

第8章 存储管理

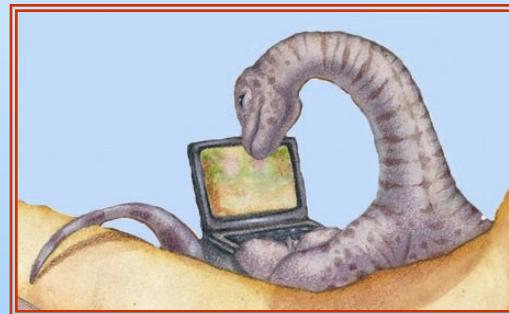




内容

1. 背景知识
2. 连续分配
3. 分页
4. 分段
5. 段页式
6. 各种内存管理方案的比较

1、背景知识

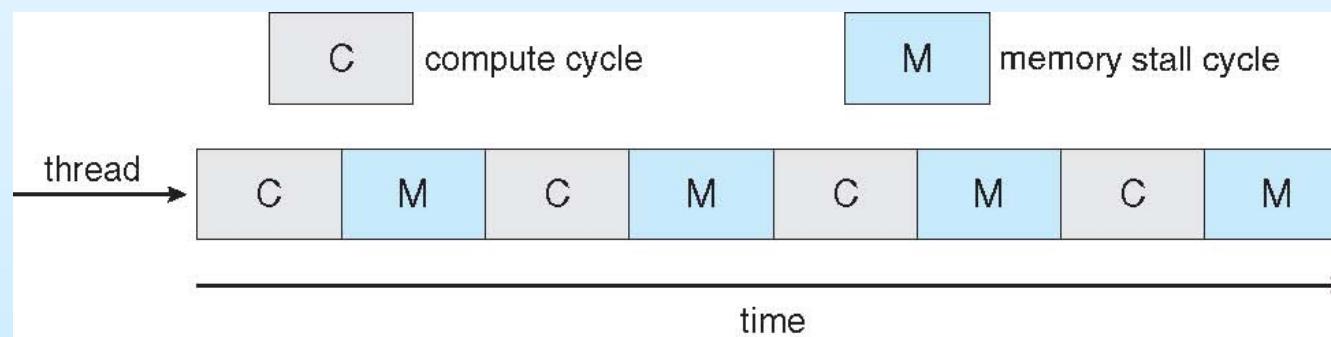


基本硬件
地址绑定
逻辑地址与物理地址
动态加载
动态链接



基础知识

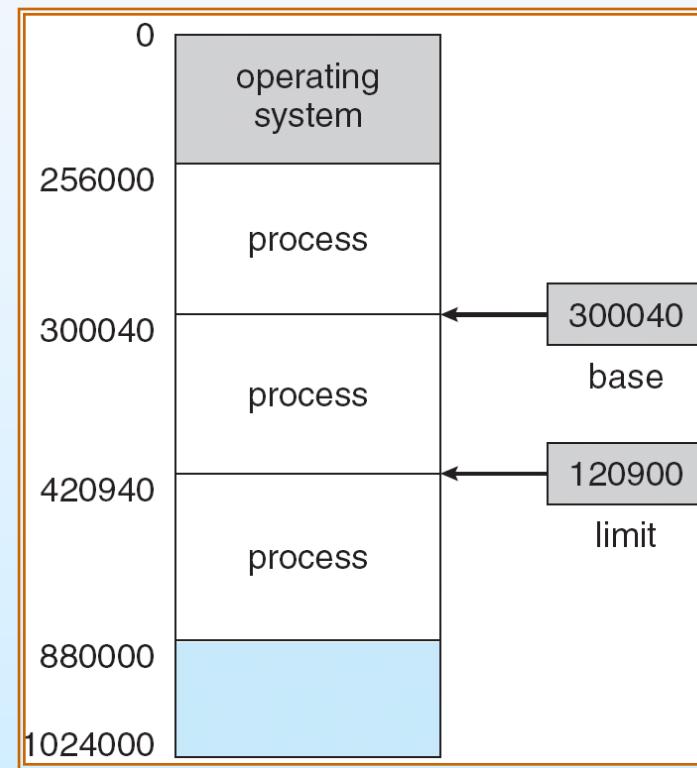
- 内存是现代计算机运行的中心，它是由字或字节组成，每个字或字节都有自己的地址，程序必须装入内存才能被执行
- CPU可以直接访问的存储器只有主存和寄存器
 - 寄存器通常可以在一个（或少于一个）CPU时钟周期内完成访问
 - 完成主存访问可能需要多个CPU时钟周期
- CPU暂停（Stall）：在读取内存数据时，CPU空闲
- 解决方案：在内存和CPU之间，增加高速内存来协调速度差异，这种内存缓冲区称为高速缓存Cache
- 内存保护需要保证正确的操作（操作系统内核不被用户进程访问）





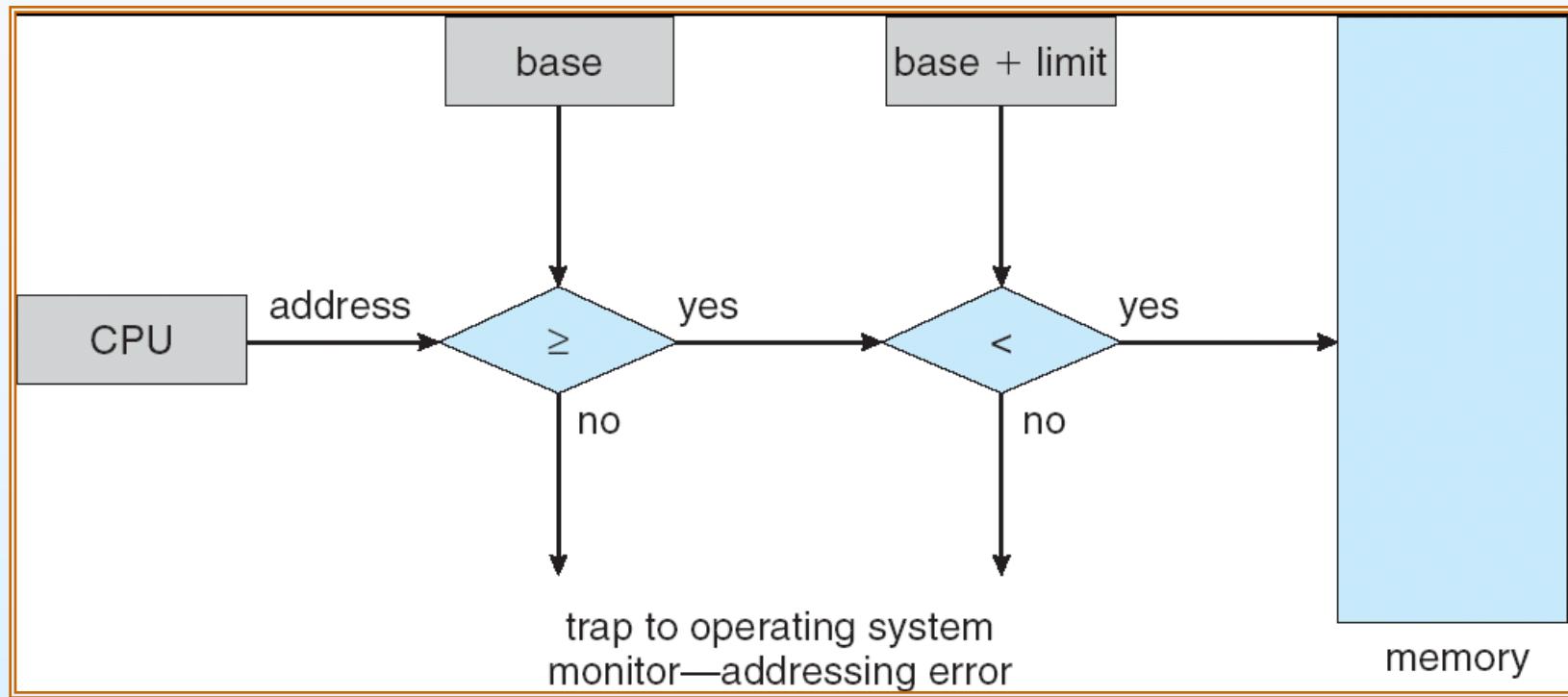
独立运行内存空间

- 基址寄存器（**Base**）：进程最小的合法物理内存地址
- 界限寄存器（**Limit**）：进程地址的长度
- CPU在执行指令时，需要进行地址合法性验证





基址寄存器和界限寄存器的硬件支持





逻辑地址和物理地址

■ 逻辑地址 *Logical address*

- 由CPU产生
- 在进程内的相对地址
- 也称：虚拟地址、程序地址、相对地址

■ 物理地址 *Physical address*

- 内存单元看到的地址
- 所有内存统一编址
- 也称：绝对地址、实地址





逻辑地址和物理地址

- 编译和加载时的地址绑定生成同样的逻辑地址和物理地址，此时重定位是静态的；
- 执行时的地址绑定导致不同的逻辑地址和物理地址，此时的重定位称为动态重定位

- 由程序所产生的所有逻辑地址的集合称为**逻辑地址空间**
- 与逻辑地址所对应的所有物理地址的集合称为**物理地址空间**

- 逻辑地址空间绑定到物理地址空间至关重要，是正确进行内存管理的中心。





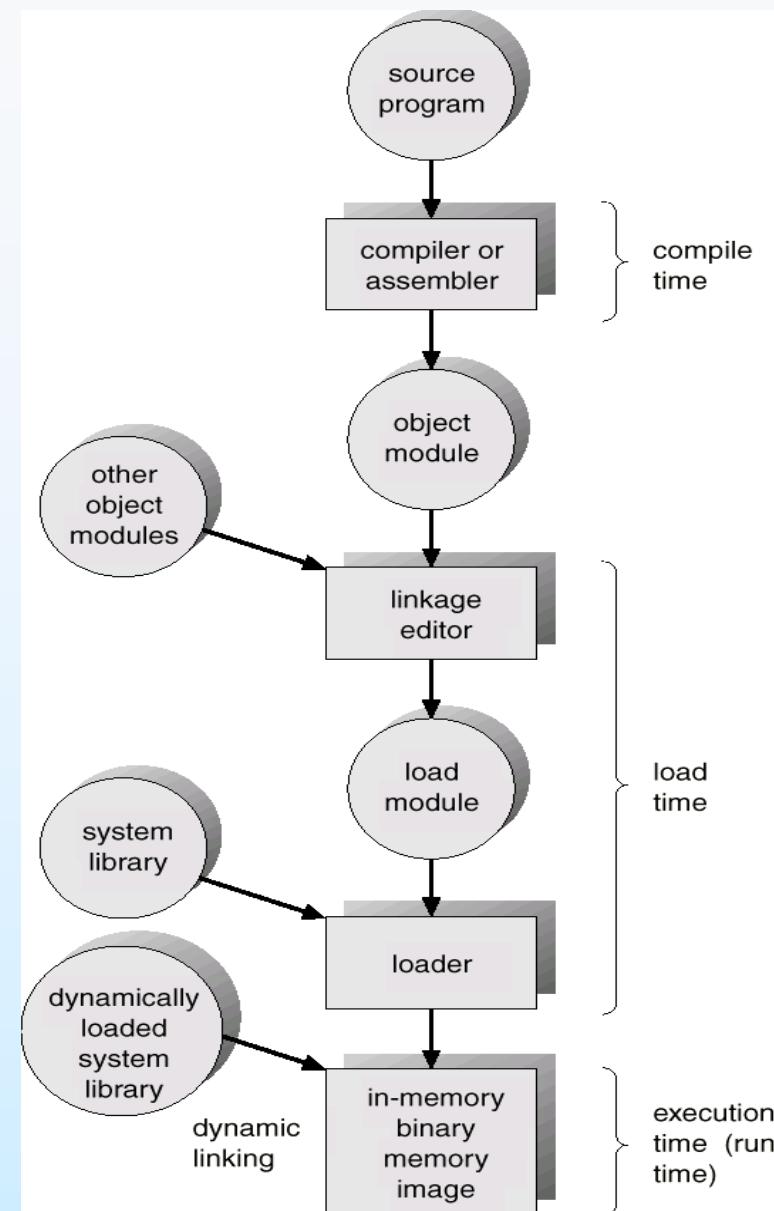
指令和数据绑定到内存

- 程序以二进制可执行文件的形式存储在磁盘上，为了执行，程序被调入内存并放在程序空间内
- 地址绑定（重定位）：在程序装入内存时，把程序中的相对地址转换为内存中的绝对地址的过程
- 指令和数据绑定到内存地址可在三个不同阶段：
 - 编译时期（**Compile time**）
 - ▶ 如果内存位置已知，可生成绝对代码（**absolute code**）
 - ▶ 如果开始位置改变，需要重新编译代码
 - 加载时期（**Load time**）
 - ▶ 如果存储位置在编译时不知，则必须生成可重定位代码（**relocatable code**），绑定会延迟到加载时进行
 - 执行时期（**Execution time**）
 - ▶ 如果进程执行时可在内存移动，则地址绑定可延迟到执行时
 - ▶ 需要硬件对地址映射的支持（例如基址和界限寄存器）





