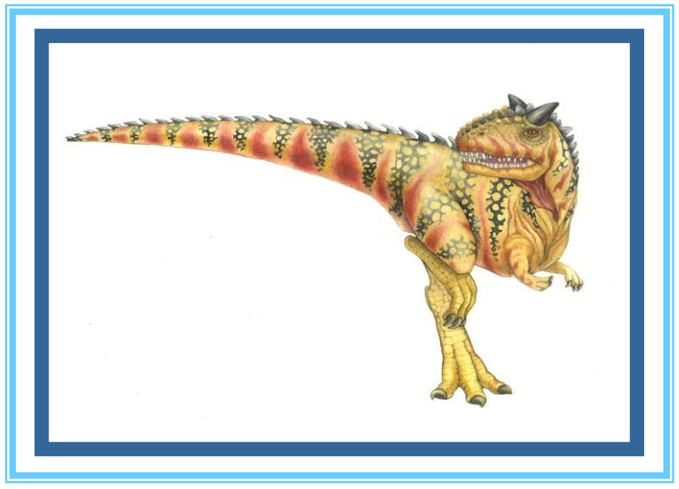
第四章 内存管理

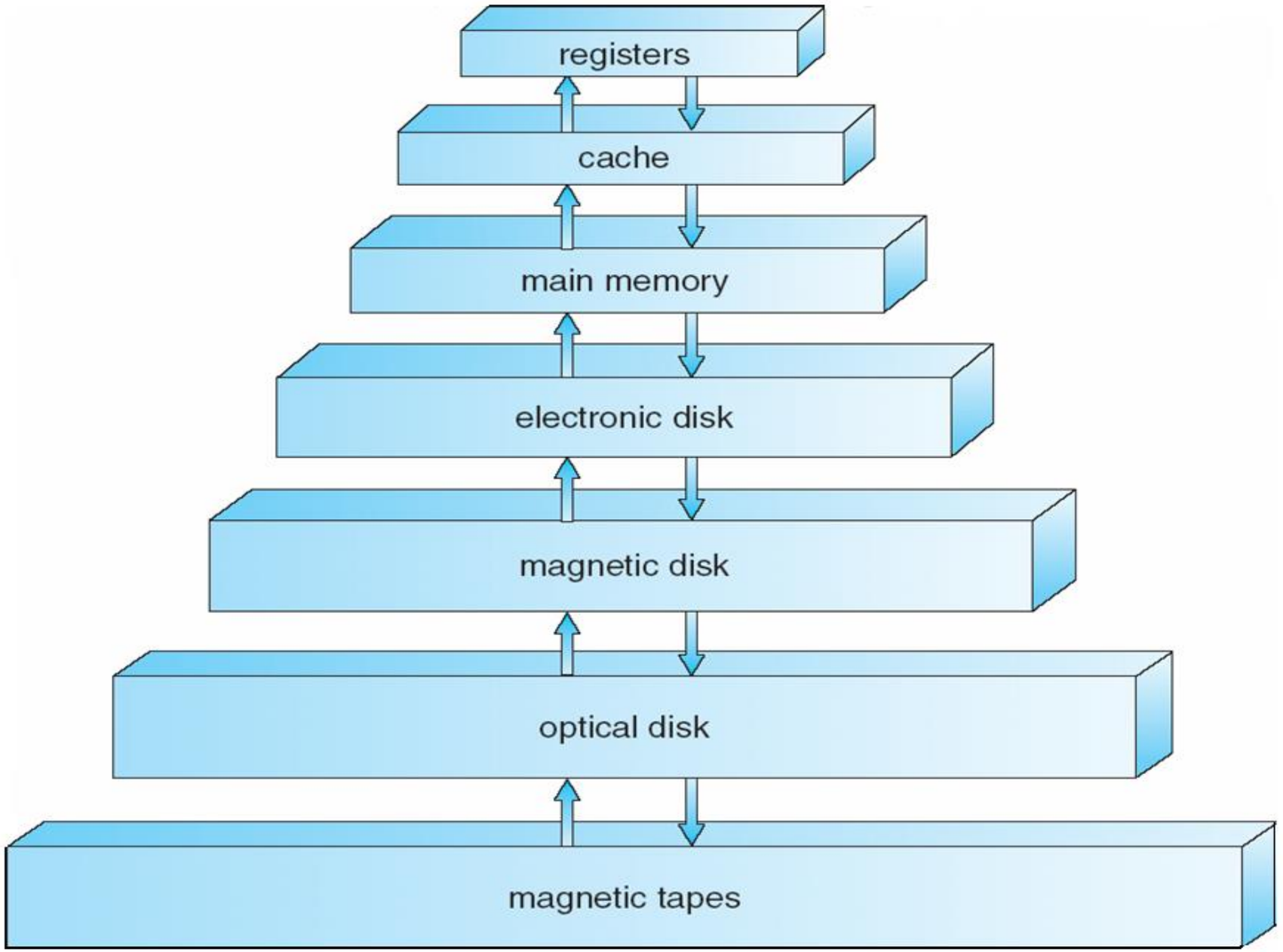


主要内容

- 存储器的层次结构
- 程序的装入和链接
- 内存管理方法
  - 连续分配方式
  - 离散分配方式
- 虚拟存储

4.1 存储器的层次结构

- 存储器架构

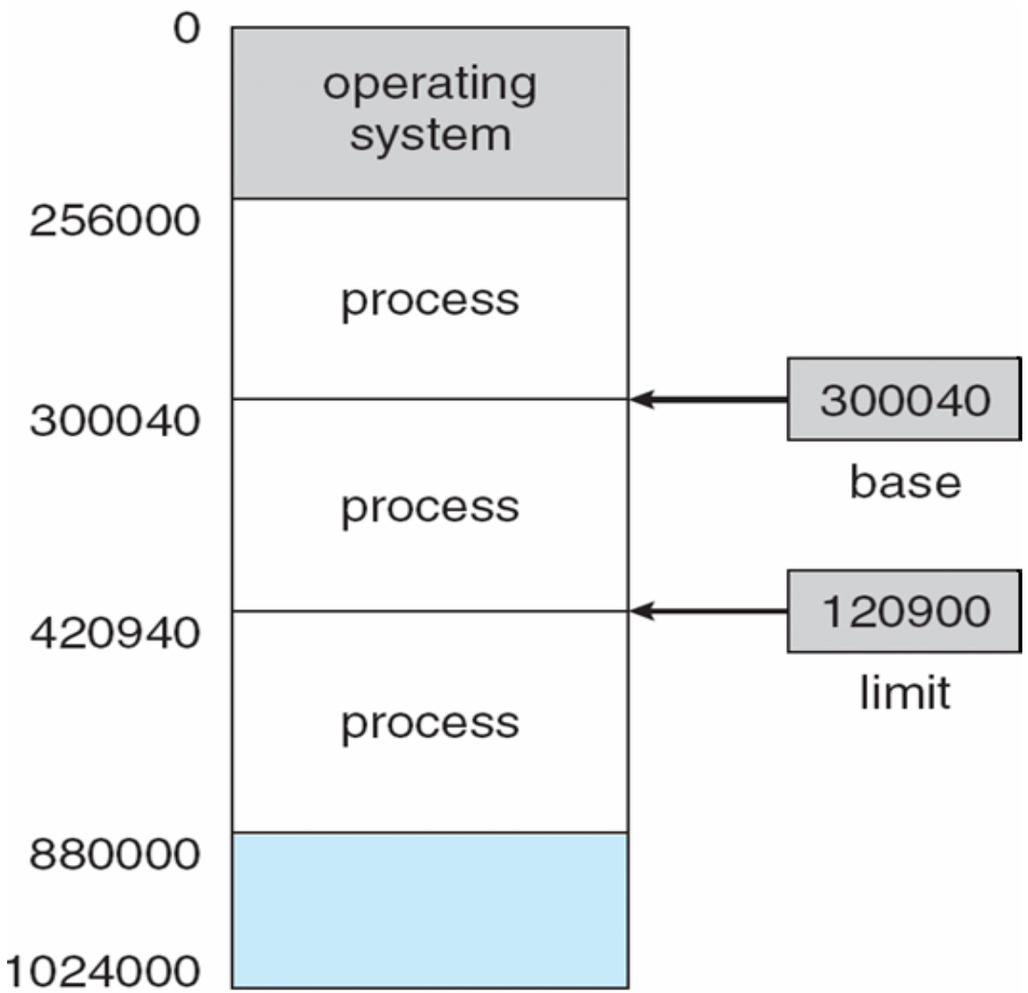


4.1 存储器的层次结构

- 内存管理的目标
  - 内存分配
  - 地址映射
    - 逻辑地址->物理地址
  - 地址保护
    - 一个进程不能随便访问另一个进程的地址空间。
  - 内存扩充

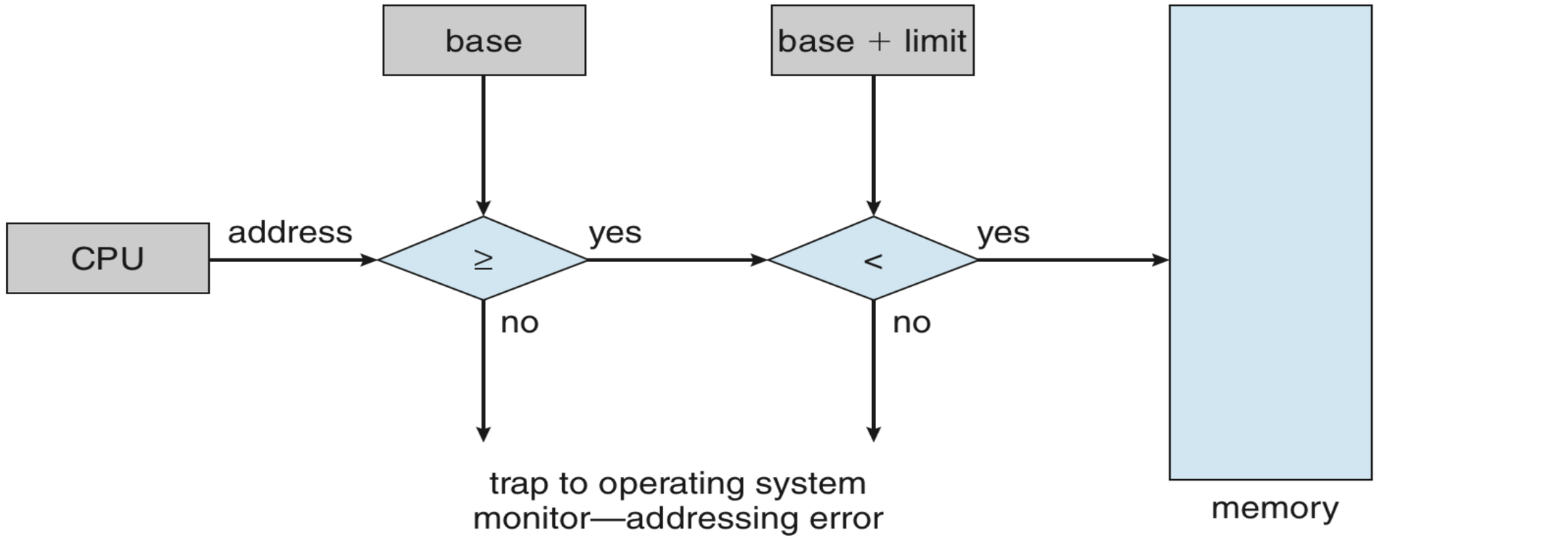
地址保护

- 每个进程都有独立的内存空间，内存保护可通过硬件实现
  - 基地址寄存器（base register）
  - 界限地址寄存器（limit register）



地址保护

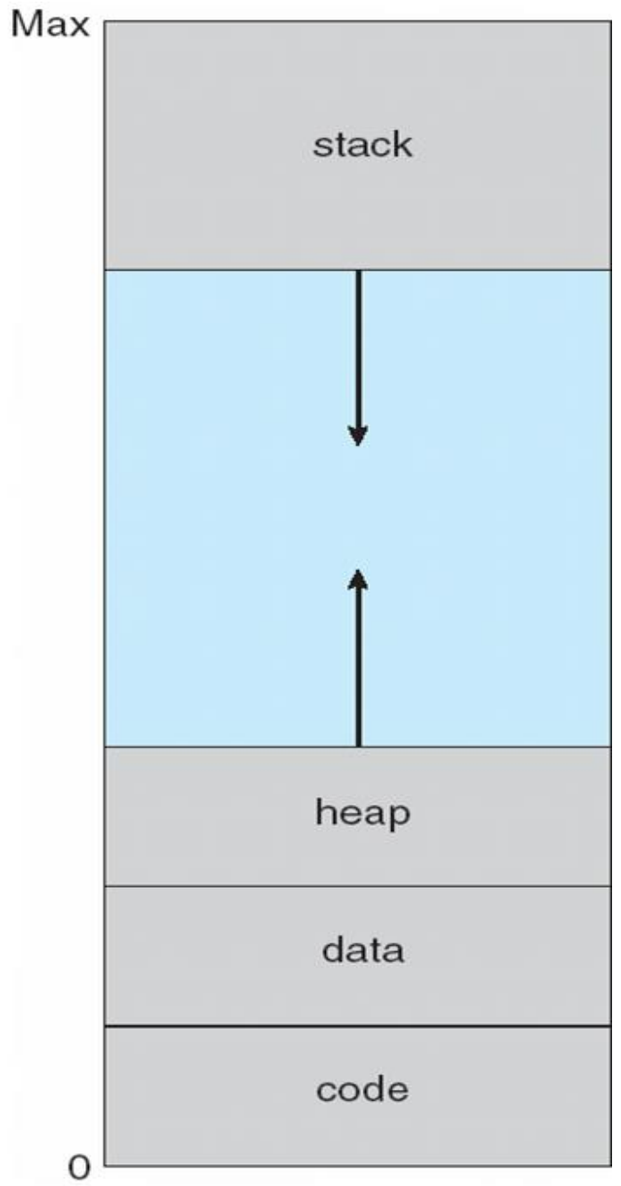
- 采用基地址寄存器和界限地址寄存器的硬件地址保护



- 只有操作系统可以加载这两个寄存器的内容，防止用户程序修改其他用户或操作系统的代码。

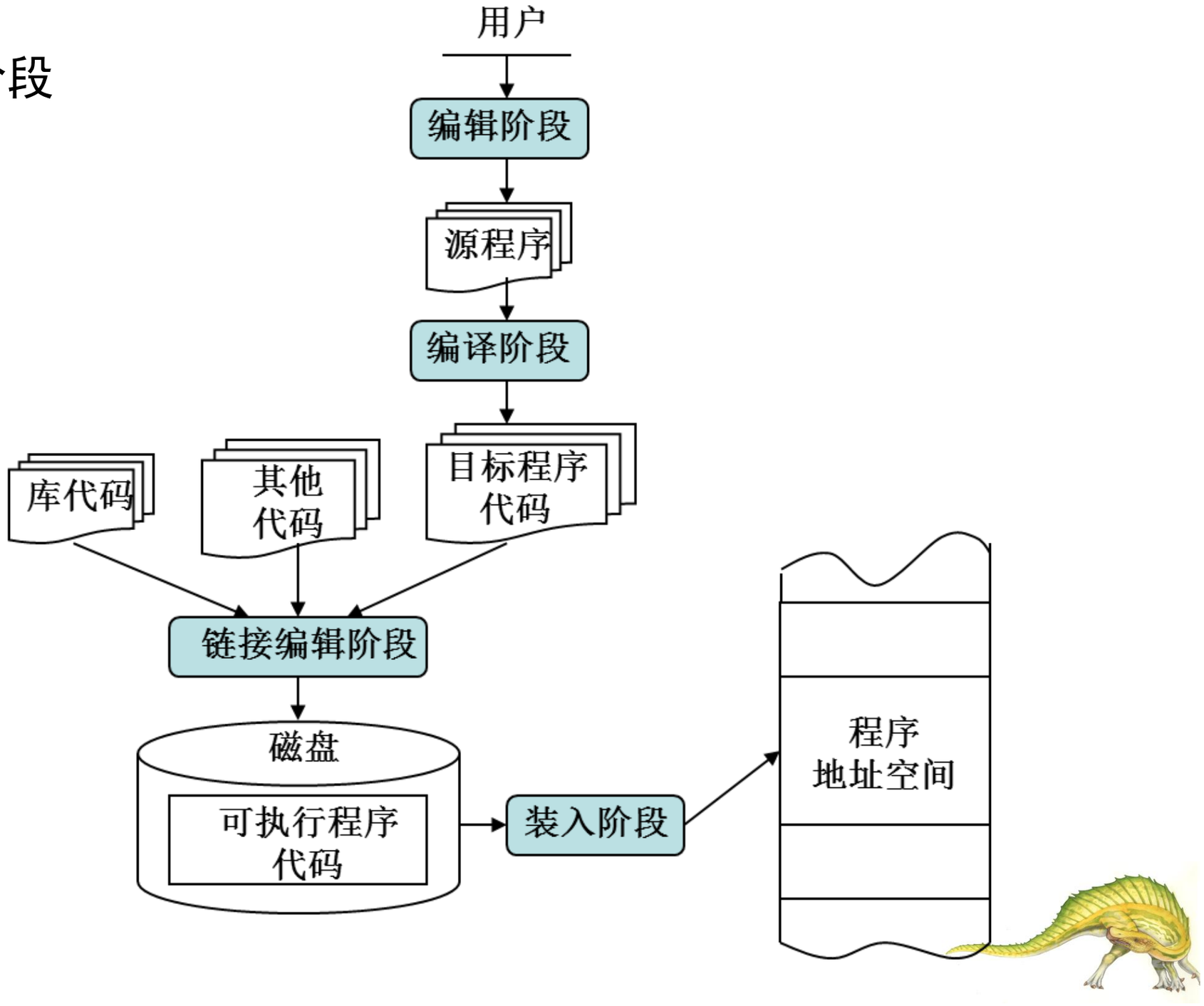
地址映射

- 从逻辑地址到物理地址的映射
  - 逻辑地址（logical address）
    - CPU生成的地址，相对地址，虚地址
  - 物理地址（physical address）
    - 内存空间的地址，绝对地址，实地址
- 地址空间
  - 逻辑地址空间（logical address space）
  - 物理地址空间（physical address space）
- 内存管理单元MMU(merory-management unit)
  - 完成逻辑地址到物理地址映射的的硬件设备



