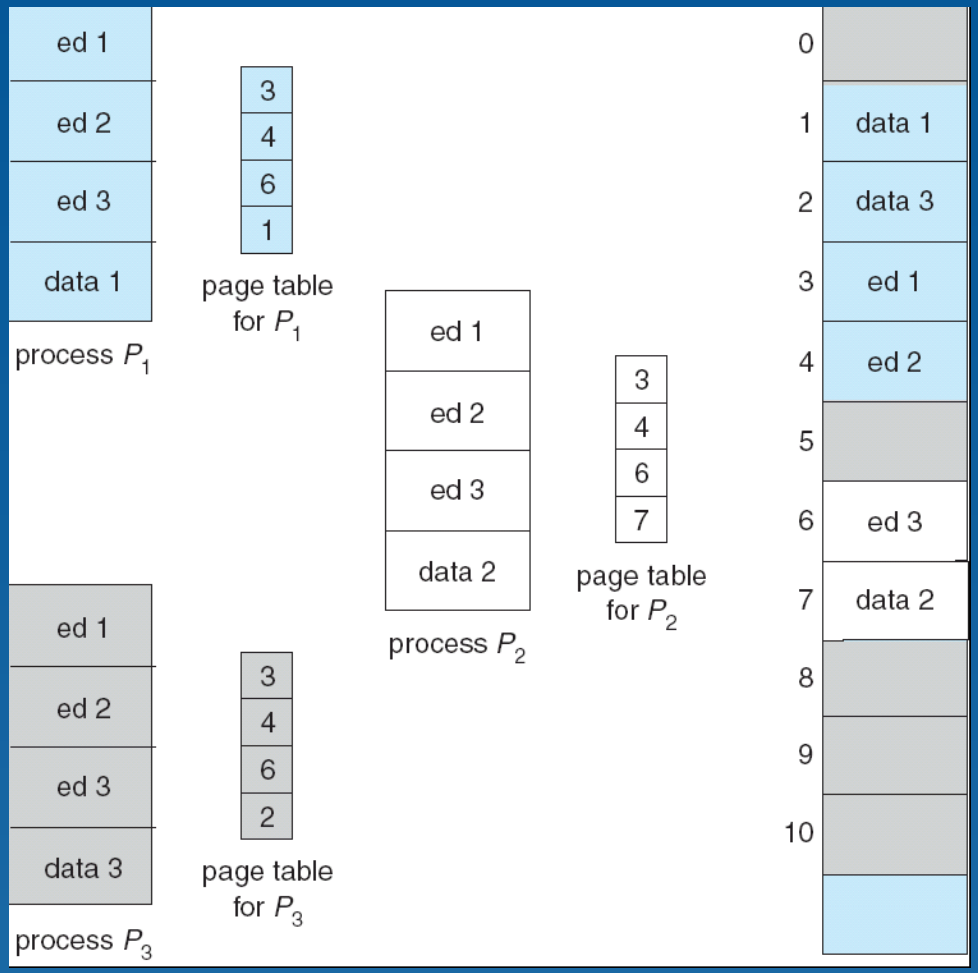
示例：页表项的有效位 (v)、无效位 (i)



共享页面

- **共享代码**
  - 只读 (可重入) 代码只需要一份，供若干进程共享 (i.e.，文本编辑器、编译器、窗口系统)
  - 对所有进程来说，共享代码必须位于逻辑地址空间的相同位置
- **进程自有代码和数据**
  - 进程各自拥有一份
  - 为自有代码、数据分配的页面，可以分布在进程逻辑地址空间的任意位置