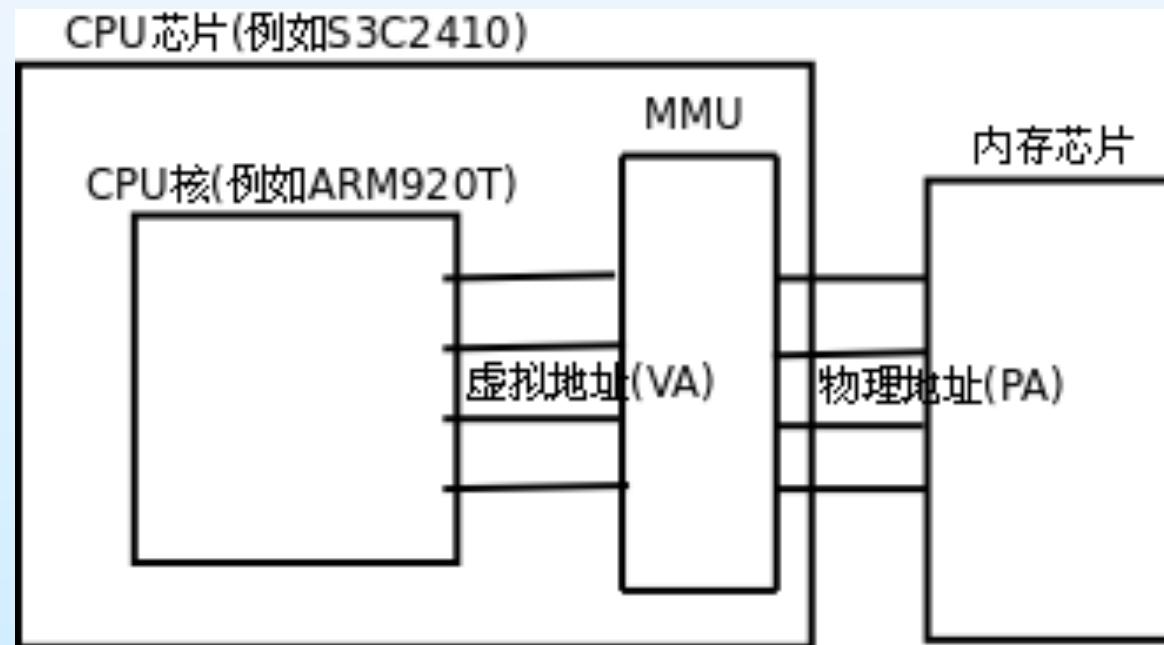
地址绑定的三个阶段





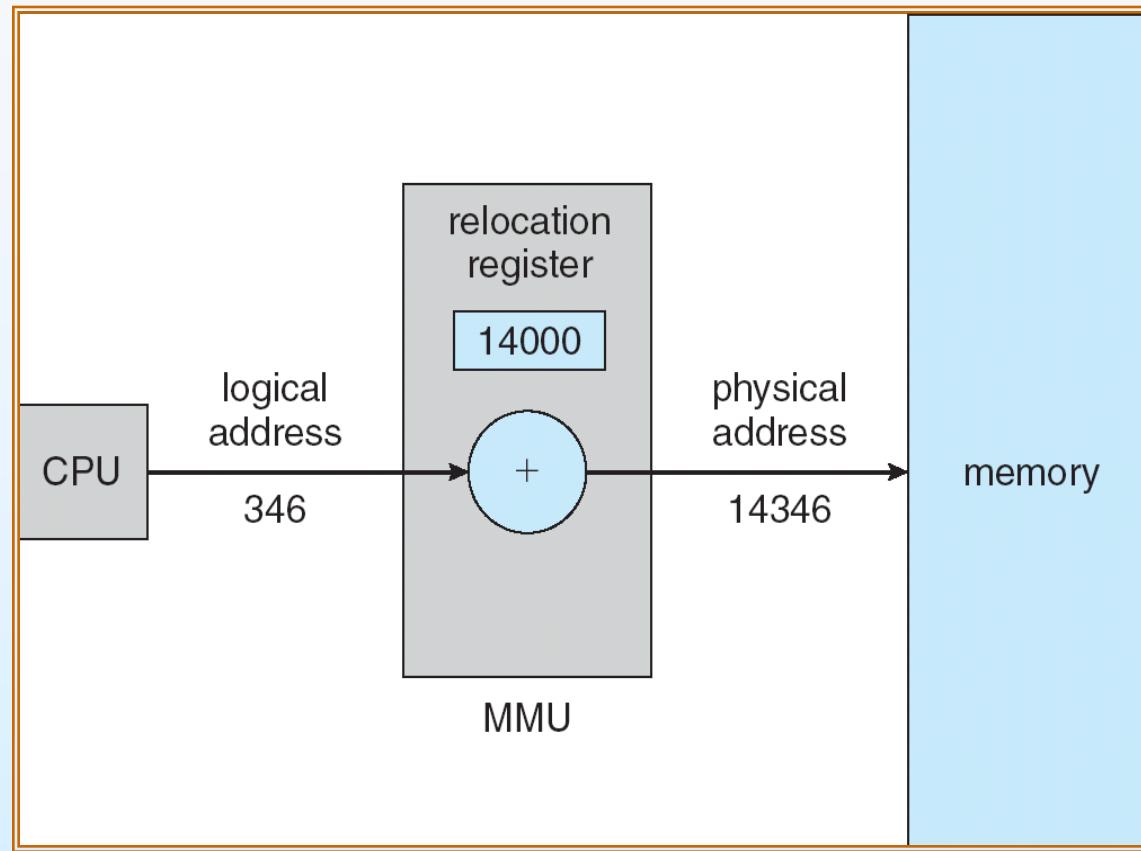
内存管理单元 (MMU)

- 内存管理单元是把虚拟地址映射到物理地址的硬件
- 是CPU用来管理内存的控制线路
- 在MMU策略中，基址寄存器中的值在其送入内存的时候被加上重定位寄存器的值
- 用户程序所对应到的是逻辑地址，绝不会看到真正的物理地址。





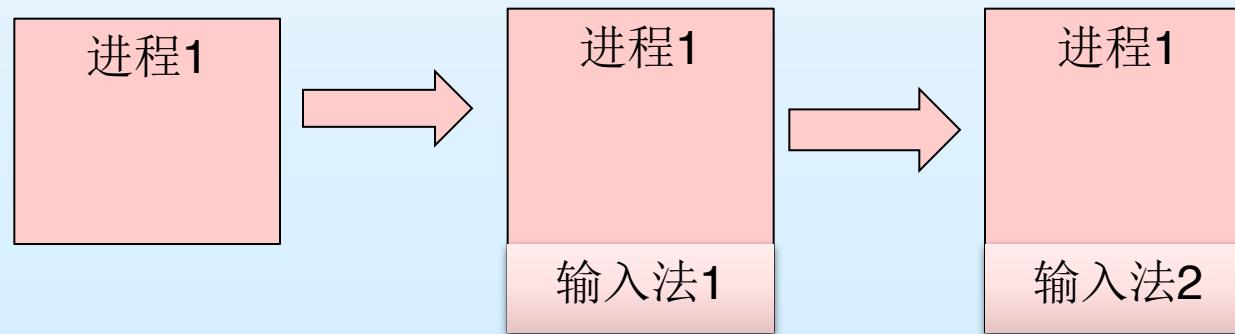
使用重定位寄存器的动态重定位





动态加载(Dynamic Loading)

- 例程在调用之前并不加载
- 优点：
 - 更好的内存空间利用率
 - 没有被使用的例程不被载入
 - 当需大量代码来处理不经常使用的功能时非常有用
- 不需要操作系统的特别支持，通过程序设计实现
- Windows 的动态链接库





动态链接(Dynamic Linking)

- 和各种库文件的链接被推迟到执行时期，称为动态链接技术
- 这一技术通常用于系统库，如语言子程序库，可以减少磁盘空间和内存空间
- 需要动态装载技术支持
- 一小段代码 - 存根，用来定位合适的驻留在内存中的库程序
- 存根用程序地址来替换自己，并开始执行程序
- 操作系统需要检查程序是否在进程的内存空间，所以需要操作系统支持





链接例子

- 静态链接：在生成可执行文件时，把sum函数相关的二进制指令和数据包含在最终的可执行文件中
- 动态链接：在程序运行时加载sum函数相关的二进制指令和数据

```
/* m.c */
int i = 1;
int j = 2;
extern int sum();
void main()
{
    int s;
    s = sum(i, j);
/* f.c */
int sum(int i, int j)
{
    return i + j;
}
```

SYMBOL TABLE:					
.....					
00000000 g	0 .data	00000004	i		
00000004 g	0 .data	00000004	j		
00000000 g	F .text	00000021	main		
00000000	*UND*	00000000	sum		
7: R_386_32	j				
b: 50			push	%eax	
c: a1 00 00 00 00			mov	0x0,%eax	
d: R_386_32	i				
11: 50			push	%eax	
12: e8 fc ff ff ff			call	13	
13: R_386_PC32	sum				
80482d6: e8 0d 00 00 00			call	80482e8	
80482db: 83 c4 08			add	\$0x8,%esp	





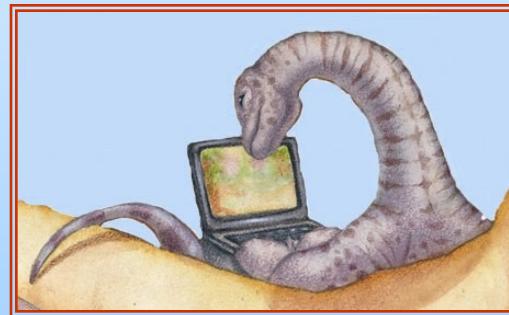
动态链接、加载、绑定

- 动态链接技术的实现依赖于动态加载和动态绑定
- sum需要动态加载装入进来，需要动态绑定技术找到物理地址
- 动态绑定：逻辑地址到物理地址的绑定
- 动态链接：符号名到地址的转换





2、连续分配





存储管理的功能

- 内存分配
- 内存回收
- 地址绑定
- 存储保护
- 存储共享



连续内存分配

- 连续内存分配为每一个用户程序分配一个连续的内存空间
- 早期内存分配模式，运用于内存较少系统
- 三个类型
 - 单一连续分配、固定分区分配、可变分区分配
- 主存通常被分为两部分
 - 为操作系统保留部分，中断向量通常保存在内存低端
 - 用户进程保存在内存高端
 - 操作系统可以位于内存低端，也可位于高端，影响这一决定的主要因素是中断向量的位置，由于中断向量通常保存在内存低端，因此操作系统通常也驻留在内存低端





连续内存分配

■ 单一连续分配

- 单道程序环境下，仅装有一道用户程序，即整个内存的用户空间由该程序独占
- 内存分配管理简单，内存利用率低
- 用于单用户、单任务的操作系统
- CP/M、MS-DOS、RT11
- 实际使用中，并未采用存储器保护措施，这样做可以节省硬件，同时又不影响系统安全性





连续内存分配

■ 固定分区分配

- 最早、最简单的可运行多道程序的内存管理方式
- 预先把可分配的内存空间分割成若干个连续区域，称为一个分区
- 每个分区的大小可以相同也可以不同，分区大小固定不变，每个分区装一个且只能装一个程序
- 内存分配时，如果有一个空闲分区，则分配给进程；进程运行结束时，其分区被收回，重新分配给其他进程

■ 划分分区两种方法：分区大小一样或不一样

- **分区大小一样：** 缺点是缺乏灵活性，程序太小浪费内存，程序太大又可能装不下。在某些场合，如利用计算机控制某些相同对象时比较适合。
- **分区大小不同：** 为了高效使用，可以设置多个小分区，适量中分区，少量大分区。





连续内存分配

固定分区使用表

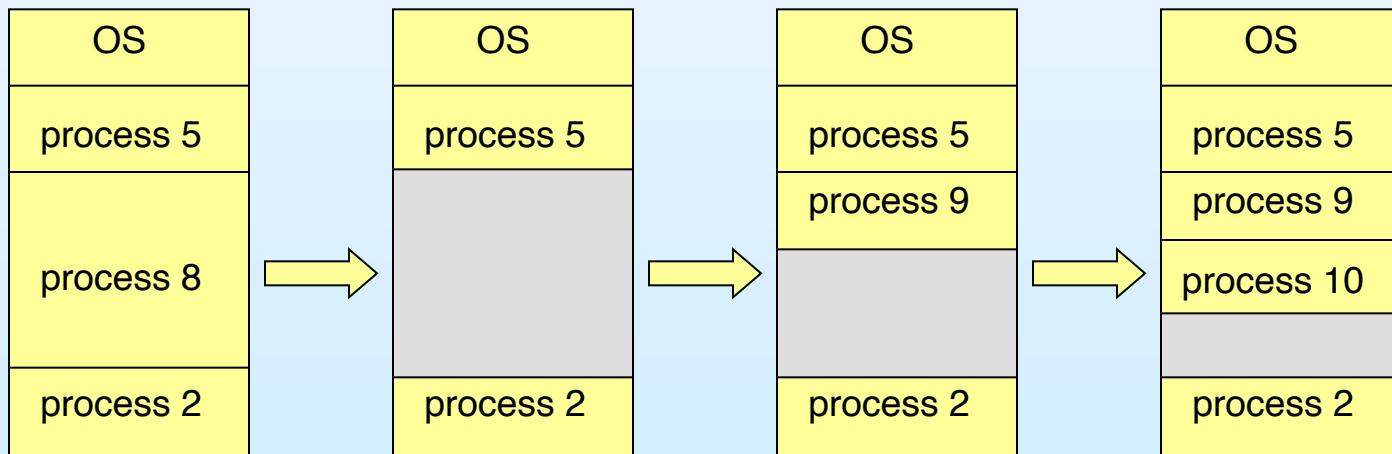
分区号	大小(K)	起始地址(K)	状态
1	12	20	pidA
2	32	32	pidB
3	64	64	pidC
4	128	128	未分配





连续内存分配

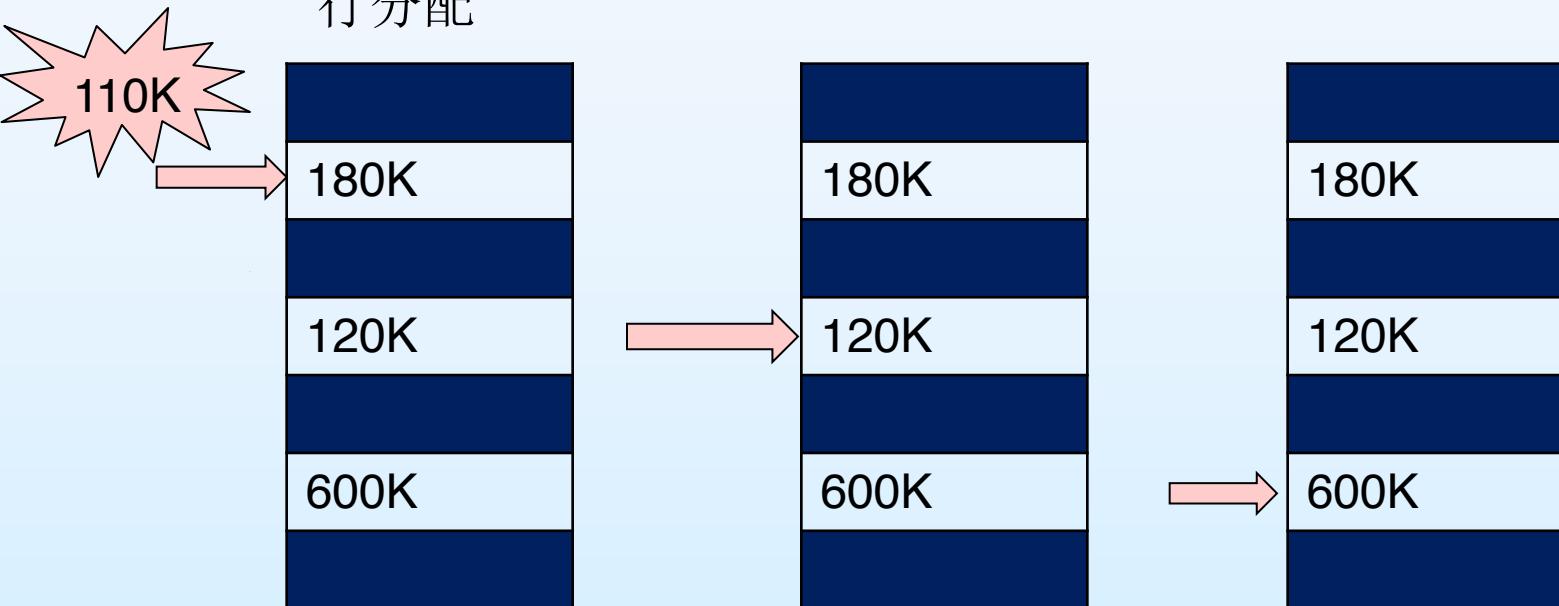
- 可变分区分配，是固定分区方案的延伸，主要用于批处理系统
- 分区(孔，*Hole*)—可用的内存块，不同大小的分区分布在整个内存中
- 根据进程的需要，动态的分配内存空间，即当一个进程到来的时候，它将从一个足够容纳它分区中分配内存，分区中未分配的内存仍然是可用的，可以下次再使用。
- 操作系统包含以下信息：
 - a) 已分配的分区 b) 空的分区-空闲分区表