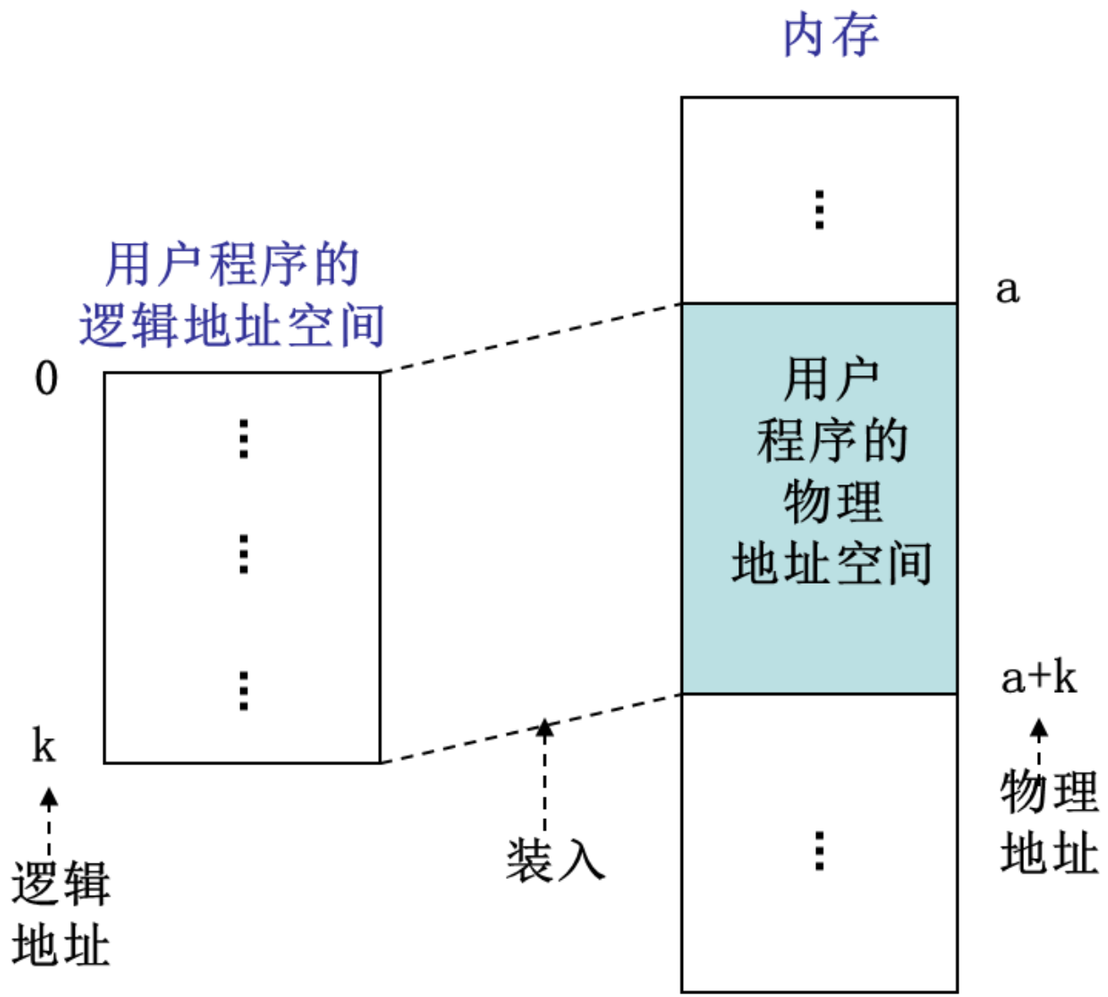
地址映射

- 用户程序的处理阶段
  - 编译（compile）
  - 链接（link）
  - 装入（load）
  - 运行（execute）



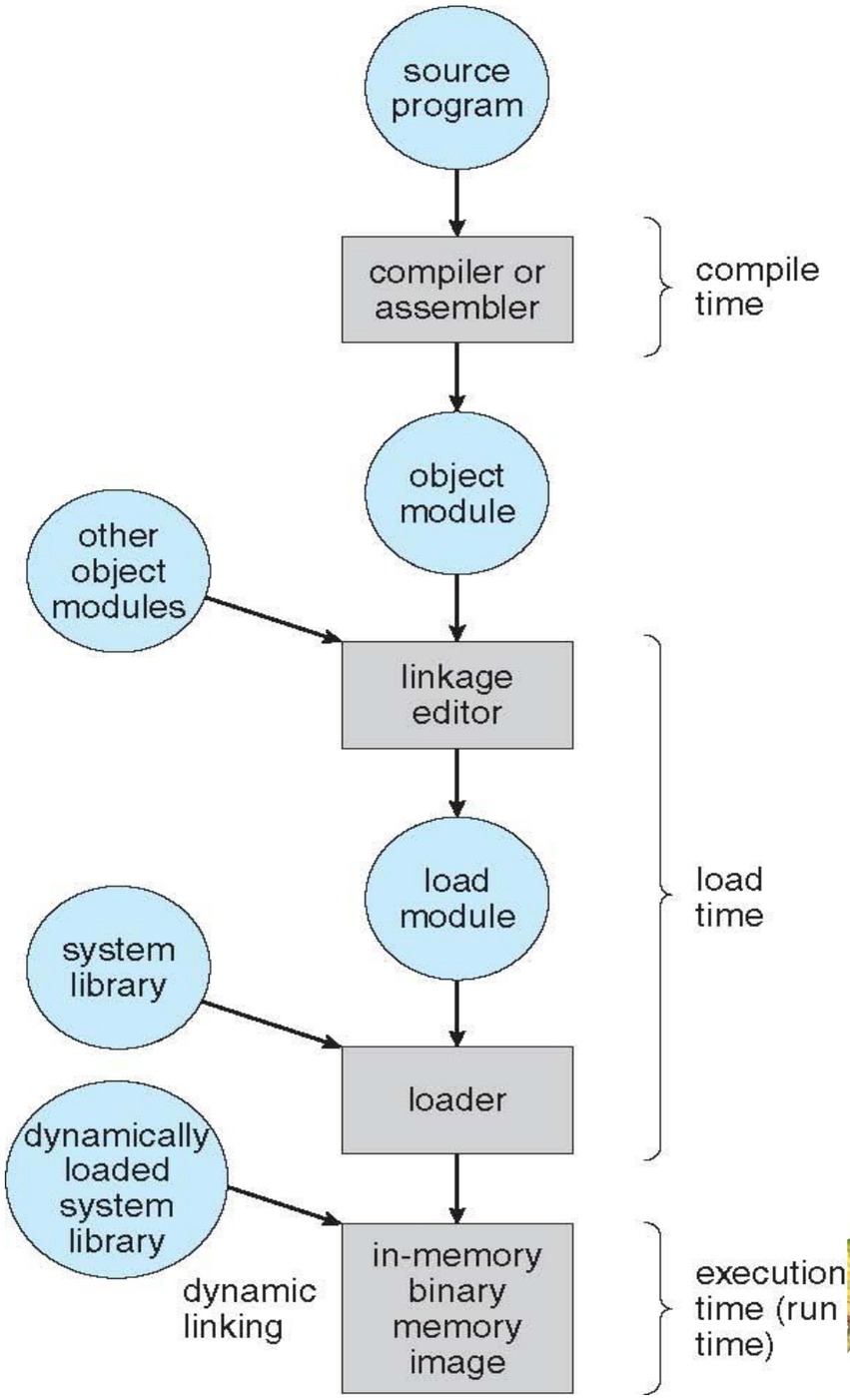
地址重定位

- 地址重定位
  - 物理地址/绝对地址：内存单元的地址
  - 逻辑地址/相对地址：通过编译链接，产生出相对于“0”计算的地址空间



地址绑定

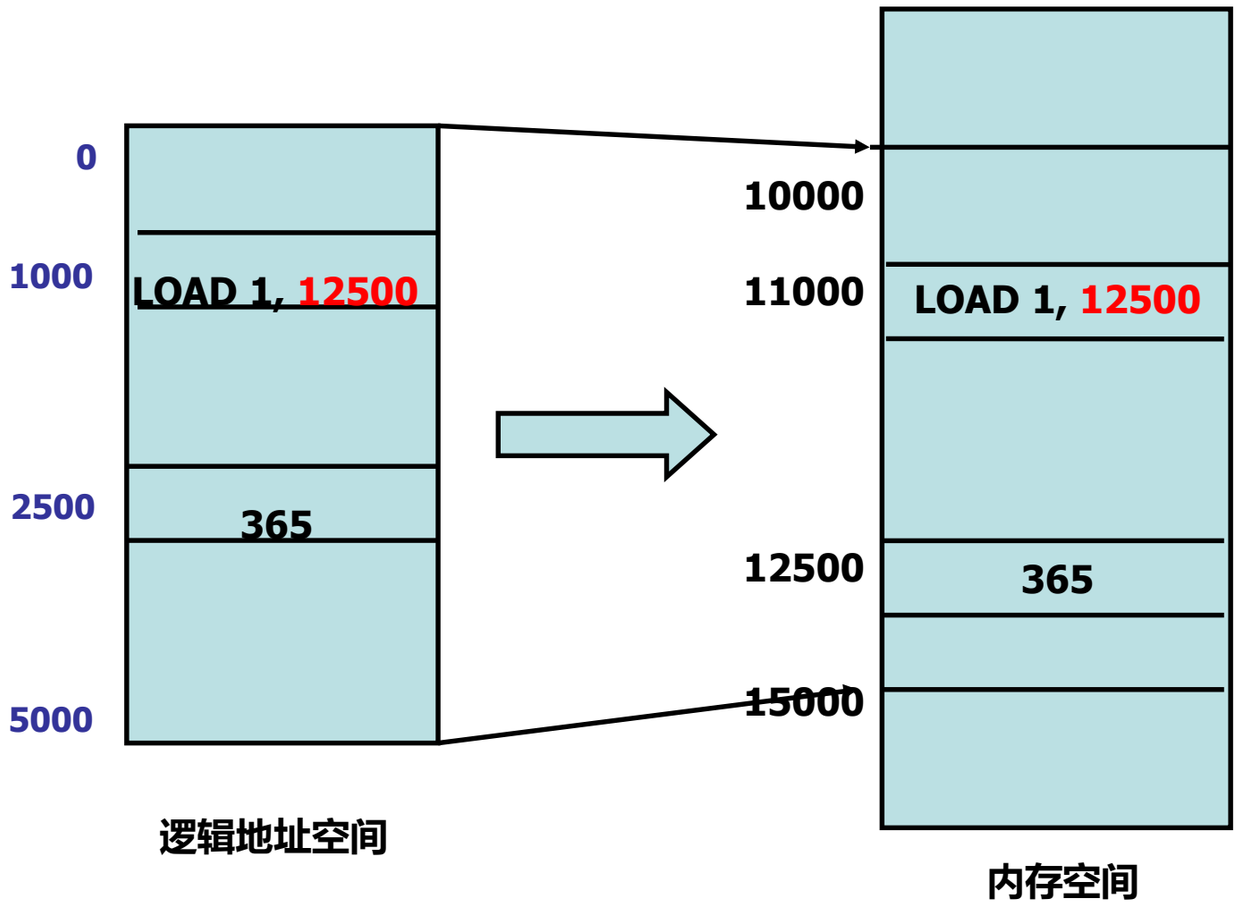
- 地址绑定（bind）的三种情况
  - 绝对定位方式
    - 编译时生成绝对地址
  - 静态重定位方式
    - 编译时生成可重定位代码
    - 加载时生成绝对地址
  - 动态重定位方式
    - 编译时生成可重定位代码
    - 执行时进行地址转换
    - 需要硬件支持，如：重定位寄存器



1、绝对定位方式

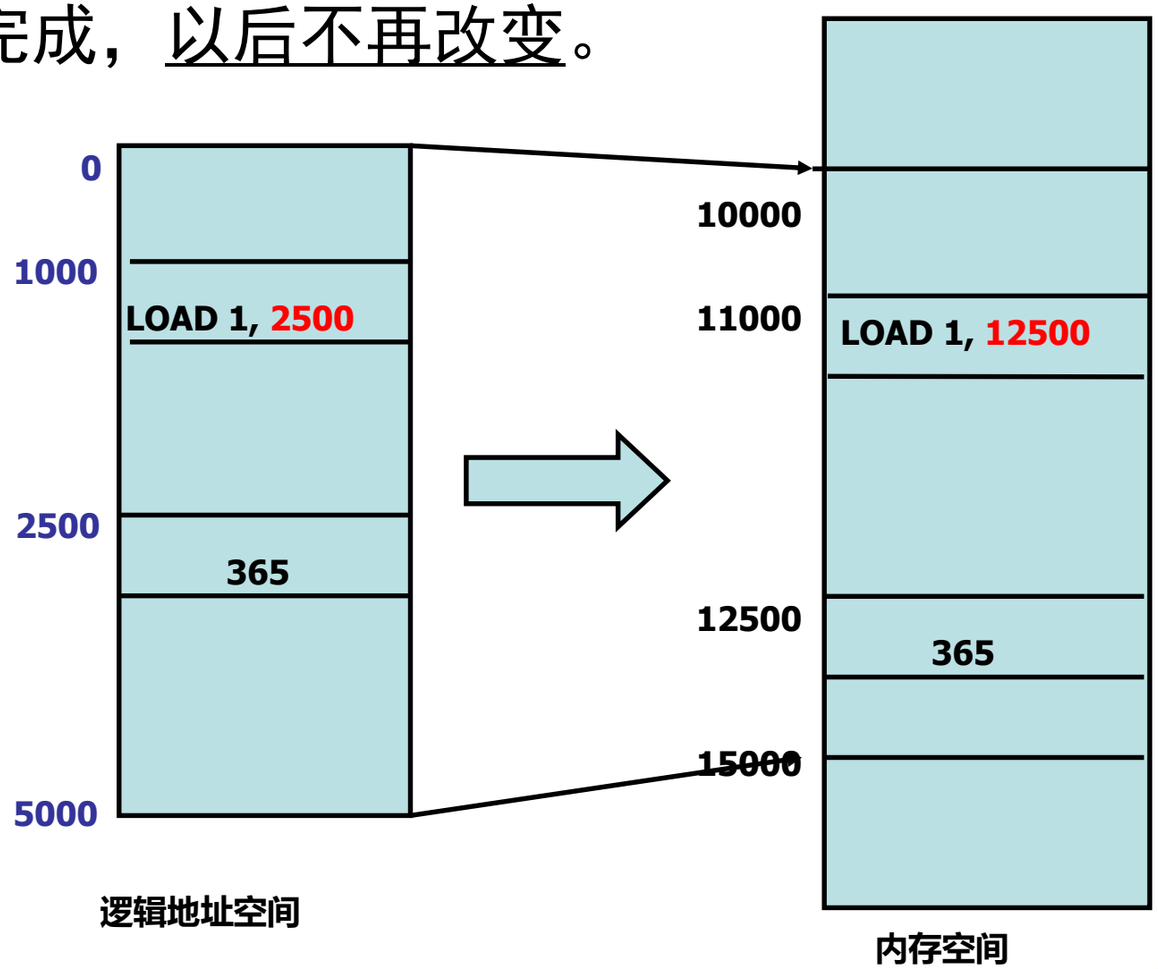
- 编译程序将产生绝对地址的目标代码；
- 根据绝对地址将程序和数据装入内存；

- 特点
  - 编程人员要熟悉内存
  - 程序在内存不能移动
  - 不适用多道程序设计环境



2、静态重定位方式

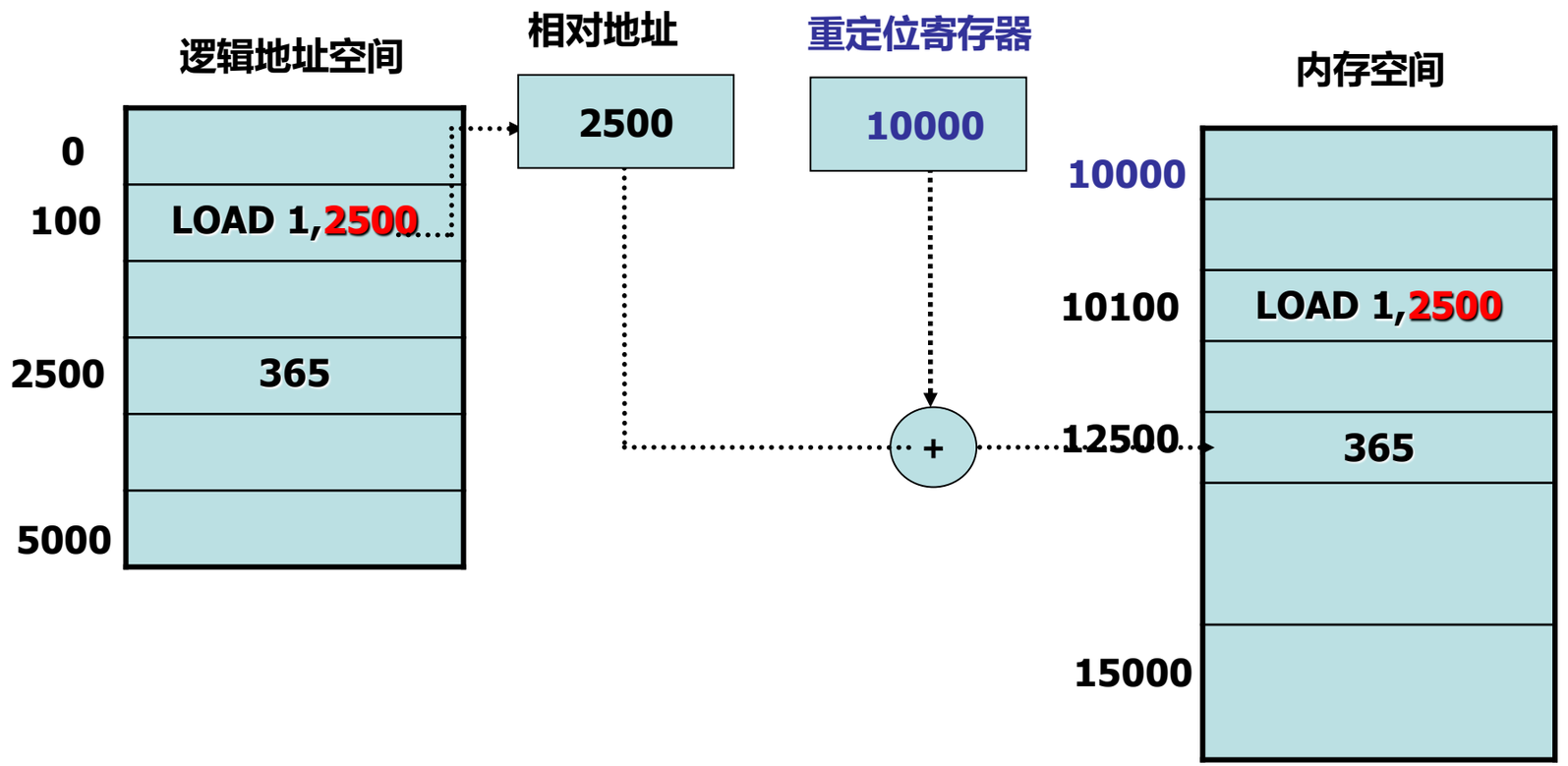
- 编译程序将产生相对地址的目标代码；
- 装入时需要地址映射（相对地址->绝对地址）；
- 地址变换只是在装入时一次完成，以后不再改变。