示例：共享页面


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10

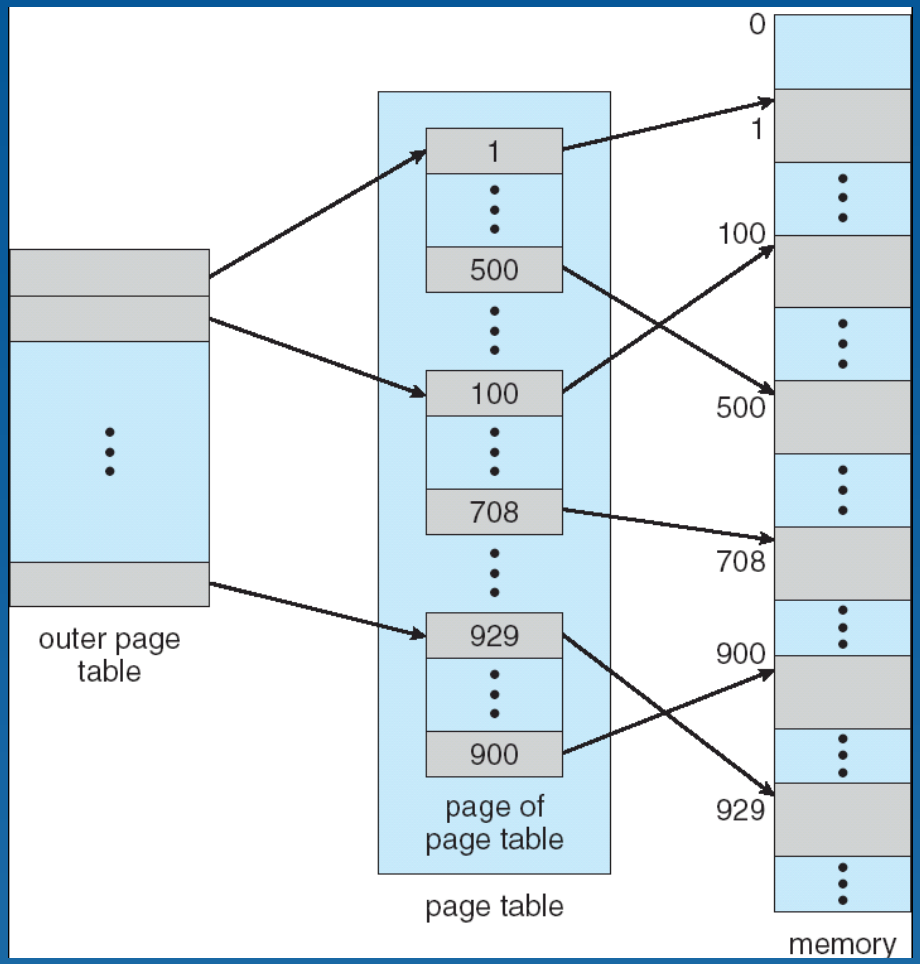
页表的数据结构

- Hierarchical Paging
- Hashed Page Table
- Inverted Page Table

多级页表 (Hierarchical Page Table)

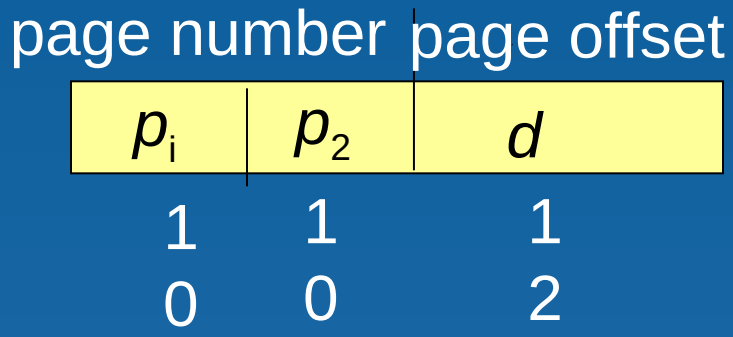
- 将页表的逻辑地址拆分成多张页表
- 一种简单的技巧：二级页表

二级页表策略


0
1
100
500
708
900
929
1
500
100
708
929
900
outer page table
page of page table
page table
memory