存储分配算法

- 首次适应（First-fit）：分配最先找到的合适的分区
- 最佳适应（Best-fit）：搜索整个序列，找到适合条件的最小的分区进行分配
- 最差适应（Worst-fit）：搜索整个序列，寻找最大的分区进行分配



在速度和存储空间的利用上，首次适应和最佳适应要好于最差适应。首次适应法和最佳适应法在空间上利用差不多，但首次适应法更快些。

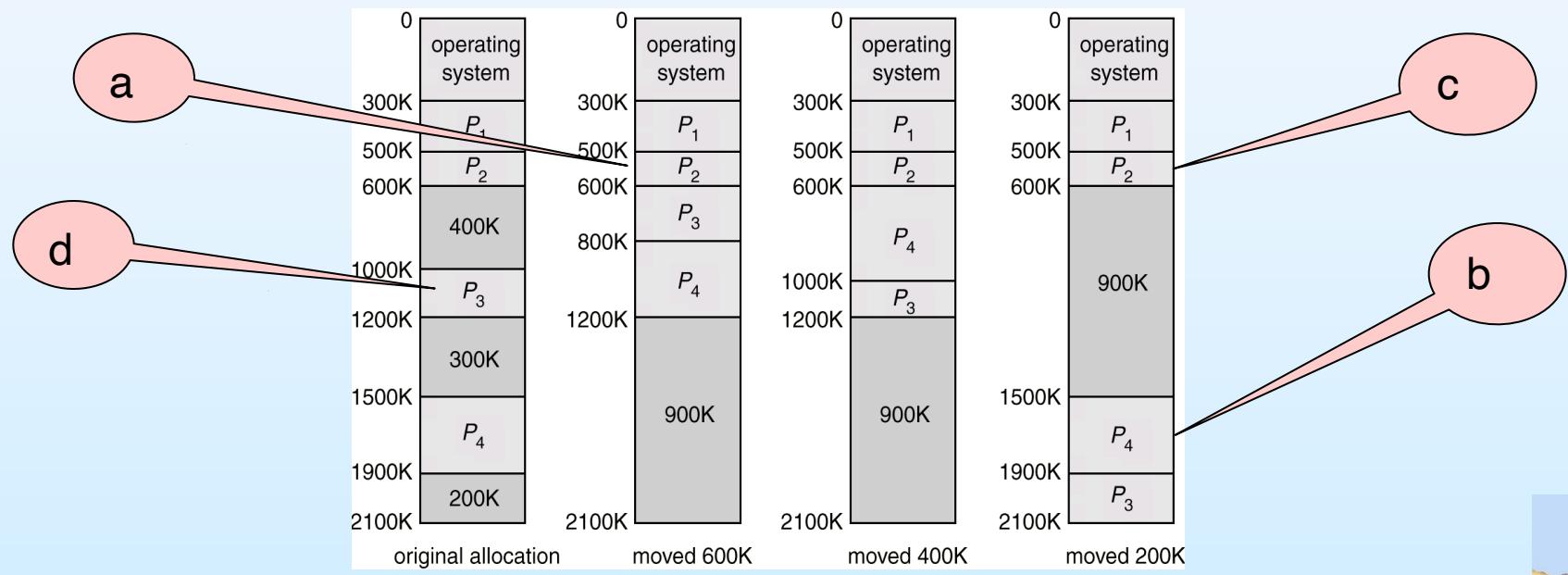




回收方法

■ 4种情况

- a. 回收内存块前后无空闲块
- b. 回收内存块前有后无空闲块
- c. 回收内存块前无后有空闲块
- d. 回收内存块前后均有空闲块





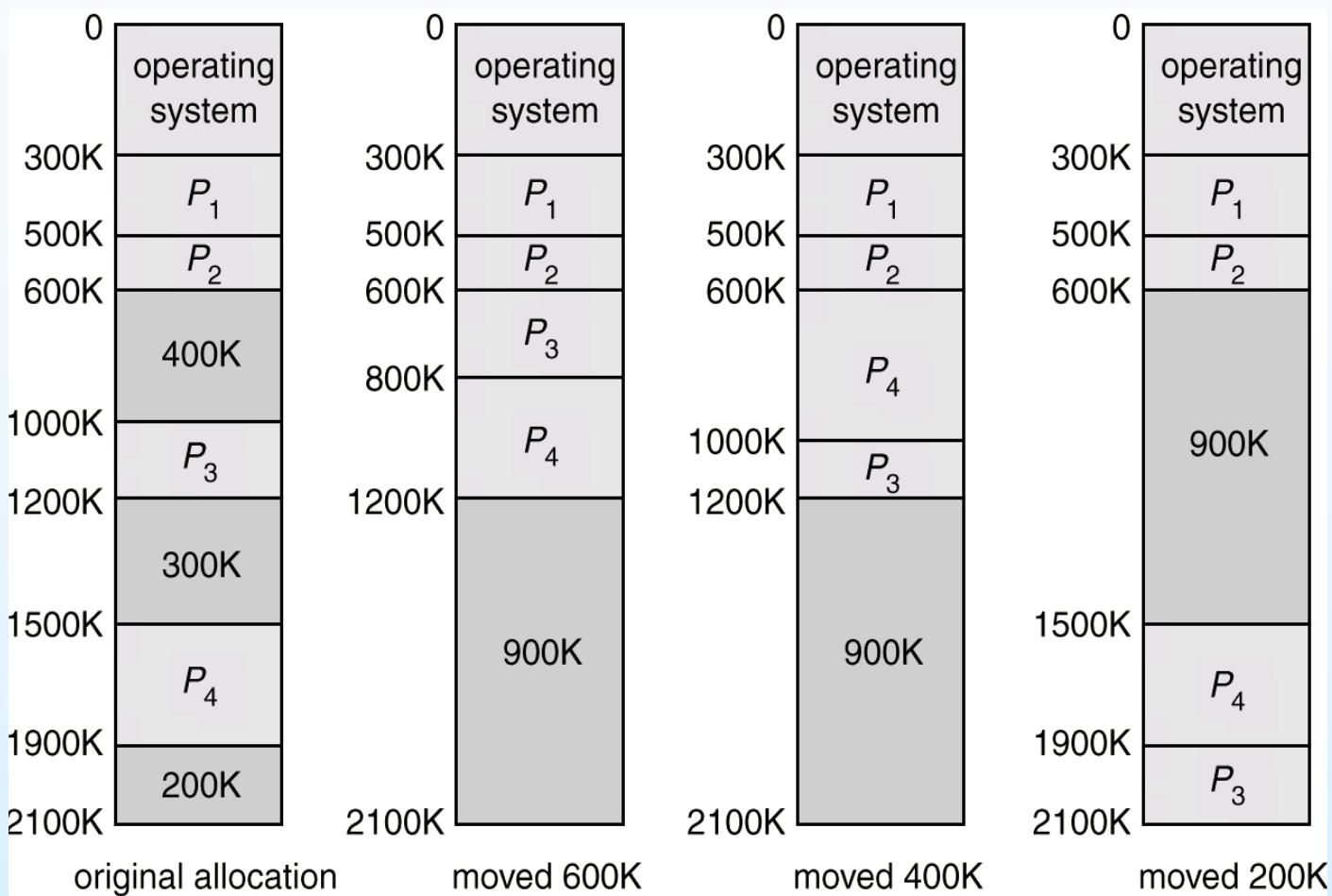
碎片

- 外碎片 - 整个可用内存空间可以用来满足一个请求，但它不是连续的
 - 首次适应法和最佳适应法都有这个问题，这个问题可能很严重，最坏的情况下，每两个进程之间都有空闲块被浪费
- 内碎片 - 分配的内存可能比申请的内存大一点，这两者的数字差称为内碎片，这部分内存存在分区内部，但又不被使用
- 可通过紧缩来减少外碎片
 - 把一些小的空闲内存结合成一个大的块。
 - 只有重定位是动态的时候，并且在运行时，才有可能进行紧缩
 - 最简单的合并算法是简单地将所有进程移动到内存的一端，所有的空闲分区移动到另一端，这种方法开销较大。为减少开销，可以选择移动内容最小的一种。



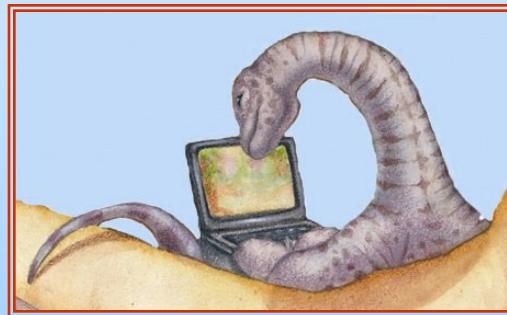


紧缩例子





3、分页





离散内存管理方案

- 分页内存管理方案—现代操作系统常用方案
- 分段内存管理方案
- 段页式内存管理方案



分页 (Paging)

- 内存管理方案允许进程逻辑地址空间不连续
 - 只要有可用的物理内存，就可以分配给进程
- 基本方法：
 - 把物理内存分成大小固定的块，称为帧 (**Frame**)，也叫页框
 - 把逻辑内存也分位固定大小的块，称为页 (**Page**)
 - 帧和页的大小是由硬件决定的，通常为2的幂，根据计算机结构不同大小不同，早期：512B-8KB，现在：4KB-64KB
- 系统保留所有空闲帧的记录
- 运行一个有N页的程序，需要找到N个空帧来装入程序
- 系统建立一个页表，记录页与帧之间的映射关系，进程运行时通过查找页表，实现逻辑地址转换为物理地址
- 传统上分页是由硬件处理的，最新的设计时通过将硬件和操作系统相配合来实现（尤其是64位的微处理器）





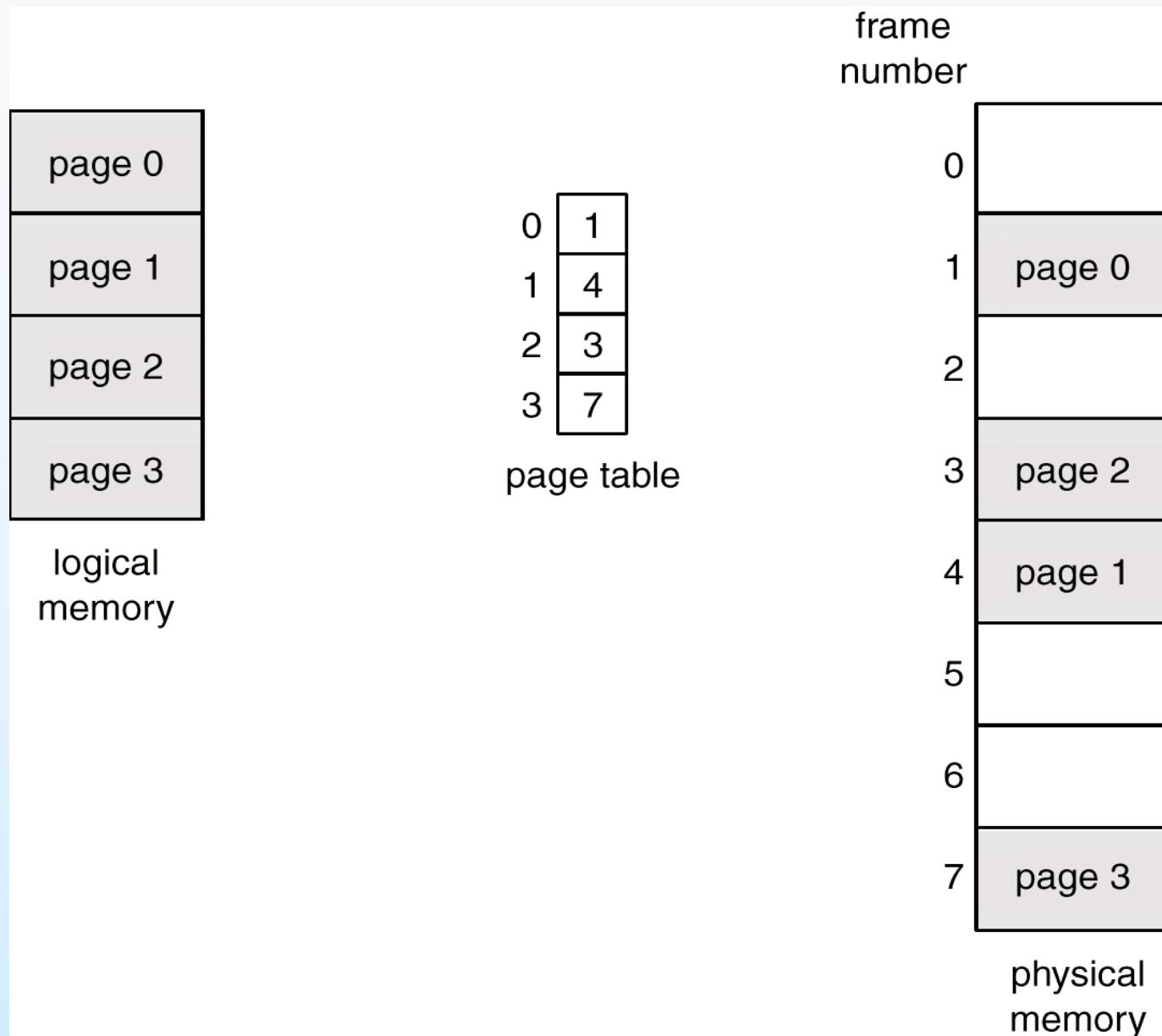
分页 (Paging)