- 特点
  - 适用于多道程序环境
  - 程序在内存中不能移动；

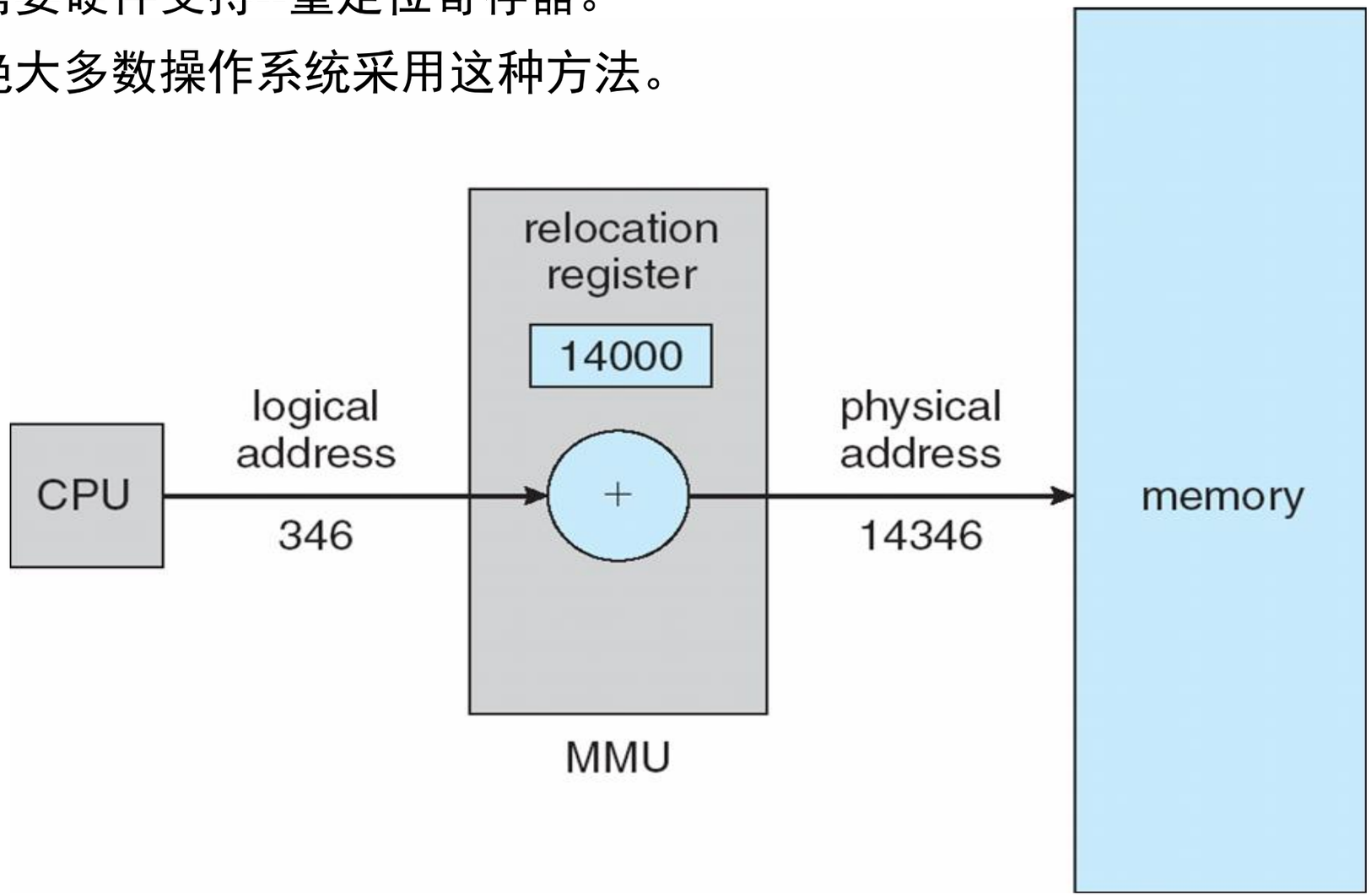
3、动态重定位方式

- 编译程序将产生相对地址的目标代码；
- 装入时，并不立即把相对地址转换为绝对地址，而是把这种地址转换推迟到程序要执行时才进行。



3、动态重定位方式

- 特点
  - 程序装入内存后可移动
  - 需要硬件支持--重定位寄存器。
  - 绝大多数操作系统采用这种方法。



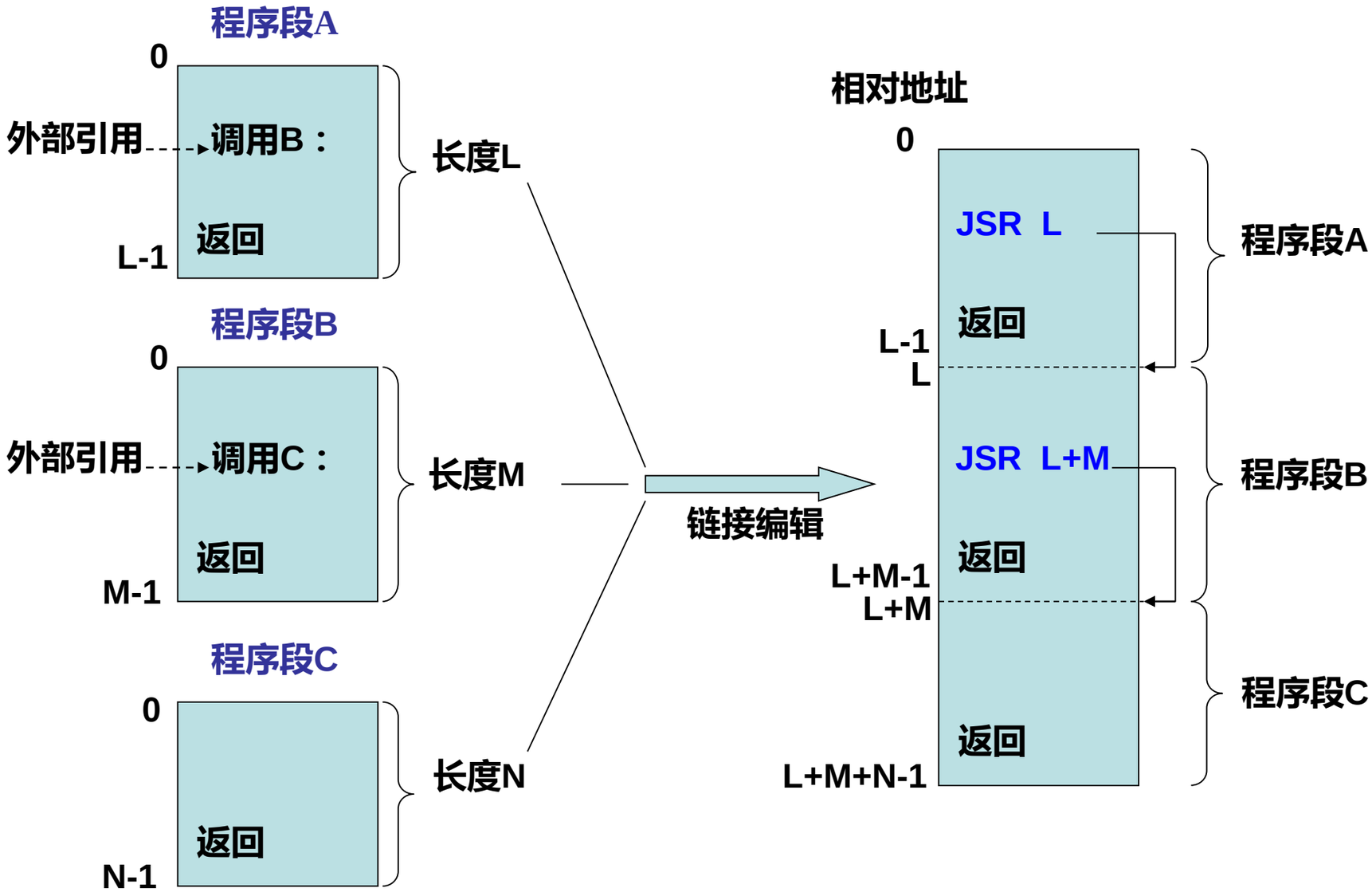
程序的链接

- 将经过编译后所得到的目标模块和它们所需要的库函数，装配成一个完整的装入模块。
- 链接的三种方式：
  - 静态链接
  - 装入时动态链接
  - 运行时动态链接

程序的链接

- 须解决的问题：
  - 变换外部调用符号
    - 将每个模块中所用的外部调用符号，变换为相对地址。
  - 对相对地址的修改
    - 由编译产生的所有目标某块的地址都是相对地址；

程序的链接


程序段A
0
外部引用
调用B：
返回
L-1
长度L
程序段B
0
外部引用
调用C：
返回
M-1
长度M
程序段C
0
返回
N-1
长度N
链接编辑
相对地址
0
JSR L
返回
L-1
L
JSR L+M
返回
L+M-1
L+M
返回
L+M+N-1
程序段A
程序段B
程序段C

动态链接

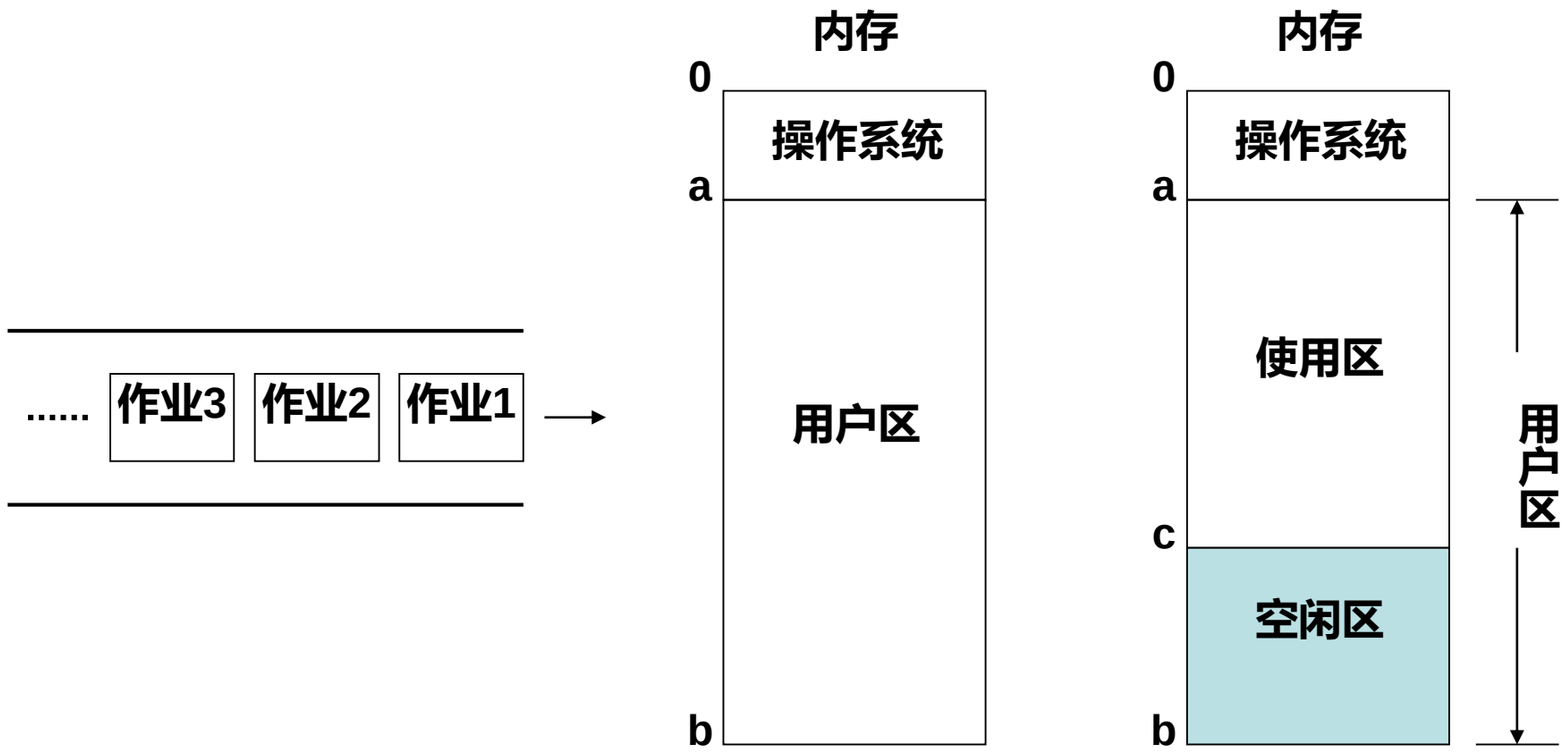
- 静态链接（static linking）
  - 整个链接过程发生在程序运行之前
  - 链接时，将库的内容加入到可执行程序中
  - 静态链接生成的可执行文件较大，占用内存空间
- 动态链接（dynamically linking）
  - 把链接的过程推迟到了运行时再进行
  - 这一特点通常用于系统库，所有的进程可以共享库代码
    - Windows：dynamic link library (DLL)
    - Linux：Shared Library

连续内存分配

- 连续分配方式
  - 每个进程分配一段地址空间连续的内存空间。
- 分类
  - 单一连续分配
  - 分区式分配
    - 固定分区分配
    - 动态分区分配
- 可重定位分区分配

单一连续分配

- 内存为两个部分
  - 系统区：供操作系统使用
  - 用户区：供用户程序使用，只能有一个用户程序

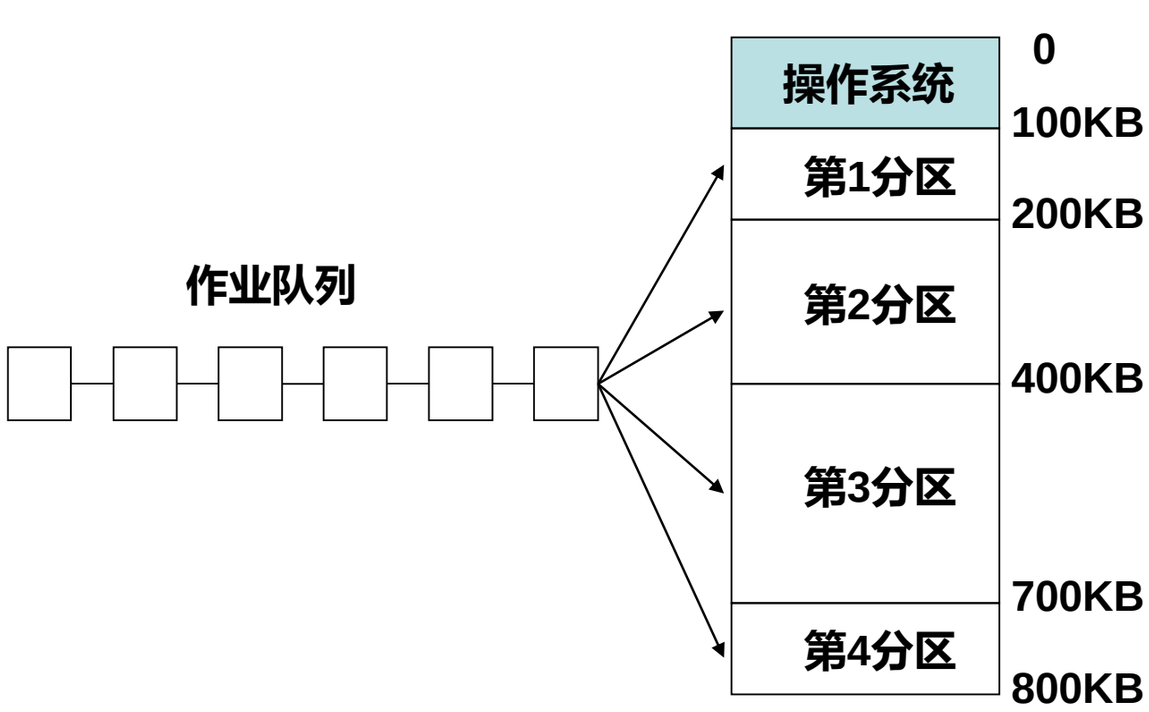


单一连续分配

- 特点
  - 只适用于单道程序的情况。
  - 若用户作业比用户区大，则无法运行
  - 若用户作业比用户区小，则造成内存浪费
  - 设置界限寄存器，限制用户程序访问操作系统。

固定分区分配

- 基本原理
  - 将内存空间划分为若干个固定大小的分区（大小可以不等）
  - 每个分区中可以装入一道作业，
  - 当有空闲分区时，选择一个适当大小的作业装入该分区；
  - 当作业结束时，释放该分区。



分区分配表

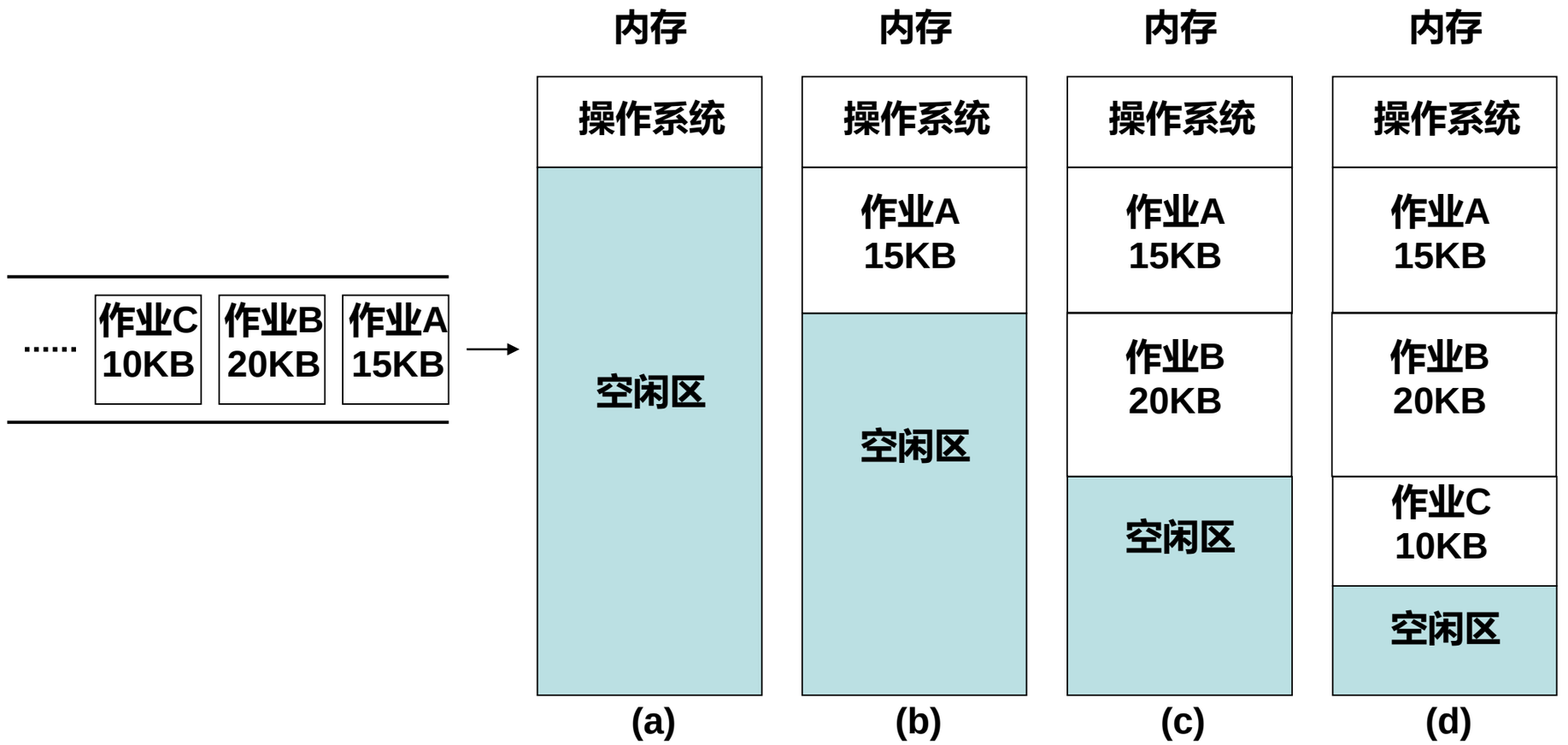
分区号	长度（K)	起址(K)	状态
1	100	100	已分配
2	200	200	已分配
3	300	400	已分配
4	100	700	未分配

