

存储管理

阅读任务

- 第13节 “The Abstraction: Address Space”
- 第14节 “Interlude: Memory API” *char * p = malloc(100);
free(p), 为什么不能这样调用这
个参数?*
- 第16节 “segmentation” *分段*
- 第17节 “Free Space Management” *空间管理*

本章教学目标

- 理解存储管理的作用
 - 地址重定位
 - 存储保护
 - 存储共享
- 掌握存储管理的方法
 - 连续空间存储管理 *任意长度的连续空间*
 - 分页存储管理 *固定长度*
 - 分段存储管理 *逻辑?? 可变长*

大纲

- 存储管理的需求和作用
- 连续空闲存储管理
- 分页存储管理
- 分段存储管理

存储管理的作用

- 单道程序设计中，内存被分为两部分：^①操作
系统使用的内存，^②和用户运行程序使用
的内存
- 多道程序设计中，用户部分的内存需要进
一步划分以容纳更多的进程，在内存中保
持适量的进程，从而有效利用处理器资源。
- 操作系统负责内存的分配、回收和管理。

存储管理的需求

- 地址重定位
- 内存保护
- 内存共享
- 逻辑组织
- 物理组织

存储管理的需求

- 地址重定位

逻辑地址
(虚地址)

mapping

(物理地址)

物理地址

- 程序员无需知道程序在执行时在内存中所处的位置

引起



- 程序执行过程中，有可能被交换到磁盘，并在之后重新被交换回内存中不同的位置

- 代码中的内存访问需要被转换成物理内存地址

进程对编址的需求

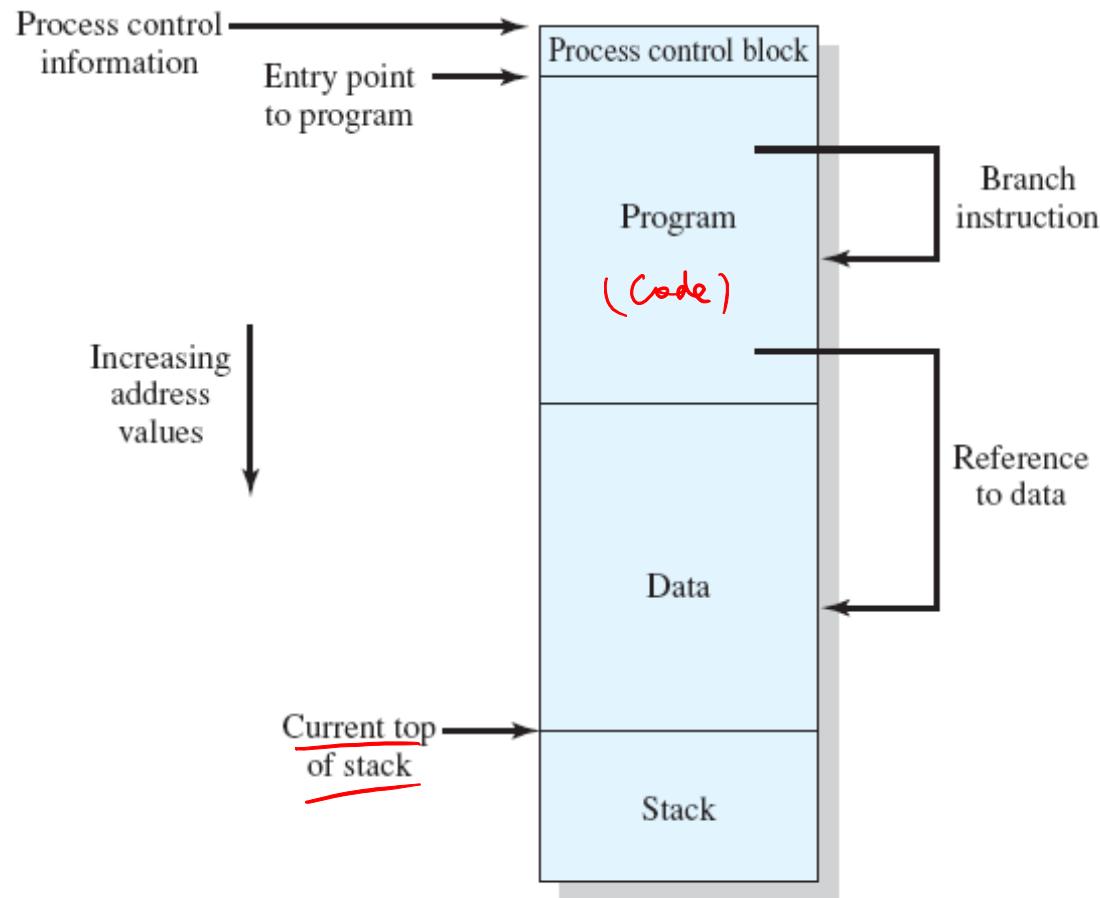


Figure 7.1 Addressing Requirements for a Process

存储管理需求

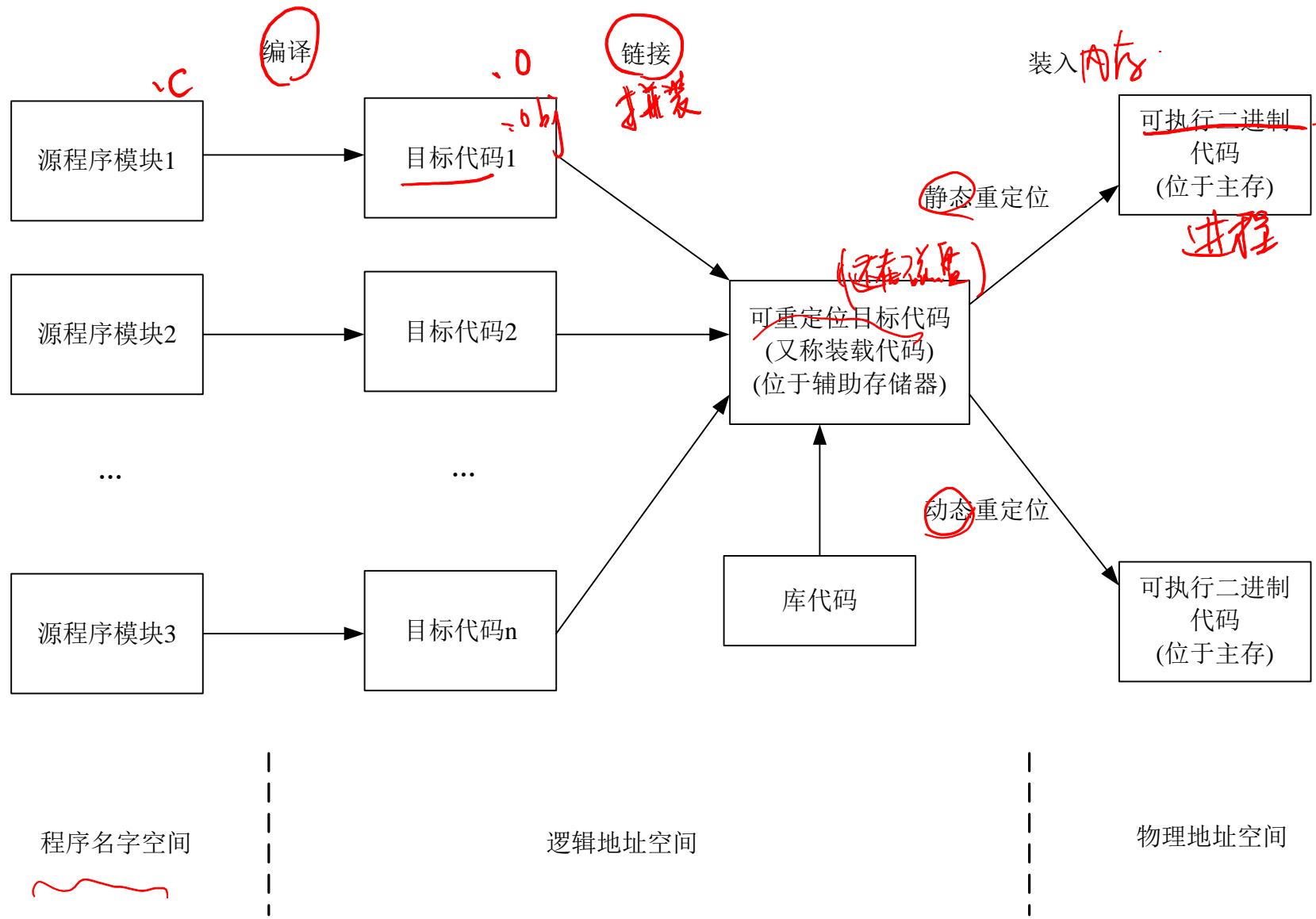
- 内存保护
 - 在未被允许的前提下，一个进程不能访问另一个进程的内存空间 X
 - 地址保护的检查无法在编译和链接时刻进行，
因为程序会被重定位到内存中不同的位置，并且
访问的地址会在运行时生成
 - 地址保护必须在程序执行时由硬件实现 必须 知识
 - 即便操作系统也无法预期一个程序将要访问的所有
内存地址
 - 当某个进程的指令越界访问其它进程的内存空间时，
CPU要能阻止该行为，通常是产生一个异常

存储管理的需求

- 共享
 - 允许多个进程访问同一块内存
 - 多个进程执行同一个程序, 让每个进程都拥有一份程序代码的拷贝将浪费内存空间
 - 协作进程之间共享数据
 - 内存管理系统应允许对内存区域的受控共享访问

地址转换与存储保护

- 用高级语言或汇编语言编写的程序称为源程序
- 源程序是不能被计算机直接运行的，需要经过编译、链接、装入三个阶段的处理才能装入主存运行



程序的编译、链接、装入和执行

编译

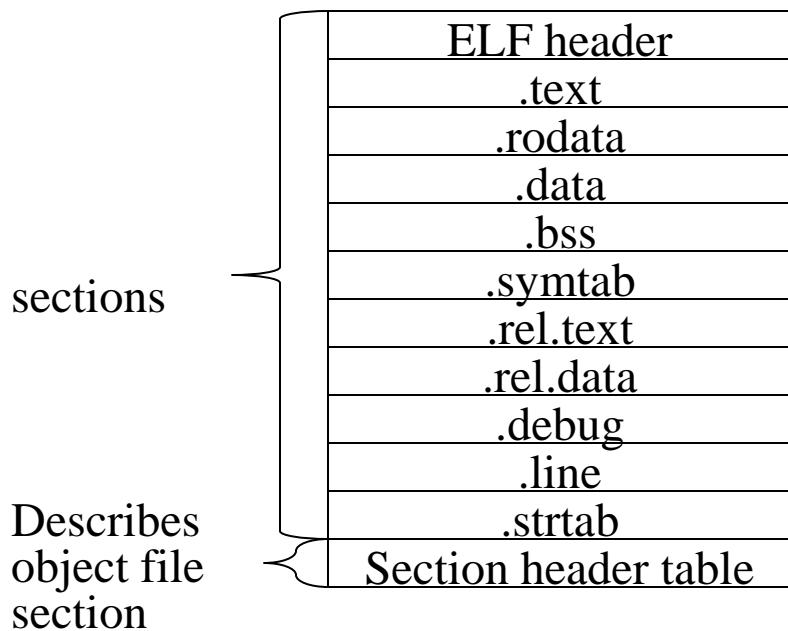
- 编译程序对源程序进行处理，得到**目标模块**
– 如gcc -c -o (编译、链接)
- 一个程序可以由独立编写且具有不同功能的多个源程序组成
- 编译程序负责记录引用的发生位置，将产生多个目标模块，每个模块都有一张符号表给出本模块定义和引用的符号信息。 EH?

目标文件

- 可重定位目标文件 .o
 - 包括二进制代码和数据
 - 可以与其它可重定位目标文件被编译合并成可执行目标文件
- 可执行目标文件 -exe
 - 包括二进制代码和数据
 - 可以直接被载入内存并执行
- 共享目标文件
 - 一种特殊的可重定位目标文件
 - 可以在装入时或运行时被动态链接

可重定位目标文件格式(ELF)

- 用于 Unix 和 Linux 系统



- .text: 指令代码
- .rodata: 只读数据, 如 printf 的格式化字符串, switch 语句的跳转表
- .data: 初始化的全局 C 变量
- .bss: 未初始化的全局 C 变量(在磁盘不占空间)
- .symtab: 符号表, 程序定义和引用的函数和全局变量信息
- .rel.text: 当该模块与其它模块链接时, .text 中需要被修改的位置列表
- .rel.data: 当该模块与其它模块链接时, .data 中需要被修改的位置列表, 如全局变量用其它全局变量或函数的地址来赋值的
- .debug: 调试符号表, 仅当采用 -g 选项时才有
- .line: C 源文件和 .text 区中指令行的映射关系, 仅当采用 -g 选项才有
- .strtab: .symtab 表和 .debug 表中用到的字符串表

链接

- 链接程序的作用是把多个目标模块链接成一个完整的可重定位程序

① 符号解析

- 每个目标模块都定义和引用符号
- 符号解析的目的是将每个符号引用与唯一的符号定义关联

② 重定位

- 每个目标模块都从0开始编址
- 重定位的目标是将其合并到一维的逻辑地址空间
 - 符号定义会被分配新地址
 - 所有对符号的引用需要被修改

链接

- 未链接前，每个模块中对其他模块的引用都必须以符号形式出现
- 链接程序将一组目标模块作为输入，输出一个可装载模块
 - 可装载模块是将所有输入连续地组装在一起
 - 每个模块之间的符号引用将被转换到整个装载模块的相对地址

引用 绑定 定义.

符号解析

- 全局符号分类
 - 强符号:
 - 函数名和初始化全局变量
 - 弱符号:
 - 未初始化全局变量
- Unix/Linux链接器对于全局符号的解析规则
 - 不允许多个同名的强符号
 - 一个强符号和多个弱符号, 则会选择强符号
 - 多个弱符号, 会任选^①

符号解析

```
/*foo2.c*/  
int x =123; 缺  
int main()  
{  
    return 0;  
}
```

```
/*bar2.c*/  
int x =124; 缺  
void f()  
{  
}
```

可执行文件名
gcc -o p foo2.c bar2.c
该命令能否成功? *×*

*10
..>*

符号解析

— 强
~~ 弱

```
/*foo3.c*/  
#include <stdio.h>  
void f(void);  
int x =123;  
int main()  
{  
    f();  
    printf("x= %d\n", x);  
    return 0;  
}
```

```
/*bar3.c*/  
int x ;  
void f()  
{  
    x = 128;  
}
```

引用，绑定到强符号

gcc -o p foo3.c bar3.c
该命令能否成功? ✓
程序的运行结果? ~~123~~ 128

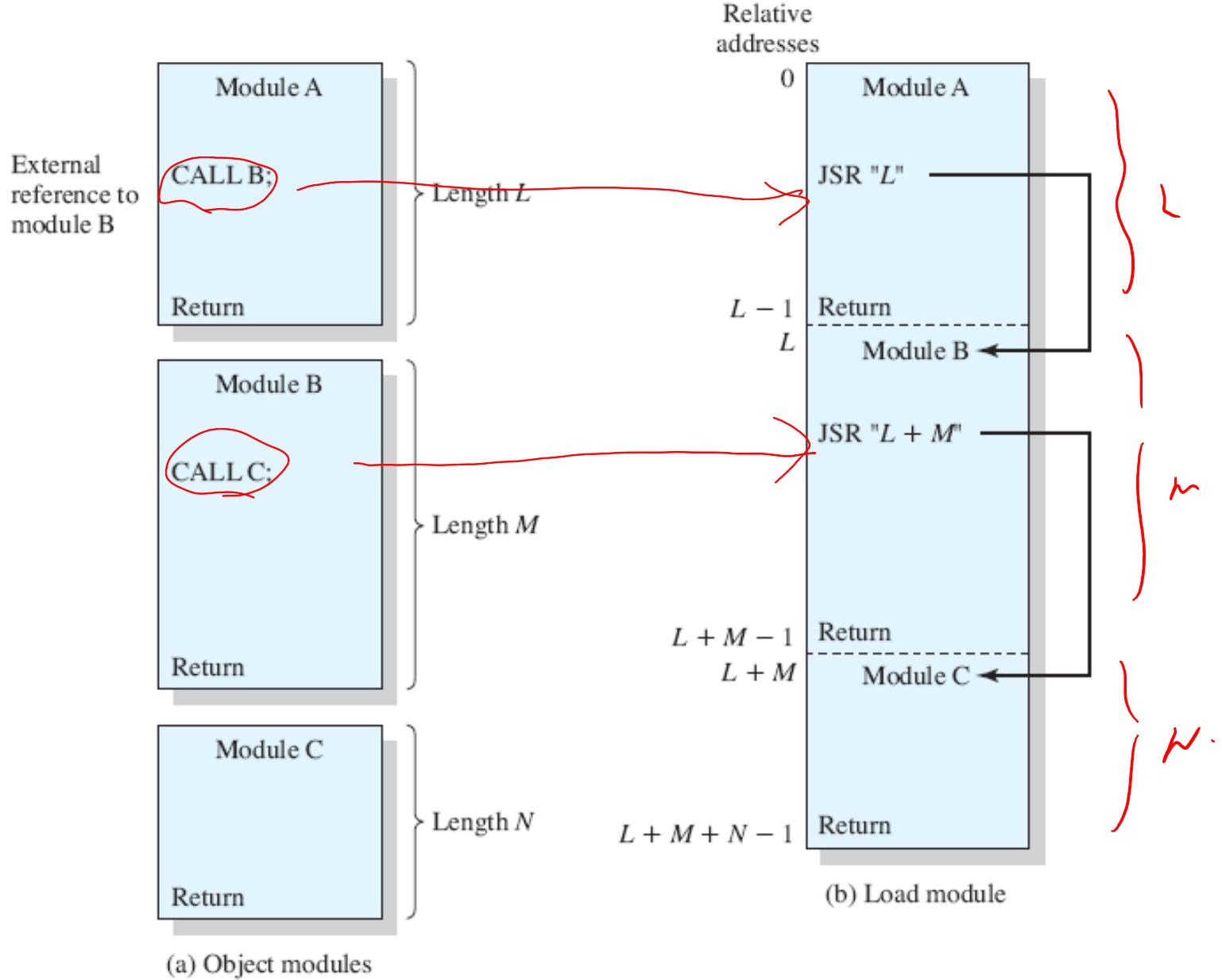


Figure 7.18 The Linking Function

链接

符号 → 相对地址

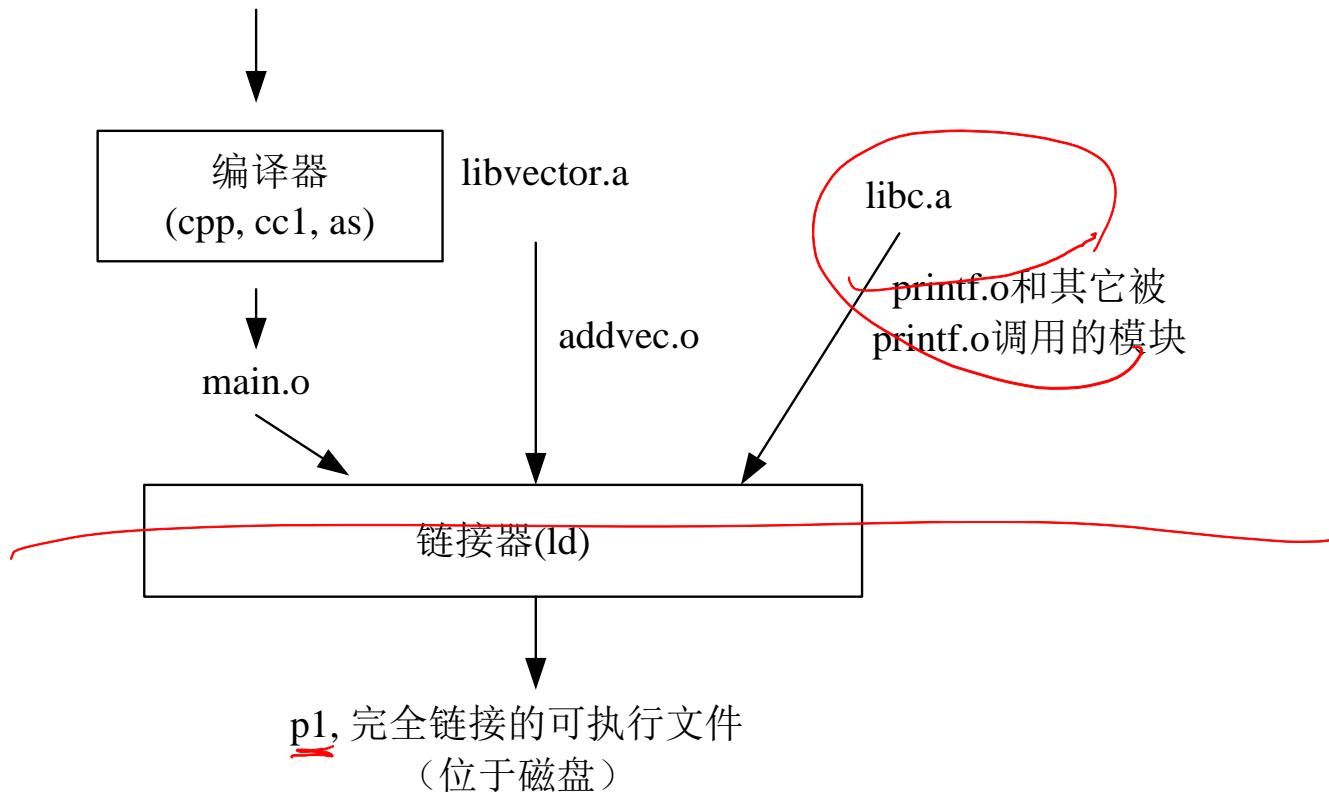
- 静态链接
 - 在链接时对所有外部引用进行转换，即给装入器的所有引用均已转换为逻辑地址空间中的相对地址
- 动态链接
 - 链接时对某些外部模块的引用不进行解析，
延迟到装入或运行时