- 采用分页技术不会产生外碎片，因为每个帧都可以分配给进程
- 分页有内碎片，进程请求的内存可能不是页的整数倍，因此最后一帧中可能存在多余的内存空间





逻辑内存和物理内存的分页模型

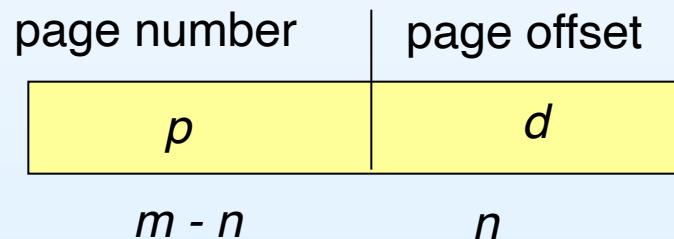




地址转换机制

■ 分页地址被分为：

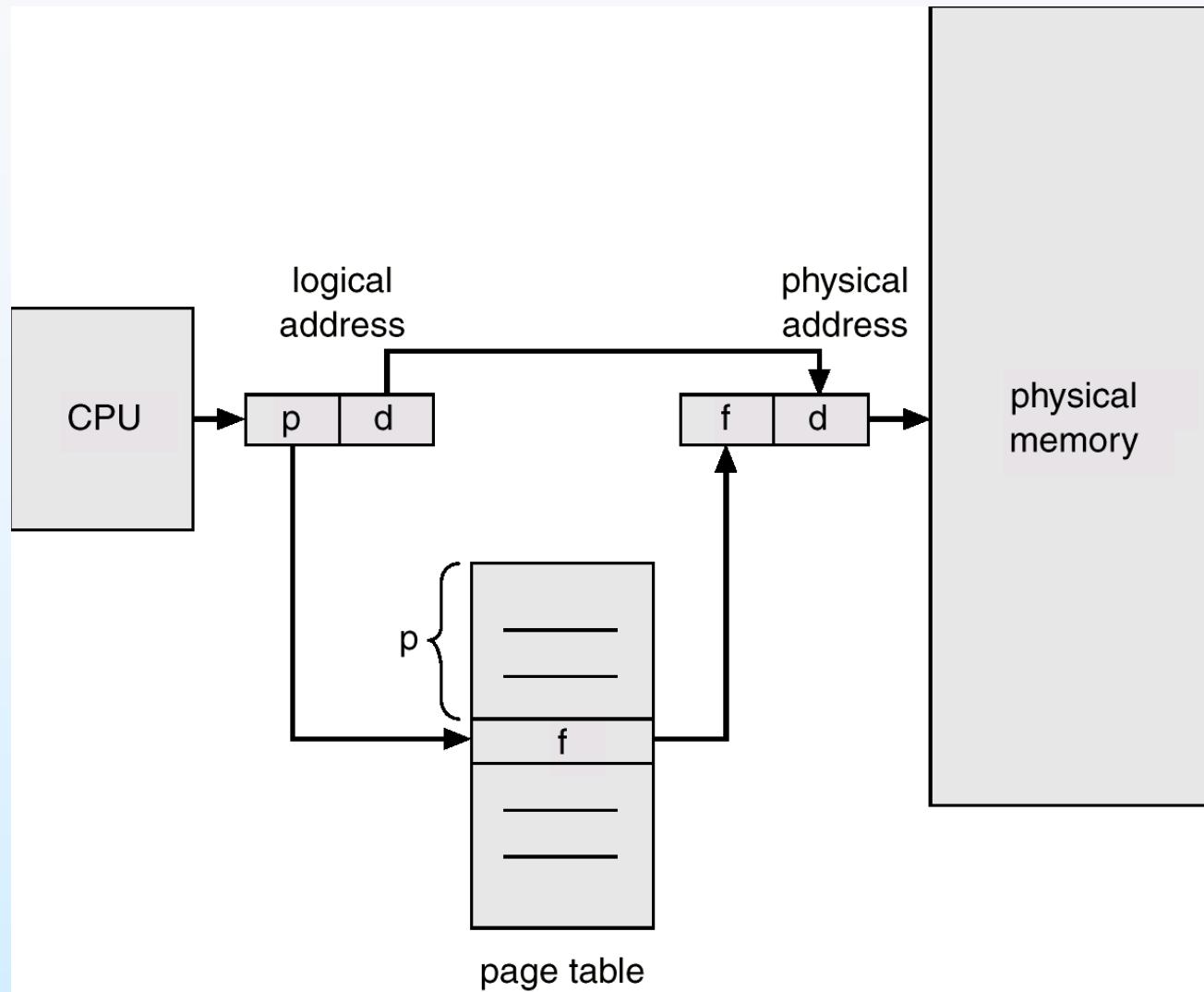
- 页号 (*p*) – 它包含每个页在物理内存中的基址，用来作为页表的索引
- 页偏移 (*d*) – 同基址相结合，用来确定送入内存设备的物理内存地址



- 对给定逻辑地址空间 2^m 和页大小 2^n ，逻辑地址的*m-n*位表示页号，低*n*位表示页偏移
 - 称这个逻辑地址为线性地址（linear address）
- ## ■ 有别于分段管理中的二维地址



分页的硬件支持



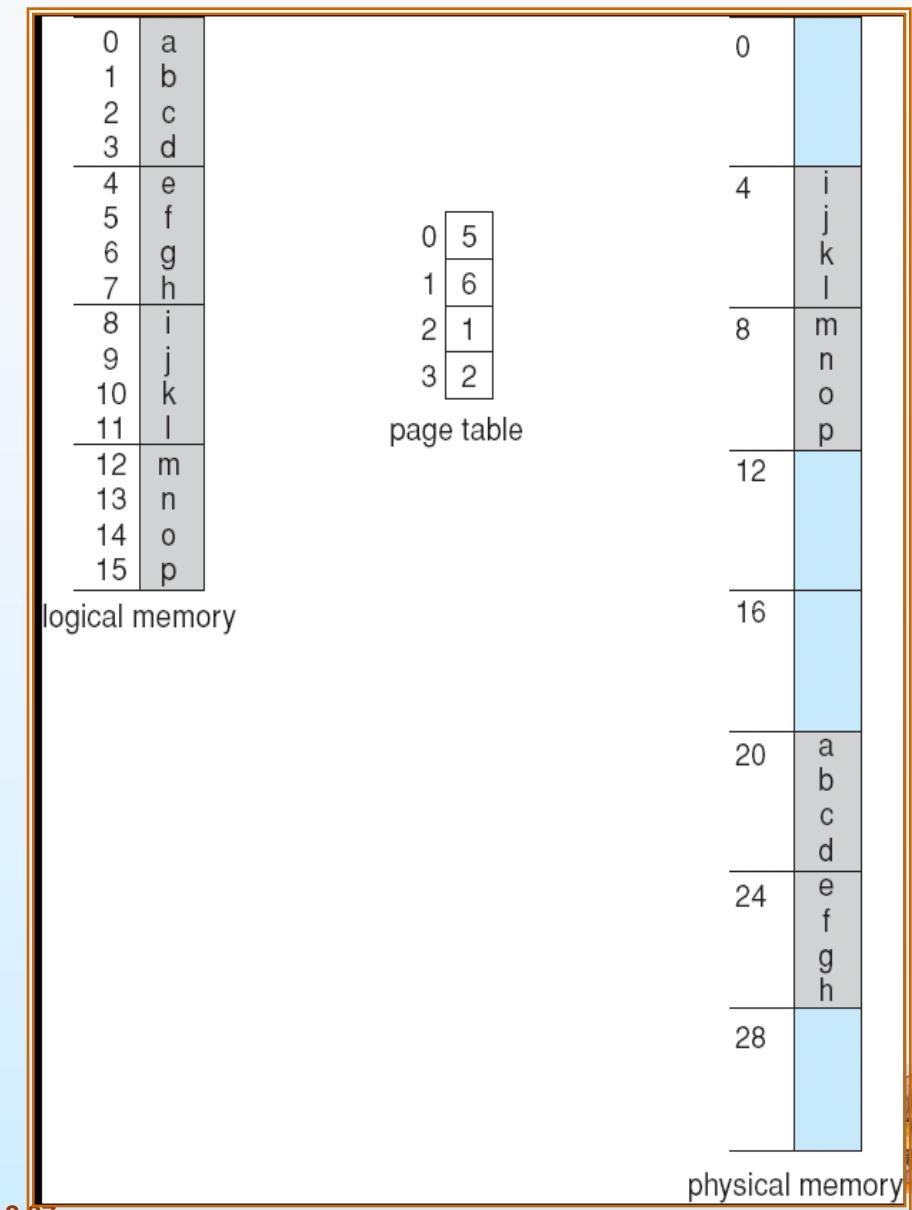


分页的例子

页大小：4-byte

物理内存：32-byte

逻辑内存：4页

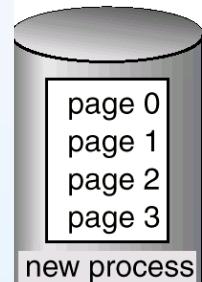




空闲帧的分配

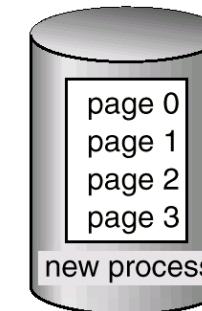
- 操作系统通过一个数据结构来管理物理内存中的空闲帧，这个结构称为空闲帧表，保存系统中所有的空闲帧
- 当进程需要n页，如果系统有至少n帧，那么就分配给新进程
- 进程的每一页装入一个帧中，帧号放入进程的页表中

free-frame list
14
13
18
20
15



(a)

free-frame list
15



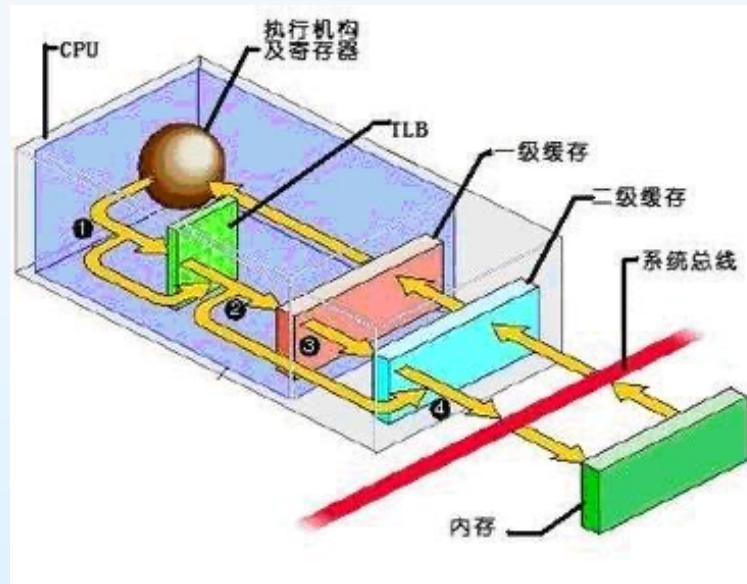
(b)





页表的实现

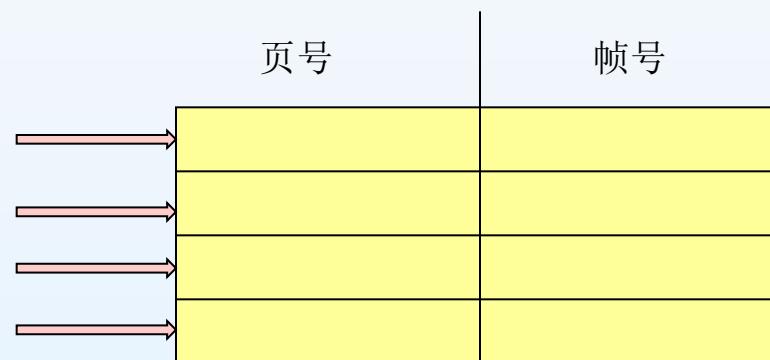
- 页表被保存在内存中
- 页表基址寄存器(**PTBR**)指向页表
- 页表限长寄存器(**PRLR**)表明页表的长度
- 在这个机制中，每一次的数据/指令存取需要两次内存存取，一次是**存取页表**，一次是**存取数据/指令**
- 解决两次存取的问题，是采用小但专用且快速的硬件缓冲，这种缓冲称为**转换表缓冲器(TLB)**或联想寄存器或快表





联想寄存器

■ 联想寄存器—并行查找



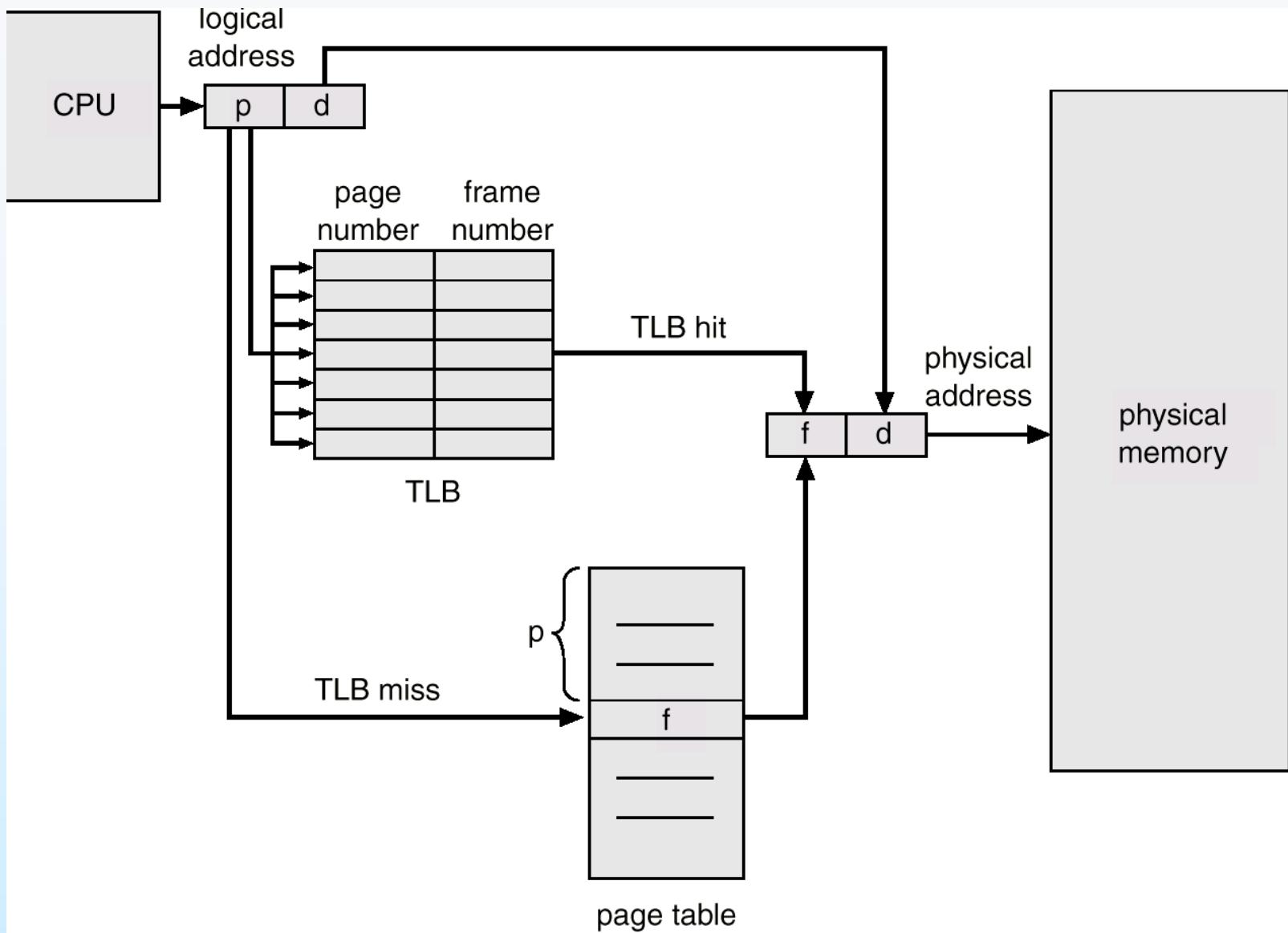
地址转换 (A' , A'')