固定分区分配

- 特点
  - 主要用于批处理系统，实现多道程序的内存管理
  - 程序的大小受分区大小的限制；程序数受分区数限制
  - 每个分区都有可能产生内部碎片，引起内存的浪费。

动态分区分配

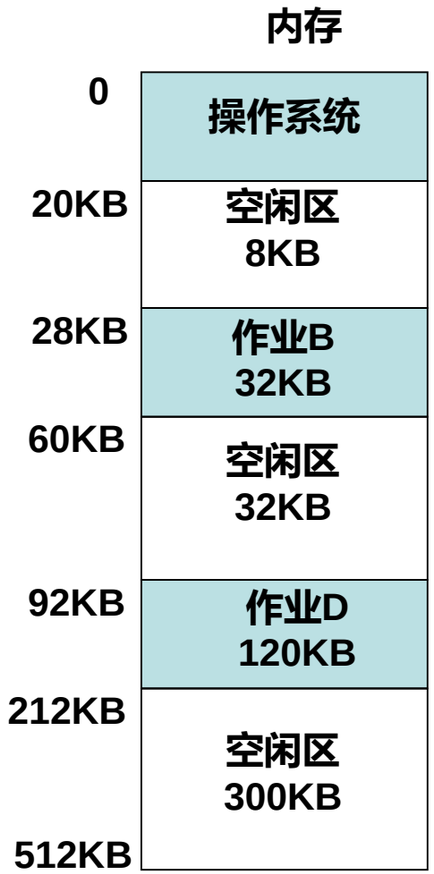
- 基本思想
  - 作业要求装入内存时，依照作业的大小划分分区。
  - 每个分区容纳一个进程。



动态分区的管理

- 可变分区的管理与组织方式
  - 表格法
    - 空闲分区表
    - 已分配分区表

- 特点
  - 管理简单
  - 需要占用一部分内存空间



(a)

序号	起始地址	尺寸	状态
1	—	—	空
2	28KB	32KB	作业B
3	—	—	空
4	92KB	120KB	作业D
5	—	—	空
⋮			

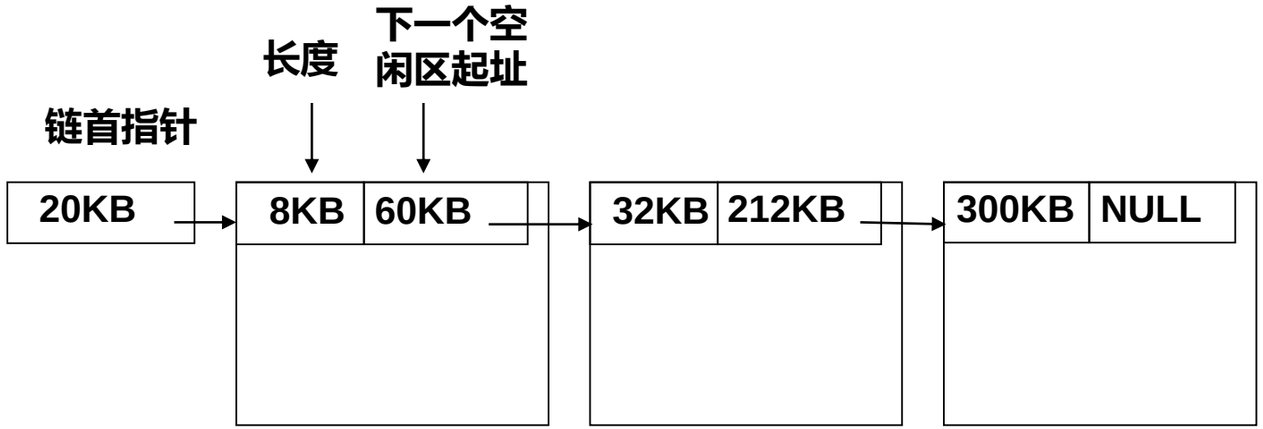
(b) 已分配区表

序号	起始地址	尺寸	状态
1	20KB	8KB	空闲
2	60KB	32KB	空闲
3	212KB	300KB	空闲
4	—	—	空
5	—	—	空
⋮			

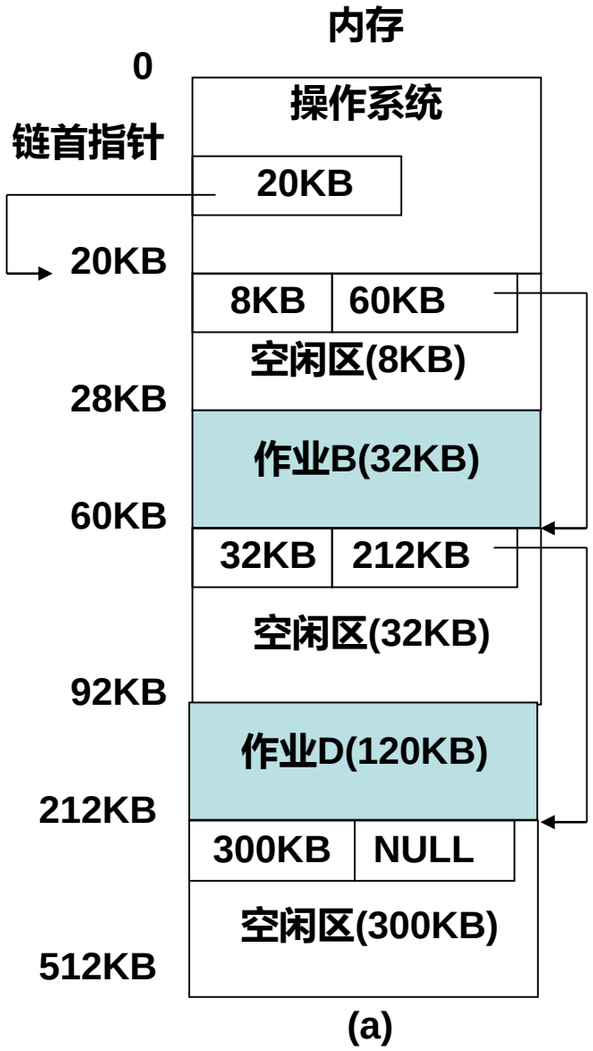
(c) 空闲区表

动态分区的管理

- 可变分区的管理与组织方式
  - 链表法



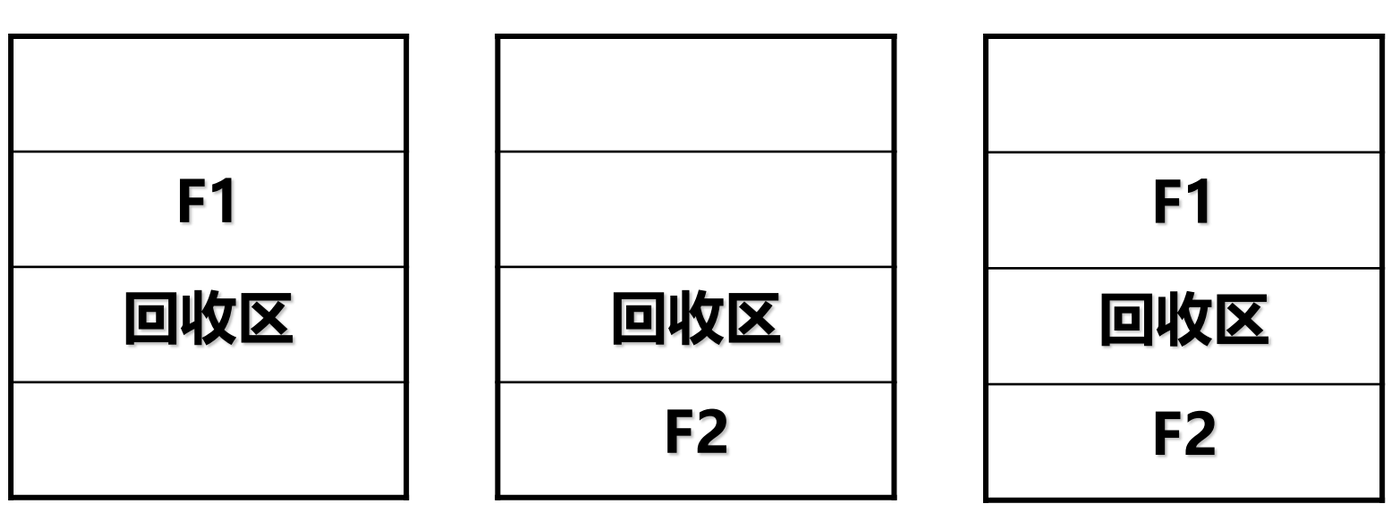
(b) 空闲分区链



(a)

动态分区分配

- 内存回收
  - 上邻空闲区：合并，改大小
  - 下邻空闲区：合并，改大小，首址。
  - 上、下邻空闲区：合并，改大小。
  - 不邻接，则建立一新表项。



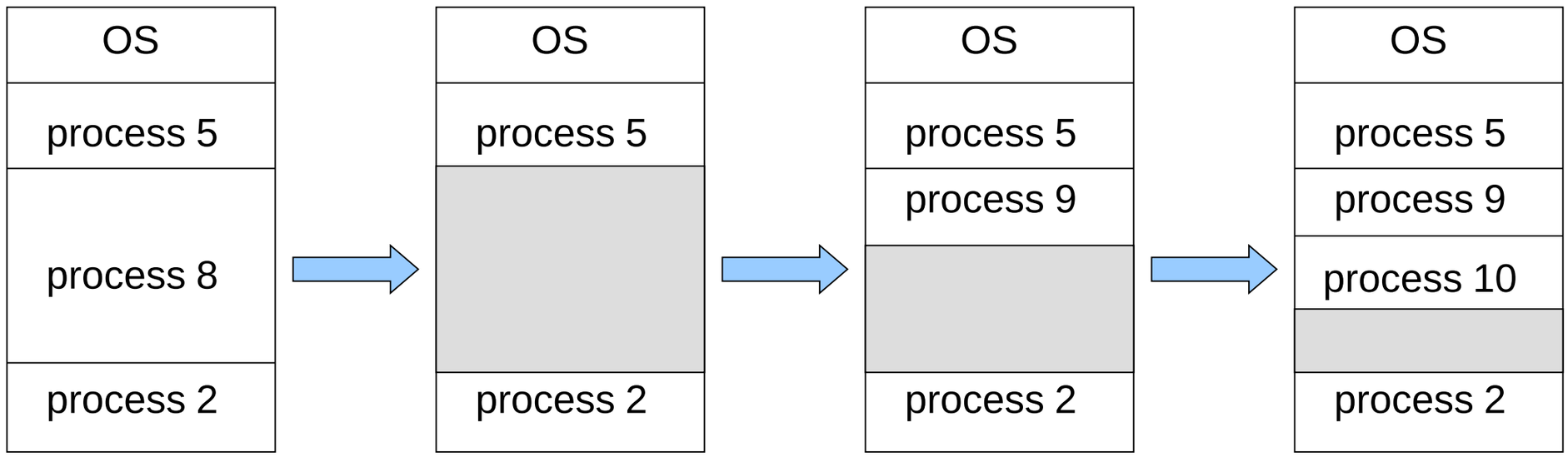
动态分区分配

- 如何从空闲分区中选择一个合适的分区

- 常用分配算法
  - 首次适应
  - 循环首次适应算法
  - 最佳适应算法
  - 最坏适应算法

1、首次适应算法

- 算法
  - 将空闲分区按地址顺序 排列。
  - 进行内存分配时，低地址开始顺序查找，分配第一个足够大的分区；



- 优点
  - 优先分配低地址部分的空闲分区，保留高地址部分
- 缺点
  - 在低址部分集中了许多小分区，难以利用；

2、循环首次适应算法

- 算法
  - 将空闲分区按地址顺序构成循环链表。
  - 进行内存分配时，不再从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，循环查找；
- 优点
  - 内存中的空闲分区分布均匀
- 缺点
  - 缺乏大的空闲分区

3、最佳适应算法

- 算法
  - 空闲分区按大小递增排序构成队列。
  - 从队列头开始查找，当找到第一个满足要求的空闲区时，则停止查找；
- 优点
  - 找到的空闲分区最接近要求的大小
- 缺点
  - 会产生非常小的碎片，难以利用

4、最差适应算法

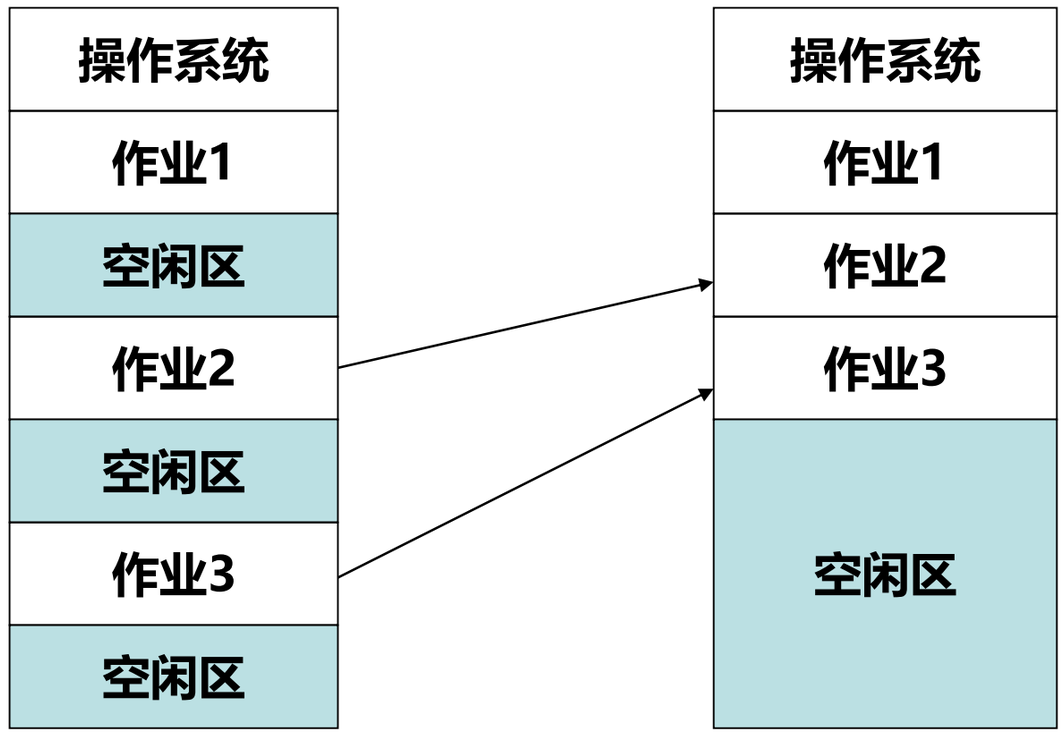
- 算法
  - 空闲分区按大小递增排序构成队列。
  - 查找最大的空闲区；
- 优点
  - 剩余的分区空间最大
- 缺点
  - 在空间利用率方面较差

碎片

- 内部碎片（ Internal Fragmentation ）
  - 分区内部的无法利用的小的空闲区域
- 外部碎片（ External Fragmentation ）
  - 随着进程的装入和移出，空闲内存被分为小片段。
- 解决方法
  - 紧凑（compaction）
  - 允许物理地址空间不连续
    - 分页（paging）
    - 分段（segmentation ）

紧凑

- 紧凑/拼接（compaction）
  - 移动内存，使所有空闲区域合并为一整块空闲区域。
  - 仅在地址是动态重定位的情况下可行。



基本分页存储管理

- 离散分配方式
  - 允许将一个进程分散的分配到许多不相邻的分区中；
  - 程序全部装入内存；
- 分类
  - 分页存储管理
  - 分段存储管理
  - 段页式存储管理

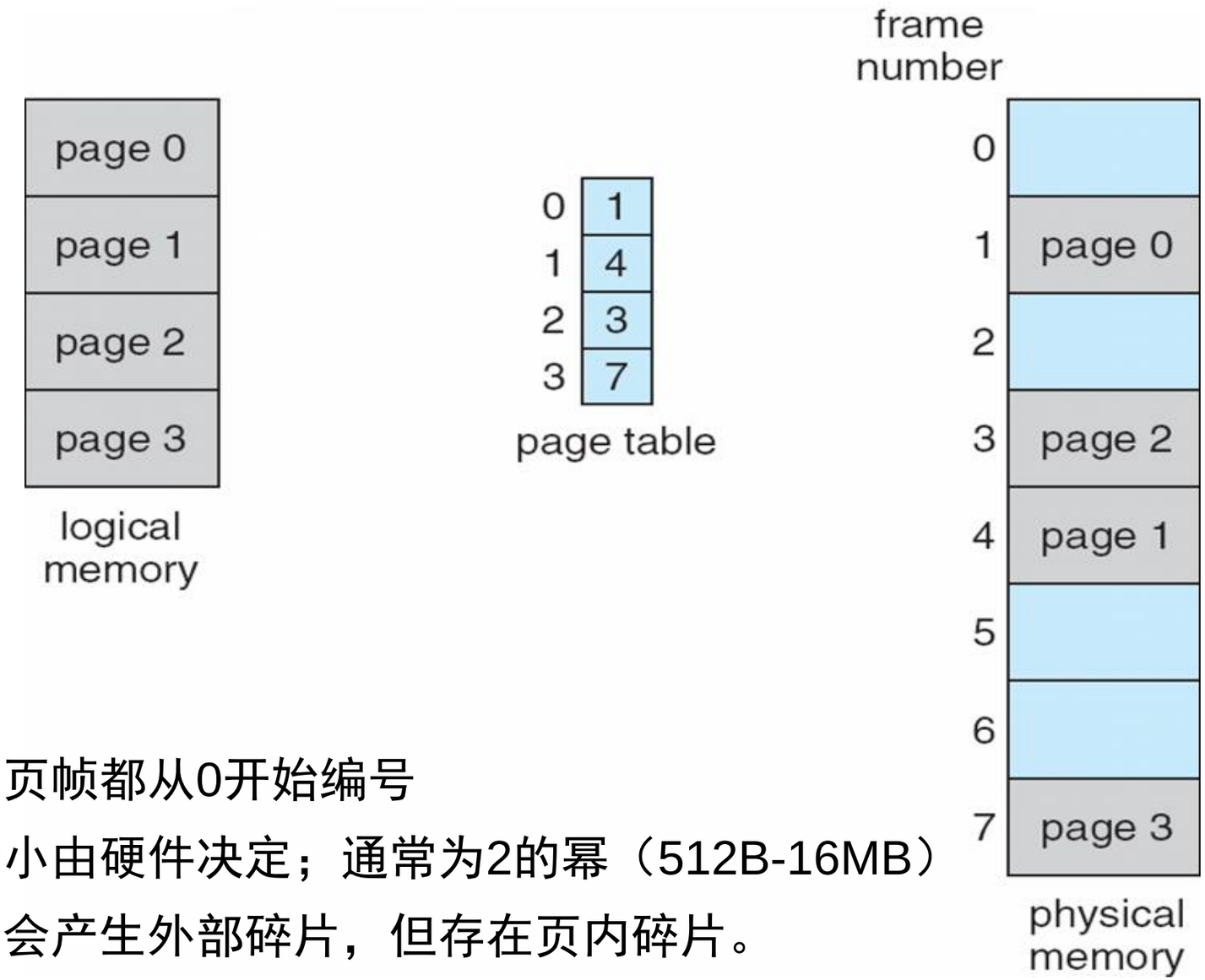
分页

- 为什么要引进分页技术
  - 碎片问题
  - 程序大小受限
- 解决方法
  - 允许进程的物理地址空间可以不连续—分页
  - 程序在装入内存时，无须全部装入，只装入部分就可运行—虚拟存储