动态链接

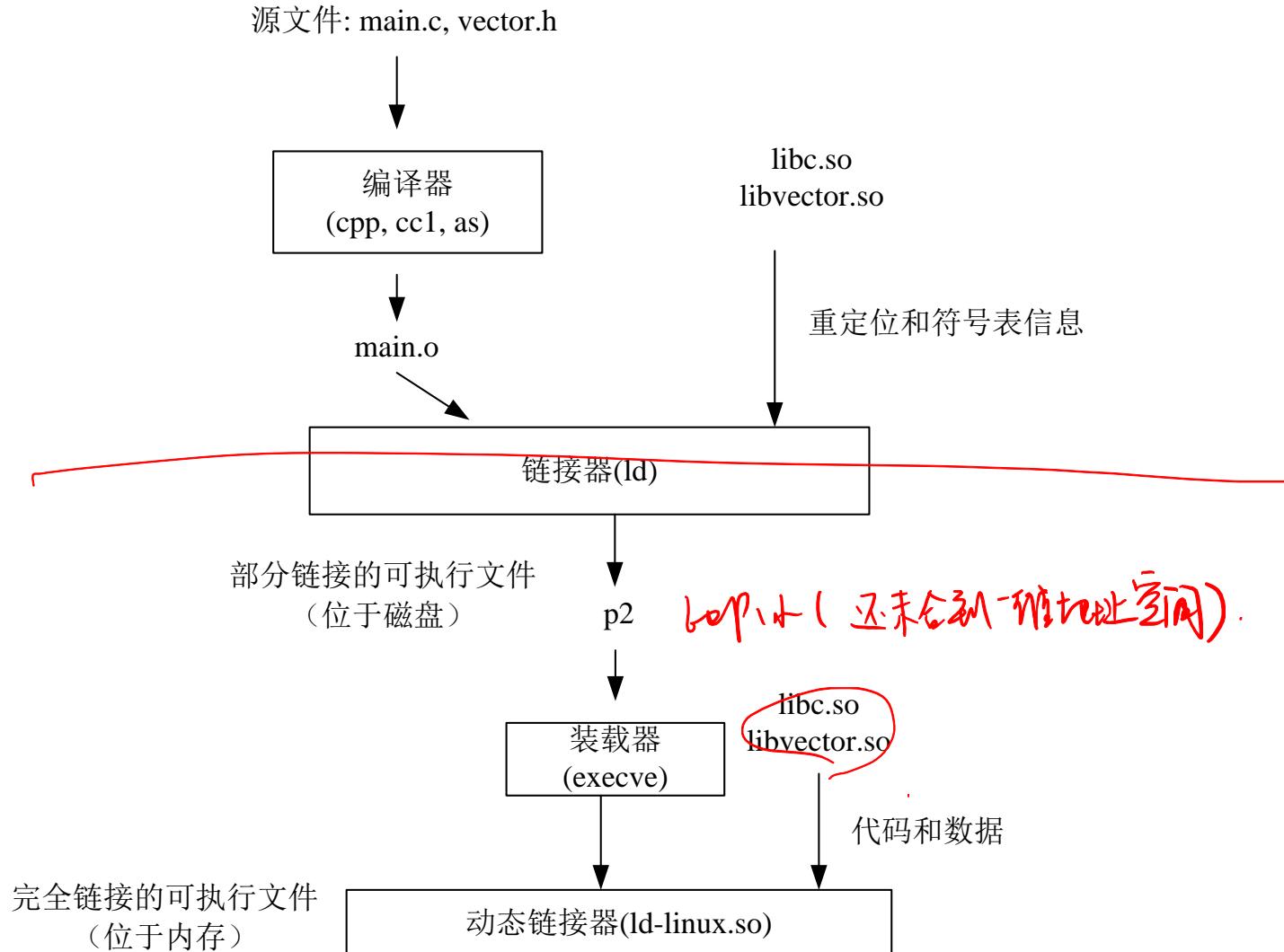
- 装入时动态链接
 - 装入时如遇到对外部模块的引用，则装入器找到并装入该模块，并将对模块的引用修改为该模块相对于整个应用模块起始地址的相对地址
 - 更容易应对目标模块的改变
 - 若是静态链接，则目标模块的每次变化都要重新链接整个程序模块
- 运行时动态链接
 - 对某些目标模块的外部引用在装入时并不进行转换，而是保持在装入到内存的程序中
 - 当真正运行时执行到该模块时，操作系统定位并装入模块，并与调用模块进行链接。
 - 这类模块通常是共享的，如Windows中的DLL

静态编译和链接过程

源文件: main.c, vector.h



动态链接过程



装入

- 在加载一个装载代码之前，存储管理程序将分配① 一块物理内存给进程
- 装入程序根据分配的内存地址，再次修改和调整被装载模块中的逻辑地址，② 将逻辑地址绑定到物理地址
- 装入方式分类（依据绑定时刻）：
 - 基于绝对地址的装入（编译时绑定）~~已通用~~
 - 基于地址重定位的装入
 - 静态重定位--装入时绑定 ✗ 静态址？(只读公用)
 - 动态重定位--运行时绑定 ✓ 可装入不同的物理地址

基于绝对地址装入

- 编译和链接程序生成绝对地址
 - 程序只能装入到固定的物理内存位置

基于地址重定位装入

- 编译和链接程序生成**逻辑地址空间**
- 重定位方式:
 - 静态重定位
 - 装入过程进行逻辑地址和绝对地址的转换
 - **进程在执行过程中无法移动位置**
 - ~~★~~ 运行时动态重定位
 - 支持进程在内存和对换区进行交换，**并在对换进来时装入到不同的物理地址**
 - 将绝对物理地址的计算延迟到运行时
 - 装载器装入时按相对地址进行
 - **采用硬件机制进行绝对地址的计算**
 - 重定位寄存器
 - 地址转换机构

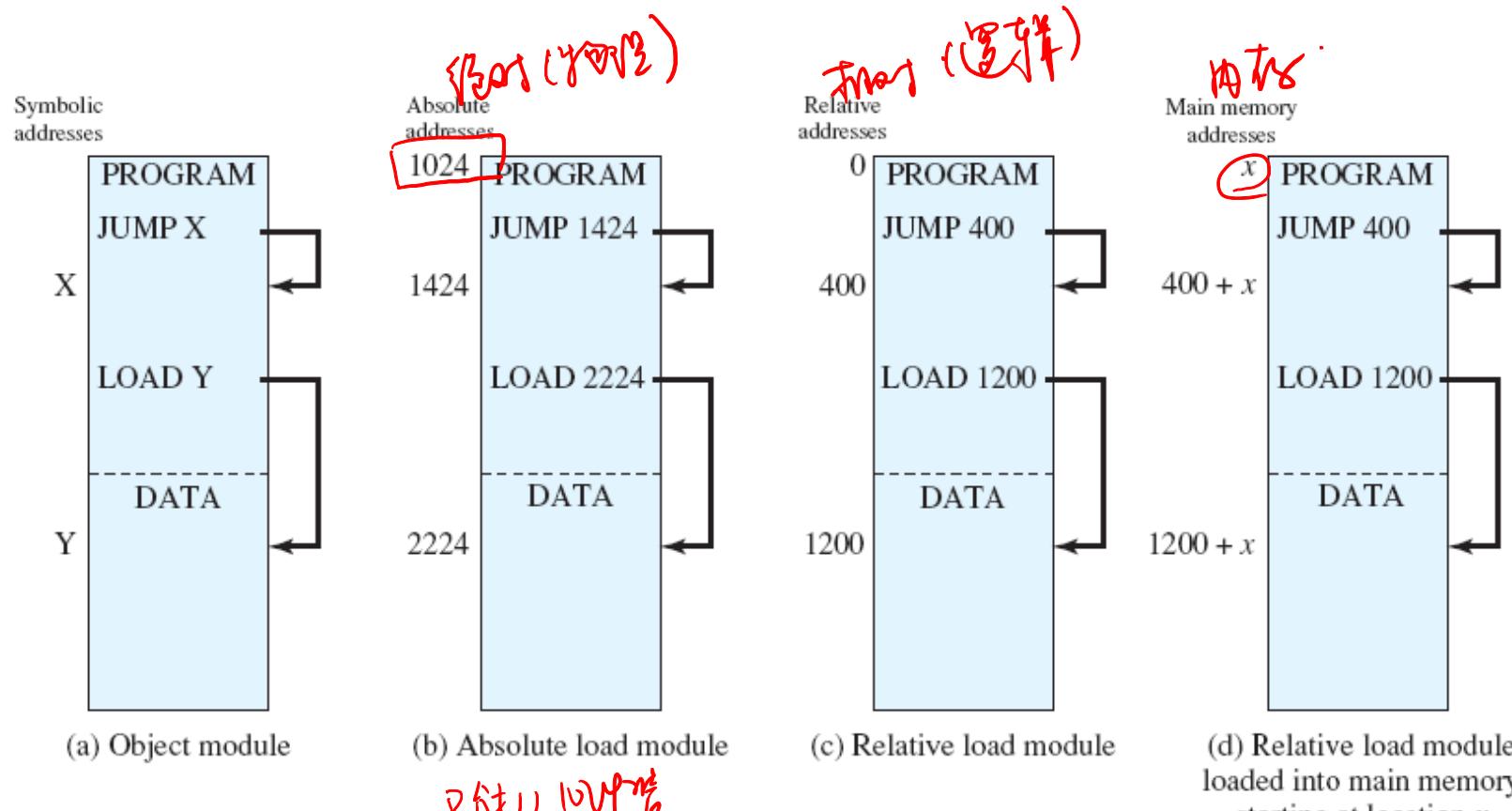
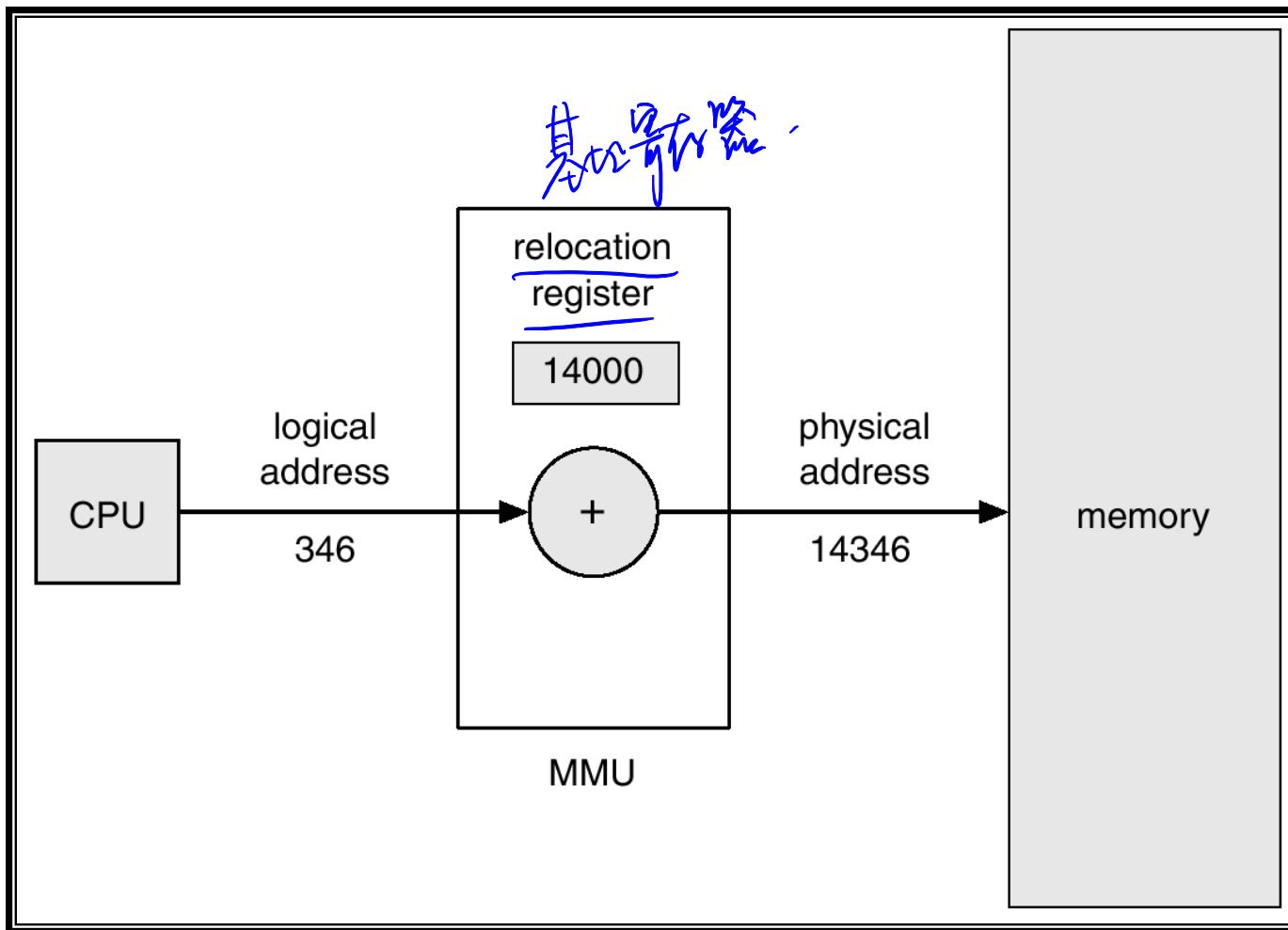


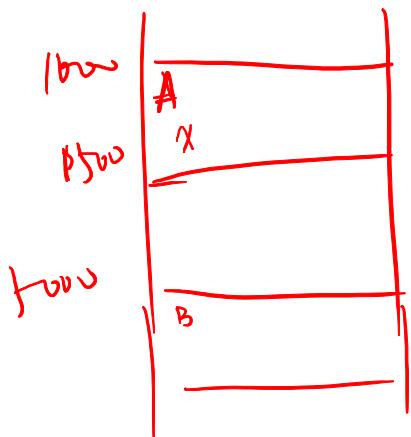
Figure 7.17 Absolute and Relocatable Load Modules



动态地址重定位示意

动态地址重定位(more)

- 重定位寄存器的内容通常保护在进程控制块中
- 当进行进程上下文切换时，当前运行进程的重定位寄存器的内容将被~~old~~^{old}保存，新进程的重定位寄存器的内容会被~~new~~^{new}恢复 ~~进上文部~~



$$\begin{aligned}B_p &= 1000 & x \text{ offset} &= 500 \\B_p &= 1500\end{aligned}$$

大纲

- 存储管理的需求和作用
- 连续空间存储管理



存储空间管理方法

- 连续存储空间管理
 - 固定分区存储管理
 - 可变分区存储管理
 - Buddy算法
- 分页存储管理
- 分段存储管理
- 虚拟存储管理

固定分区存储管理

- 固定分区划分方法
 - 一种方法是将主存分成大小相等的分区
 - 任何小于分区大小的进程可以被载入一个可用的分区
 - 若所有分区都满了，则操作系统可以将某个进程对换出内存
 - 另一种方法是将主存分成数目固定不变的，但大小不同的分区。
如何设置？ $2^n \cdot (1, 2, 4, 8, \dots)$