- 如果 A' 在联想寄存器中，把帧号取出来
- 否则从内存中的页表中取出帧号





使用TLB的分页硬件





有效访问时间

- 命中率—在联想寄存器中找到页号的百分比，比率与联想寄存器的大小有关
- 命中率 = λ
- 有效访问时间 (EAT)，根据概率对每种情况进行加权得到，类似于平均访问时间
- 联想寄存器的查找需要时间单位a微秒
- 假设内存一次存取需要b微秒

$$EAT = \lambda (a + b) + (1 - \lambda) (a+2b)$$

例：查找TLB需要20ns，访问内存需要100ns，命中率为80%，如果TLB命中，那么需要120ns；如果未命中，那么需要220ns

$$EAT=0.8*120+0.2*220=140\text{ns}$$

如果命中率为98%， $EAT=122\text{ns}$ ， 比内存访问只慢了22%





内存保护

■ 简单方法：

- 把页号和页表限长寄存器(PRLR)比较

■ 通过与每个帧相关联的保护位来实现，这些位保存在页表中。

- 例如，可用一个位来定义一个页是可读可写或只可读

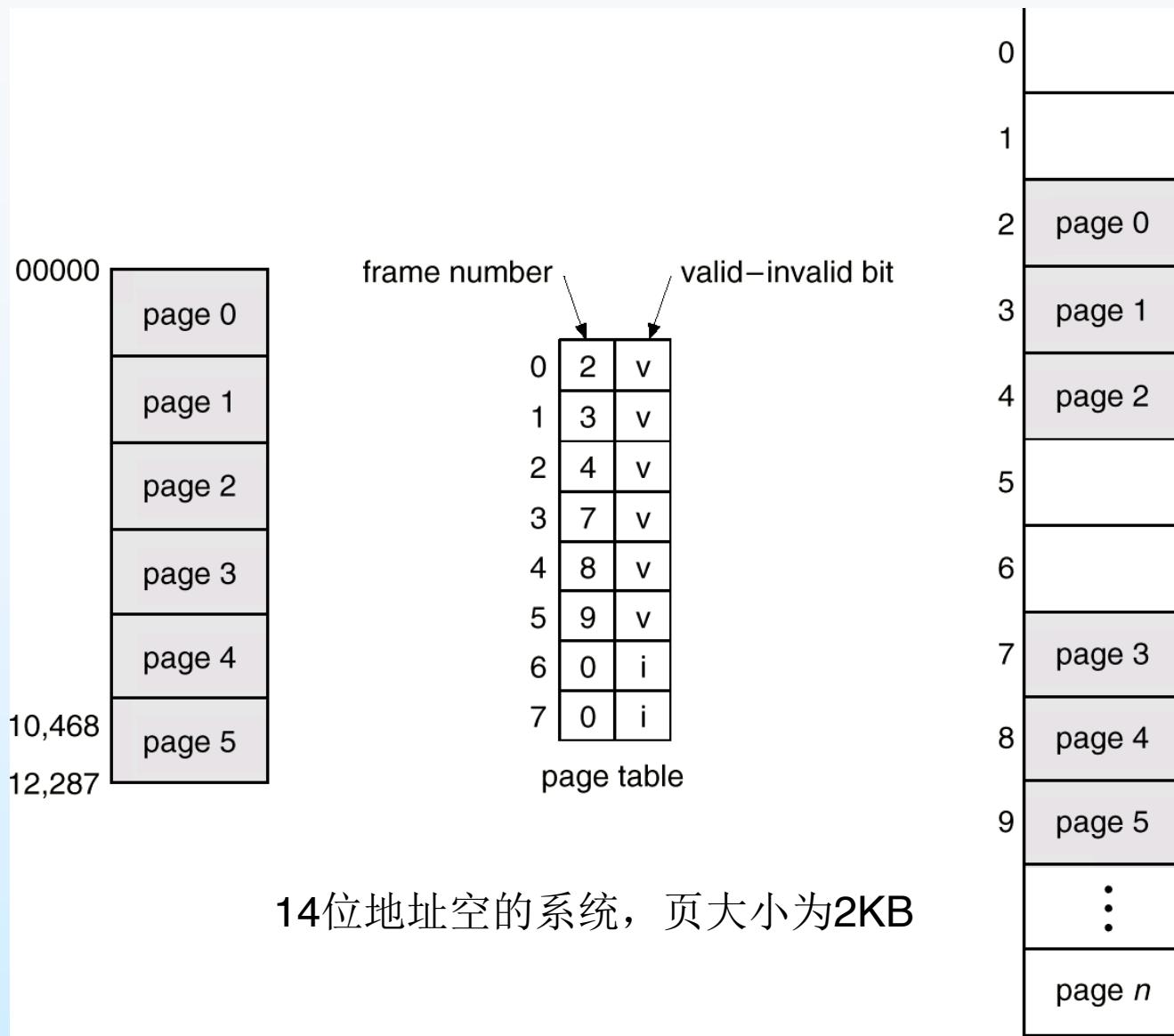
■ 有效-无效位附在页表的每个表项中：

- “有效” 表示相关的页在进程的逻辑地址空间，并且是一个合法的页
- “无效” 表示页不在进程的逻辑地址空间中





页表中的有效位和无效位





页共享

■ 分页的优点：可以共享公共代码

■ 共享代码

- 如果代码是可重入代码（或称为纯代码），可以在进程间共享（如文本编辑器，编译器，数据库系统， window 系统等）
- 共享代码必须出现在所有进程的逻辑地址空间的相同位置

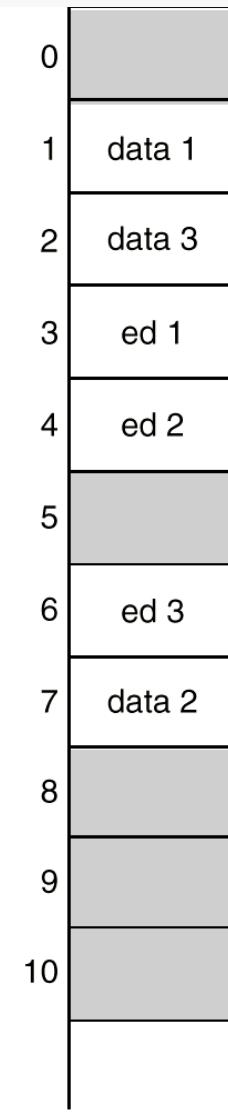
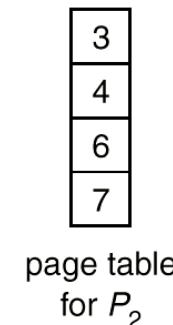
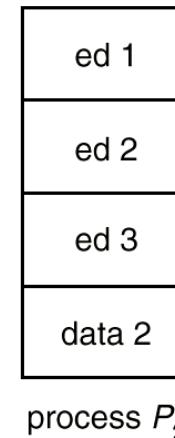
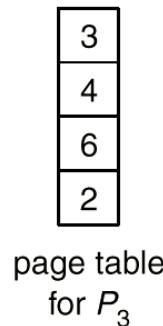
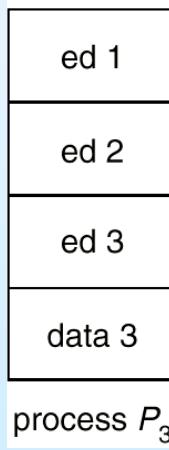
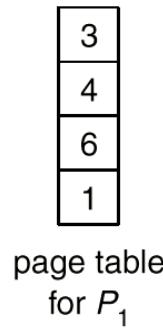
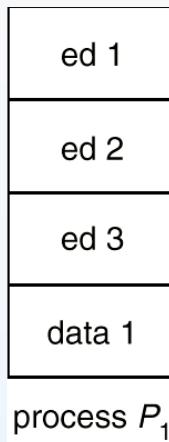
■ 私有代码和数据

- 每个进程单独保留一个代码和数据副本，还可以有自己的私有代码和数据
- 存有私有数据和代码的页能够出现在逻辑地址空间的任意位置





一个共享页的例子



页表结构





页表结构

■ 例子：

- 32位逻辑地址
- 页大小为4KB（也就是 2^{12} ）
- 那么一个页表最多可包含1M（ $2^{32}/2^{12}=2^{20}$ ）个表项
- 假设每个页表项占用4个字节，那么每个进程需要4MB物理空间，也就是1024个连续页面来存储页表
- 需要这么多个连续页面来存放页表不一定能实现

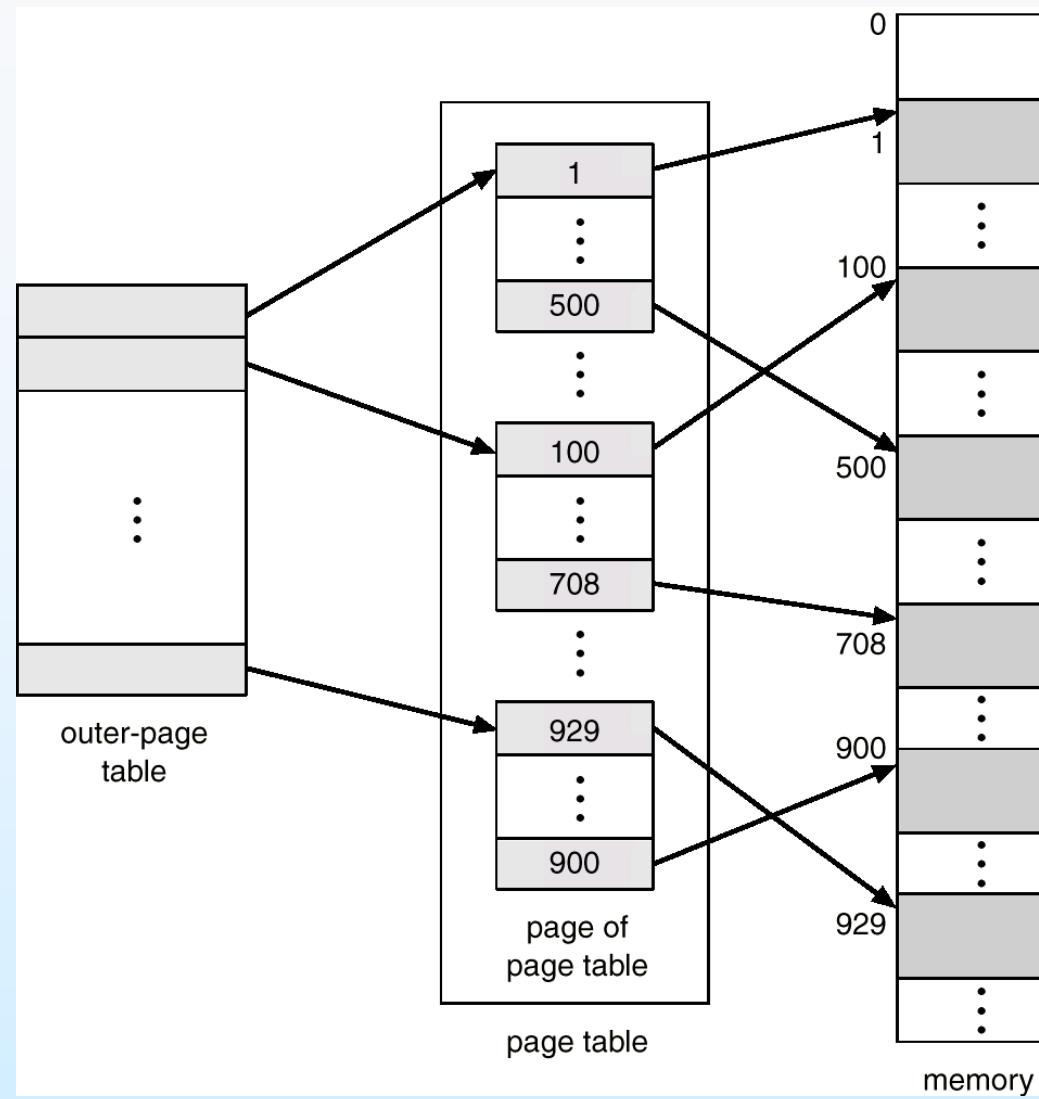
■ 解决方法：

- 层次页表
- 哈希页表
- 反向页表





两级页表机制





一个两级分页的例子

- 一个逻辑地址被分为 (在4K页大小的32位系统上)：
 - 一个20位的页号
 - 一个12位的页偏移
- 由于页表所在页也被分页，页号被进一步分为：
 - 一个10位的页号
 - 一个10位的页偏移
- 因此，逻辑地址表示如下：

page number		page offset
p_1	p_2	d
10	10	12

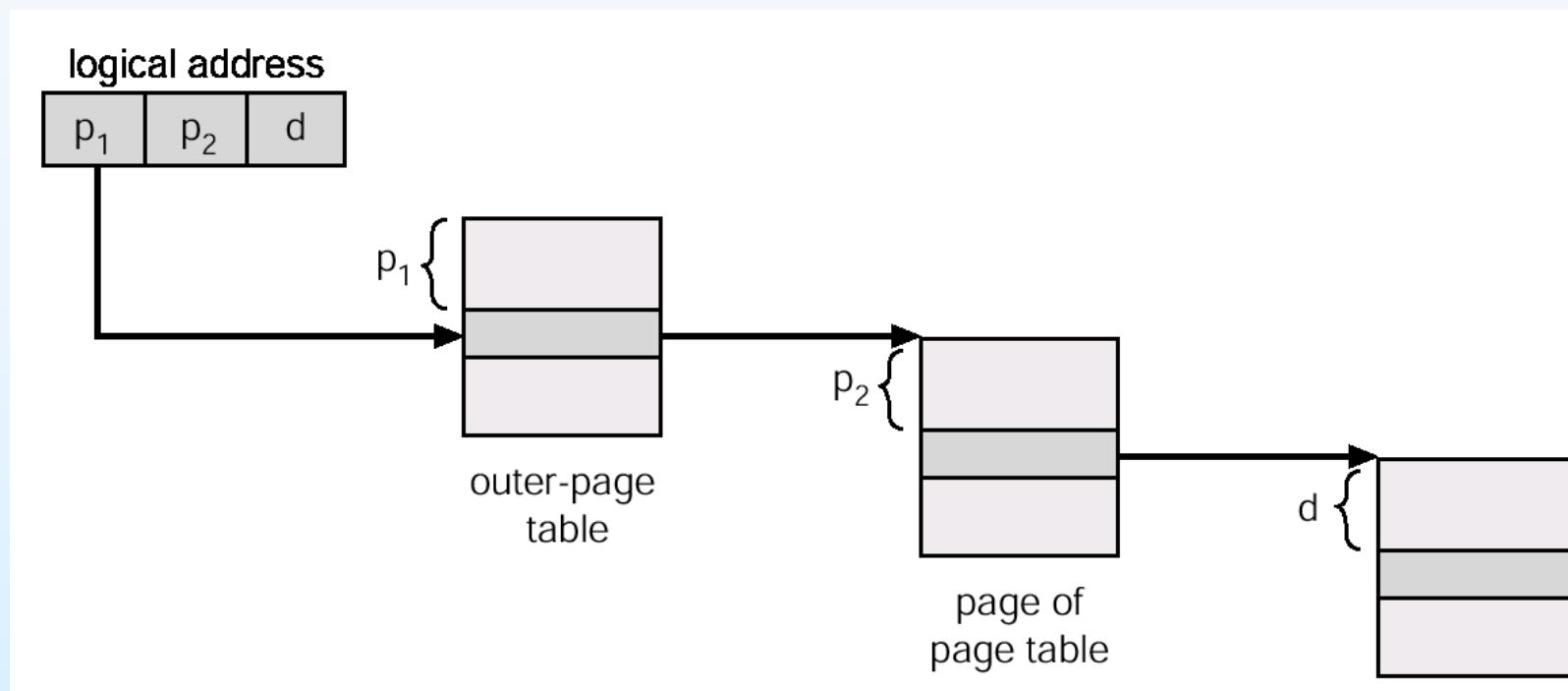
p_1 是外页表的索引，而 p_2 是外部页表的页偏移





地址转换机制

- 一个两级32位分页结构的地址转换机制





三级分页机制

outer page	inner page	offset
p_1	p_2	d
42	10	12

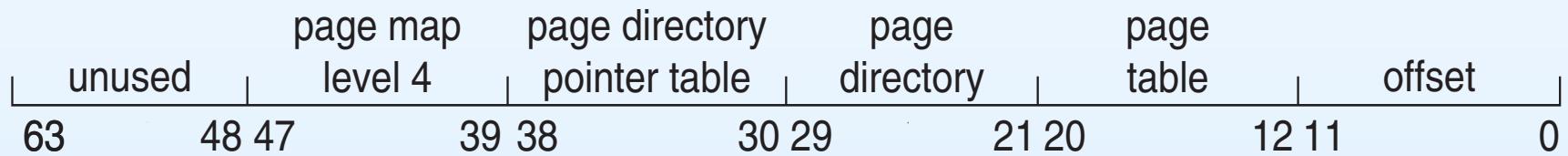
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12