分页

- 基本概念
  - 帧/物理块(frame)
    - 物理内存分为固定大小的块，帧号：0#、1#、2#...
  - 页(page)
    - 逻辑内存分为同样大小的块，页号：0#、1#、2#...
  - 页表(page table)
    - 页面映射表，记录了页号和帧号的映射关系

分页模型



- 说明
  - 页面和页帧都从0开始编号
  - 页面大小由硬件决定；通常为2的幂（512B-16MB）
  - 分页不会产生外部碎片，但存在页内碎片。
  - 页面之间地址不再连续,需要有相应地址变换机构

地址结构

- CPU生成的每个地址分为两个部分：
  - 页号（p）
  - 页内偏移（d）

page number	page offset
p	*d*
m - n	*n*

- 说明
  - 如果逻辑地址空间为2^m，页面大小为2^n，那么逻辑地址的高m-n位表示页号，低n位表示页内偏移

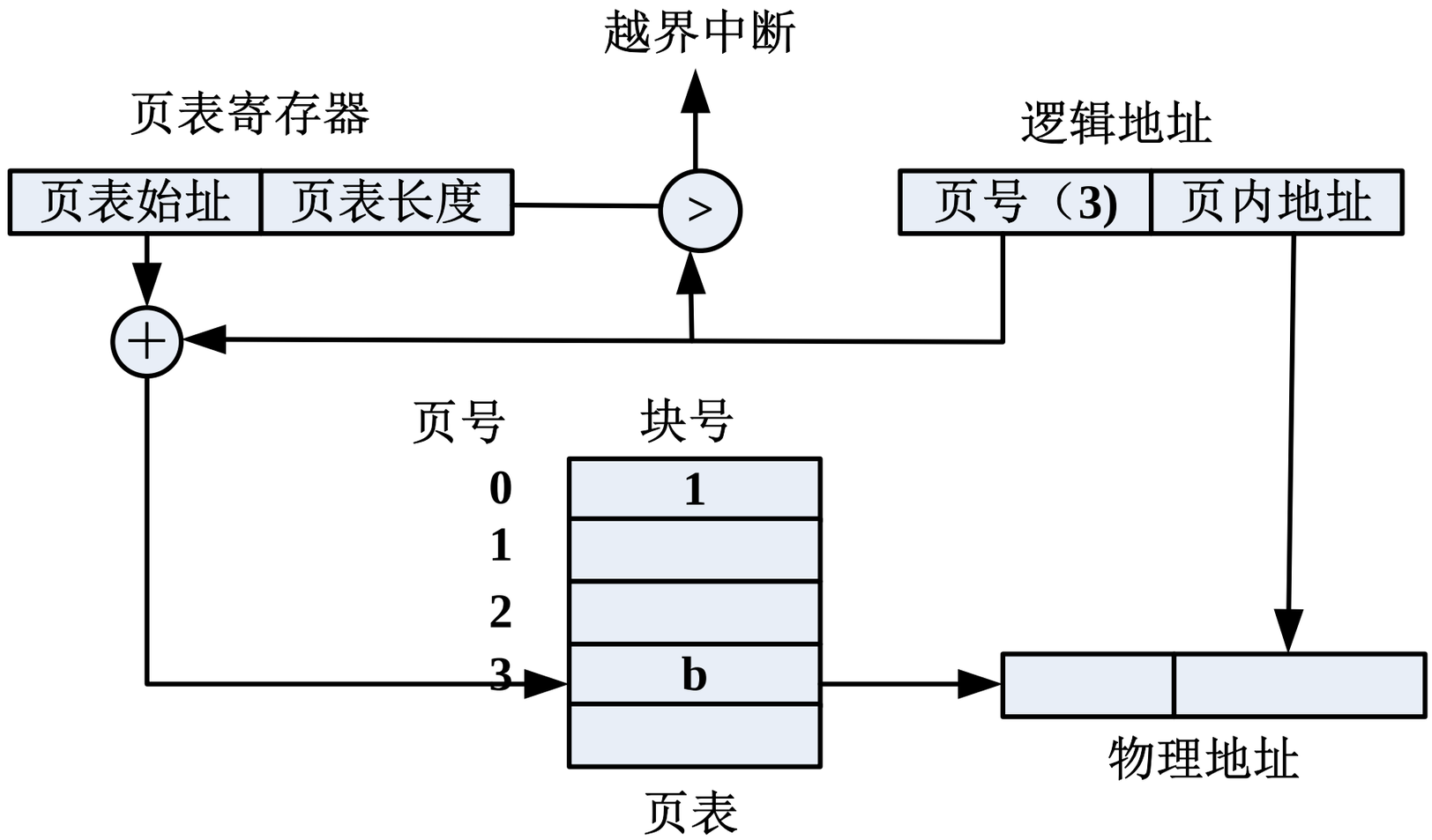
4.4.1 页面与页表

- 页面与页框如何建立联系
  - 页面映射表/页表
    - 用来说明页号和物理块号的对应关系
    - 每个进程有一张页表；

页号	块号
0	2
1	3
2	6
3	8
4	9
…	…

图4-13 分页系统的地址变换机构

- 地址变换过程



分页系统地址变换机构

4.4.1 页面和页表

- 说明
  - 操作系统为每一个进程建立一个页表
  - 页表的指针存储进程控制块中。
  - 页表项

页号	物理块号	状态位P	访问字段A	修改位M	外存地址

1、基本地址变换机构

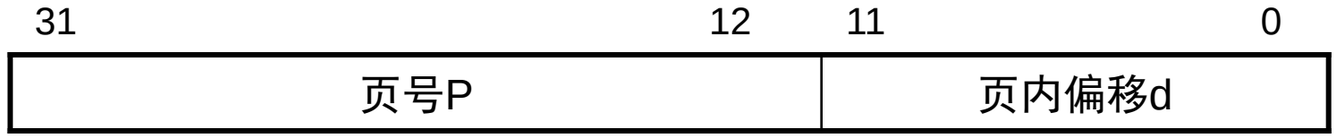
- 如何将逻辑地址转换为页号和页内偏移
  - 假定一个逻辑地址空间中的地址为A，页面大小为L，则页号P和页内地址d为：

$$P = INT\left[\frac{A}{L}\right] \qquad d=[A]\ MOD\ L$$

  - 例如：逻辑地址为：A＝2170B，页面大小为1KB
    则P=2, d=122

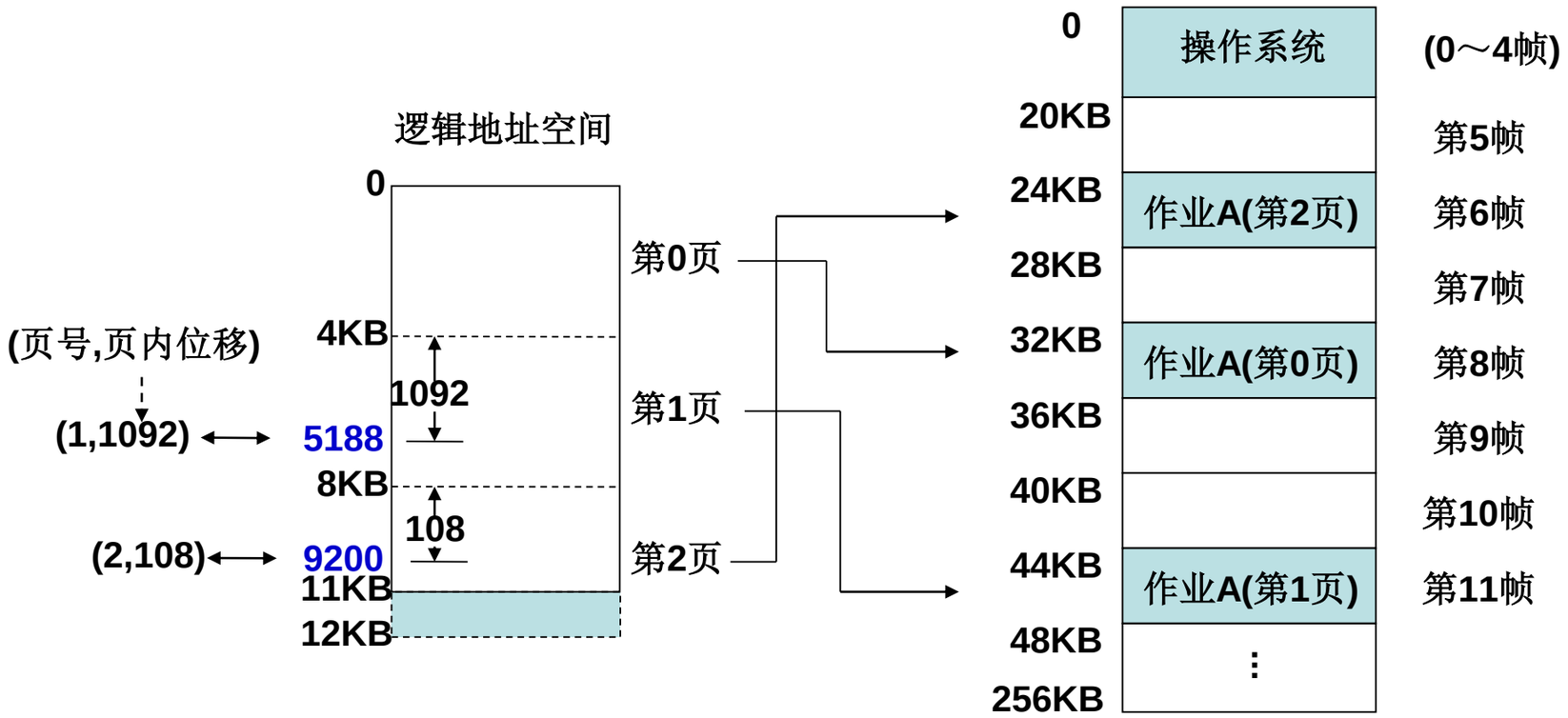
地址变换

- 例子：假设32位的寻址系统，如果页面大小为4KB，则页面数为2^{20}，计算逻辑地址对应的物理地址。（1）5188；（2）9200



页表

页号	帧号
0	8
1	11
2	6

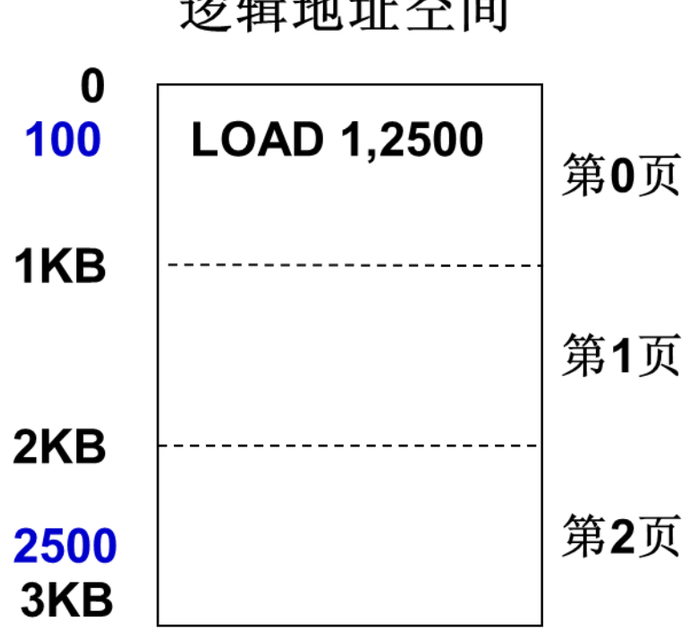


练习

- 设页面大小为1KB，指令LOAD 1,2500的逻辑地址为100，通过图示的页表求出该指令中对应的物理地址。
  ① 100
  ② 2500

页表

页号	帧号
0	2
1	3
2	8



练习

- 某虚拟存储器的用户编程空间共32个页面，每页为1KB，内存为16KB。假定某时刻一用户页表中已调入内存的页面对应的物理块号如下表，试将逻辑地址0A5C和093C变换为物理地址。

页号	物理块号
0	5
1	10
2	4
3	7

分页机制

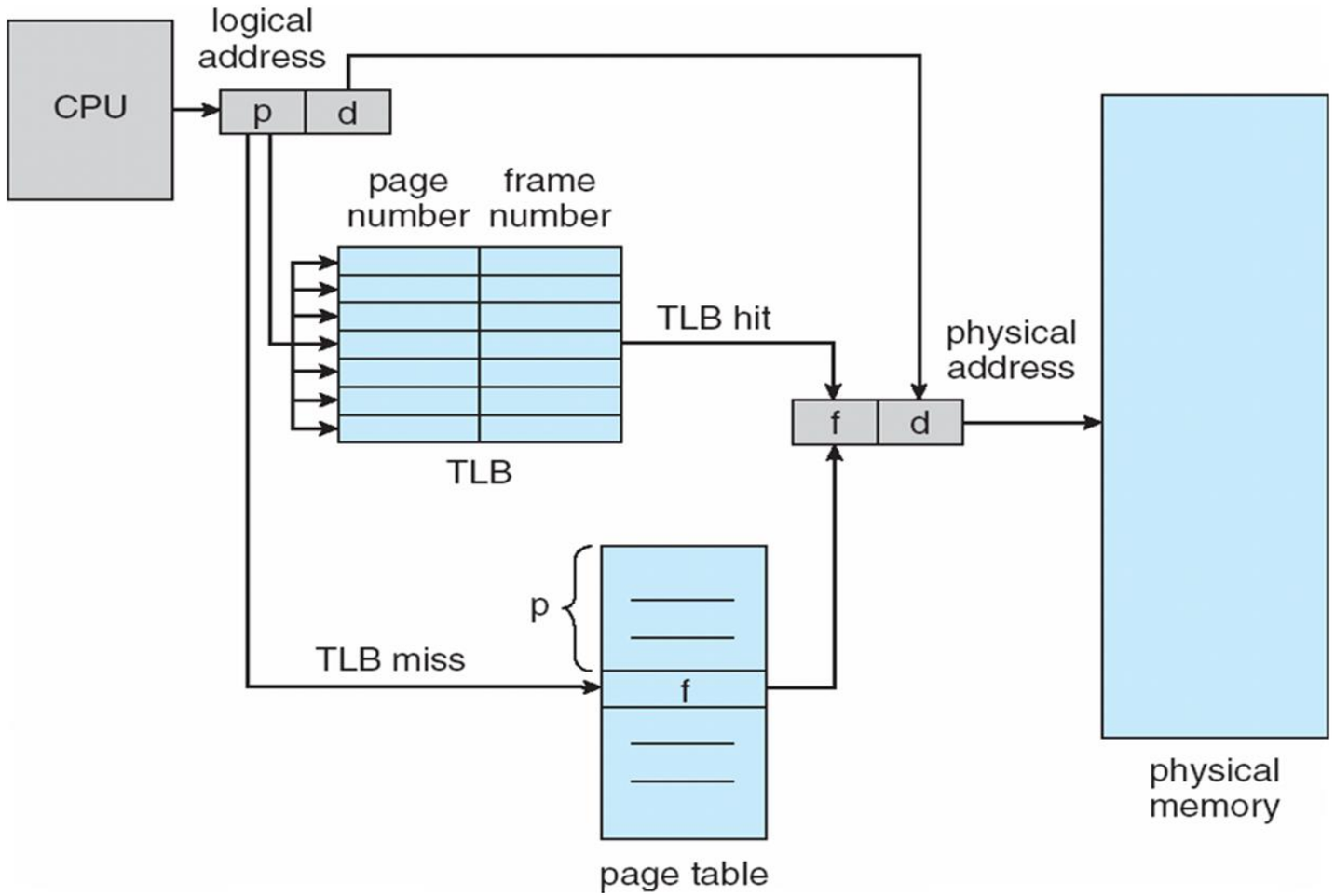
- 分页机制的一个重要特点是很清楚的分离了内存的用户视角和实际的物理内存。
  - 用户程序把内存看作一个单一的连续空间，只存储了自己这一个程序。
  - 事实上，用户程序在物理内存中是分散的，而不是连续的。
- 用户进程不可以访问不属于自己的内存。
  - 它无法对页表之外的内存进行寻址，而且页表也只包含了进程所拥有的那些页。

页表的实现

- 操作系统为每一个进程建立一个页表
- 页表的指针存储在进程控制块中
- 页表的硬件实现
  - 一组专用寄存器（页表比较小，如256个条目）
  - 页表放在内存，并将页表寄存器指向页表
    - 访问数据需要两次访问内存
  - 采用专用的硬件缓冲
    - TLB表：转换表缓冲区（translation look-aside buffer），也称为快表或联想寄存器（Associative Memory,AM）
    - TLB表的条目一般在64-1024之间
    - TLB表是一个具有并行查寻能力的高速缓冲寄存器，配合内存中的页表，一起完成地址转换的工作。

2、具有快表的地址变换过程

- 快表的工作机制
  - 先用页号与快表中所有表项同时（即并行）比较，得到匹配页号所对应的页帧号。
  - 如果在快表里没有匹配的页号时，才去访问页表并得到页帧号。并且将该页表项添加到快表中。