固定分区管理的例子

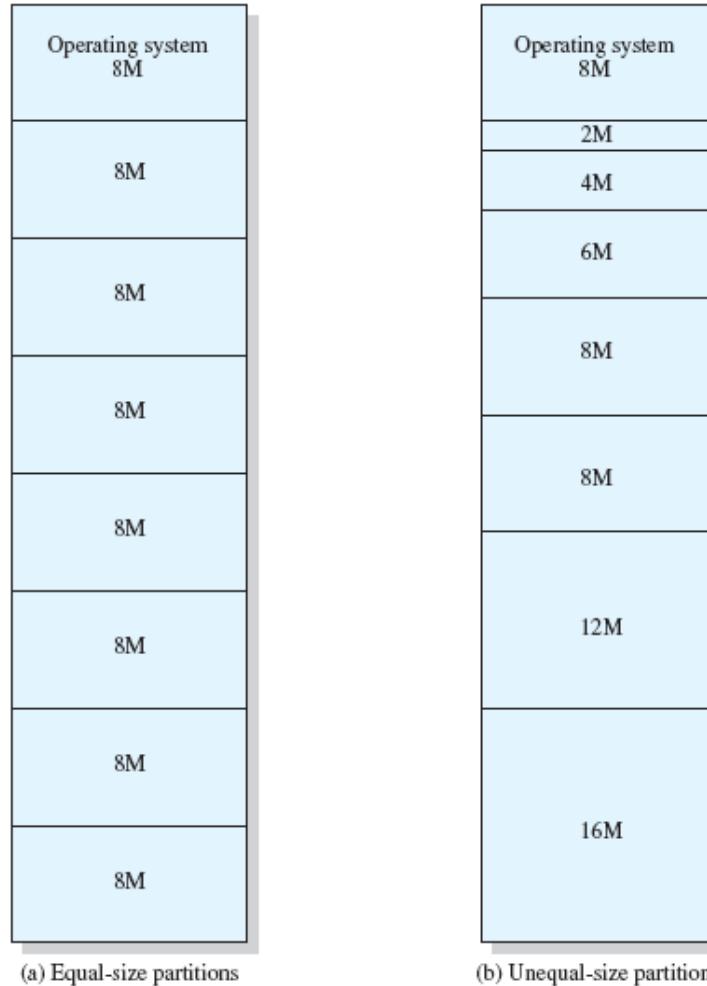


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

分区分配算法

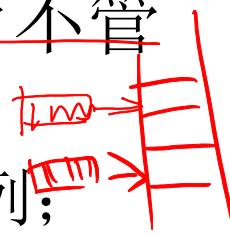
- 分区大小相同

~~分区大小~~
无所谓

- 分区大小不同

~~适配~~ 将适合进程的最小分区分配给该进程，而不管该分区是否被占用 ~~抢占!!~~

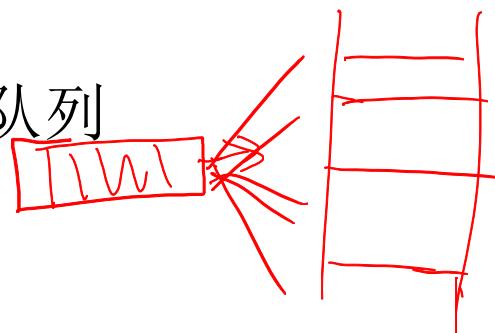
min

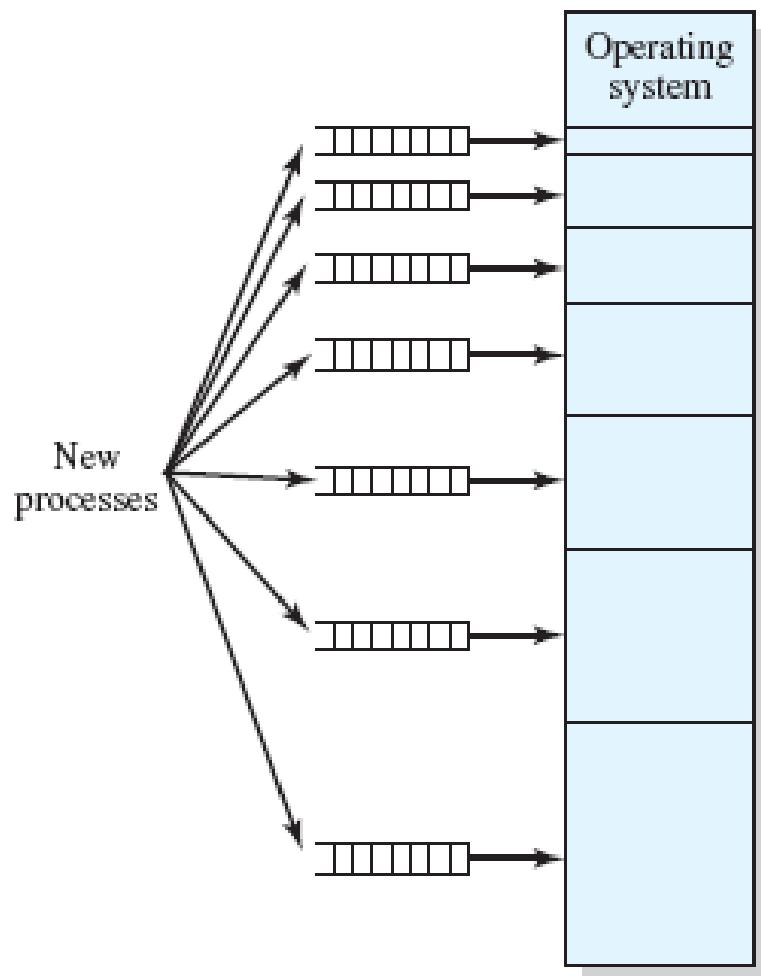


- 通常对应于每个分区都有一个单独的排队队列；

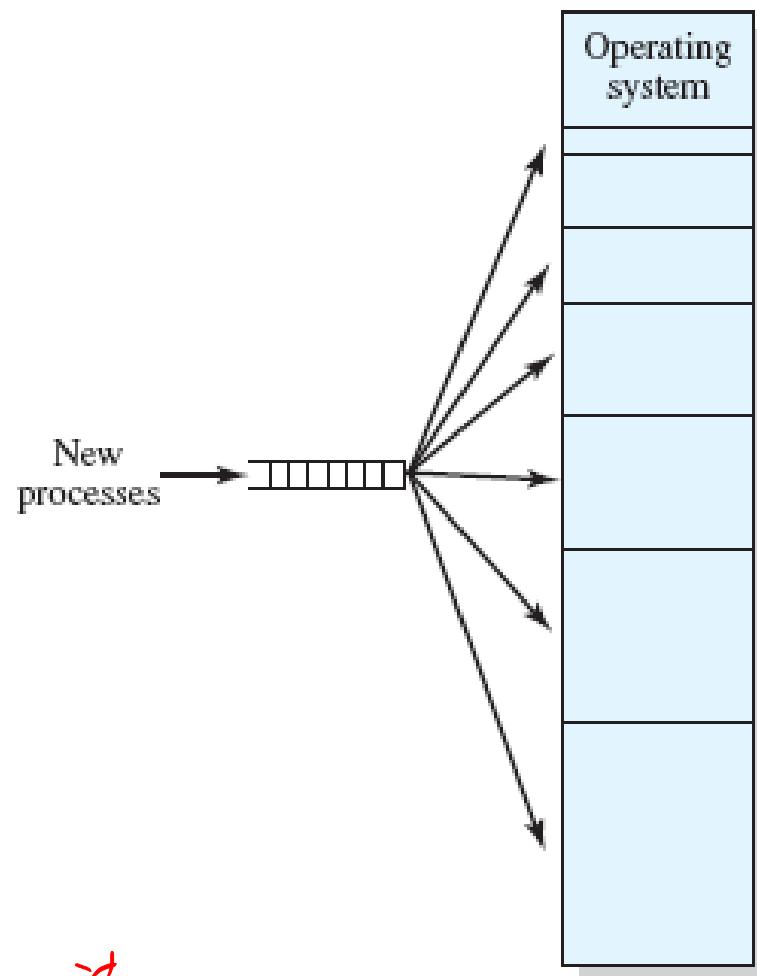
~~适配~~ 将当前未被占用的适合进程最小分区分配给进程

- 对应于所有作业排成一个等待队列





(a) One process queue per partition
-
zh-



(b) Single queue
-
zh=

Figure 7.3 Memory Assignment for Fixed Partitioning

固定分区管理—主存分配表

分区号	起始地址	长度	占用标志
1	8KB	8KB	0
2	16KB	16KB	Job1
3	32KB	16KB	0
4	48KB	16KB	0
5	64KB	32KB	Job2
6	96KB	32KB	0

固定分区管理的优缺点

- 优点
 - 实现简单
- 缺点
 - 大作业可能无法装入
 - 主存利用率低
 - 一个程序无论多小，都将占用整个分区，大于程序大小的部分称为内部碎片。(被浪费) 低利用率
 - 运行过程中扩充主存困难
 - 限制了多道程序的道数

可变分区内存管理

- 按照进程的需求为其分配恰好^{按需分布}的内存空间
 - 初始时，整个用户区是一个大空闲分区 ~~并不划分~~
 - 当装入作业时，根据作业需求查看是否存在足够的连续空间
 - 若存在，则按作业大小分割一块连续存储空间
 - 若不存在，则令作业等待主存资源
- 优点：
 - 按需分配，有利于多道程序设计，提高主存资源利用率
- 缺点：
 - 随着内存的不断分配、回收，最终主存中会出现许多分散的小空闲区，这通常被称为外部碎片。~~和进程使用~~
 - 需要对内存进行压缩，对进程进行移动，从而将小空闲区合并成大的空闲区。

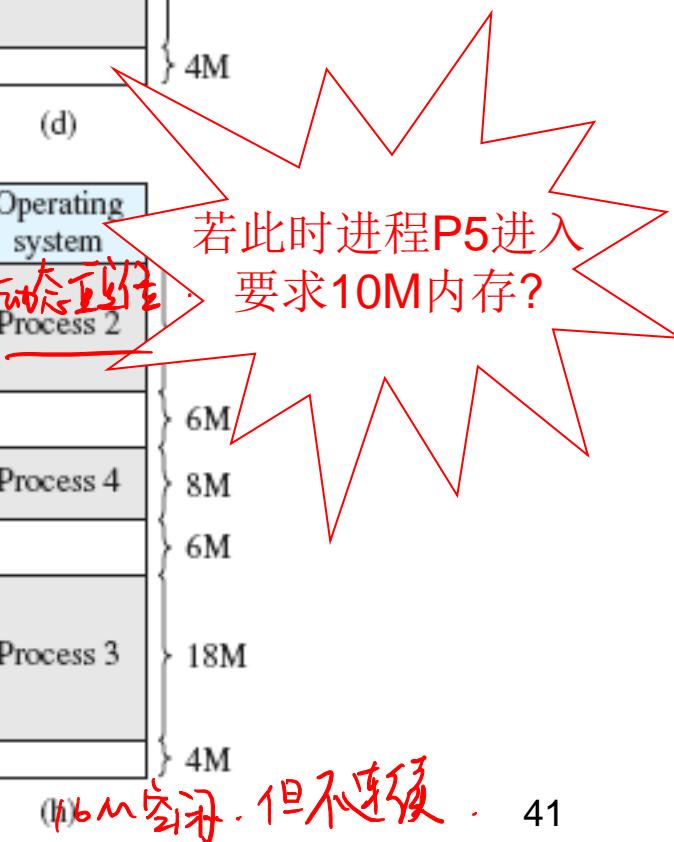
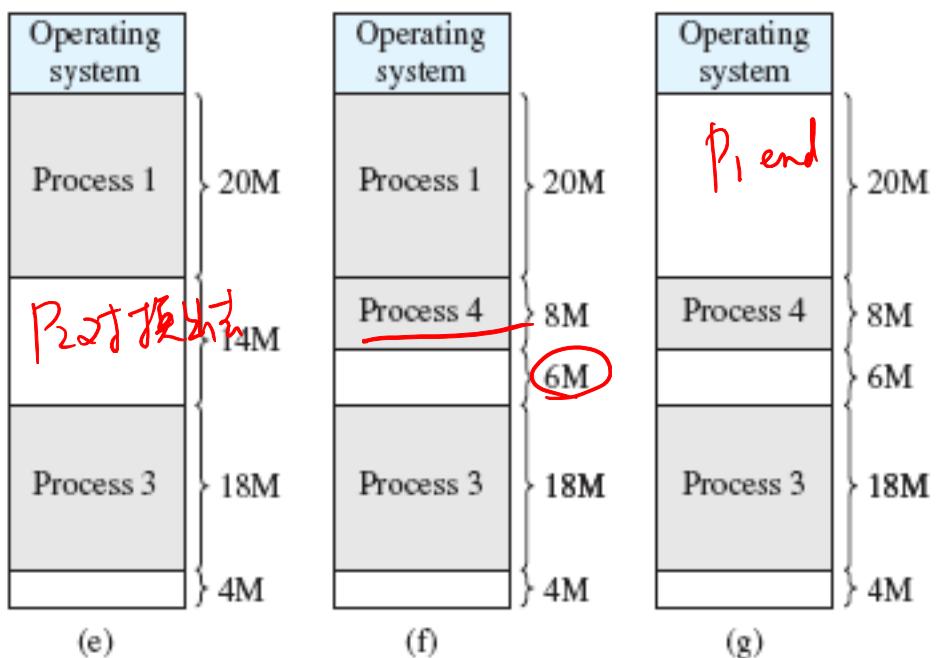
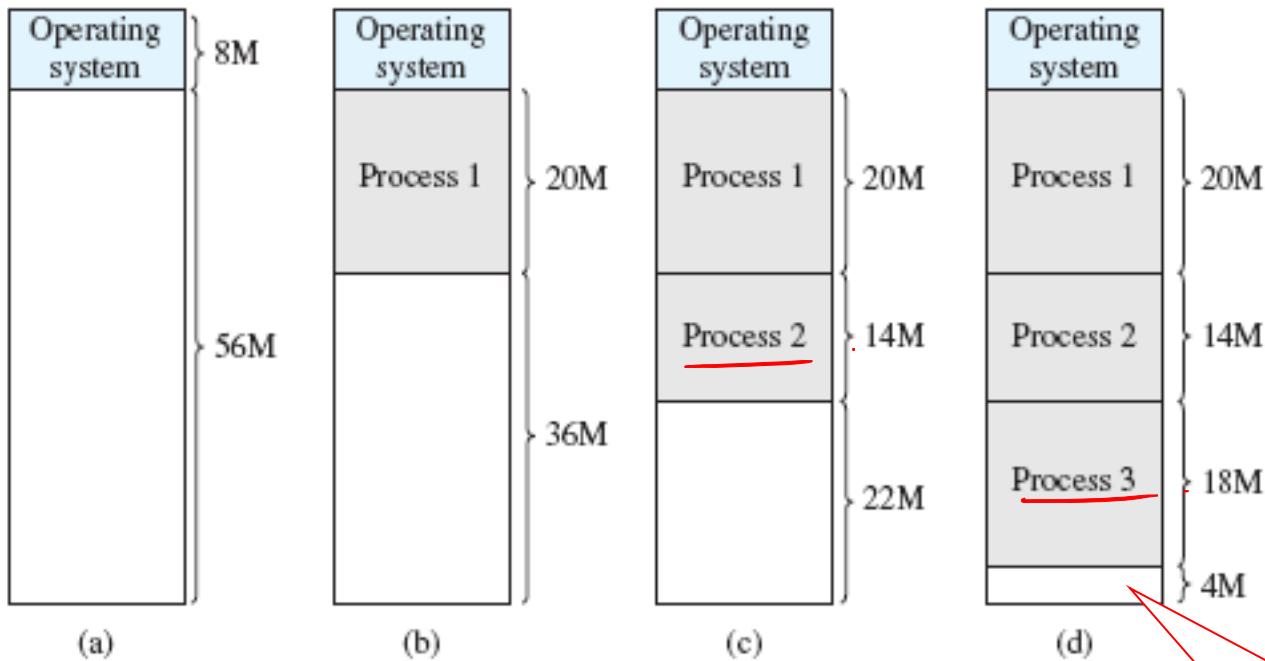


Figure 7.4 The Effect of Dynamic Partitioning

可变分区管理的数据结构

- 主存中分区的数目和大小随着进程的执行而不断改变
- 两类管理方法
 - ① 空闲区表格管理法
 - ② 空闲区链表管理法

①

空闲区表格管理法

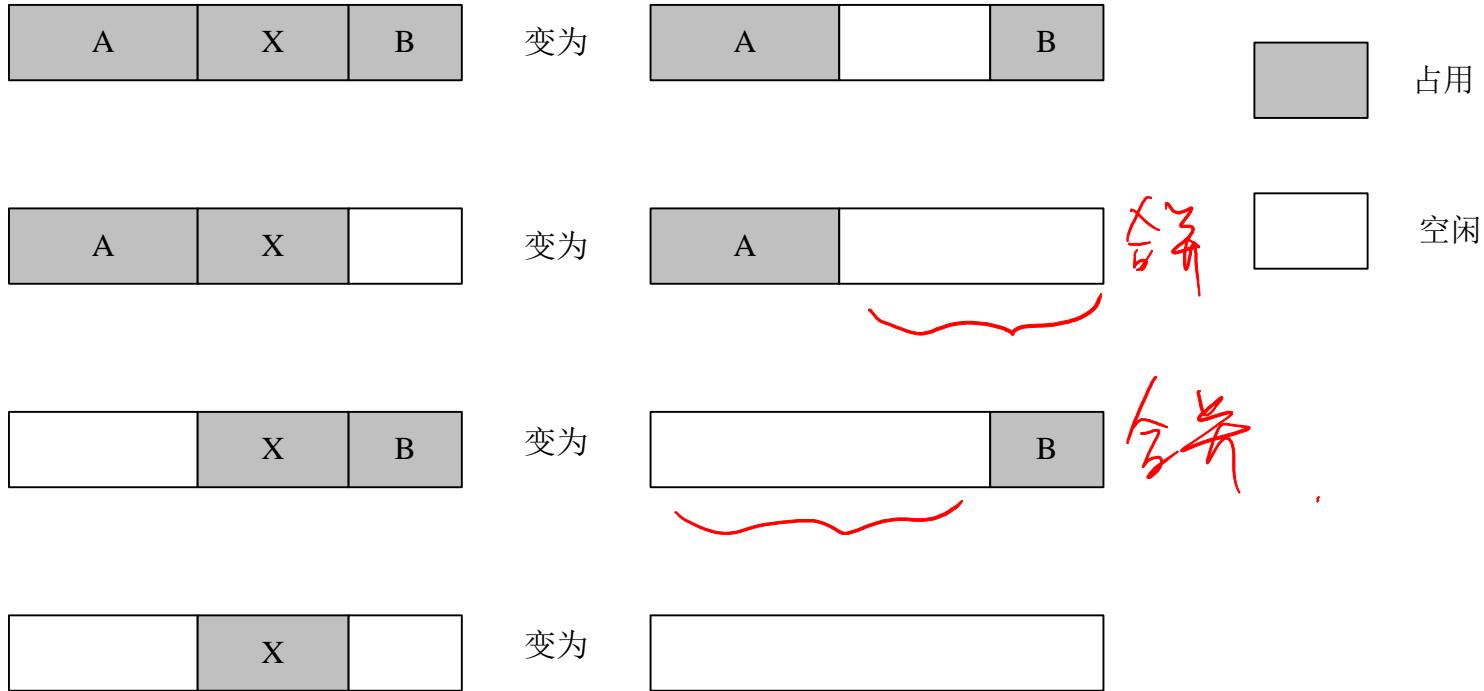
- 两张表格
 - 空闲区表（未分配区表）
 - 占用区表（已分配区表）
- 内存的分配与回收
 - 分配:
 - 从空闲区表中找出一个足够容纳进程的空闲区
 - 将其分为两部分，一部分装入进程，成为占用区，另一部分（若有）仍为空闲区
之后地址长过
 - 回收:
 - 已占用区表中的相应状态为空
 - 将收回的分区登记到空闲区表中
 - 若有相邻的空闲区，则需要将其合并后登记

空闲区表

分区号	起始地址	长度	占用标志
1	A	L1	0
2	C	L3	0

占用区表

分区号	起始地址	长度	占用标志
1	B	L2	Job1



作业X结束时，可变分区回收情况

②

空闲区链表管理法

(linked list structure)

- 链指针将所有空闲分区链接起来
- 每个主存空闲区的开头单元存放本空闲区长度及下一个空闲区起始地址指针
- 系统设置指向空闲区链的头指针
- 分配与回收
 - 分配
 - 沿链表查找并取得一个长度能满足要求的空闲区给进程，修改链表；
 - 回收
 - 将此空闲区链入空闲区链表的相应位置，有可能要合并相邻的空闲区

可变分区内存管理的分配算法

- 为什么需要分配算法?
 - 内存压缩代价很高, 因此操作系统应该采用合适的分配算法, 降低外部碎片的产生频率。
- 常用分配算法
 - 最先适应分配算法(first fit)
 - 下次适应分配算法(next fit)
 - 最优适应分配算法(best fit)
 - 最坏适应分配算法(worst fit)
 - 快速适应分配算法(quick fit)

最先适应分配

第一个满足要求的空闲区

- 从低地址开始顺序查找未分配区表或链表，直至找到第一个能满足要求的空闲区
- 分割此分区，一部分分配给进程，另一部分仍为空闲区（若有）。
- 高地址部分尽可能少用，有利于大作业的装入。
- 低地址和高地址两端的分区利用不均衡

下次适应分配算法

循环

- 从未分配区的上次扫描结束处顺序查找未分配区表或链表，直至找到第一个能满足长度要求的空闲区为止
- 分割该未分配区，一部分分配给作业，另一部分仍未空闲区（若有）
(从头开始搜索)
- 最先适应分配算法的变种，能缩短平均查找时间，存储空间利用率更均衡。

最优适应分配算法

~~遍历 (按地址排)~~

- 扫描整个未分配区表或链表，从空闲区挑选一个能满足用户进程要求的最小分区进行分配。
- 保证不会分割更大的分区，使大作业装入容易得到满足
- 易于产生外部碎片
- 可将空闲区按长度递增顺序排列