Intel x86-64

■ 仅用48位：

- 页大小： 4KB, 2MB, 1GB
- 四级页表



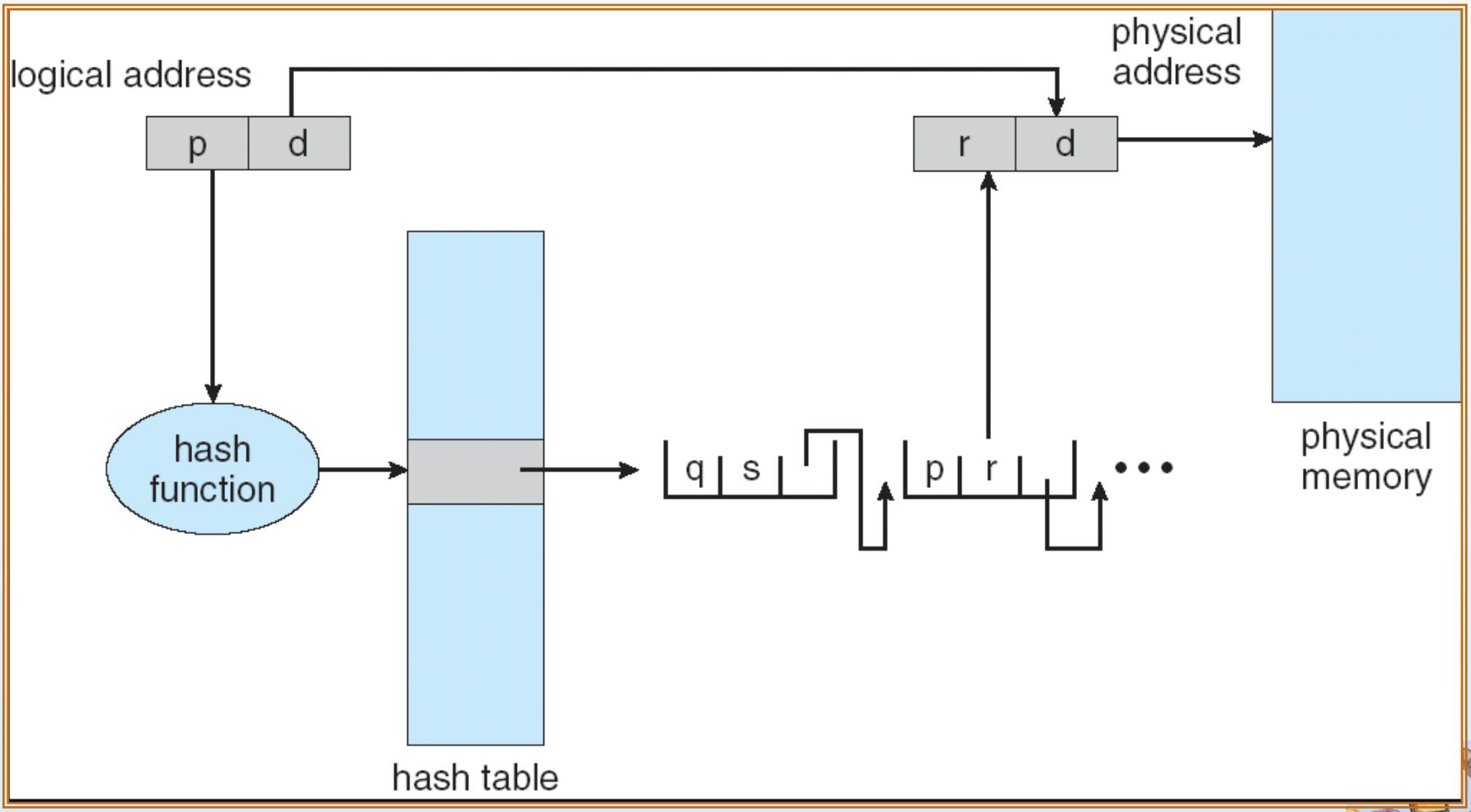


哈希页表

- 通常地址空间 > 32 位
- 虚拟页号被散列到一个页表中。这种页表的每一个条目都包括了一个链表元素，这些元素哈希在同一位置。
- 每个元素有三个域：1) 虚拟页号；2) 所映射的帧号；3) 指向链表内下一个元素的指针
- 虚拟页号与链表中的每个元素相比较，找到匹配项。如果匹配，则相应的物理帧被取出。



哈希页表



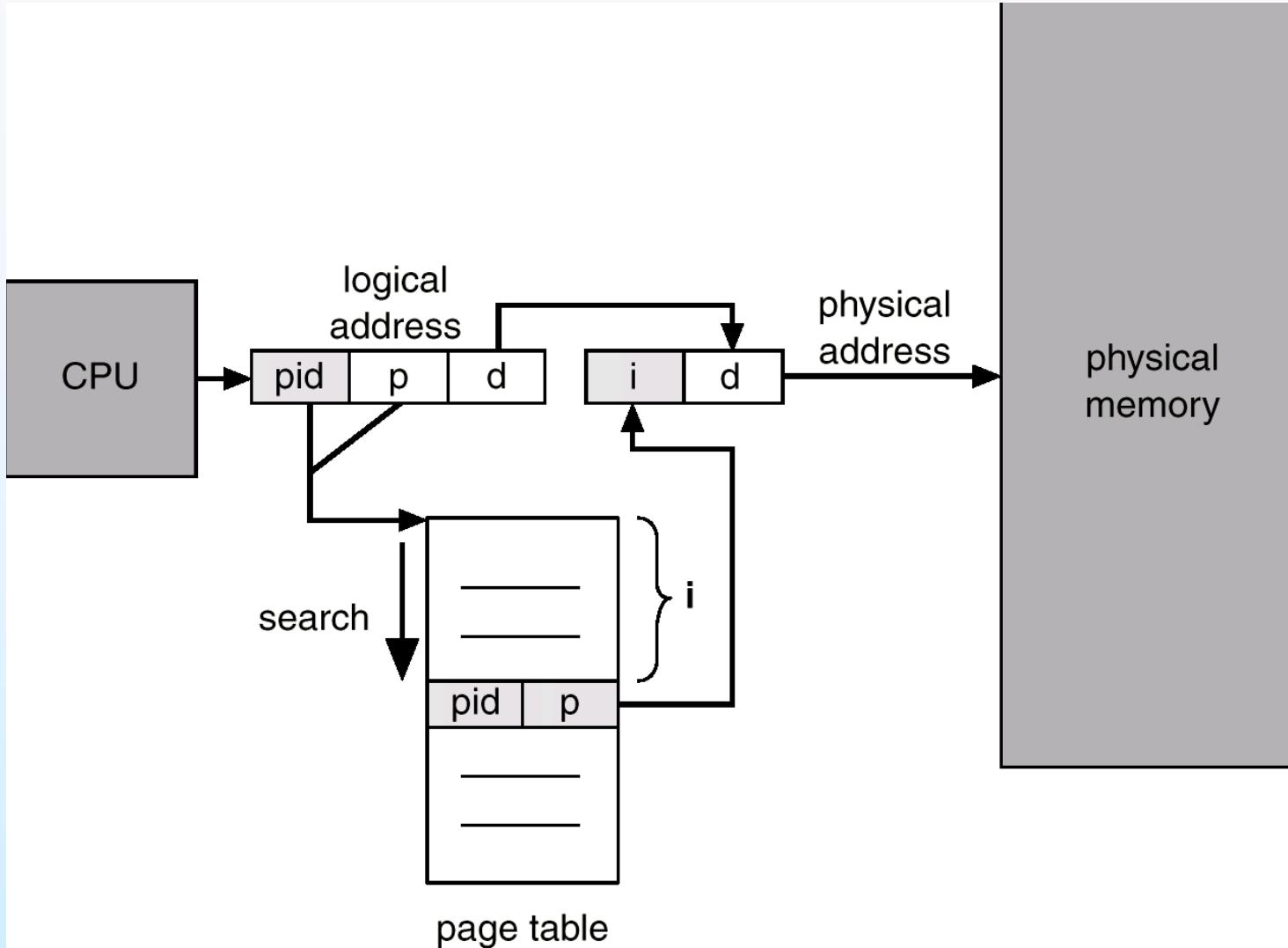


反向页表(inverted page table)

- 对于每个真正的内存页或帧才有一个条目。
- 每个条目保存在真正内存位置的页的虚拟地址，以及包括拥有这个页的进程的信息。
- 反向页表的条目中需要一个地址空间标志符，以确保一个特定进程的一个逻辑页可以映射到相应的物理帧。



反置页表机制





反向页表讨论

- 减少了需要储存每个页表的内存，但是当访问一个页时，增加了寻找页表需要的时间。
- 使用哈希表来将查找限制在一个或少数几个页表条目。
- 实现共享内存困难
 - 每个物理页只有一个虚拟页条目，所以一个物理页不可能有两个或更多的共享虚拟地址。





4、分段





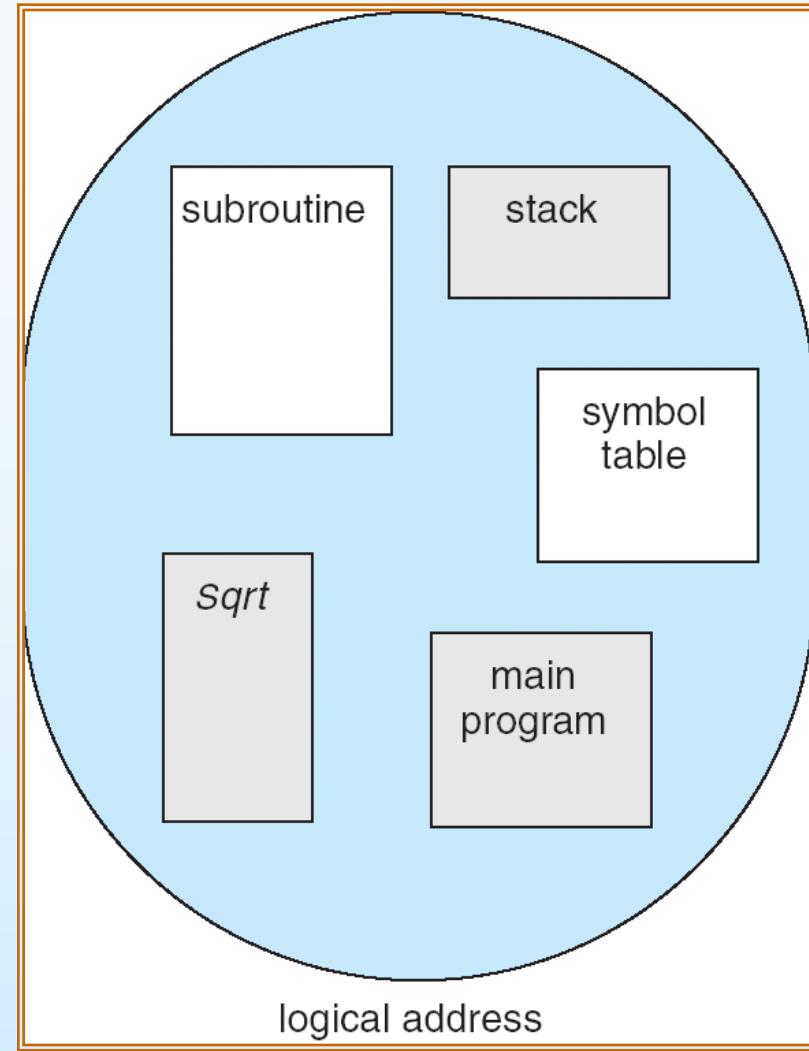
● 分页内存管理存在的问题：

- 用户视角的内存和实际物理内存的分离，即用户视角的内存与实际物理地址不一样。
- 没有用户愿意将内存看作一个线性字节数值，用户通常愿意将内存看成一组长度不同的段的集合，段是逻辑上有意义的单位，而且段与段之间没有一定的顺序。





用户眼中的程序





分段(Segmentation)