具有TLB表的地址变换

- 命中率（ Hit ratio ）
  - 在TLB 中找到指定页号的概率。
- 有效内存访问时间（ Effective Access Time， EAT ）

$$T = h \times t_1 + (1-h) \times t_2$$

  - h：快表的命中率
  - t1：在快表中找到页号的时间
  - t2：在快表中没有找到页号所需的时间

具有TLB表的地址变换

- 例：假定CPU访问内存的时间为100ns，访问快表的时间为20ns，若TLB表命中率可达80%，试问：
  1. 不采用TLB表，有效内存访问时间是多少？
  2. 采用TLB表，有效内存访问时间是多少？
  3. 当命中率为98%时，有效内存访问时间是多少？

- 不采用TLB表，有效内存访问时间为两次访问主存的时间：
  - 100 + 100＝200
- 采用TLB表，有效访问时间为：

  （100＋20）×80%＋(100+100+20)×(1-80%)＝140
- 命中率为98%时，有效访问时间为：

  （100＋20）×98%＋(100+100+20)×(1-98%)＝122

练习

- 有一页式系统，页表存放在主存中：
  - 如果对主存的一次存取时间需要1.5μs（微秒），试问实现一次页面访问的存取时间是多少?
  - 如果系统中增加TLB表，平均命中率为85%，当页表项在快表中，其查找时间可以忽略，试问此时的存取时间是多少？

练习

解：

- 若页表存放在主存中，则需两次访问主存：
  - 第一次：访问页表，确定所存取页面的物理地址。
  - 第二次：根据该地址存取页面数据。

  存取访问时间=1.5*2=3(μs)
- 增加快表后的存取访问时间

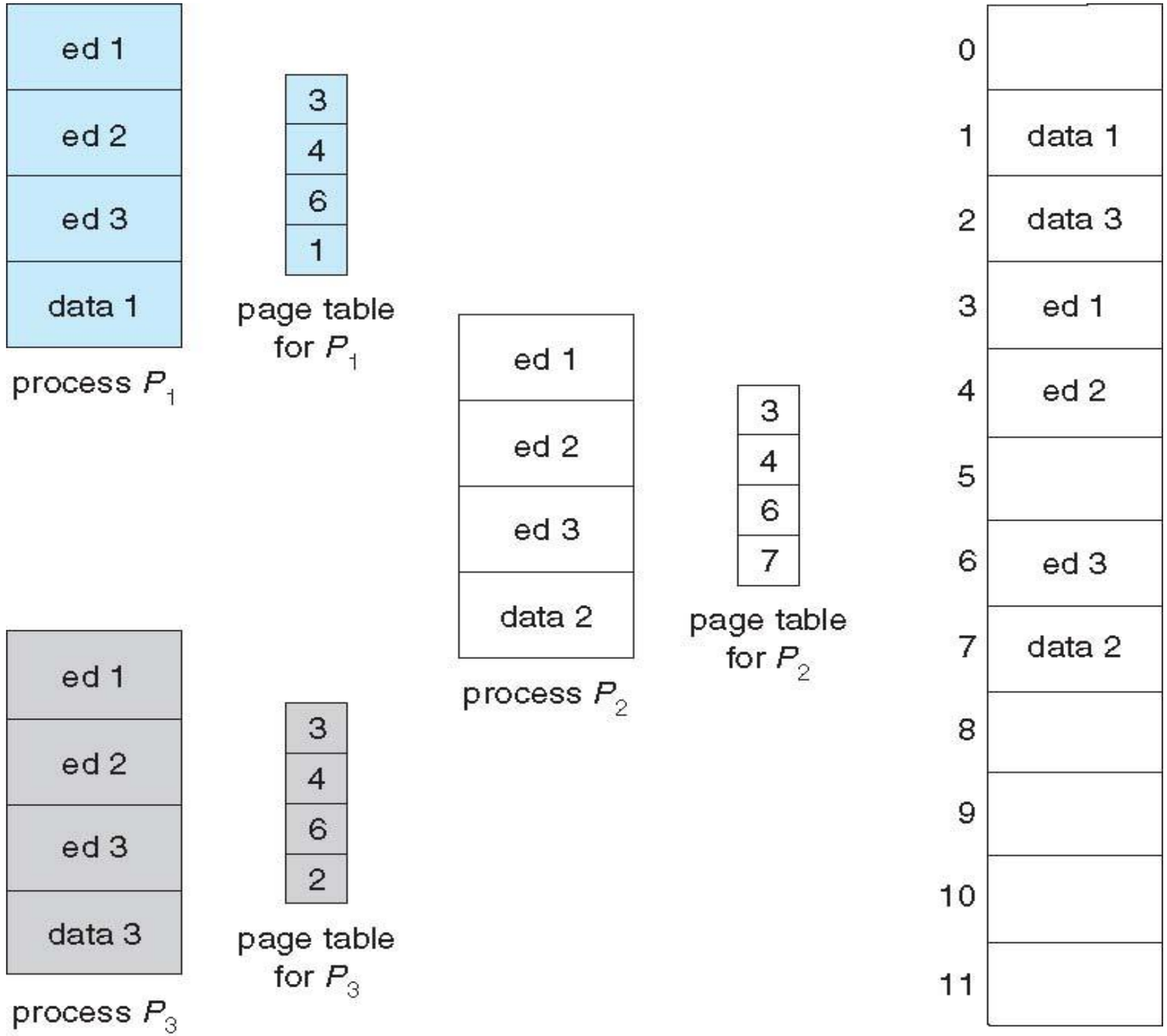
  T=h*t1+(1-h)*t2

  =0.85*1.5+(1-0.85)*2*1.5=1.725(μs)

共享页

- 例：一个支持40个用户的系统，每个用户都执行一个文本编辑器。如果文本编辑器包括150KB的代码和50KB的数据区
  - 如果不共享
    - 则需要200KB*40=8000KB
  - 如果代码是可重入代码（纯代码），即代码区可以共享
    - 150KB+50KB*40=2150KB

分页环境下的代码共享

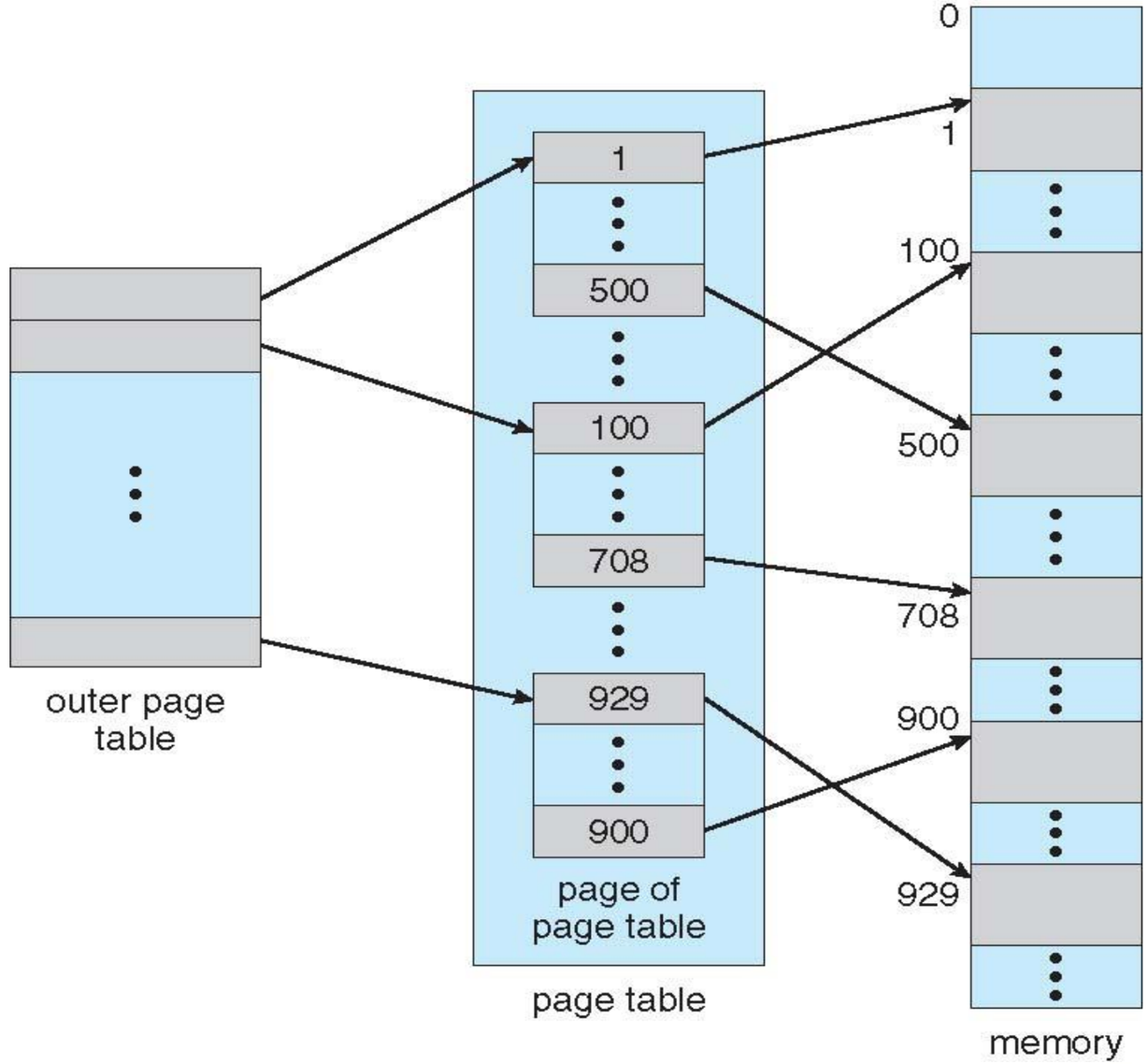

ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11

页表结构

- 多级页表
  - 为什么引入多级页表
    - 当逻辑地址空间很大，导致页表很大。
    - 假设32位逻辑地址空间，页面大小为4KB(2^{12}),则页表项可以达到1M（2^{20}）个，假设每个页表项占4B，则页表需要4MB
  - 解决方案
    - 两级页表：对页表进行“分页”处理
    - 将当前需要的页表项调入内存，其余页表项放磁盘。

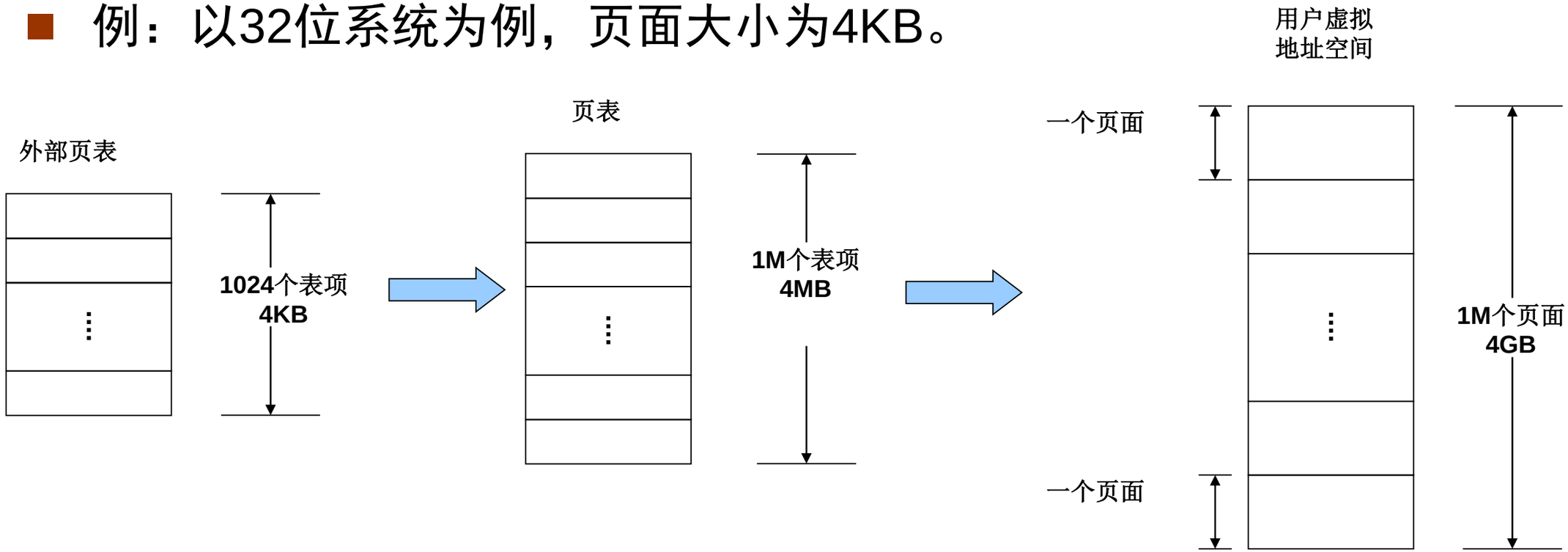
两级页表

- 两级页表
  - 先对虚拟地址空间分页，形成页表；
  - 再对页表分页，形成页表的页表，或外部页表。



两级页表

- 例：以32位系统为例，页面大小为4KB。



page number		page offset
p_1	p_2	d
10	10	12

page number	page offset
p	d
20	12

- p1 是外层页表的索引
- p2是外层页表页的偏移量

两级页表的地址转换

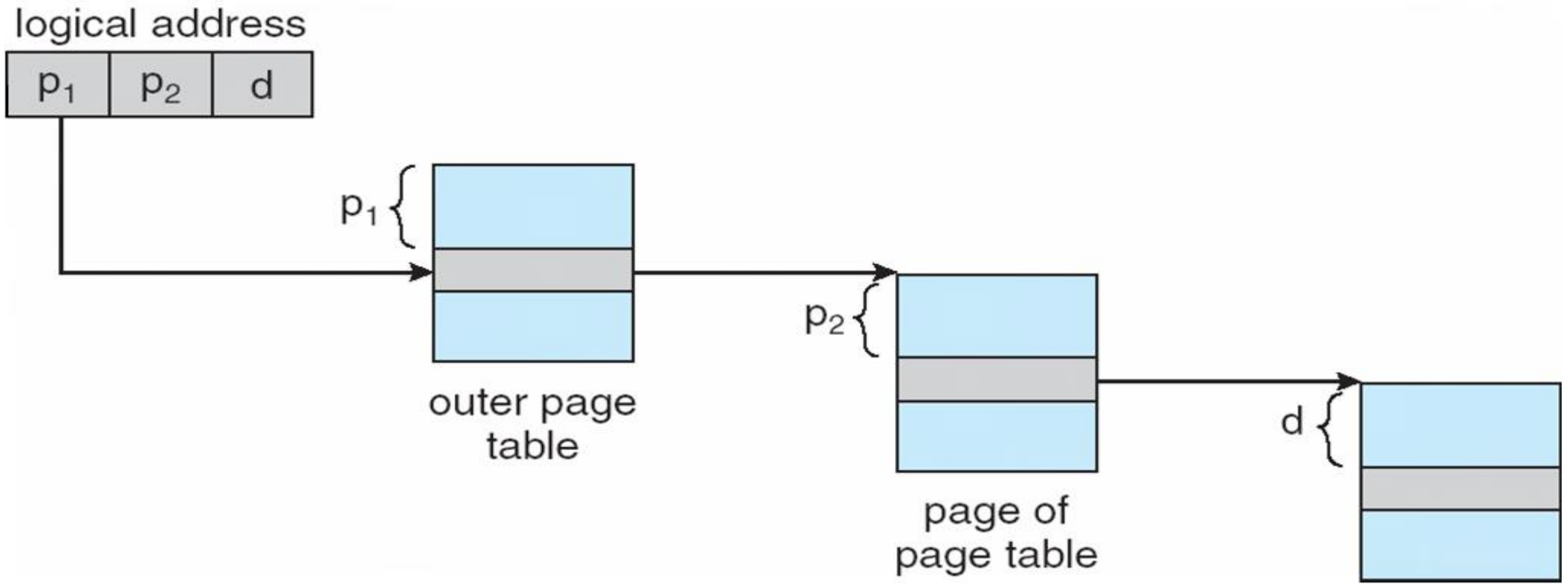
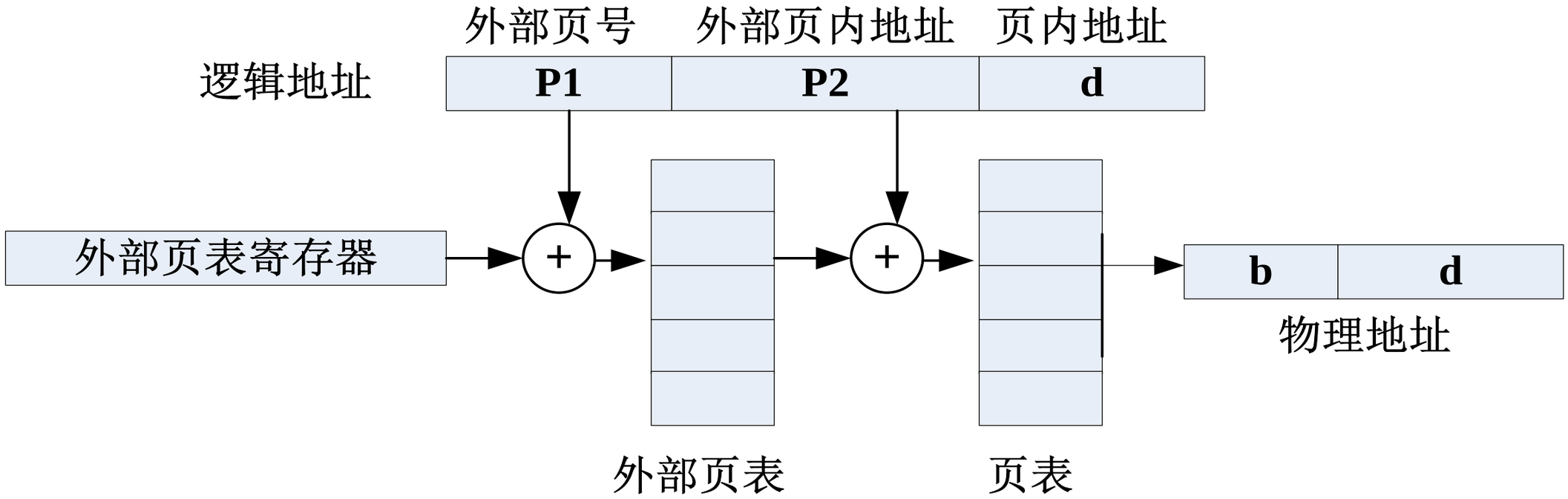