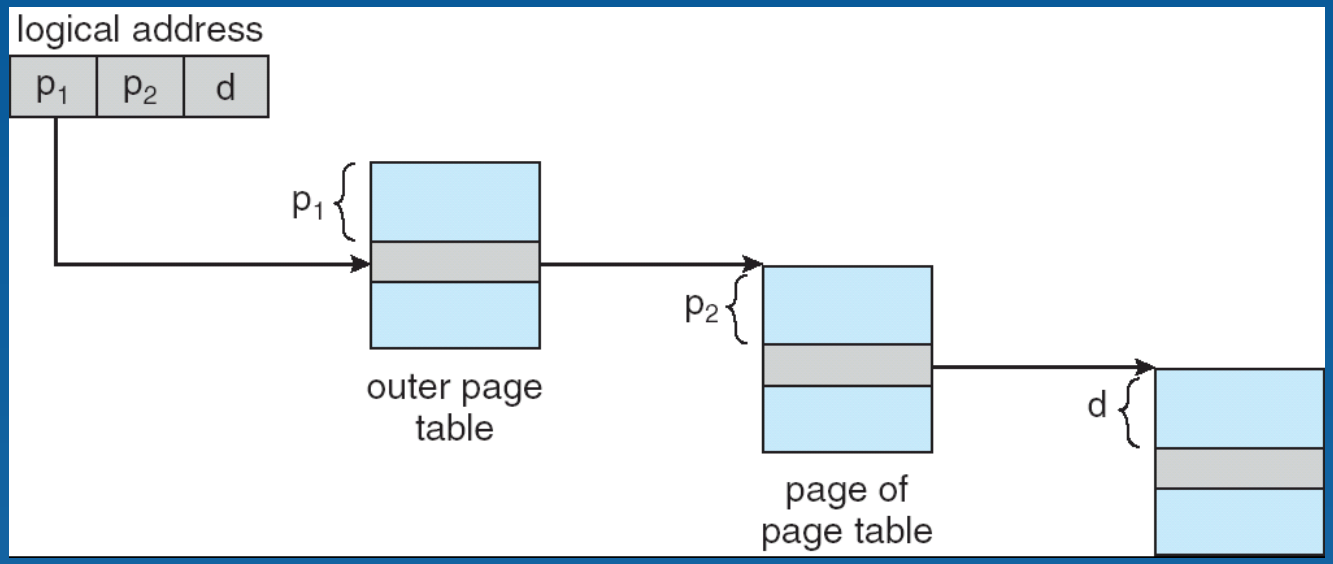
示例：二级页表

- 逻辑地址 (32 位 CPU ，页长 4KB) 分割成两部分
  - 页号， 20 位
  - 页内偏移量， 12 位
- 页表被进一步分页，其页号分割成两部分
  - 页号的页号， 10 位
  - 页号的页内偏移量， 10 位
- 因此，一个逻辑地址分割成三部分



- 其中， p_i 是外层页表的下标， p_2 是外层页表内部之位移

二级页表策略的地址翻译



三级页表的策略

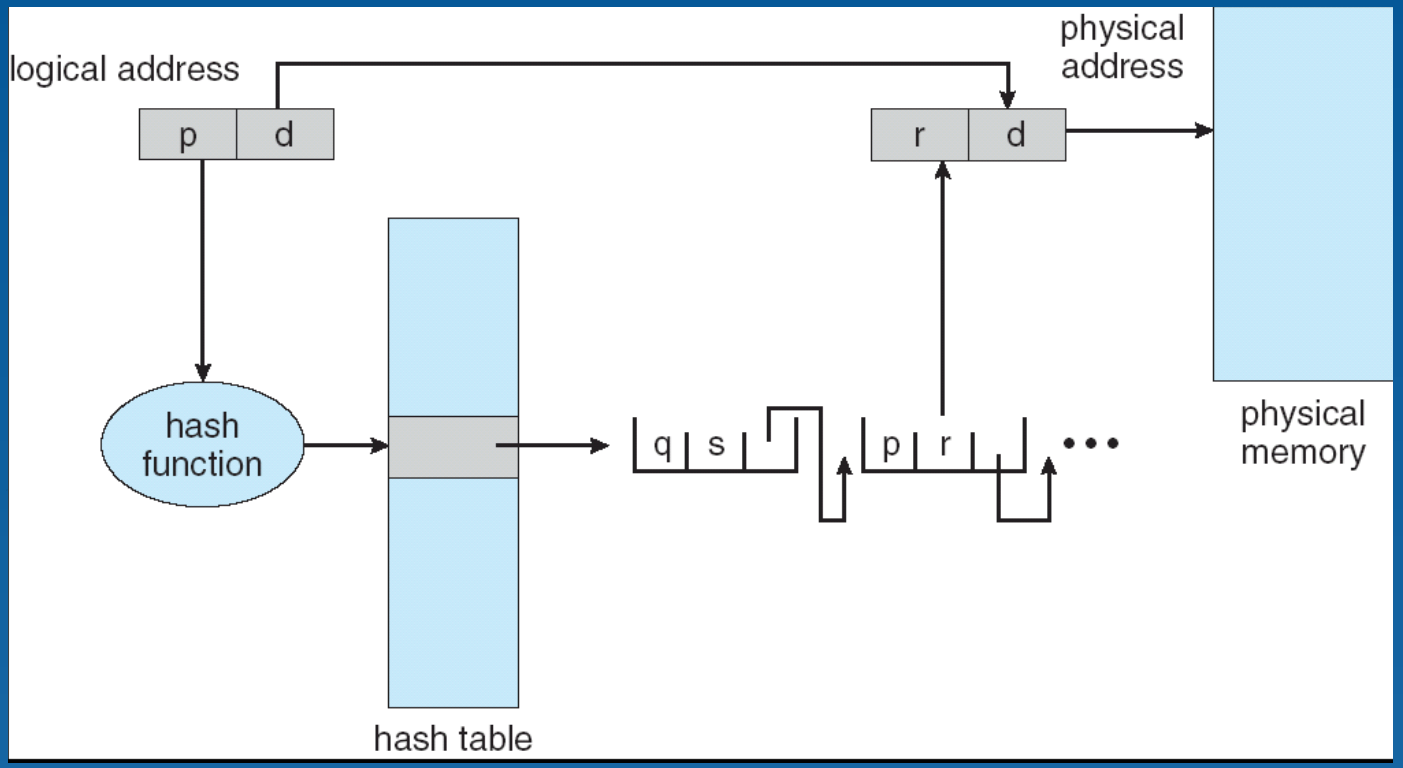
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Table