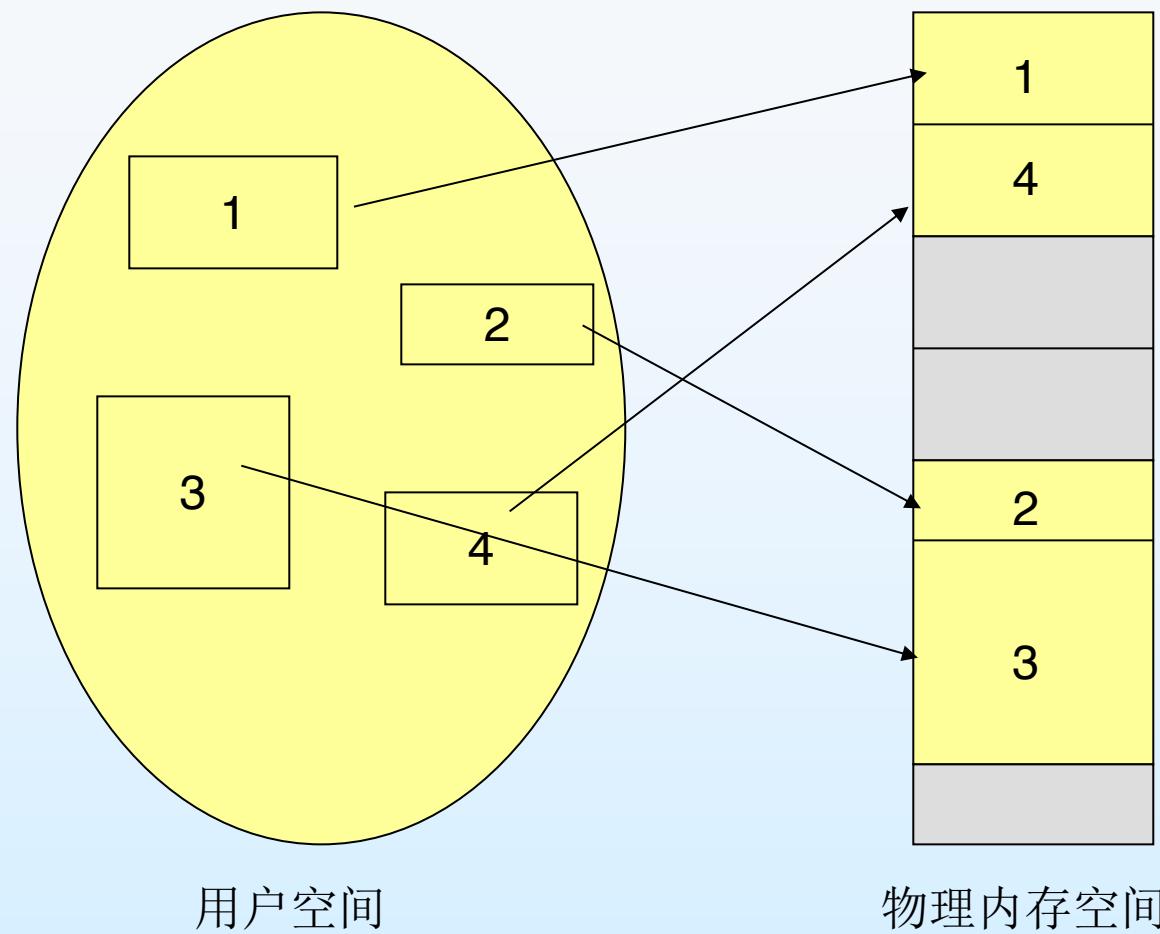
- 支持用户观点的内存管理机制。
- 逻辑地址空间是由一组段组成的，每个段都有名称和长度，地址指明了段名称和段内偏移。
- 一个逻辑地址是两个向量的集合：
 - <segment-number, offset>



分段的逻辑视图

分段的逻辑视图

- 在编译用户程序时，编译器会自动根据输入程序来构造段
- 如一个C语言程序，可能会构造如下段：代码、全局变量、堆、栈、标准的库函数。
- 加载程序会装入所有这些段，并为它们分配段号





分段机制 (1)

- 一个逻辑地址是两个向量的集合:

<segment-number, offset>

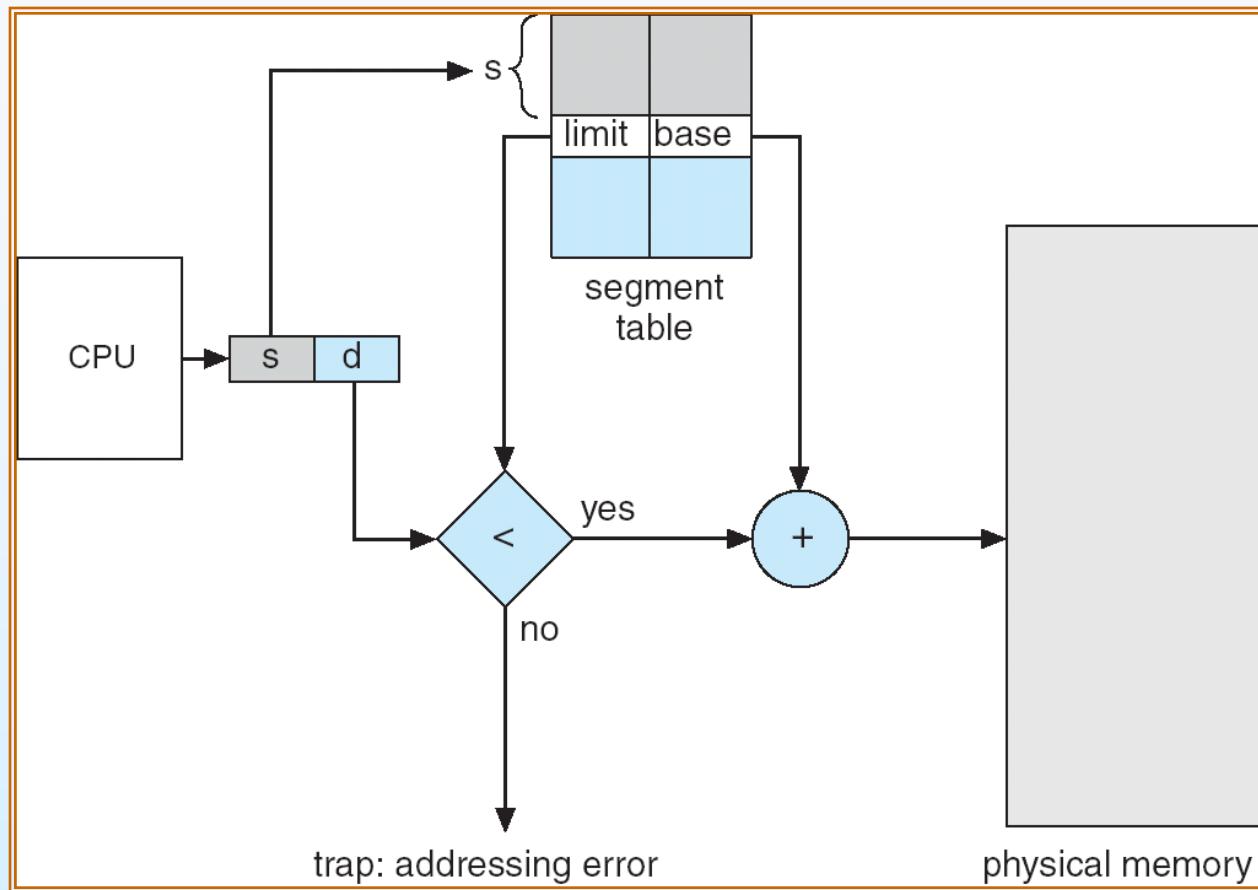
- 段表 (**segment table**) - 映射二维物理地址, 每个表项包括:
 - 基址 **base**- 包括内存中段物理地址的起始地址
 - 限长 **limit**- 指定段的长度
- 段表基址寄存器(**STBR**)指向段表在内存中的地址
- 段表限长寄存器(**STLR**)表明被一个程序所使用的段的数目

segment number s is legal if $s < \text{STLR}$

则段号是合法的

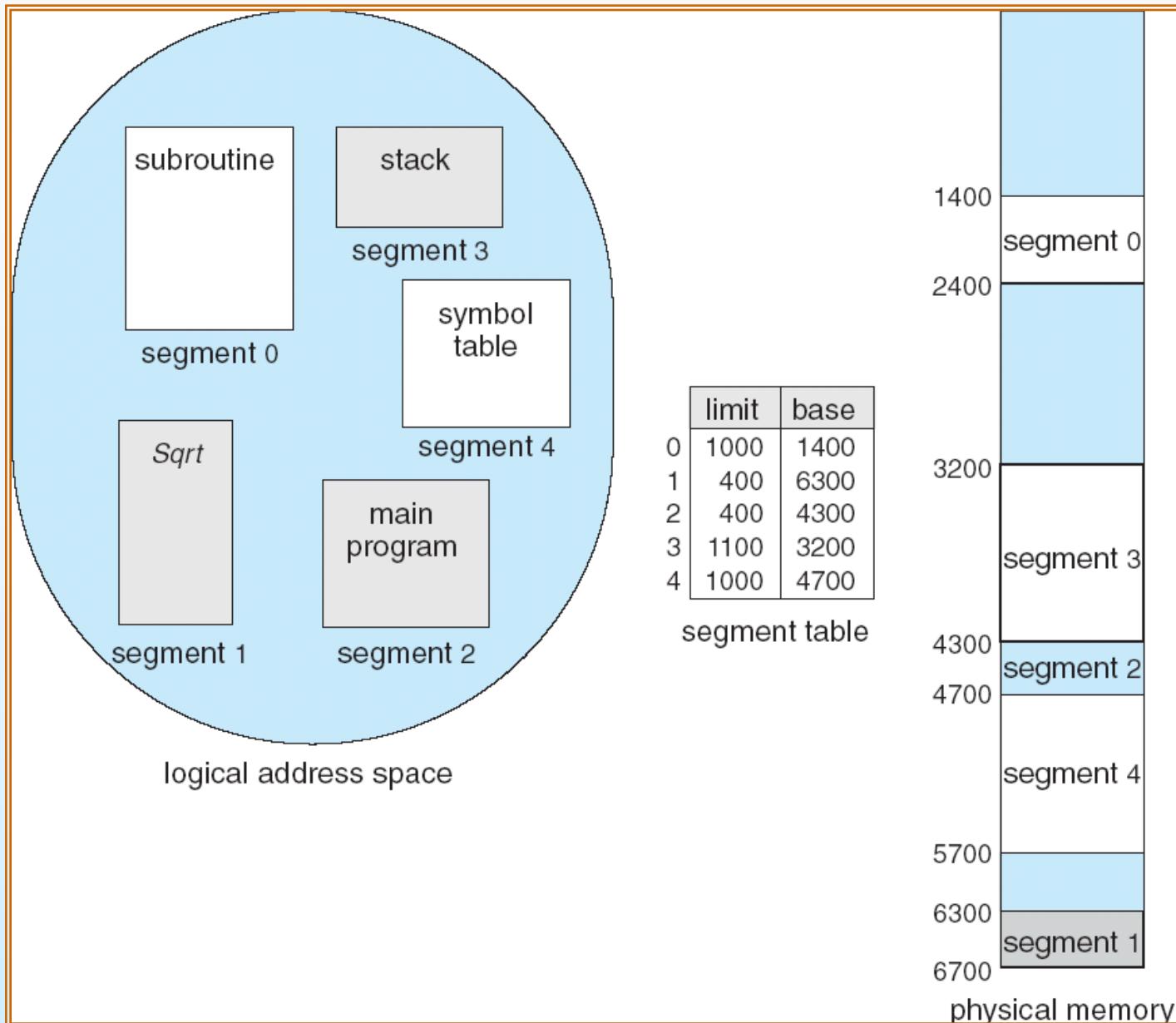


分段硬件





分段例子



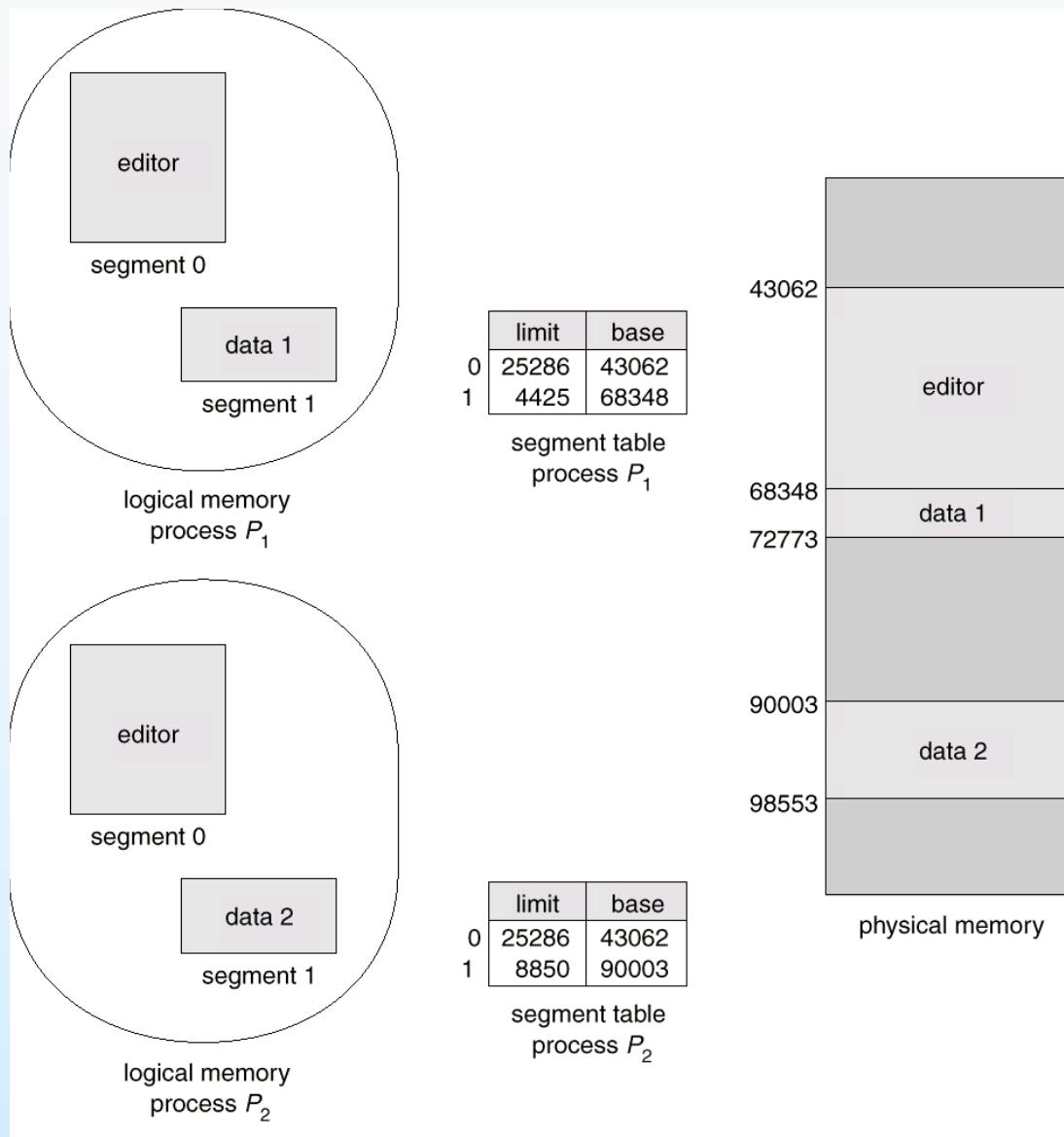


分段机制（2）

- 由于段的长度各不相同，因此这种内存分配实际上是一个动态存储分配问题，可采用可变分区方案实现。
- 内存分配
 - 首先/最佳适应法
 - 外碎片问题
- 重定位
 - 动态
 - 由段表来执行
- 共享
 - 段是逻辑上有意义的单位
 - 实现更方便
 - 比页共享更容易实现