~~(按长度排)~~

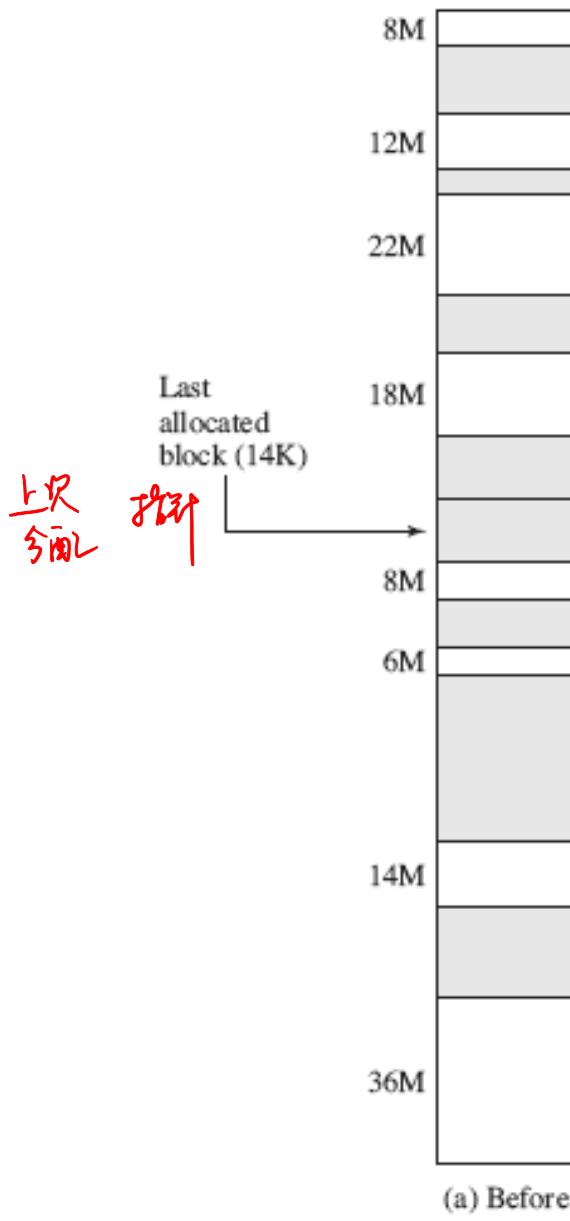
~~归还后按长度~~

最坏适应分配算法

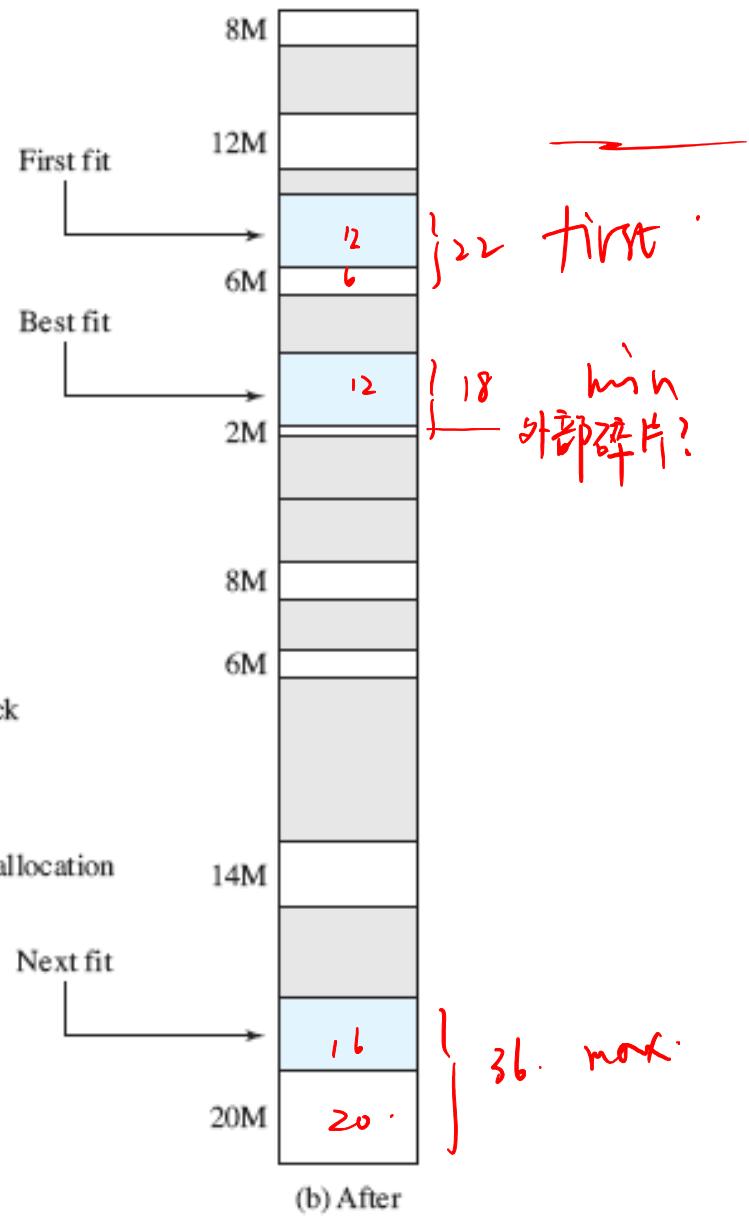
- 扫描整个未分配区表或链表，挑选一个最大的空闲区分割给作业使用
- 剩下的空闲区不致于过小，对中小型作业有利。 避免产生外部碎片。
- 可将空闲区按长度递减排列

快速适应分配算法

- 为经常用到的长度的空闲区设立单独的空闲区链表。
 $< 2M$ 
 $2M \sim 4M$ 
 $4M \sim 8M$ 
- 优点
 - 查找十分快速
- 缺点
 - 归还主存空间时与相邻空闲区的合并较复杂



(a) Before



(b) After

Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

连续内存分配的地址转换和存储保护

- 基址寄存器(base register/relocation register)
$$\text{BR} + \text{offset}$$

(逻辑地址)

 - 存放分配给进程使用的分区的起始地址
- 限长寄存器(limit register)
 - 存放进程所占用连续存储空间的长度
- 界限寄存器(bounds register)
 - 存放进程所占用的连续存储空间的最大地址
 - 界限寄存器值 = 基址寄存器值 + 限长寄存器值 - 1
- 限长寄存器和界限寄存器择一就行。

三
基址寄存器

连续内存分配的地址转换和存储保护

地址转换:

物理地址=基址寄存器值+逻辑地址值

(Offset)

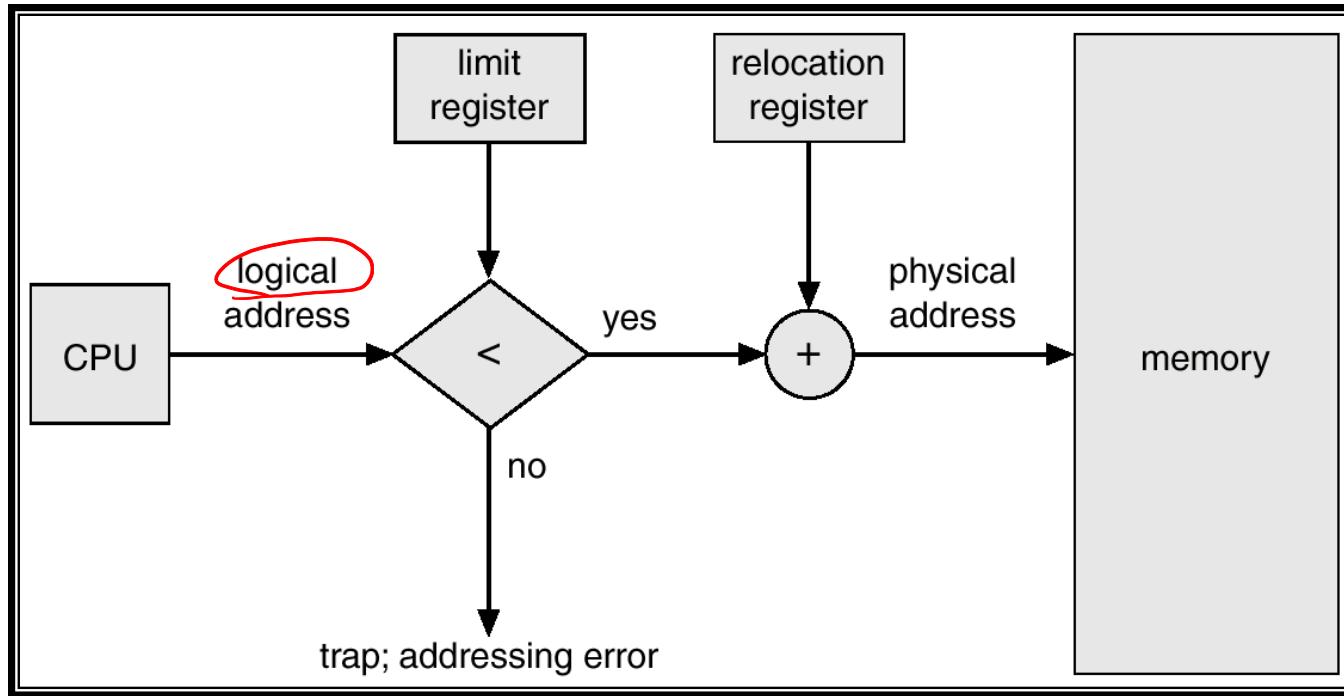
存储保护:

方法一：逻辑地址是否小于限长寄存器的值？

方法二：物理地址是否小于界限寄存器的值？

$$bound = BR + limit - 1$$

*if (logical < limit)
+ BR . → physical*



基于基址/限长寄存器的存储保护

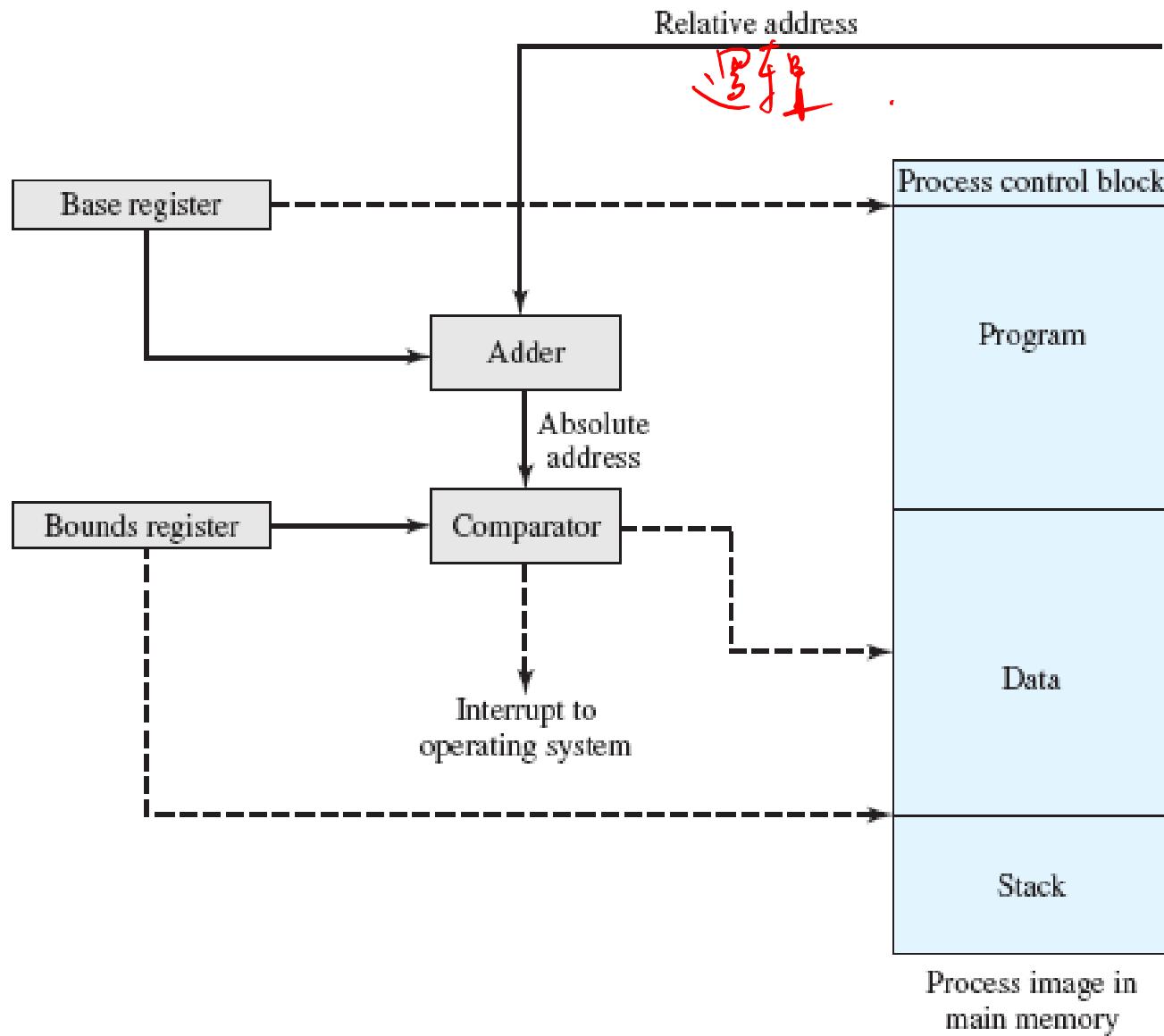


Figure 7.8 Hardware Support for Relocation

基于基址/界限寄存器的存储保护

Buddy算法/伙伴系统

二分法

- 整个存储空间可以被看成是大小为 2^U 的块
- 任何尺寸为 2^i 的空闲块都可以被分为两个尺寸为 2^{i-1} 的空闲块，这两个块称作伙伴，它们可以被合并成尺寸为 2^i 的空闲块。
- 如果请求的内存 s 满足 $2^{U-1} < s \leq 2^U$ ，则分配整个块
- 否则，将块分成两个相同的伙伴，直至分到大于等于 s 的最小的块为止，并将其中之一分配给进程

Buddy 系统的例子

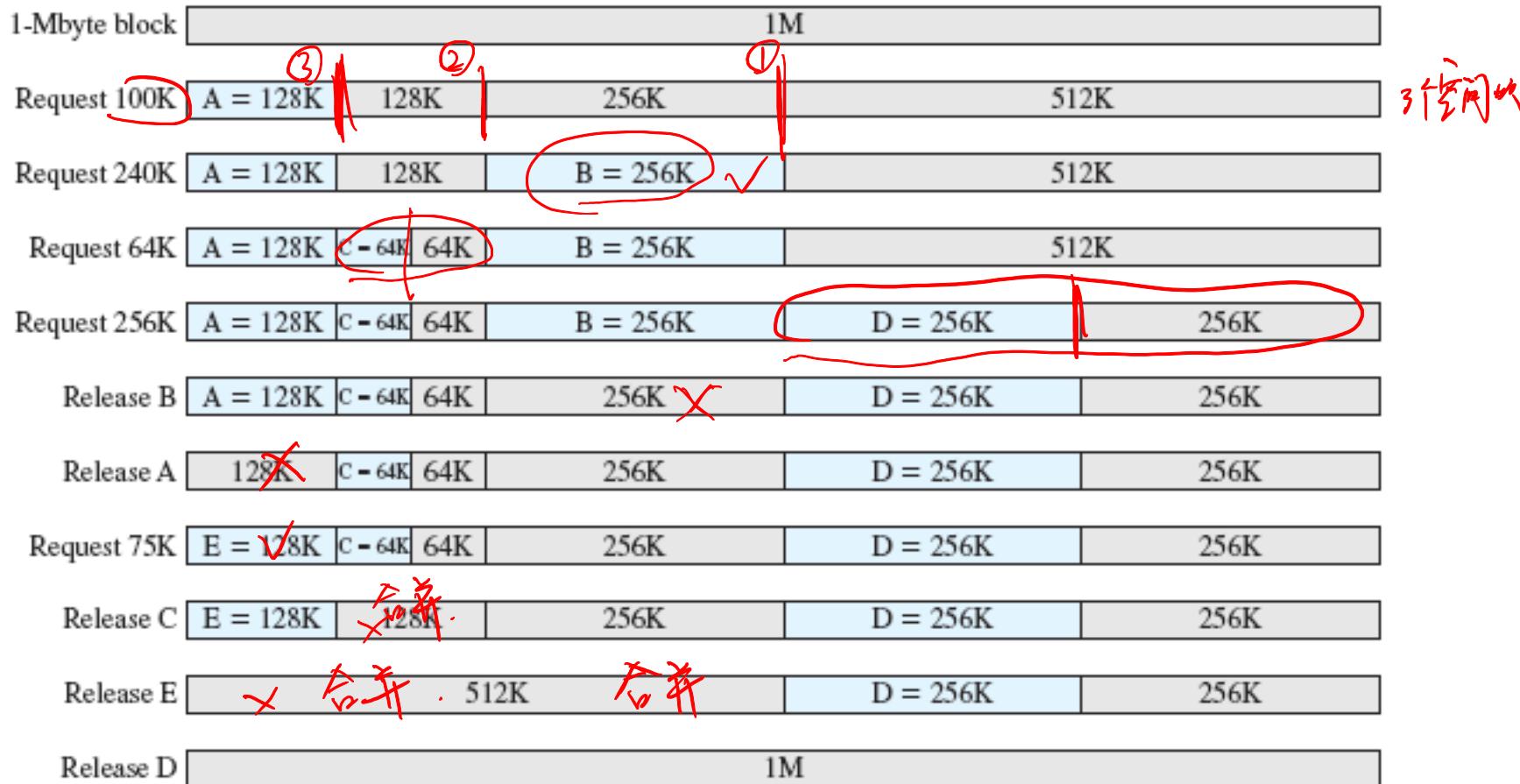


Figure 7.6 Example of Buddy System

Buddy系统的二叉树表示

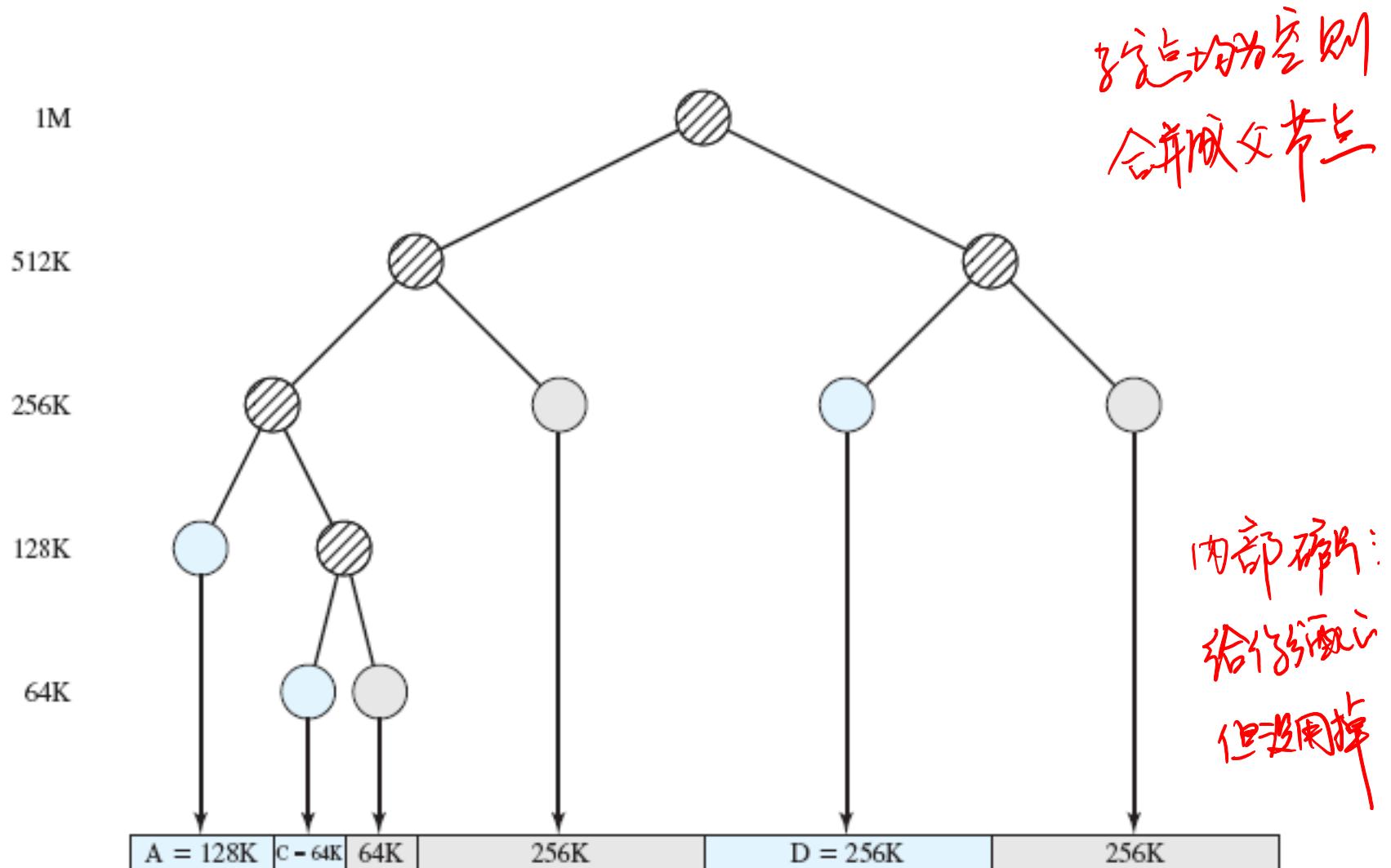


Figure 7.7 Free Representation of Buddy System

主存不足的存储管理技术

① 移动技术

- 可以解决可变分区中由于“碎片”问题导致的主存不足
- 将已在主存中的分区进行移动，使得分散的空闲区汇集成更大的空闲区，以满足新进程的内存请求

合并
并
碎

② 对换技术

- 当内存空间不足时，将某个处于阻塞态的进程移出主存，腾出空间给其它进程使用；同时将磁盘中的某个进程换入主存，让其投入运行
- 应该选择哪个进程对换出主存？应该将进程的哪些信息移出去？什么时候对换？