- 多见于地址空间大于 32 位的 CPU
- 虚拟页号经过哈希函数转换后，指向页表中某个页表项
- 哈希函数值相同的虚拟页号，指向同一个页表项，它们在那个页表项下组成一个链表
- 地址翻译时，由虚拟页号哈希后锁定对应链表，搜索与虚拟页号的匹配项
- 如果找到匹配项，则找到了虚拟页号对应的物理页帧

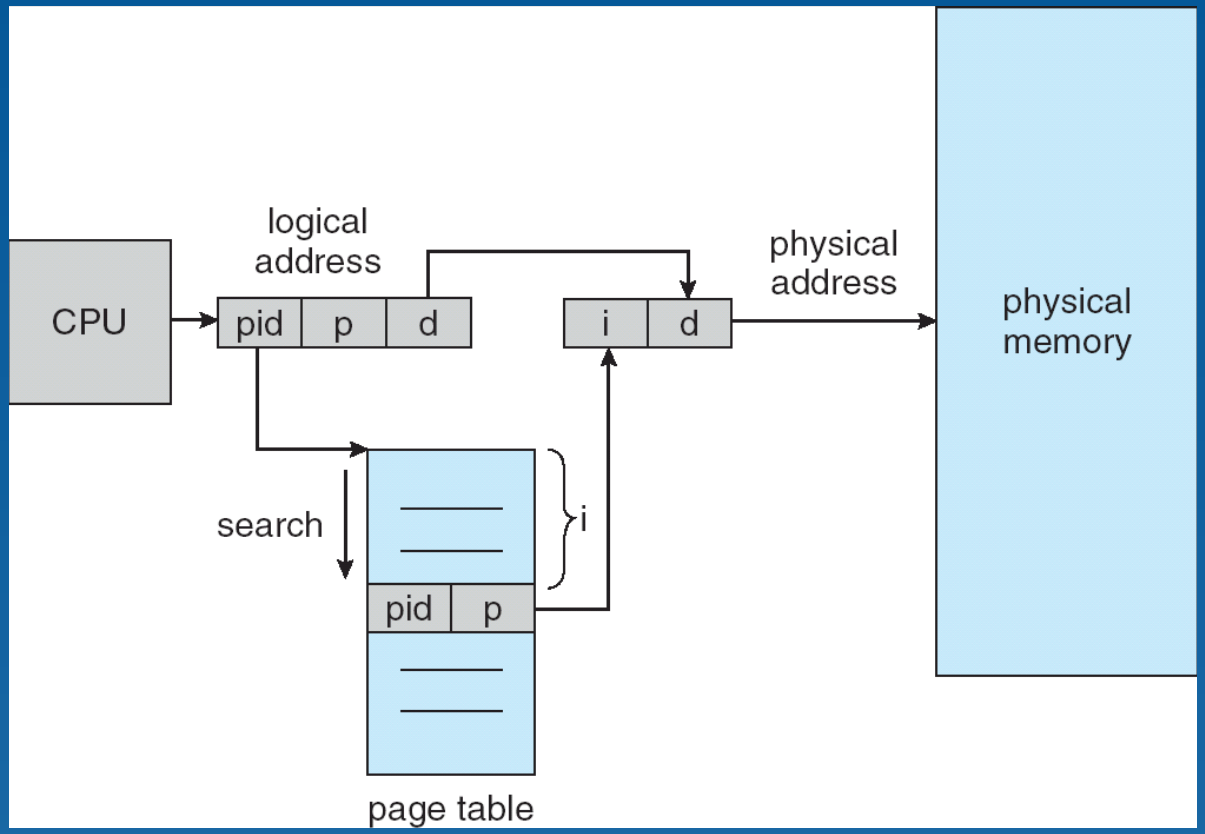
Hashed Page Table



Inverted Page Table

- 每个物理页帧，对应 Inverted Page Table 的一个表项
- 对于每个表项，它表示的物理页帧存储了某个进程的一个逻辑页。表项内容包含该进程 id 、页号
- 对比传统页表，该方法的页表空间大幅度减少
- 但是，查找页表项的时间明显增加
- 利用哈希表，使得查页表操作能一次命中，或者耗费较少的查找次数

Inverted Page Table




End