分段共享例子





分段机制（3）

- 保护，每个段表的表项有：
 - 有效位 = 0 \Rightarrow 非法段
 - 读/写/执行权利
- 保护位同段相联系，在段的级别进行代码共享





8.71



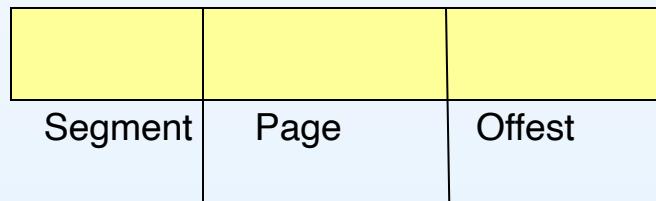
5、段页式





原理

- 分段和分页原理的结合
- 其基本原理是：先将用户程序分成若干个段，并为每个段赋予一个段号，再把每个段分成若干个页
- 逻辑地址：<段号，页号，页内偏移>



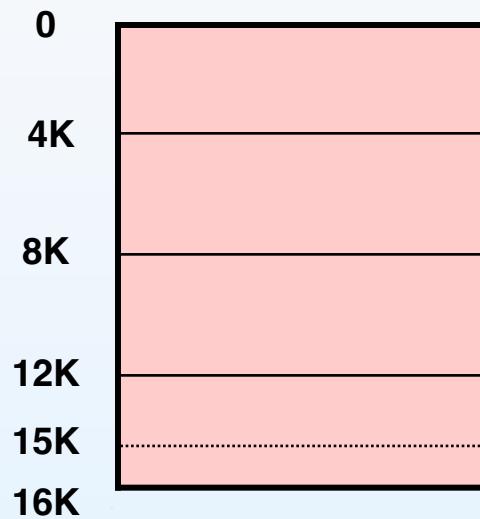
- 从用户视角看，程序被划分成了逻辑上有意义的段
- 从系统角度看，物理内存被划分成了固定大小的帧
- 结合了分页和分段这两者的优点
- 存在内碎片？
- 有无外碎片？





例子

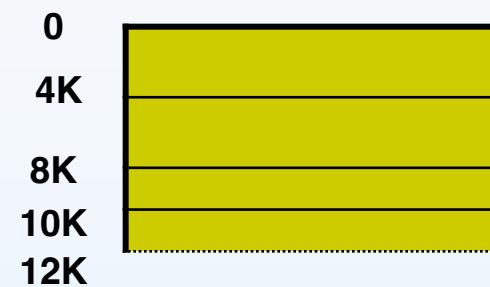
主程序段



子程序段



数据段



(a)

段号S

段内页号P

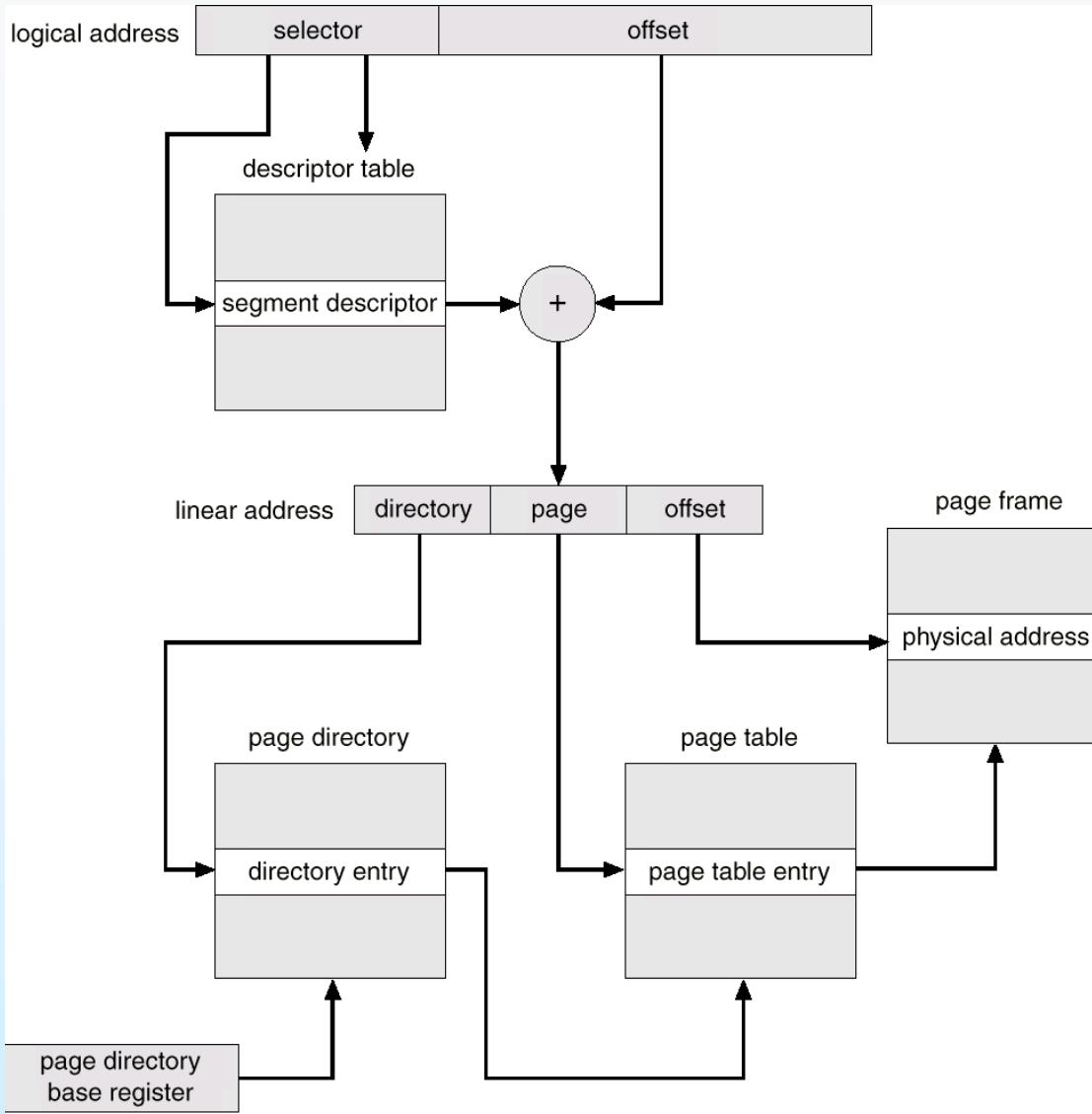
页内地址W

(b)





Intel 386的段页式存储管理



- CPU产生的逻辑地址，分为选择子selector和偏移offset
- 以选择子为索引，在段描述符表中找到对应项，取出段描述符，与偏移相加后，得到32位的线性地址
- 该线性地址是用来进行分页的地址，采用两级页表，分为三部分，页目录号，页号和页内偏移
- 以页目录号为索引在页目录表中检索，得到页表所在位置，再以页号为索引得到页所在的内存帧的位置，最后通过页内偏移确定物理位置
- 其中页目录的起始地址由页目录基址寄存器保存





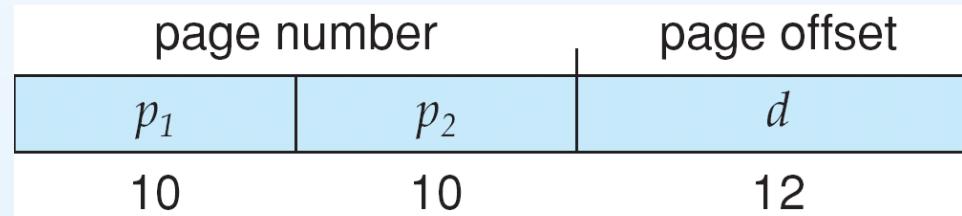
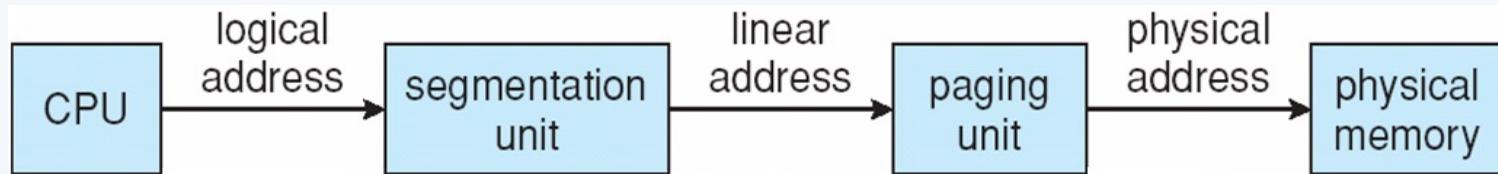
Intel 32 和 64-bit CPU段页式结构

- 32位: IA-32 (64位: IA-64)
- 支持分段和段页式
 - 每段最大: 4 GB
 - 每个进程最多: 16 K 段
 - 分成2部分:
 - 第一部分: 最大8K个段是进程私有 (保存在**local descriptor table (LDT)**)
 - 第二部分: 最大8K个段私有进程共享 (保存在**global descriptor table (GDT)**)

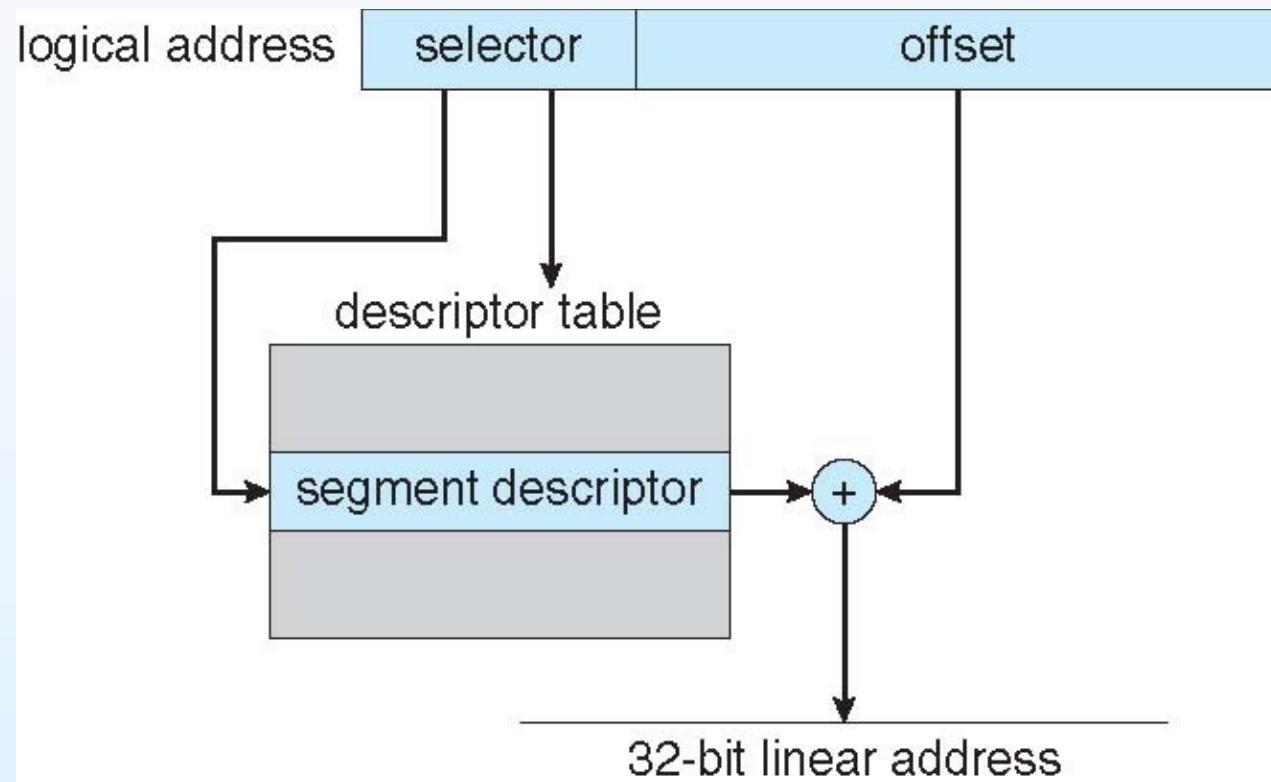




IA-32地址转换

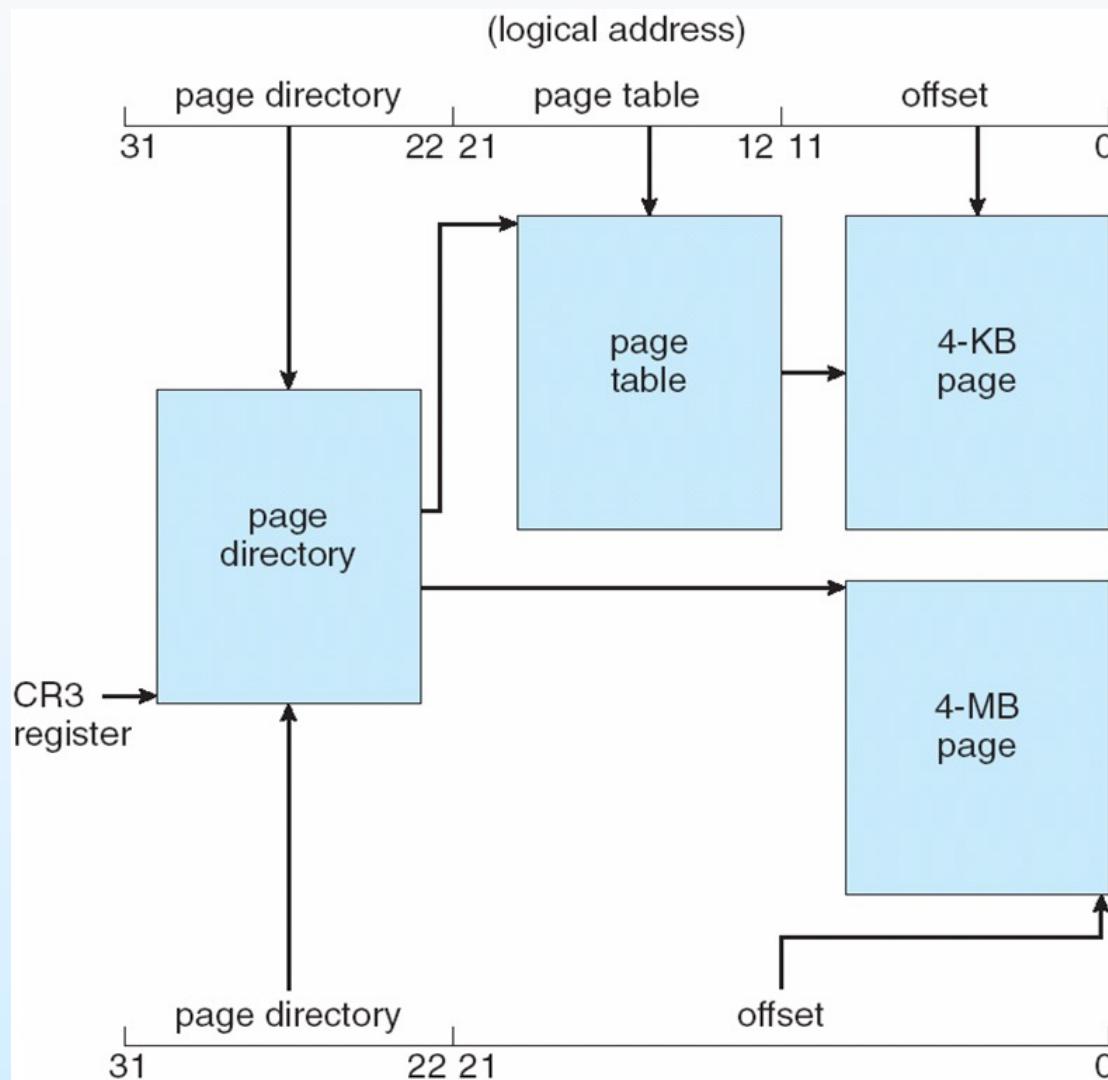


Intel IA-32 分段结构





Intel IA-32 分页结构





8.80





单元测试