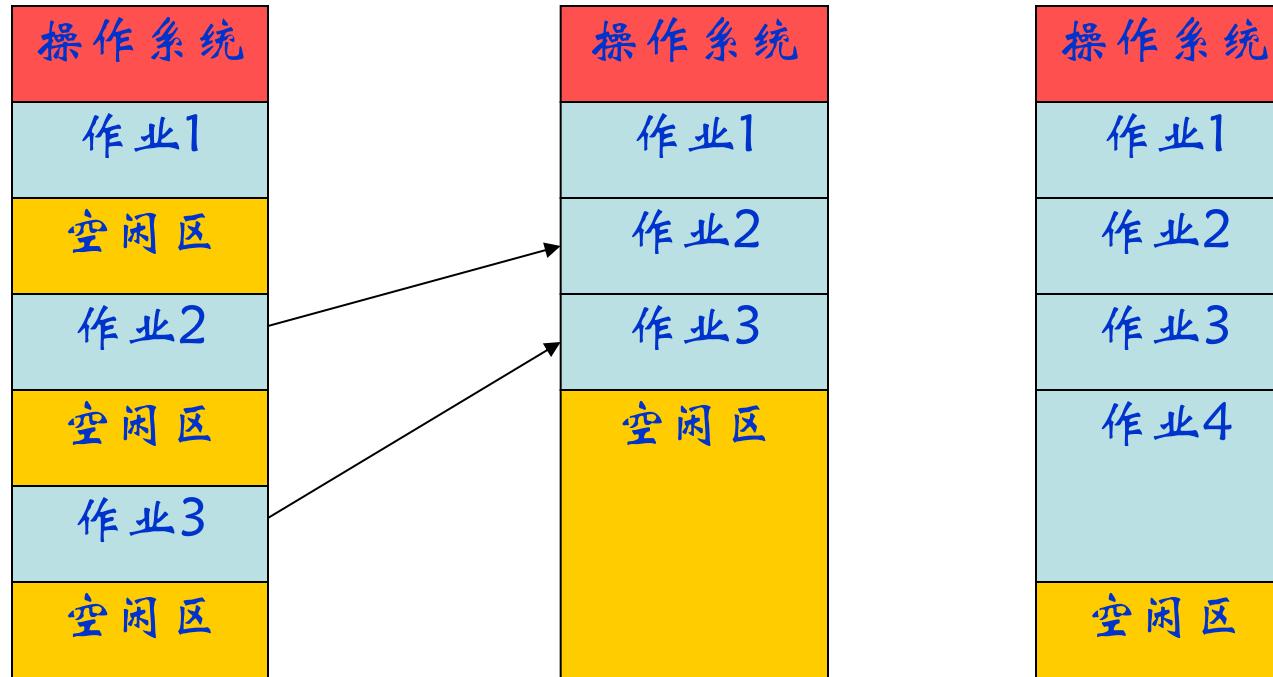
挂起

③ 覆盖技术

- 解决因程序的长度超出物理主存总和出现的主存永久性短缺问题
- 用户空间分成固定区和一个或多个覆盖区
- 把控制或不可覆盖部分放在固定区，其余按调用结构及先后关系分段并存放在磁盘上，运行时依次调入覆盖区。

$\geq \max$

虚地址请求



主存移动技术示例

大纲

- 存储管理的需求和作用
- 连续空间存储管理
- 分页存储管理 \rightarrow 离散
文件、磁盘连接扇区
- 分段存储管理

分页存储管理原理

$0 \sim 2^{32}$

- 分页存储管理允许逻辑上连续的地址空间映射到物理内存中不连续的空间中，只要存在可用物理内存，就可以分配给进程使用，而不管其是否连续。
 - 连续空间存储管理要求连续的内存空间
- 将物理内存分为固定长度的块，称为页框(frame)
 - 长度为2的幂次，如512字节，8192字节
- 将逻辑地址空间分为和页框相同长度的块，称为页面(page)
- 内存管理模块跟踪所有空闲页框
- 为了运行大小为n个页面的程序，需要为其分配n个页框，以载入程序。
- 建立页表，将逻辑地址映射到物理地址
 - 逻辑页面和物理页框的对应关系
- 为进程分配的最后一个页面可能存在内部碎片，平均每个进程的内部碎片大小为页框大小的一半

$$\text{page} = \frac{\text{frame}}{2}$$

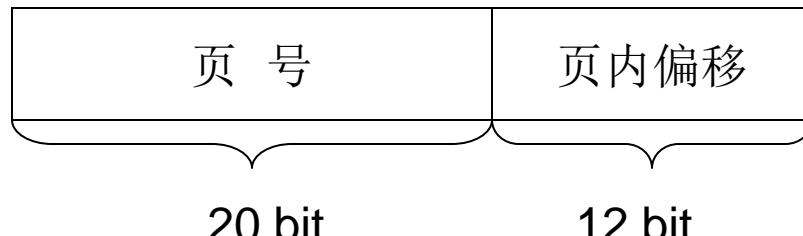
分页管理的数据结构--页表

- 将逻辑地址的页号映射到物理地址的页框号
- 页表中的每一项都指明了程序中的一个页面和物理内存页框之间的对应关系
- 在分页系统中，操作系统为每个在内存中的进程建立一张**页表**，用于支持地址转换
- 系统设置**页表基址寄存器**，用于存放当前运行进程的页表起始地址 找到页表在哪里

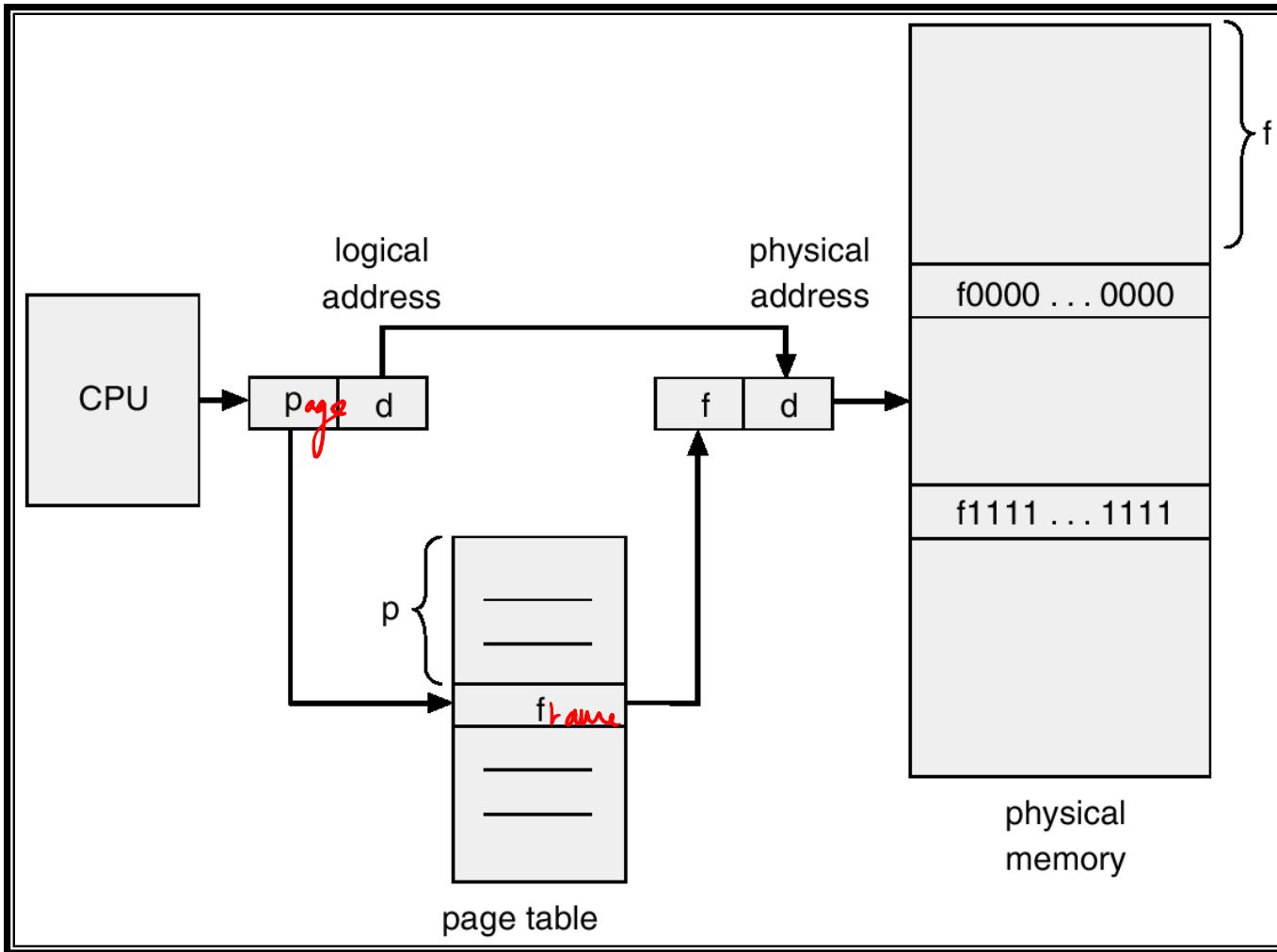
地址转换

- CPU产生的逻辑地址将分为
 - ~~页号(p)~~ - 用作查找页表的下标，页表中对应的位置存放了每个页面在物理内存中的首地址.
 - ~~页内偏移(d)~~ - 与页面在物理内存中的首地址结合，产生物理地址.

物理地址 = 页框号 \times 页面大小 + 页内偏移
物理



32位逻辑地址示意图



Virtual Address

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

Register

Page Table Ptr

Page Table

Frame #

Program

Paging Mechanism

Main Memory

Offset

Page
Frame

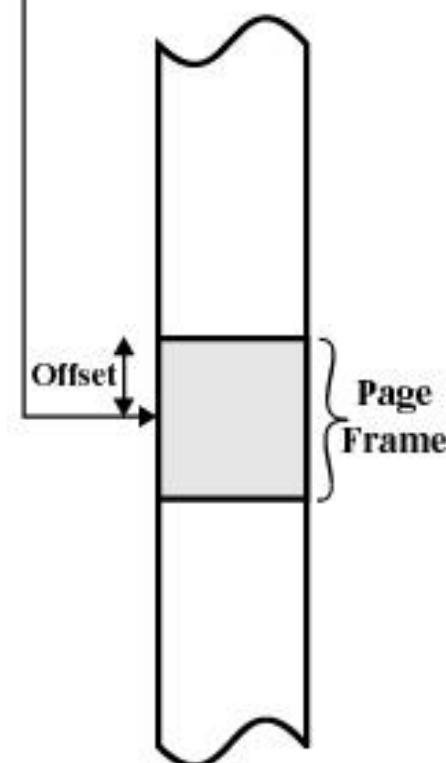
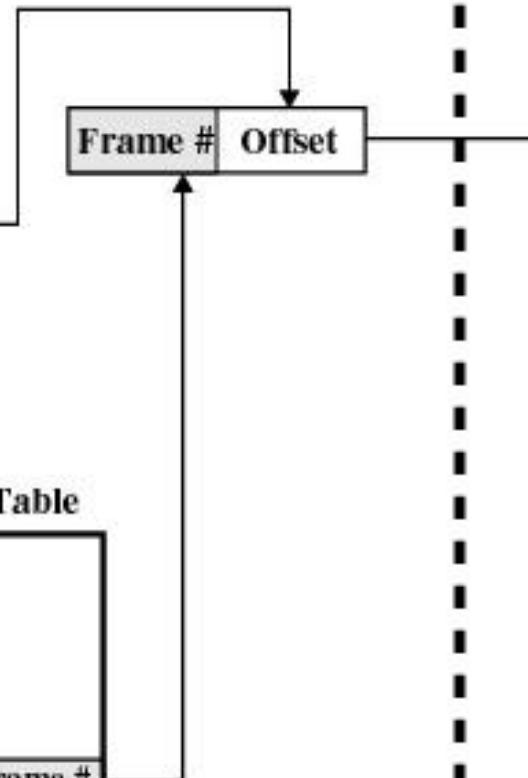
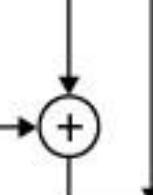


Figure 8.3 Address Translation in a Paging System

为进程的页面分配空闲页框

A: 4

B: 3

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen available frames

Main memory
A.0
A.1
A.2
A.3

(b) Load process A

Main memory
A.0
A.1
A.2
A.3
B.0
B.1
B.2

(c) Load process B

C:4

为进程的页面分配空闲页框

D:5

Main memory	
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	

(d) Load process C

Main memory	
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	

(e) Swap out B

Main memory	
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	

(f) Load process D

Figure 7.9 Assignment of Process to Free Frames

页表例子

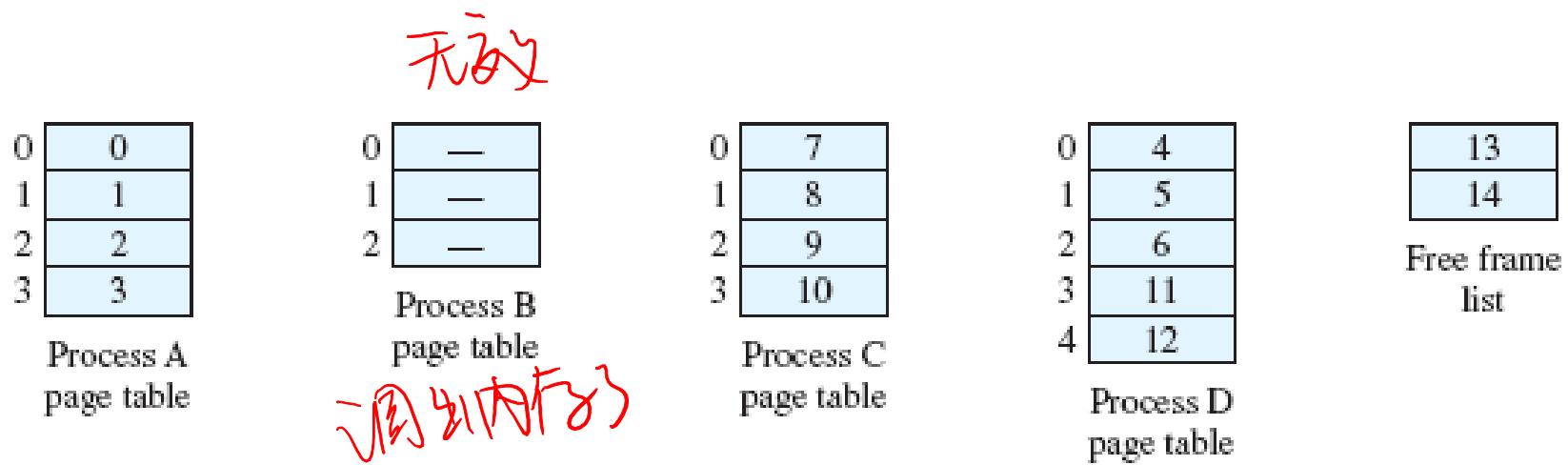


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

页表的实现

- 页表一般保存在内存中
- ① 页表基寄存器 (PTBR) 指向页表首地址
 - 进程切换时保存在进程控制块中
- ② 页表长度寄存器 (PRLR) 给出页表的大小
 - 进程上下
局部部分
- 整个系统只有一个 PTBR 和 PRLR，只有占用 CPU 的进程才占有它
- 每次取数据/指令需要两次内存访问，一次访问页表，一次访问数据/指令
 - 两次内存访问的问题可以通过一个特殊的快表 (TLB) 来部分解决

只有一个

Cache

相联存储器/快表(TLB)

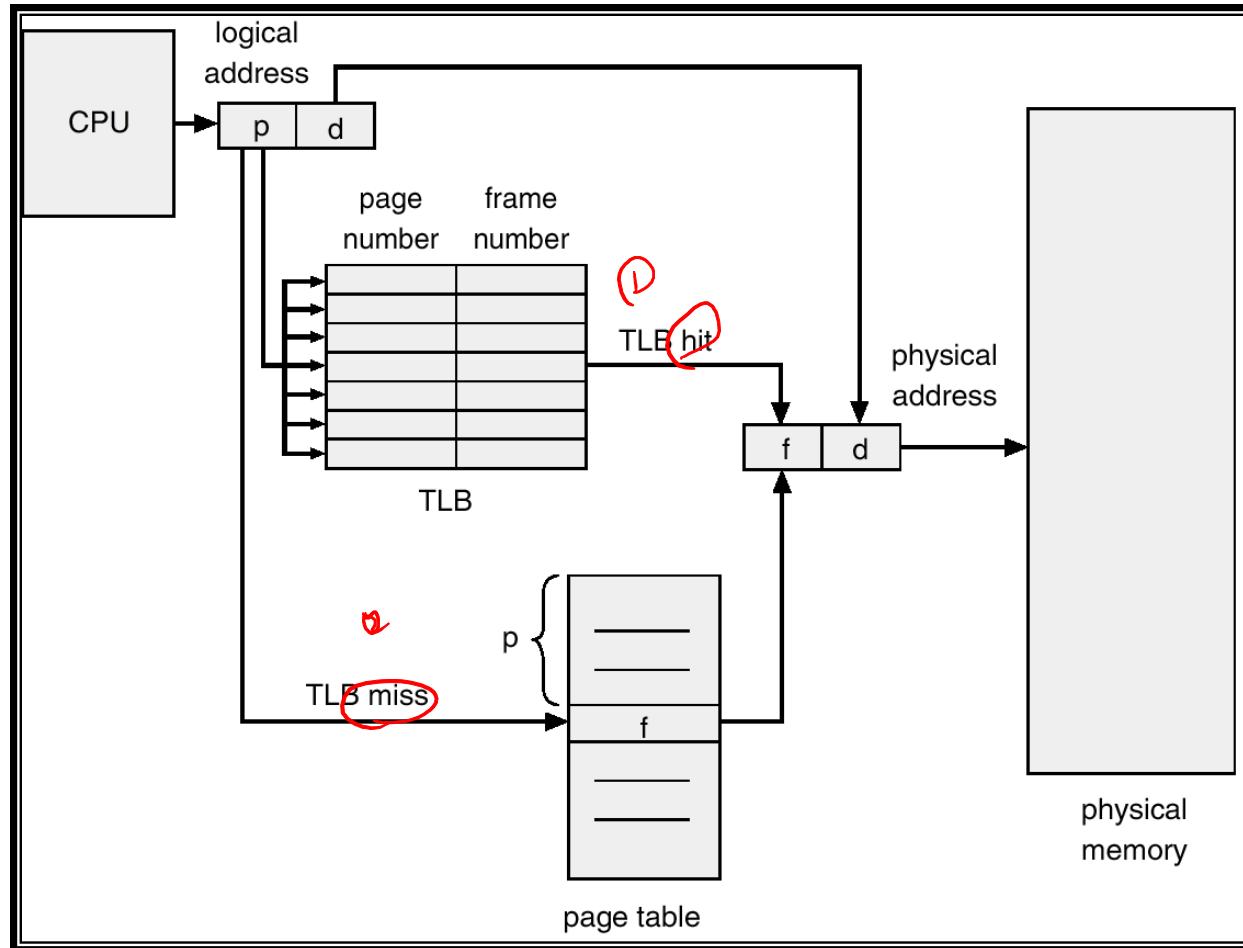
- 用于存放进程最近访问的部分页表项，以提高转换速度
- 可以进行并行匹配

Page #	Frame #

地址转换过程 (A' , A'')

- 若 A' 位于 关联存储器, 则直接获取页框号 frame #.
- 否则, 从 主存中的页表中 获取页框号 frame #

具有TLB的页面转换机制



TLB

多个进程 (进程标识符)
可不用清空 TLB.