logical address
p1
p2
d
p1
outer page table
p2
page of page table
d

图4-16 具有两极页表的地址变换机构

- 地址变换过程



具有两级页表的地址变换机构

三级页表

- 引入
  - 对于64位的逻辑地址空间，两级分页方案就不再合适。
- 例：对于64位系统，页面大小为4KB。
  - 二级页表

outer page	inner page	offset
p_1	p_2	d
42	10	12

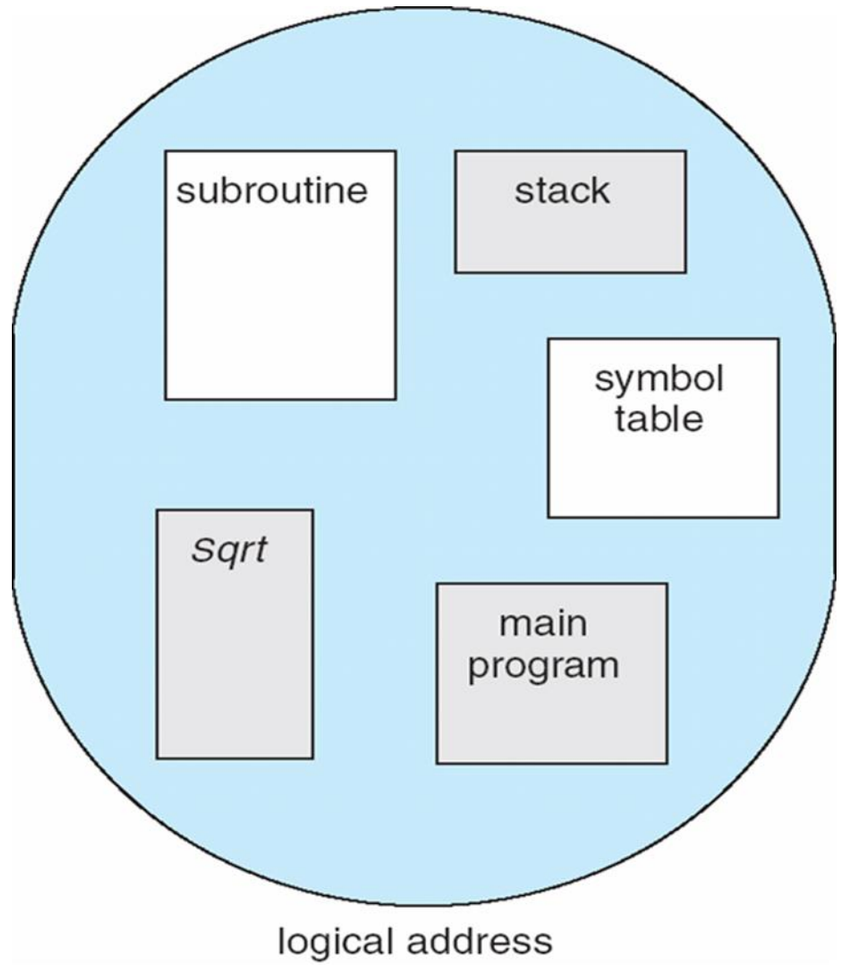
  - 三级页表

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

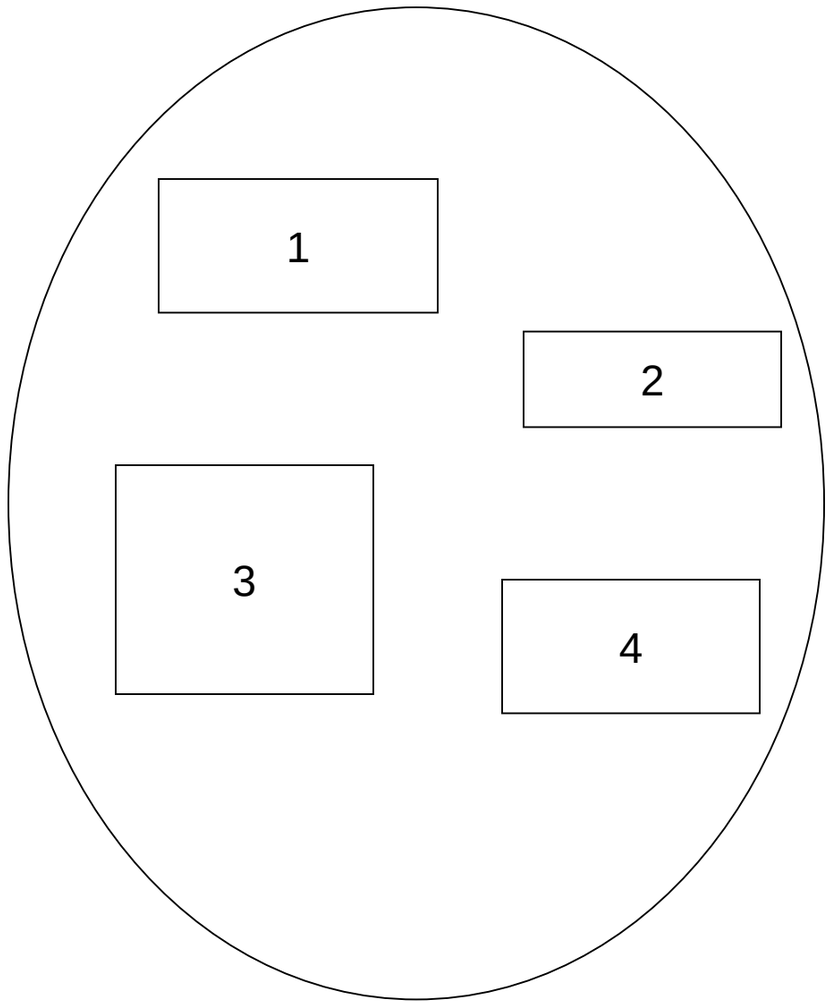
- 说明
  - 三级页表的外部页表依然很大2^{34}B
  - 访问速度降低（三次访问内存）。

分段

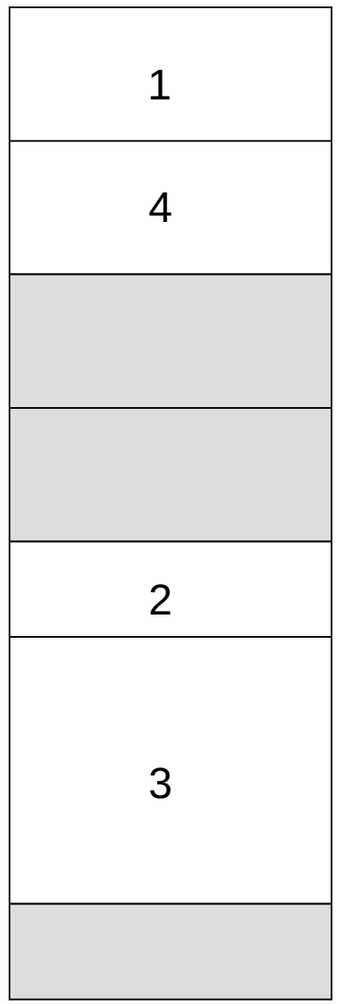
- 分段（Segmentation）是一种内存管理机制，它支持内存的用户视角。
  - 逻辑地址是段的集合。
  - 每个段都有其名称和长度。
  - 地址是由段名（段号）和段内偏移构成。
- C编译器可能会创建如下段
  - 代码
  - 全局变量
  - 堆
  - 栈
  - 标准C库函数
- 编译、连接、加载过程都以段为单位



分段



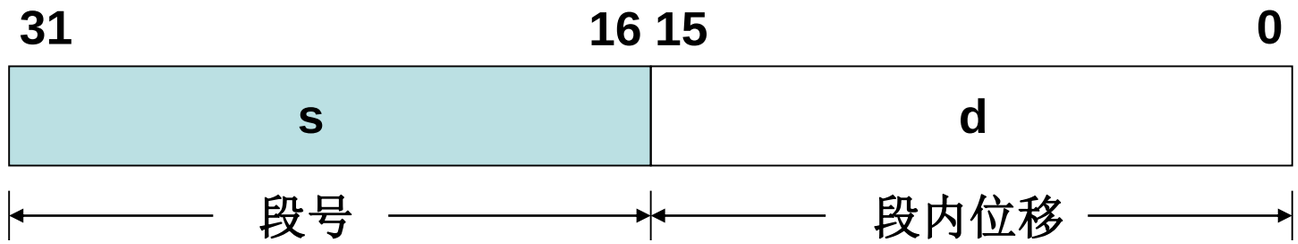
user space



physical memory space

分段存储结构的实现

- 地址结构
  - 逻辑地址应该是二维的：[段号，段内位移]。
  - 32位地址结构中，



  - 段号s：16位表示，2^{16}个段
  - 段内位移d：16位表示，每段最大长度是64KB。

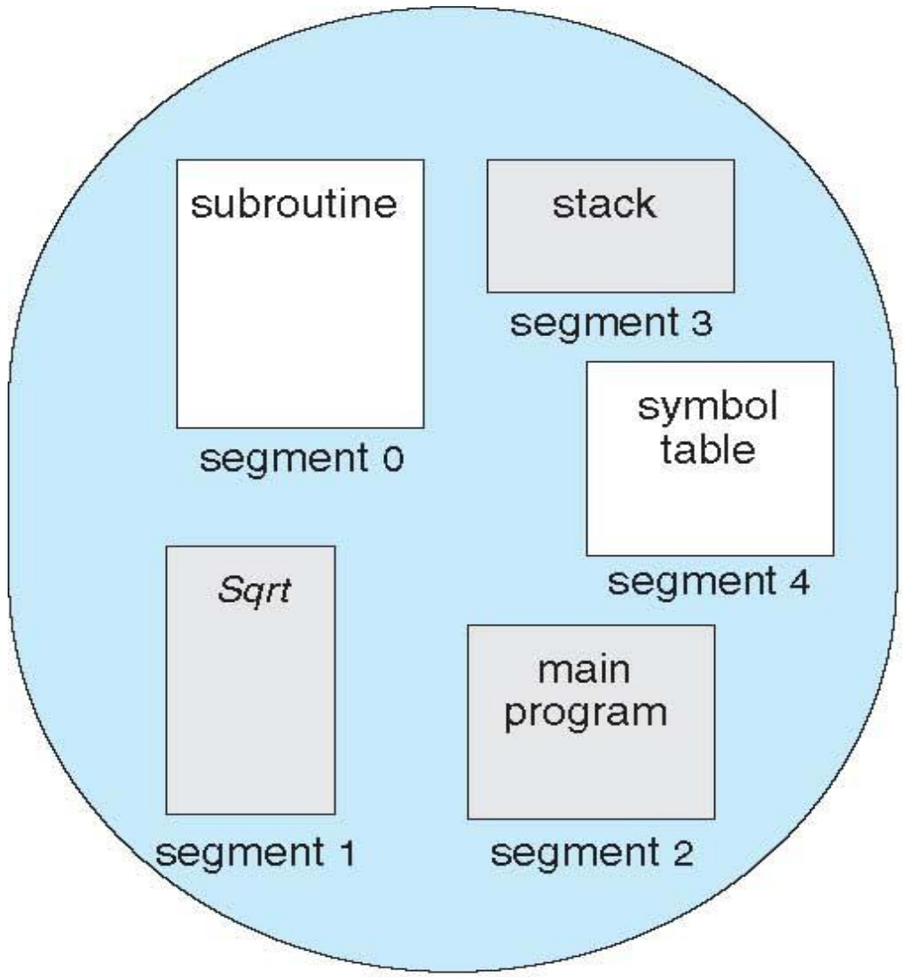
分段存储结构的实现

- 段表
  - 段映射表，该表存放的虚拟段号到该段所在内存基址的映射。
    - 段基址（segment base）：包含了段所在内存的初始物理地址
    - 段长（segment limit ）：指明了段的长度

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

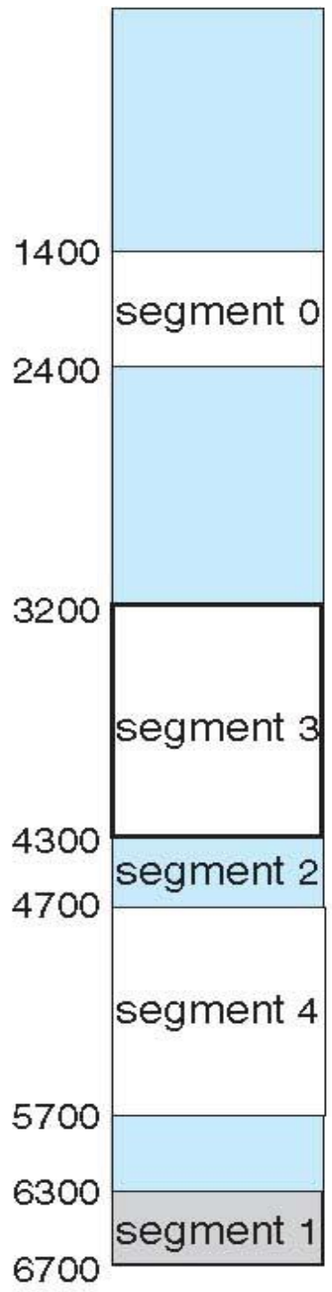
Example of Segmentation



logical address space

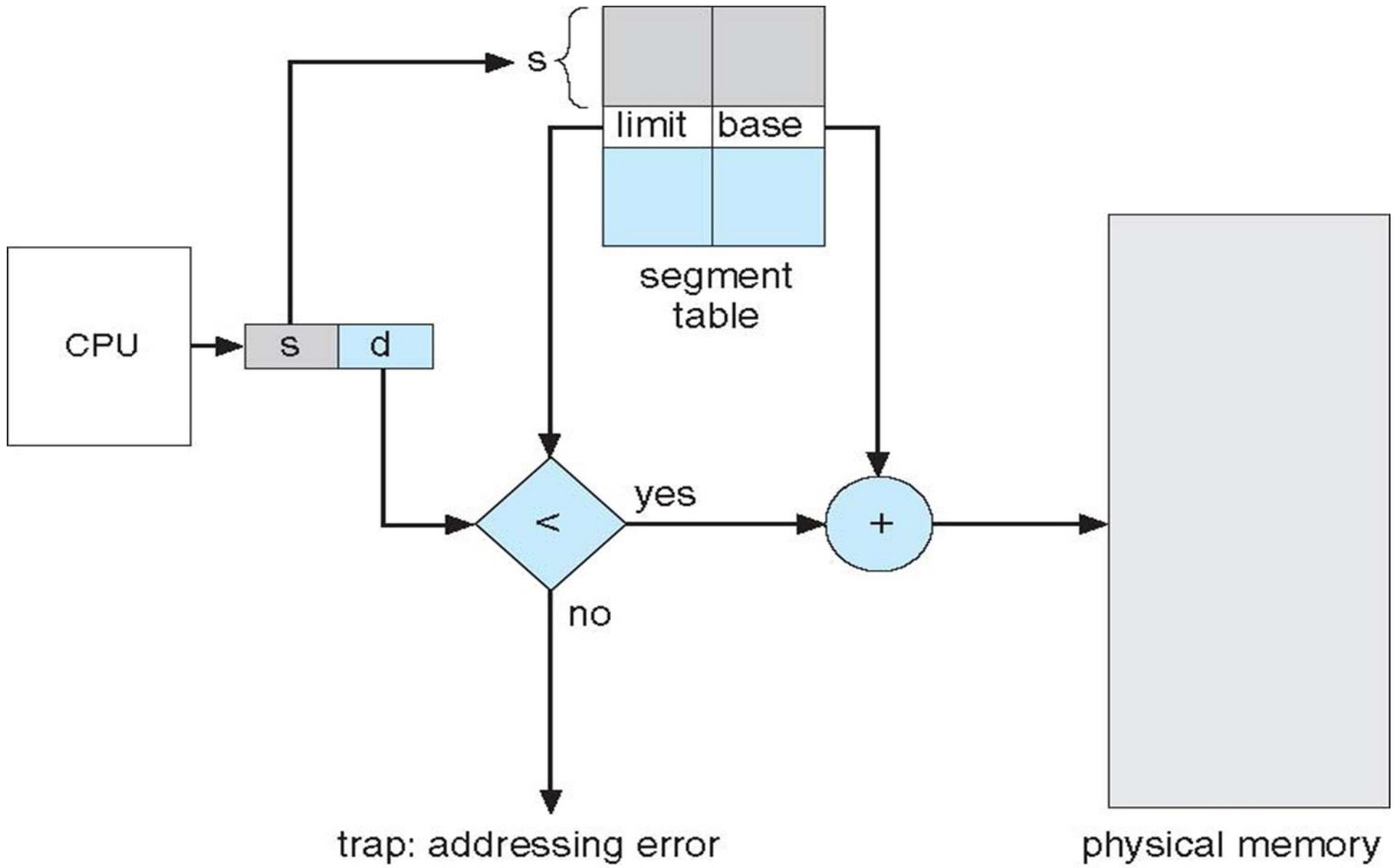
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



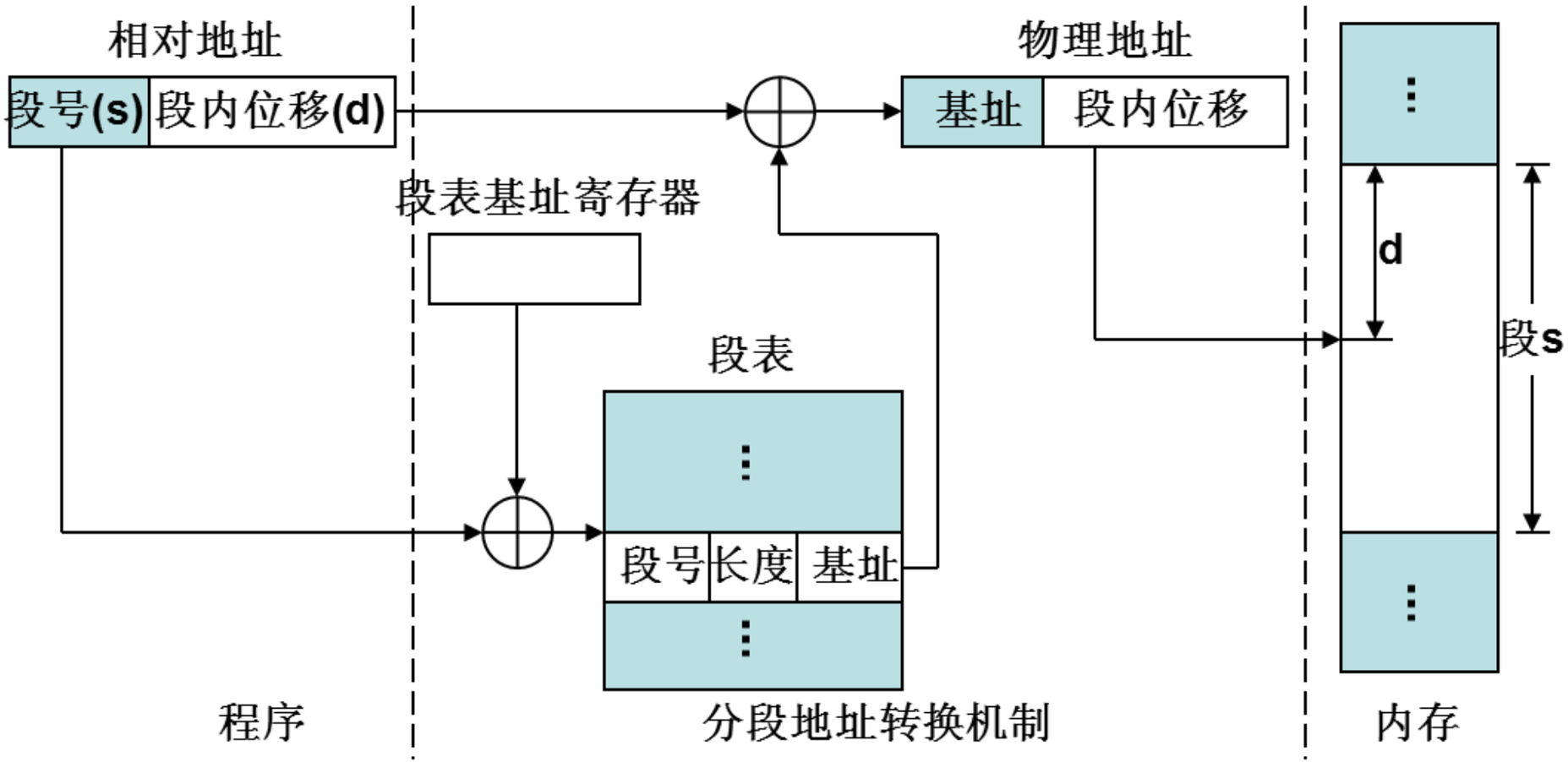
physical memory

地址变换


s
limit
base
segment table
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

4.5.2 分段系统的基本原理

- 地址变换过程



练习

- 例子：根据给出的段表，完成下列逻辑地址（段号+段内地址）到物理地址的转换
  - ① [0,430]
  - ② [3,400]
  - ③ [4,42]
  - ④ [2,500]