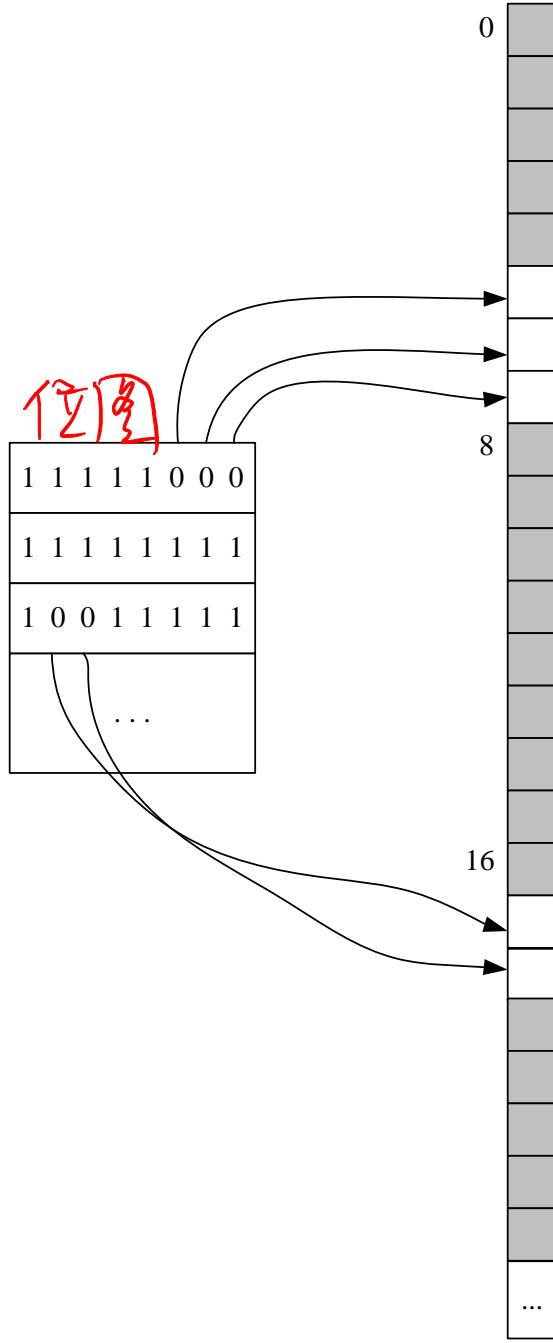
- 一些TLB还在每个TLB表项中保存ASID(地址空间标识符)，用于进程的地址空间保护。
- 当TLB用于解析虚拟页面地址时，需要保证当前运行进程的ASID与TLB中的ASID匹配。如果两者不匹配，则被视作TLB miss。
- 在TLB中存储ASID允许TLB同时存放不同进程的页面和页框映射；否则，每次进程切换时，需要清空TLB，以保证下一个执行的进程不会错误地使用之前进程的快表。

分页存储空间的分配与去配

- 用位图记录页框的分配情况，每位与一个页框相对应
1bit *物理有无被占用*
- 用0/1表示对应块为空闲/占用，另用一个专门字记录当前空闲块数。
- 分配算法
 - 若空闲块数<进程所需块数，则进程等待；
 - 否则，*查找位图*，找出为“0”的那些位，置占用标志，从空闲块数中减去本次占用块数，按所找到的位的位置计算对应的页框号，并填入对应进程的页表。

1位用

0位用



分页存储的存储保护

是否有效

法一

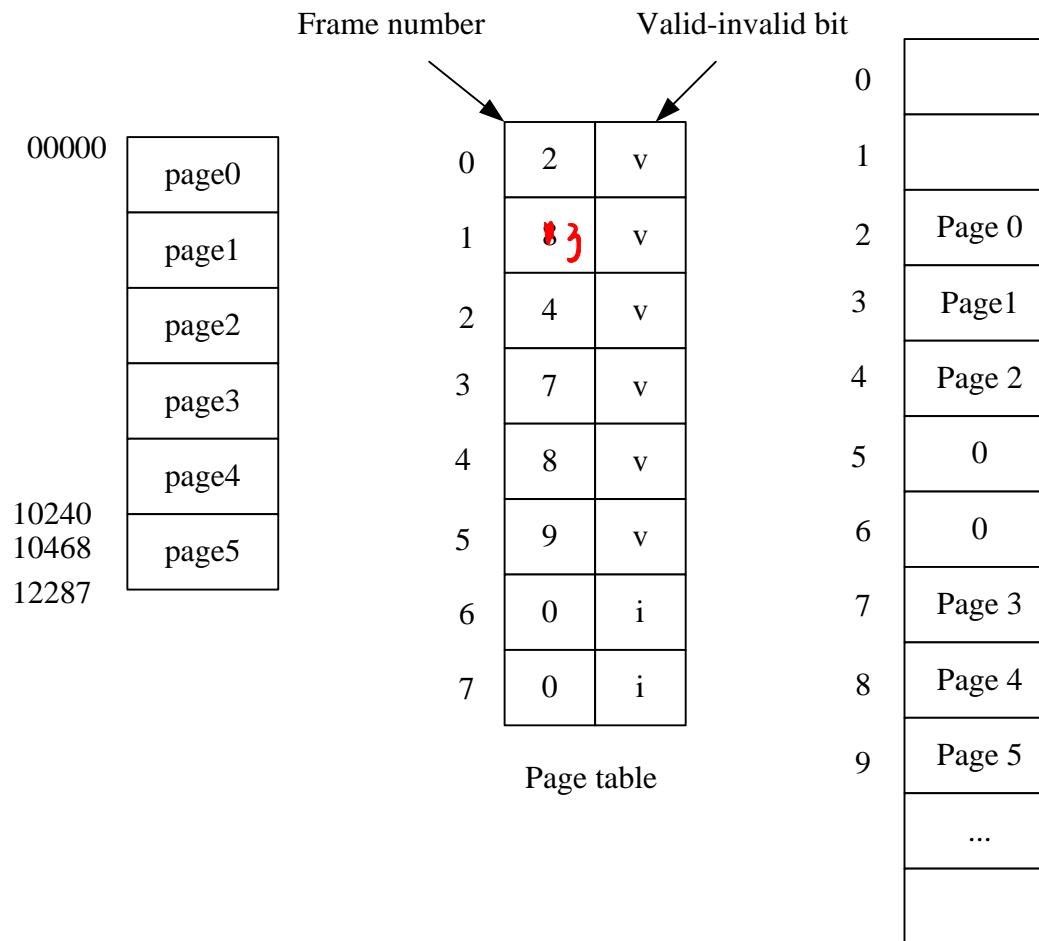
- Valid-invalid bit

- 页表中每一项后附加一位，用以标识该页是否属于进程的逻辑地址空间。
- 如果访问的地址对应的页表项标识为invalid，则产生异常。

法二

页表长度寄存器

- 很多时候，进程仅使用地址空间的一小部分。
- 为地址空间中的每一页都创建一个页表项很浪费空间
- 一些系统提供了页表长度寄存器，指出页表的长度
- 每个逻辑地址都与该值比较，如果逻辑地址的页号超过了页表长度寄存器的值，则产生异常。



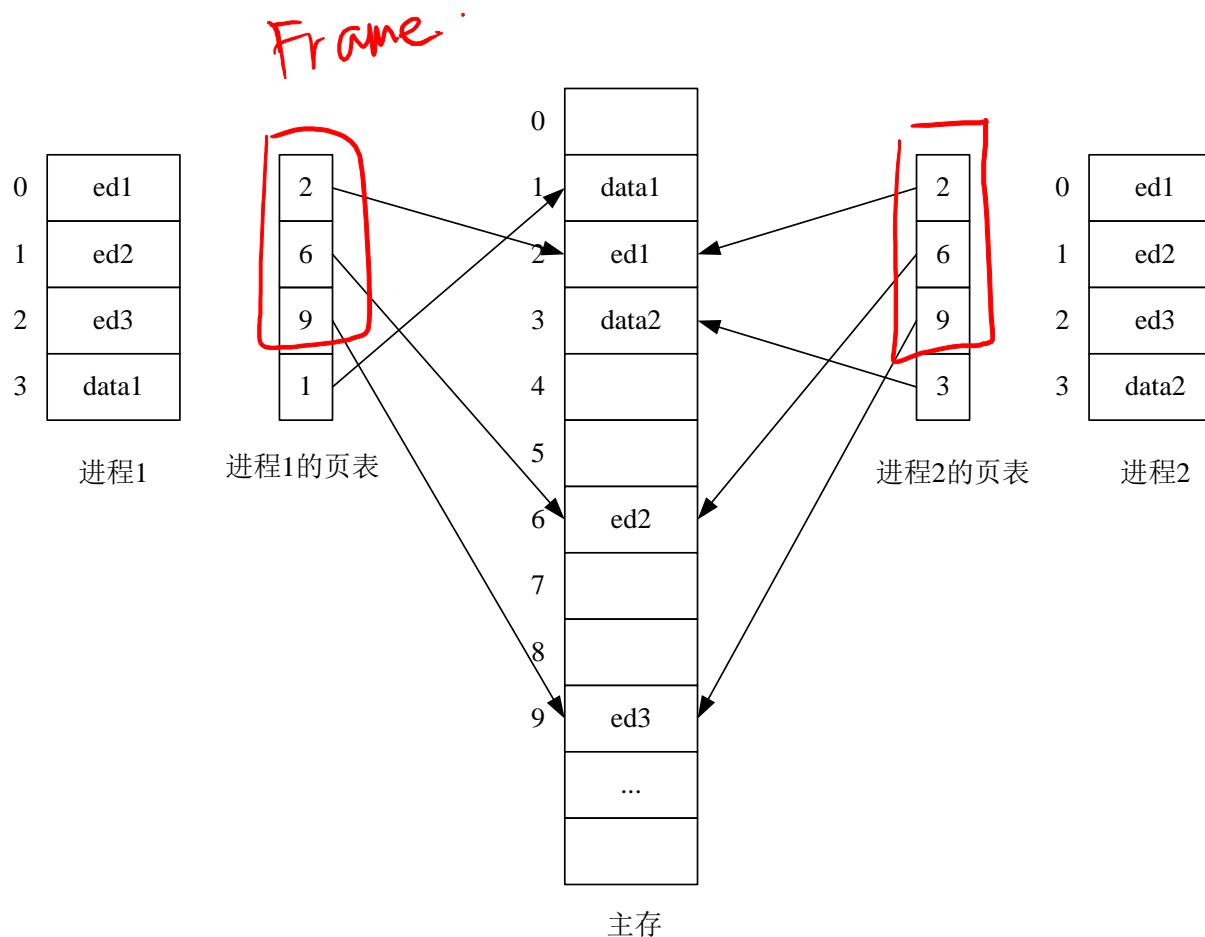
14位地址空间（0~16383），页大小为2K，每个进程可用的页数为8。

一个进程实际使用的地址空间为(0~10468)，则需要6个页。

11

$$2^3 = 8$$

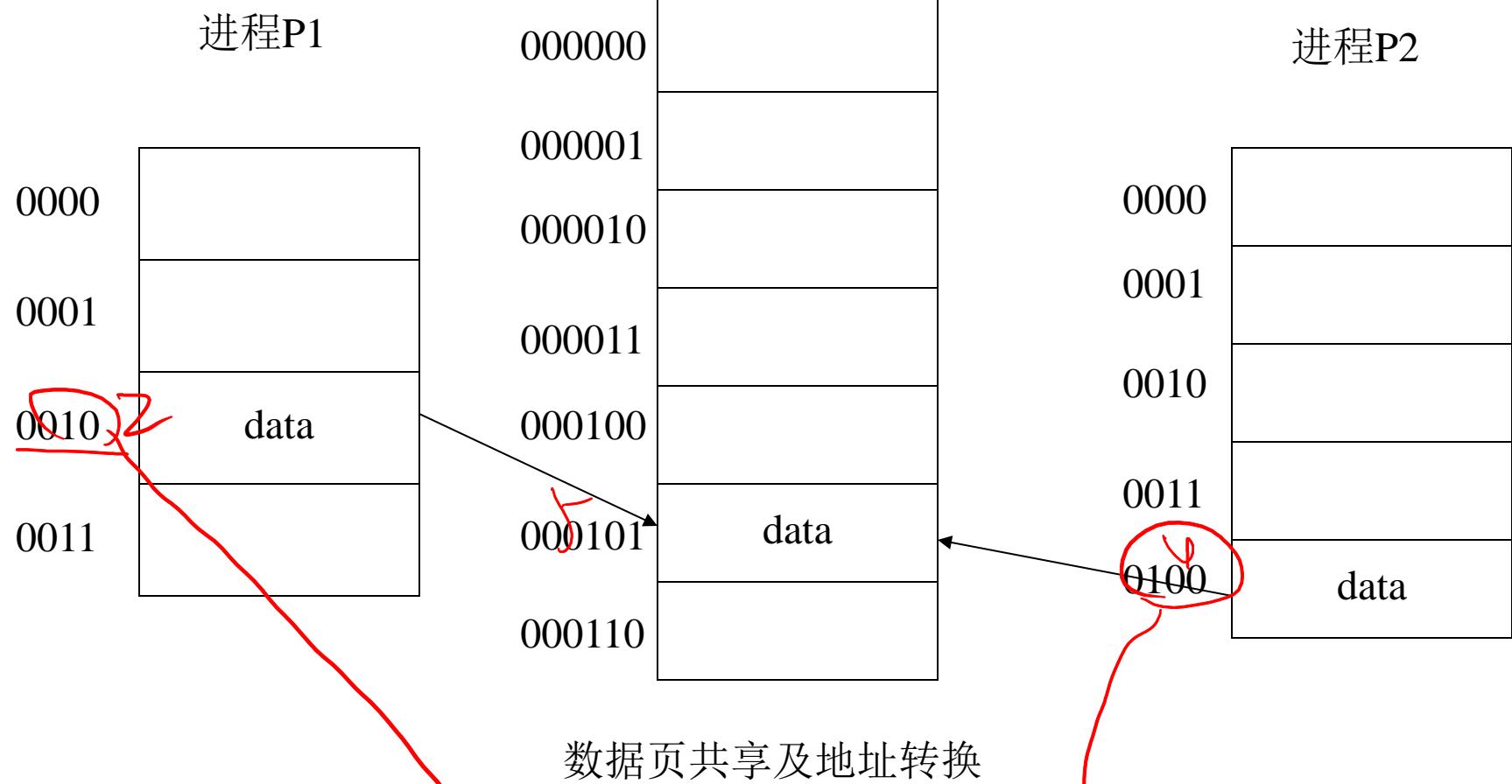
分页存储中的存储共享



页面共享时的页面设置

- 数据页共享
 - 允许不同进程对共享数据页采用不同的页号
- 代码页共享
 - 要求不同的进程为共享代码页在逻辑空间中指定同样的页号
 - **Why?**

物理内存



设页面大小为1K，逻辑地址为14位,物理地址为16位，则
 进程P1的数据页逻辑地址为0010 000000000000-0010 1111111111
 进程P2的数据页逻辑地址为0100 000000000000-0100 1111111111
 地址转换: 不同
 进程P1 0010 0101011000 → 00010101011000
 进程P2 0100 0101011000 → 00010101011000

都指向 5号页框

主存

进程P1

0号页面

JUMP(0, 234)

...

234

JUMP(0, 234)
Or JUMP(2, 234)?

只能取其一!

进程P2

2号页面

JUMP(2, 234)

...

234

edit1

edit2

程序代码共享需要在不同的进程逻辑地址空间中分配相同的页号

052不归

让页表也不连续存放

页表的组织

主机 → 4MB
64个块

- 层次化页表/多级页表
- 哈希页表
- 反置页表

层次化多级页表

- 现代计算机的逻辑地址空间很大，采用分页存储管理时，页表相当大
 - 32位逻辑地址空间，页面4KB，则页表为 2^{20} 项，若每个页表项占用4字节，则每个页表需要占用4MB的连续存储空间
- 解决办法：
 - 将页表分页，形成多级页表，从而允许页表存放在不连续的空间中
 - 最常见的是二级页表

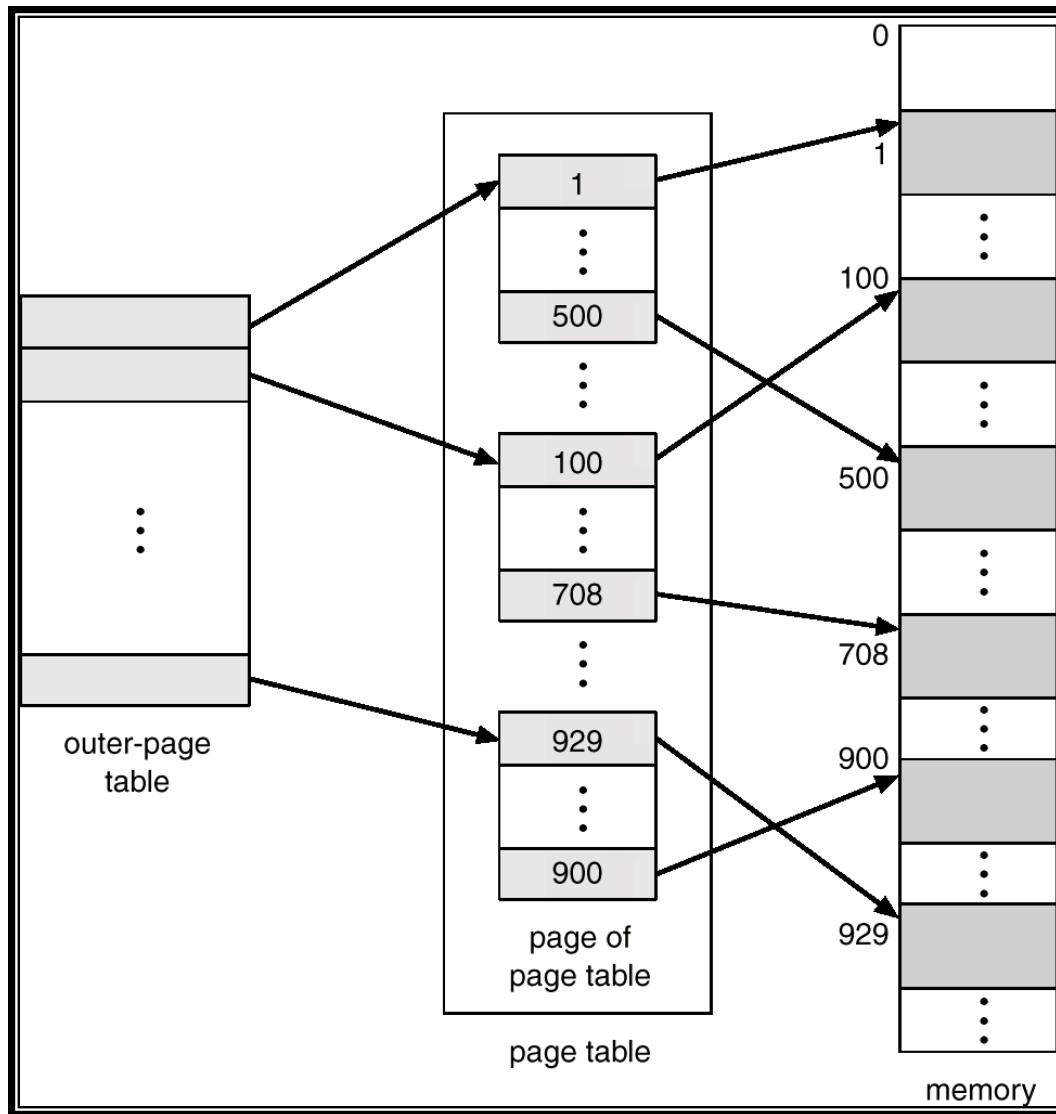
二级页表

- 例如，页面大小为4K的32位地址首先被分为：
 - 20位的页号
 - 12位的页内偏移
- 对页表进行二次分页：
 - 10位的页号
 - 10位的页内偏移（每个页表项占4字节） $2^{10} \times 4 = 4K$
- 因此，逻辑地址被表示为：

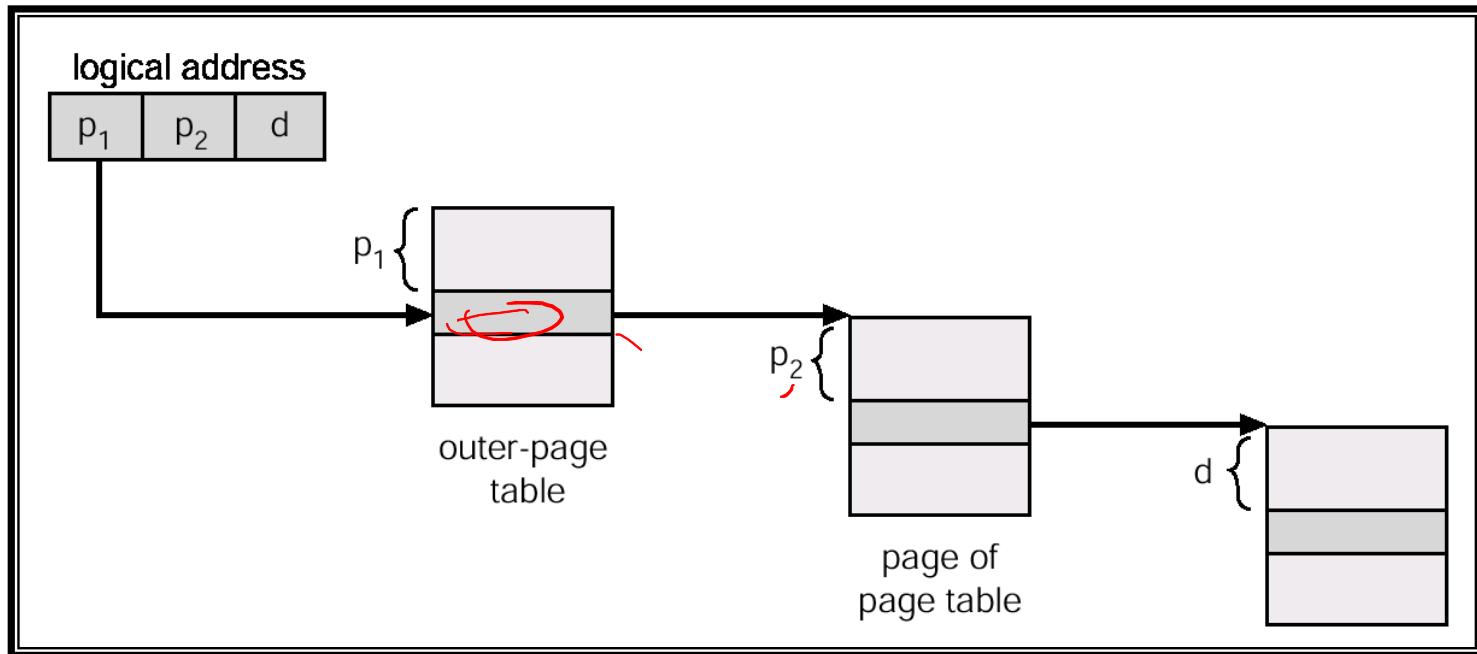
page number	page offset
p_1	p_2
10	12

其中， p_1 用于查找一级页表， p_2 用于查找二级页表。

二级页表



二级页表的地址转换



访问数据或指令需要三次内存访问

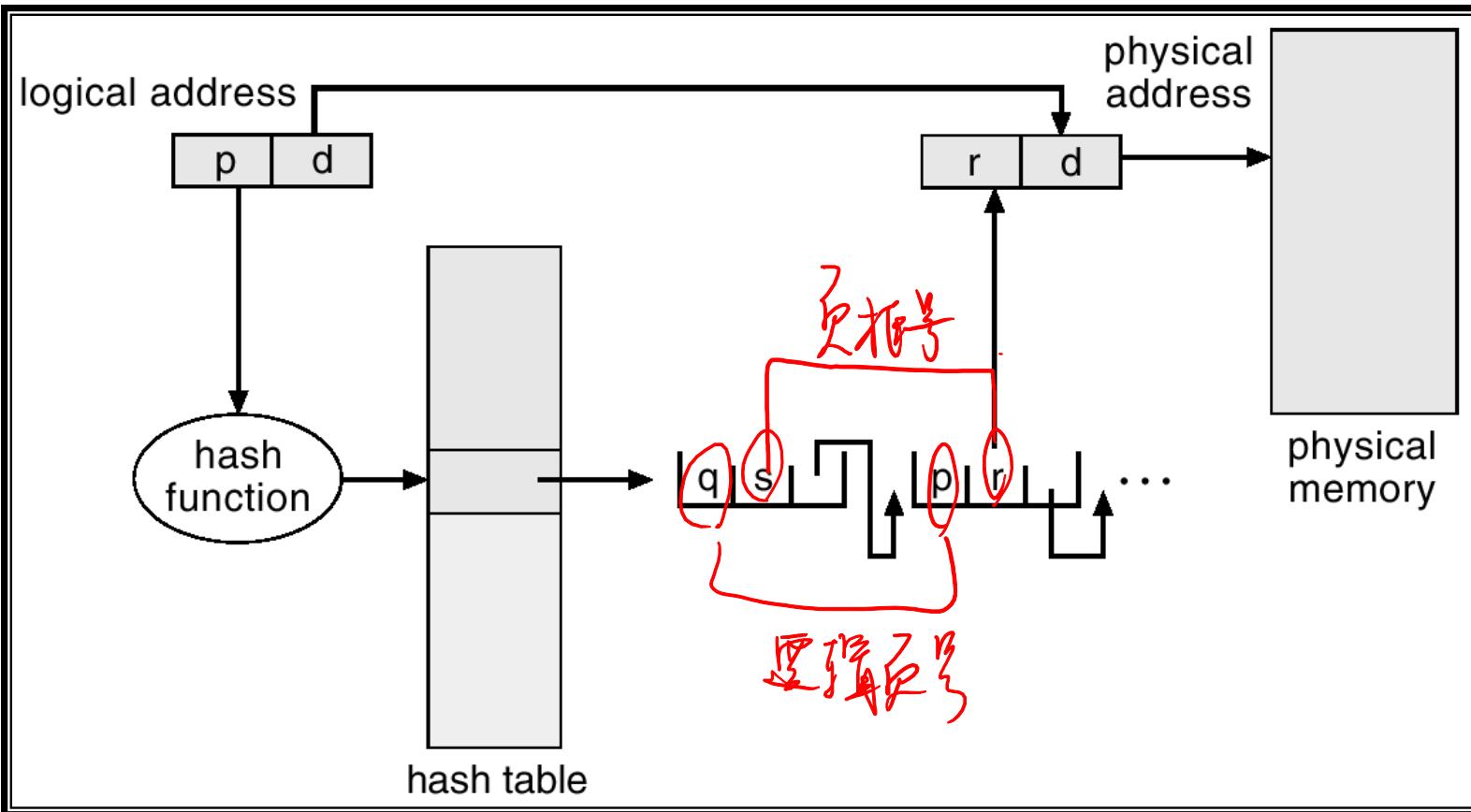
哈希页表

解决

大
→
小

把逻辑小

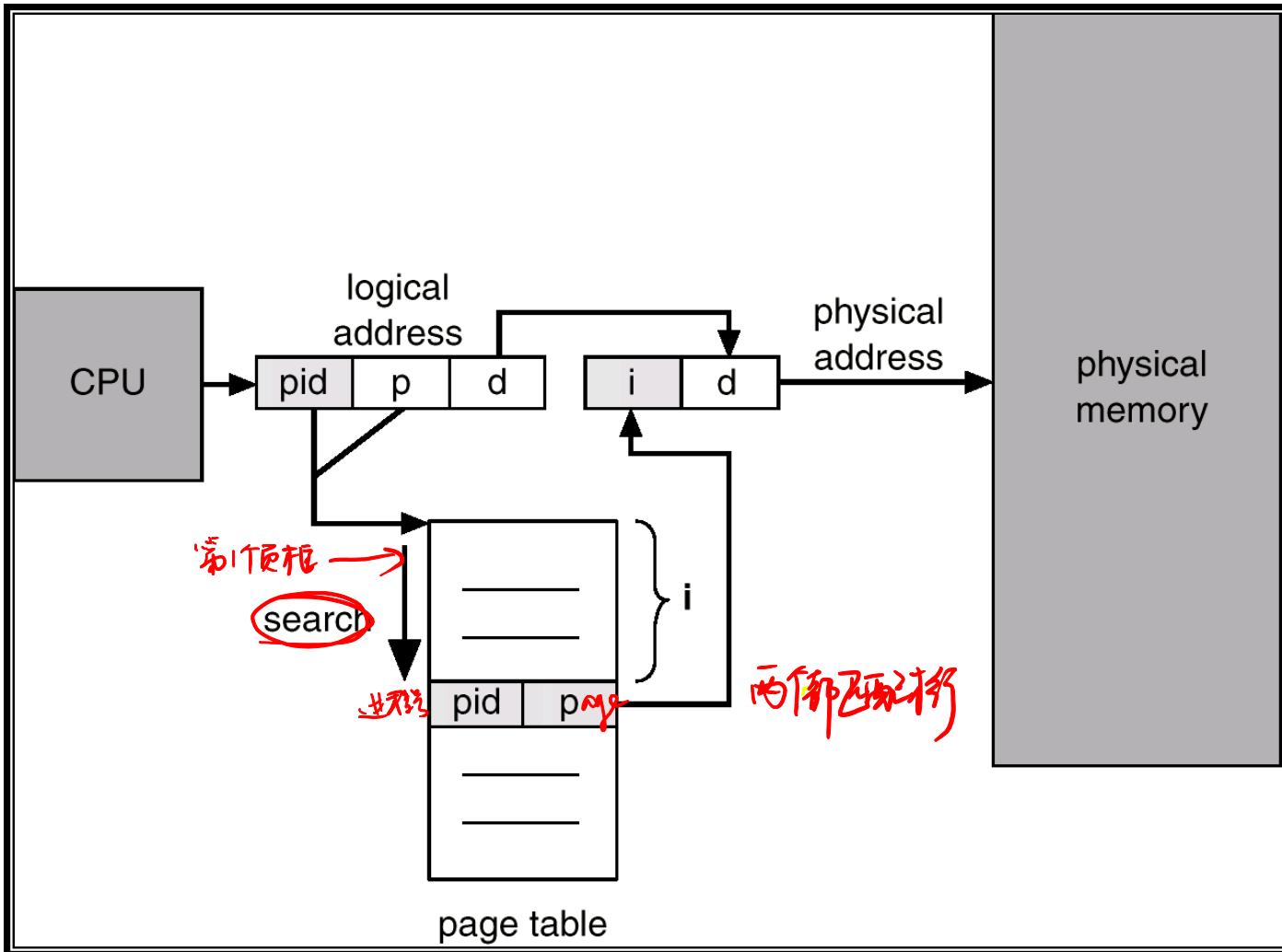
- 在地址空间大于32位时，通常采用哈希页表的方式
- 进程的逻辑页号通过哈希函数映射到一个哈希表中的值
- 每个哈希表项都有一个链表组成，以解决哈希冲突
- 每个链表元素由三个字段组成
 - 原始的逻辑页号 链接下
 - 对应的页框号
 - 指向下一个链表元素的指针

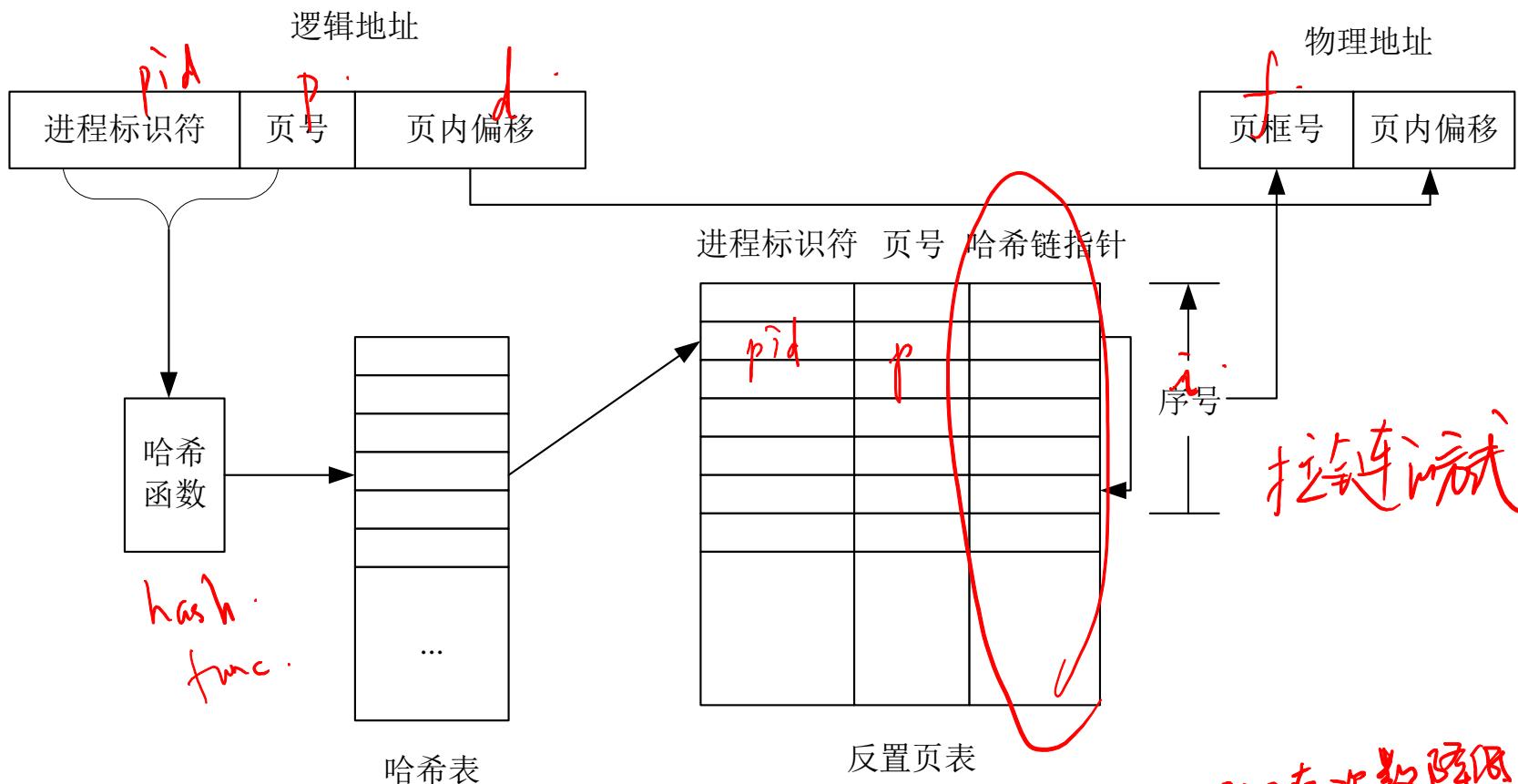


页框被哪个逻辑页号占用

反置页表

- 传统上，每个进程要有一个页表，便于进程进行地址转换，但页表占用的存储空间大。
- 反置页表为所有进程维护一张页表，每个物理内存的页框在页表中包含一项。
- 每个页表项包含占用该页框的进程号以及对应的逻辑页号。
- 降低了存储页表的开销，但增加了访存的查询时间。
– 可以采用哈希表来降低查找的次数。





基于哈希表的反置页表实现

哈希表和反置页表的哈希连指针更新较复杂

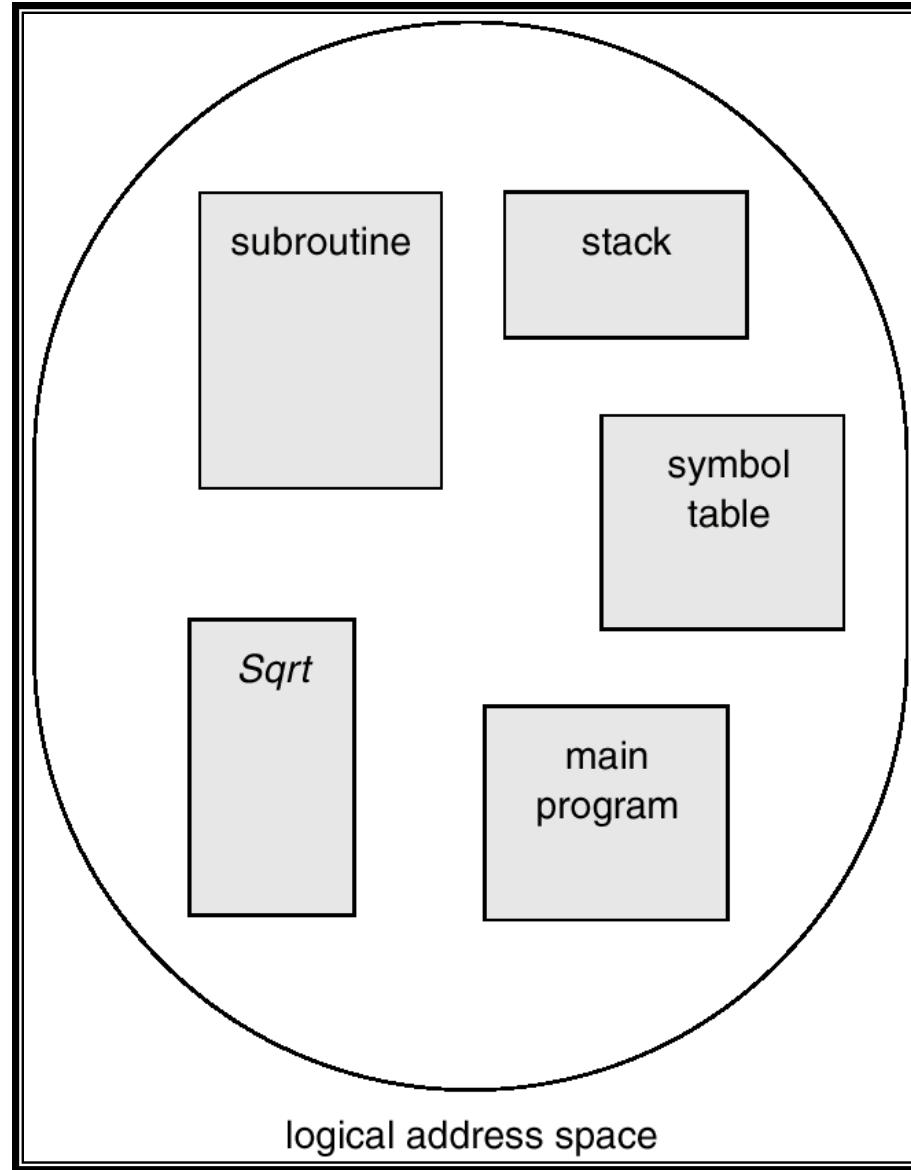
大纲

- 存储管理的需求和作用
- 连续空闲存储管理
- 分页存储管理
- 分段存储管理

分段存储管理

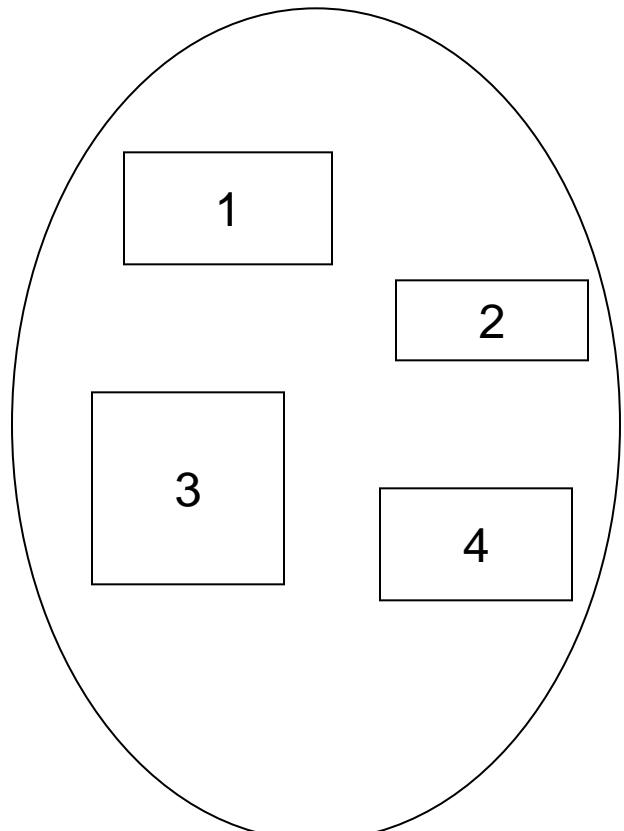
- 满足用户（程序员）编程和使用上的要求。
- 程序员并不将内存视作是一维的数组，而更愿意将内存看作是一组大小不同的段
 - 主程序
 - 对象
 - 数组
 - 栈
 - ...
- 逻辑地址空间是若干个段的集合，每个段都有自己的名称和长度，每个段由连续的逻辑地址构成，在物理内存中也连续存放
- 地址由两部分组成：段号和段内偏移