段号	段长	起始地址
0	600	219
1	14	2300
2	100	90
3	580	1327
4	96	1954

分段与分页的区别

- 页是信息的物理单位，段是信息的逻辑单位
  - 分页是系统管理的需要。
  - 分段是基于用户程序结构提出，每段都是相对完整的信息。
- 页的大小由系统确定，段的大小取决于用户程序。
- 页式地址空间是一维的，段式地址空间是二维的

信息共享

进程A的
逻辑地址空间

段0 文本
编辑程序

段1 程序段

段2 数据段

进程A的段表

段号	段长	基址
0	25286	43062
1		
2		

进程B的
逻辑地址空间

段0 文本
编辑程序

段1 程序段

段2 数据段

段3 堆栈段

进程B的段表

段号	段长	基址
0	25286	43062
1		
2		
3		

内存

操作系统

A的数据段

文本
编辑程序

B的数据段

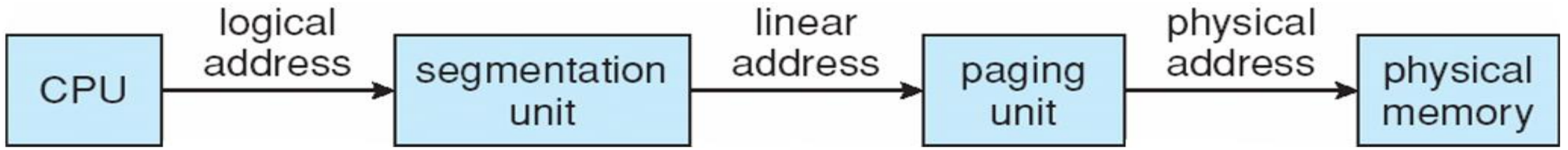
A的程序段

B的堆栈段

B的程序段

4.5.4 段页式存储管理

- 基本原理
  - 在段页式存储管理中，将用户的作业地址空间按分段来管理，系统在内部将组成该空间的每一个段按内存页帧的尺寸划分成固定大小的页。



4.5.4 段页式存储管理

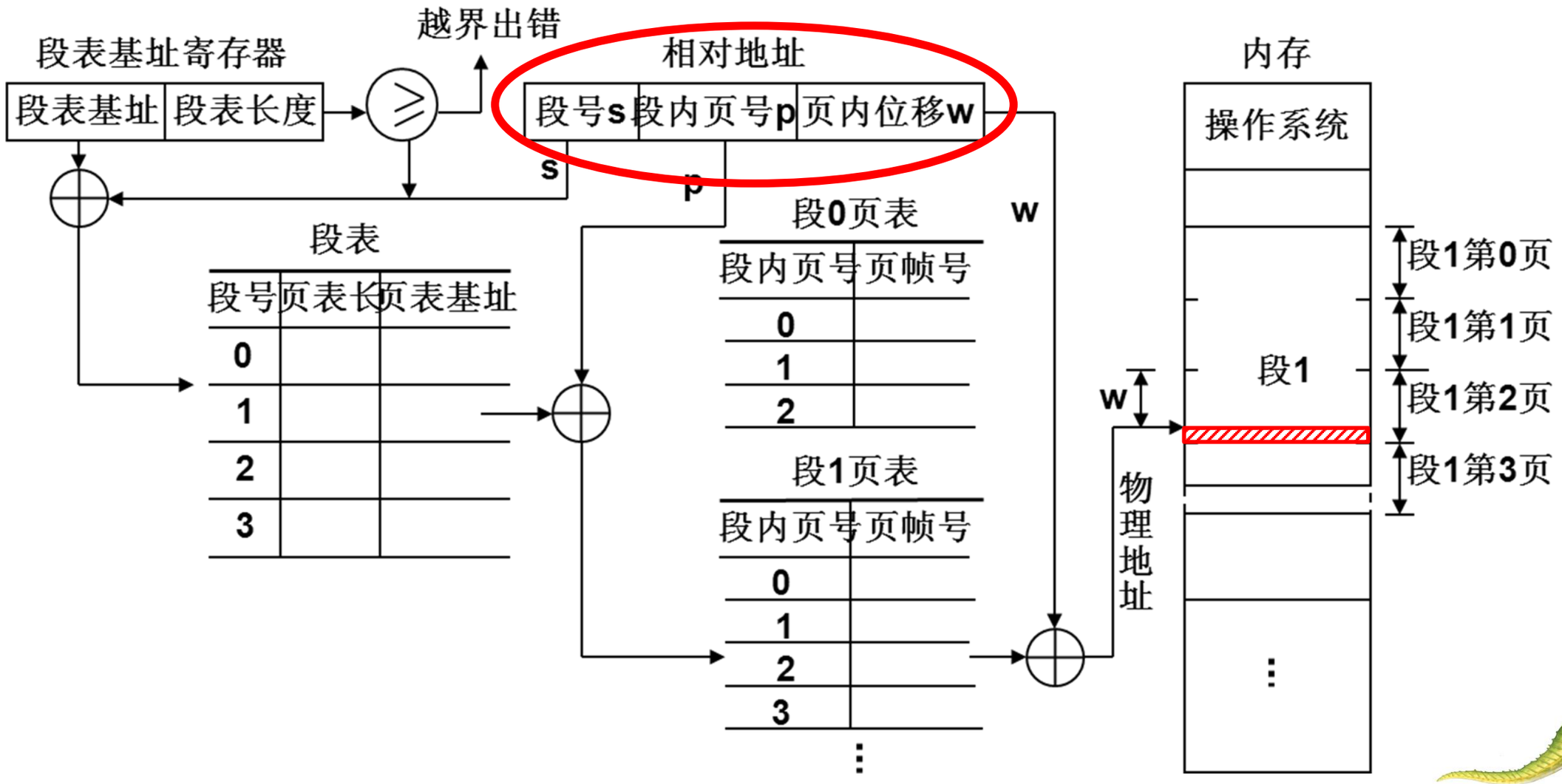
- 地址结构
  - 用户面对的是段号s和段内位移d，
  - 系统在内部又将段内位移d分解成段内页号p，页内位移w。

<table>
<tr><td rowspan="2">段号s</td><td colspan="2">段内位移d</td></tr>
<tr><td>段内页号p</td><td>页内位移w</td></tr>
</table>

  - 在这样的管理模式下，任何一个用户作业有一个段表，作业中的每一个段有一个页表。系统通过一个段表和若干个页表，实现对作业存储空间的管理和地址转换。

4.5.4 段页式存储管理

- 地址变换
  - 段表+页表

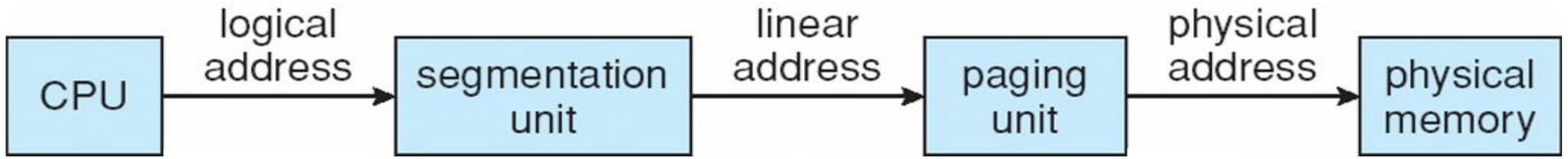


小结

- 优点
  - 综合了段式和页式两种管理方式
    - 克服了碎片，提高了存储器的利用率；
    - 有利于段的动态增长以及共享和内存保护等；
- 缺点
  - 复杂性和开销也增加；
  - 为了获得一条指令或数据，须三次访存，将会使执行速度大大下降。

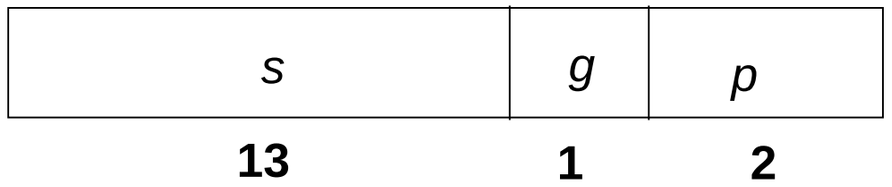
实例：Intel Pentium

- 支持
  - 单纯分段
  - 分页加分段（段页式）
- Pentium中的地址映射

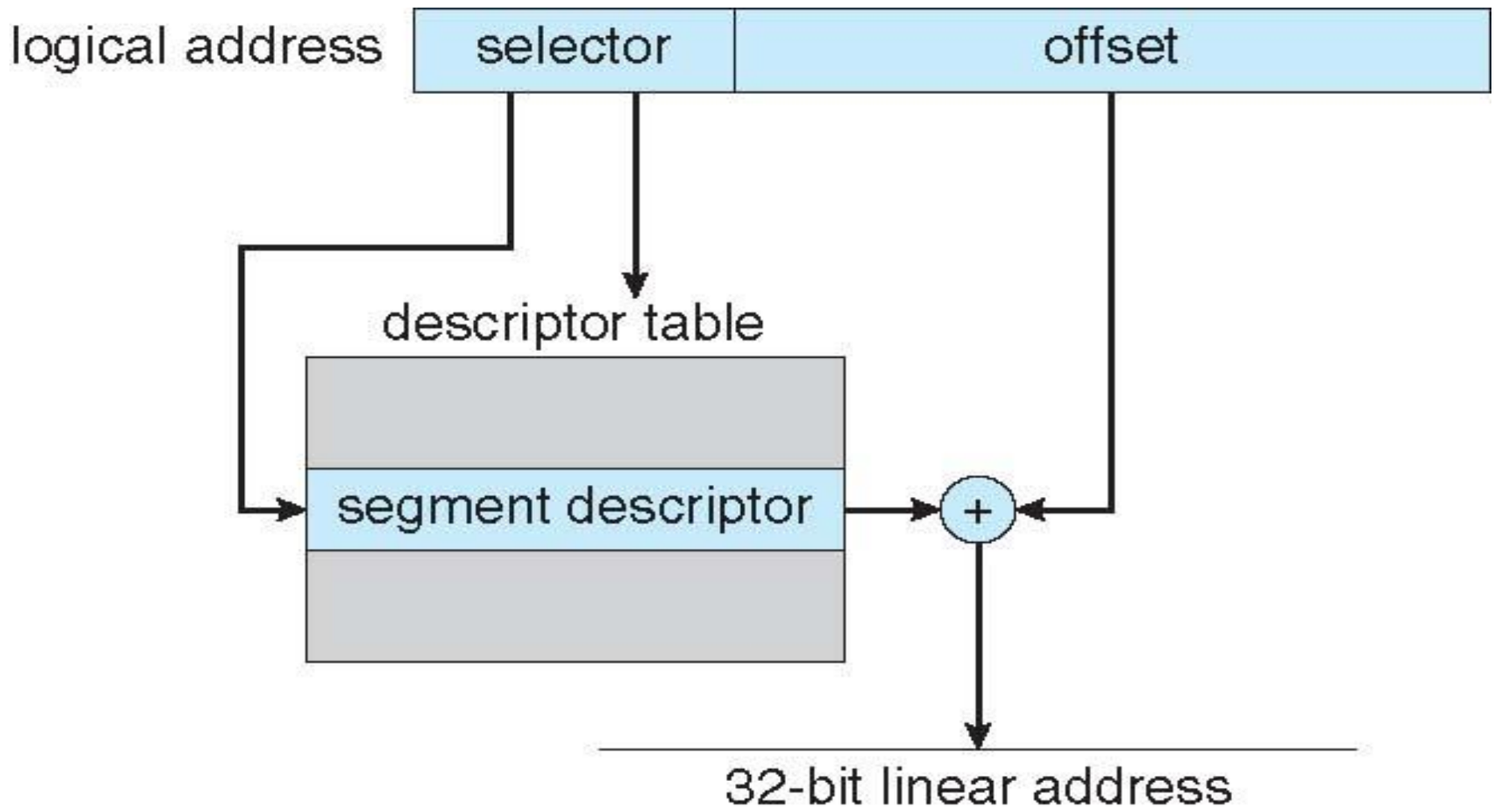


实例：Intel Pentium

- Pentium分段
  - 每个进程最多可以有 16K 个段，每个段最大可以有 4GB 。
  - 进程的私有段，最多8KB 个，本地描述符表（LDT, local descriptor table ）
  - 进程共享的段，最多 8KB 个，全局描述符表（GDT, global descriptor table）
- 逻辑地址:（selector,offset）
  - 选择器selector
    - s：段号
    - g：GDT/LDT
    - p：保护信息
  - 段偏移offset
    - 32位，4GB

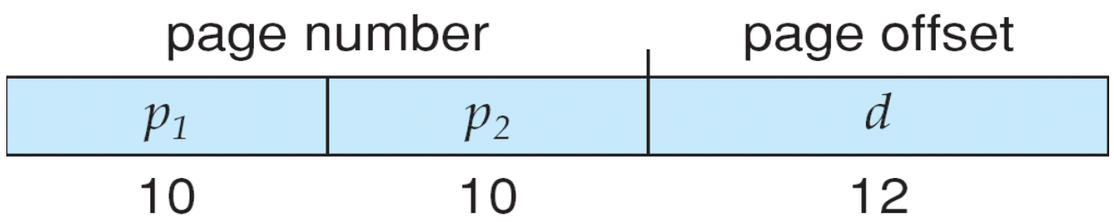


Intel Pentium Segmentation



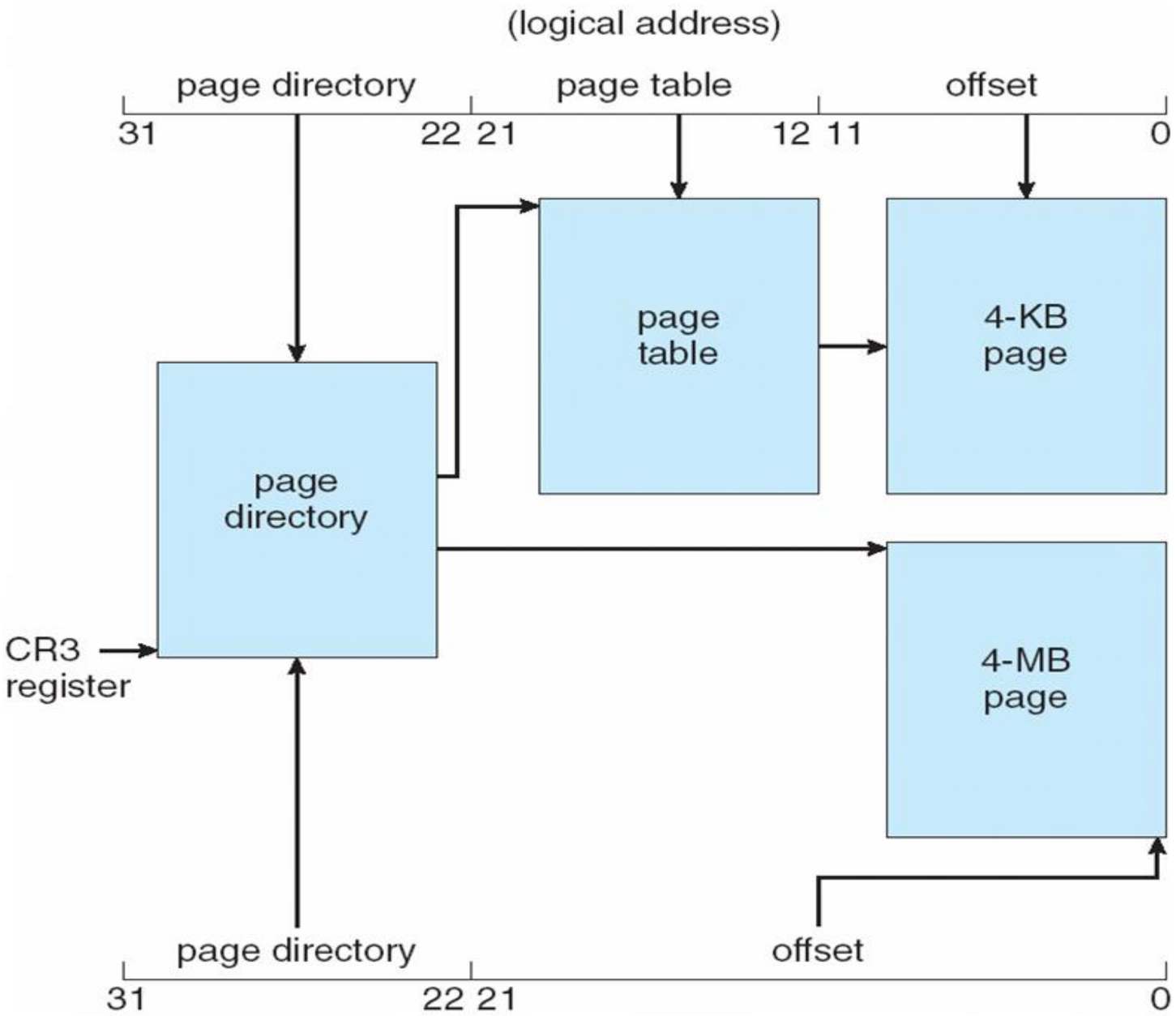
实例：Intel Pentium

- Pentium分页
  - 允许页的大小为4KB或4MB
    - 对于4KB的页，采用二级分页方案



    - 对于4MB的页，页目录直接指向4MB的页帧，绕过内层页表，最低位22位为页内偏移

Pentium Paging Architecture



小结

- 连续分配方式
  - 单道编程的内存管理
  - 多道编程的内存管理
    - 固定分区
    - 动态分区
- 离散分配
  - 分页式
  - 分段式
  - 段页式