段号	段内偏移
----	------

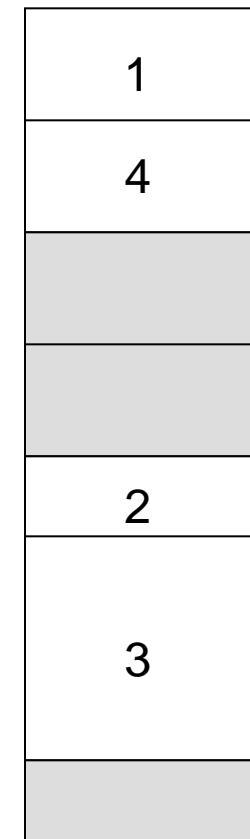


程序的分段逻辑结构

分段的逻辑视图



user space



physical memory space

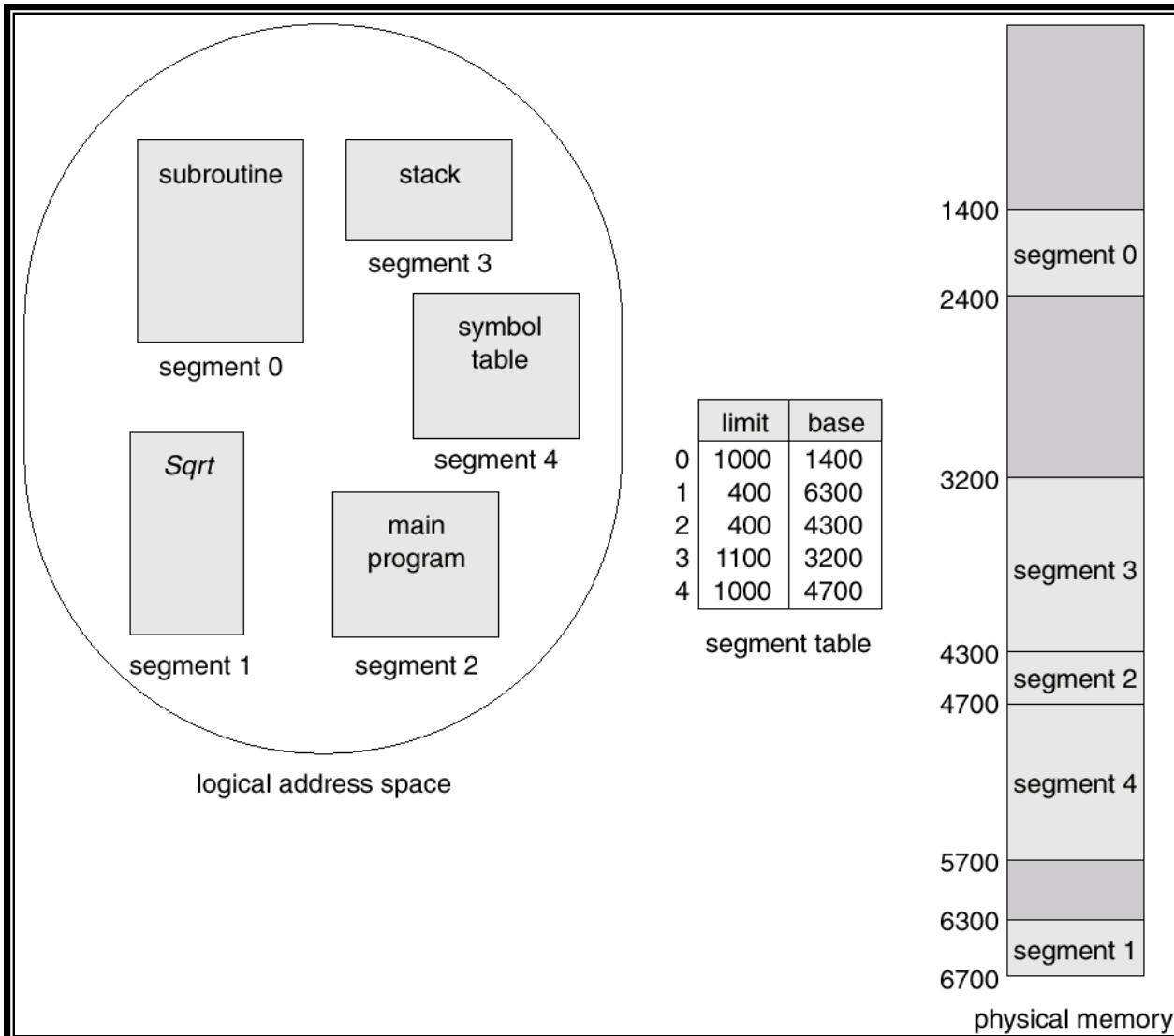
段表

- 用于将二维的逻辑空间地址映射到一维的物理地址
- 段表中的每一个表项包含：
 - 段号
 - 段起始地址
 - 段长度

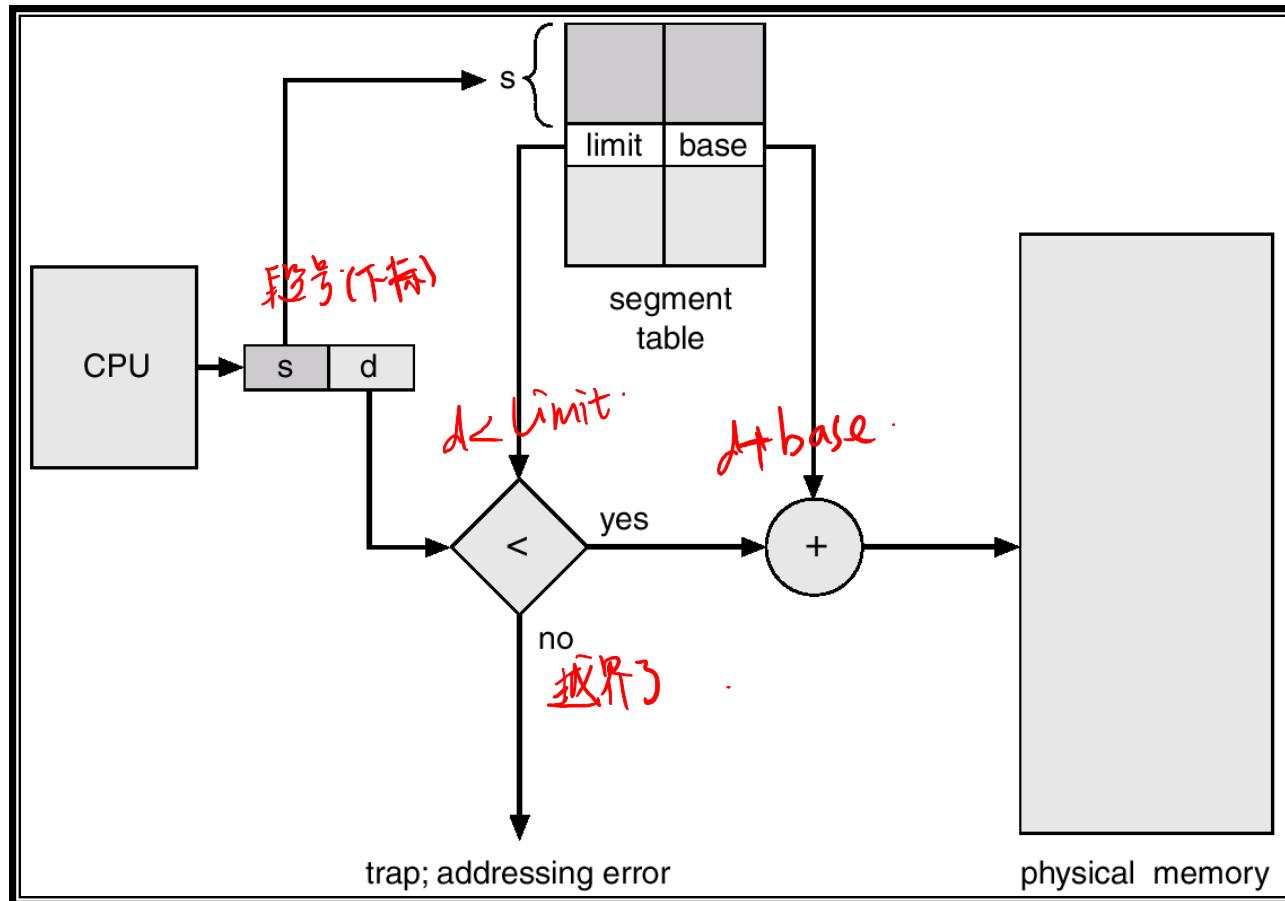
→ 基址 + 限长寄存器

段号寄存器
页表基址

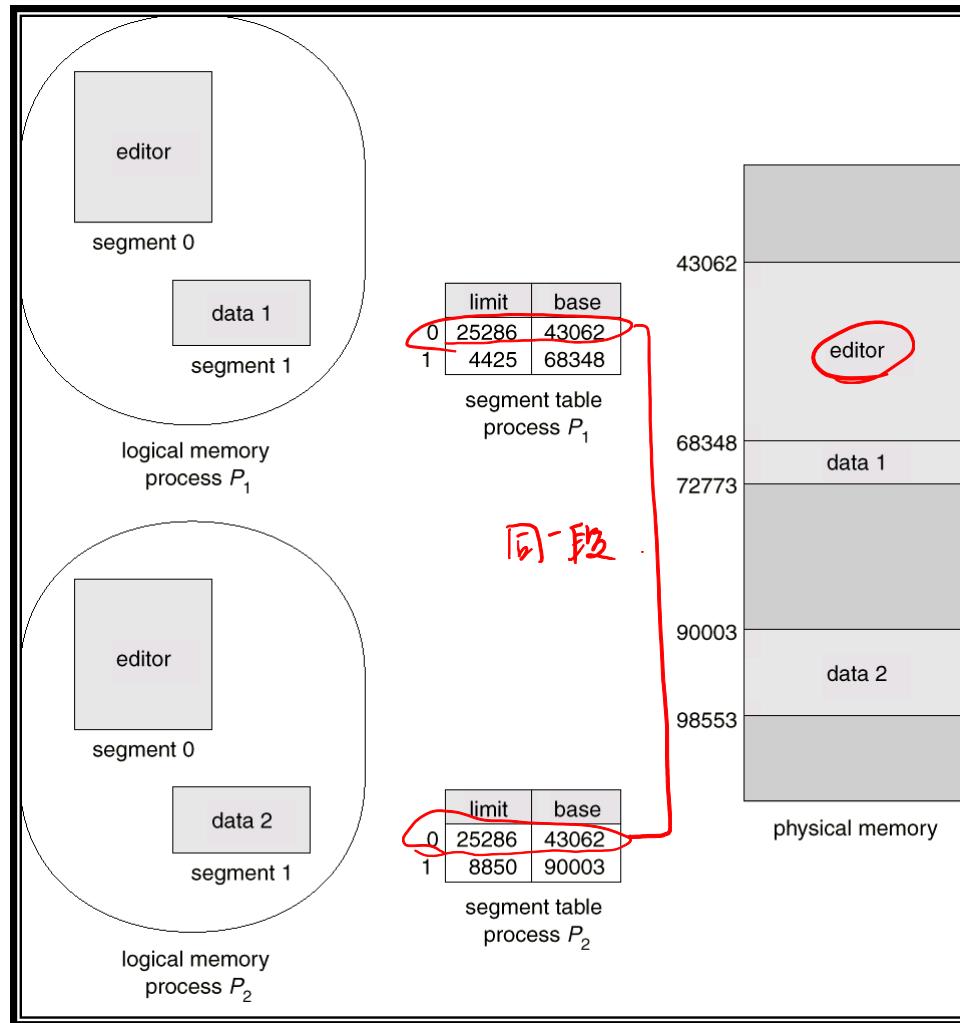
段表的例子



分段存储管理的地址转换和存储保护



段共享



分段和分页的比较

- 信息组织:

面向 coder - 分段是信息的逻辑单位由源程序的逻辑结构及含义所决定, 是用户可见的

面向 machine - 分页是信息的物理单位与源程序的逻辑结构无关, 是用户不可见的
机器支持分页

- 长度:

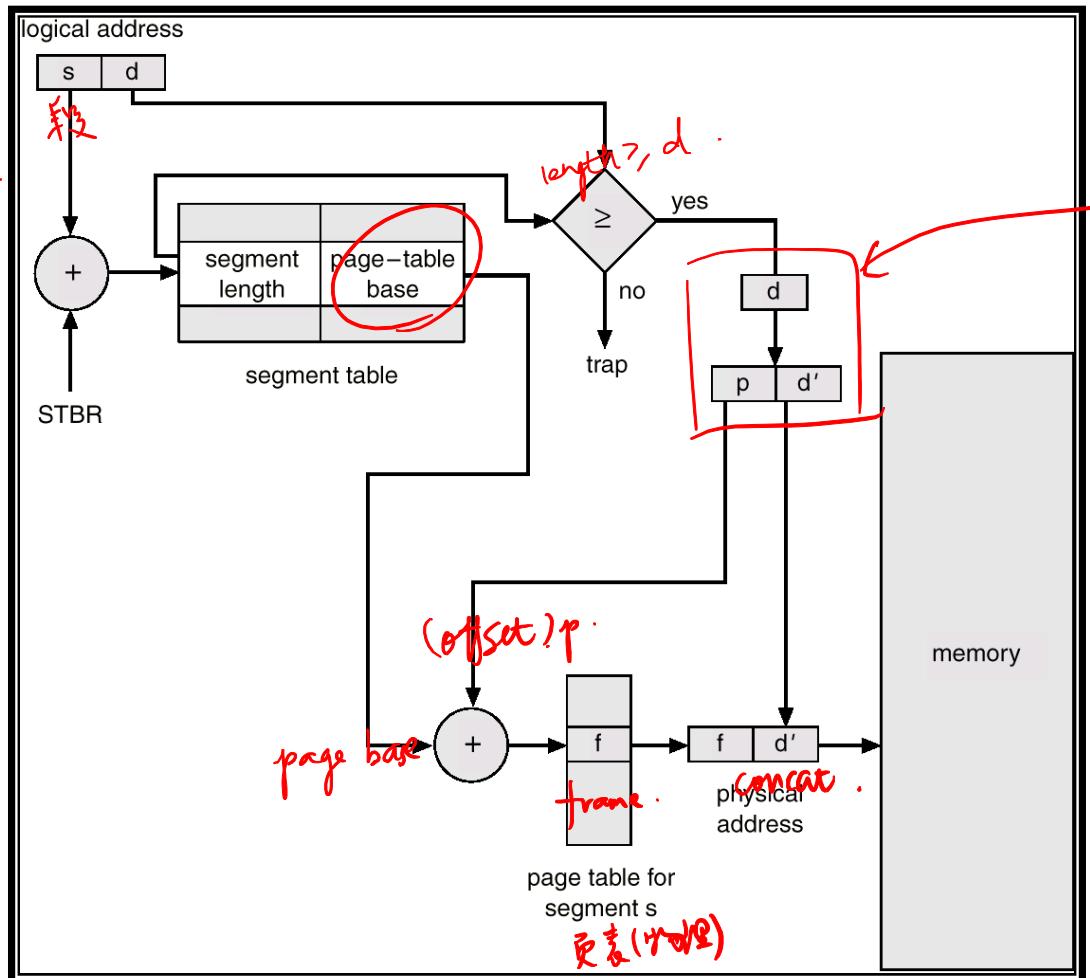
- 分段的长度由用户根据需要来决定, 每个分段在内存中是连续存放的, 体现了连续存储空间管理的思想, 段起始地址可以是任何地址;

- 页长由系统的硬件确定, 页面只能从页大小的整数倍地址开始。

分段和分页相结合

- 对段进行分页
- 段表不再存放段起始地址，而是存放该段对应的页表的起始地址

段内偏移
块内偏移



段内偏移
划分
页表
二级页表

二级页表
p1 | p2 | d'

分段和分页相结合的地址转换与存储保护

安全性问题

- 进程之间的内存访问必须受限
- 缓冲溢出攻击

缓冲溢出

- 缓冲放入比指定容量更多的数据，从而覆盖其它信息
- 攻击者利用这一漏洞来攻破系统或获得系统的控制权
- 后果：
 - 访问错误的数据
 - 非法的程序转移（破坏程序执行逻辑）
 - 内存访问越界
 - 程序异常终止

```
/*buffer1.c*/
int main(int argc, char *argv[ ])
{
    int valid = 0;
    char str1[8];
    char str2[8];

    next_tag(str1); //假设该函数用于获取系统的某个字符串，如口令；
    gets(str2); //从命令行输入一个字符串，gets()方法不检查输入字符串的长度，而会把字符串都
                 //拷贝到str指针指定的地方，若超出缓冲区长度，则覆盖临近的内存内容；
    if(strncmp(str1, str2, 8) == 0) //比较输入的字符串是否匹配后台的字符串；
        valid = 1;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

```
$gcc -o buffer1 buffer1.c
$./buffer1
START
buffer1: str1(START),str2(START), valid(1) //假设从系统取得的字符串为START
$./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

Memory Address	Before gets (str2)	After gets (str2)	Contains Value of
.....	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 v . @	42414449 B A D I	str2[0-3]
.....	

ges()前/后栈空间的内存

本章小结

- 程序员编写的源程序通过**编译、链接、装入**三个阶段载入内存
- 从程序的逻辑地址空间到物理地址空间的映射称为**地址转换、地址映射或地址重定位**，现在一般都采用**动态地址重定位技术** *暂时保留逻辑 addr*
- **连续存储空间管理**指进程总是被分配到一块连续的内存空间，包括**固定分区**存储管理和**可变分区**存储管理，前者易产生**内部碎片**，后者易产生**外部碎片** *buddy 伙伴系统*
- 可变分区存储管理通过**基址/限长寄存器**实现地址转换和存储保护

本章小结（续）

- 分页存储管理可以将连续的逻辑地址空间映射到物理上不连续的页框，能有效提高物理内存的利用率
- 分页存储的地址转换和存储保护通过页表实现
- 为了解决页表过大的问题，通常采用多级页表的方式
- 分段存储管理是为了满足用户编程和使用上的要求
- 分段存储的地址转换和存储保护通过段表实现
- 分段存储通常和分页存储结合使用，对每个段进行分页