操作系统原理

南京师范大学 计算机与电子信息学院

夏年

Tel：15052875526

Email：nian.xia@nnu.edu.cn

明理楼—108

第六章 存储管理

课程结构

操作系统原理

- 背景
  - Chapter 1 计算机系统
  - Chapter 2 操作系统
- 进程
  - Chapter 3 进程控制和管理
  - Chapter 5 并发：互斥与同步
  - Chapter 6 并发：死锁
- 内存
  - Chapter 7 存储管理
  - Chapter 8 虚拟内存
- 调度
  - Chapter 4 处理器调度
- 输入输出（I/O）
  - Chapter 9 设备管理
- 文件
  - Chapter 10 文件系统

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
  - 分页存储管理
  - 分段存储管理

大纲

- **6.1 存储管理的需求和作用**
- 6.2 连续空闲存储管理
- 6.3 分页存储管理
- 6.4 分段存储管理

6.1 存储管理的需求和作用

- 单道程序设计中，内存被分为两部分：操作系统使用的内存，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。

6.1 存储管理的需求和作用

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

6.1 存储管理的需求和作用

地址重定位

- 程序员无需知道程序在执行时在内存中所处的位置
- 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

6.1 存储管理的需求和作用

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
- 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**
  - 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

6.1 存储管理的需求和作用

内存共享

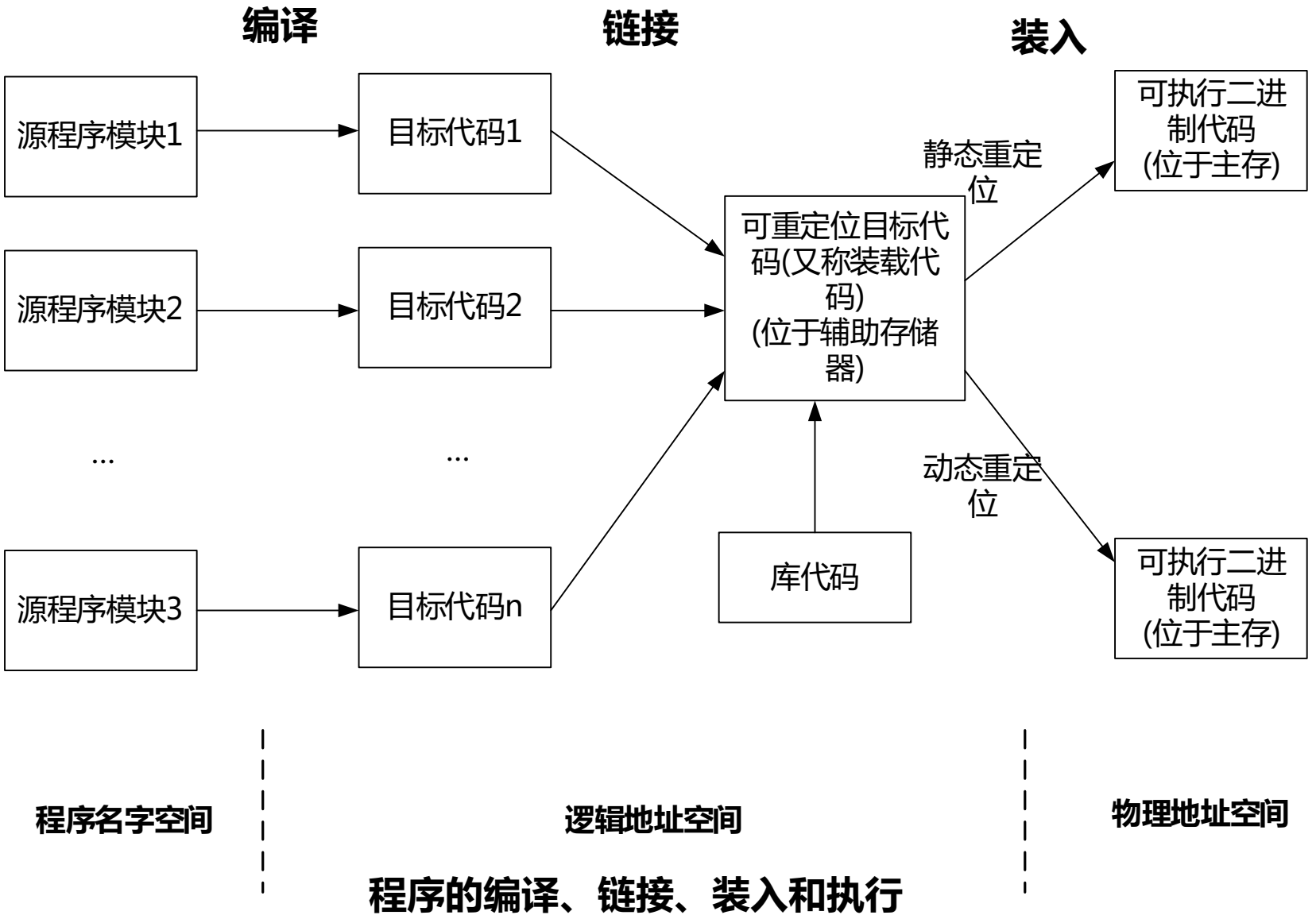
- 允许多个进程访问同一块内存
  - 多个进程执行同一个程序，让每个进程都拥有一份程序代码的拷贝将浪费内存空间
  - 协作进程之间共享数据
- 内存管理系统应允许对内存区域的**受控共享**访问

6.1 存储管理的需求和作用

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

6.1 存储管理的需求和作用



6.1 存储管理的需求和作用

编译

- 编译程序对源程序进行处理，得到**目标模块**
  - 如gcc –c
- 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
  - 文本段
  - 数据段
  - 堆栈段
- 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**。

6.1 存储管理的需求和作用

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**
  - 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
  - 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

6.1 存储管理的需求和作用

链接

- 未链接前，每个模块中对其它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**

6.1 存储管理的需求和作用

链接

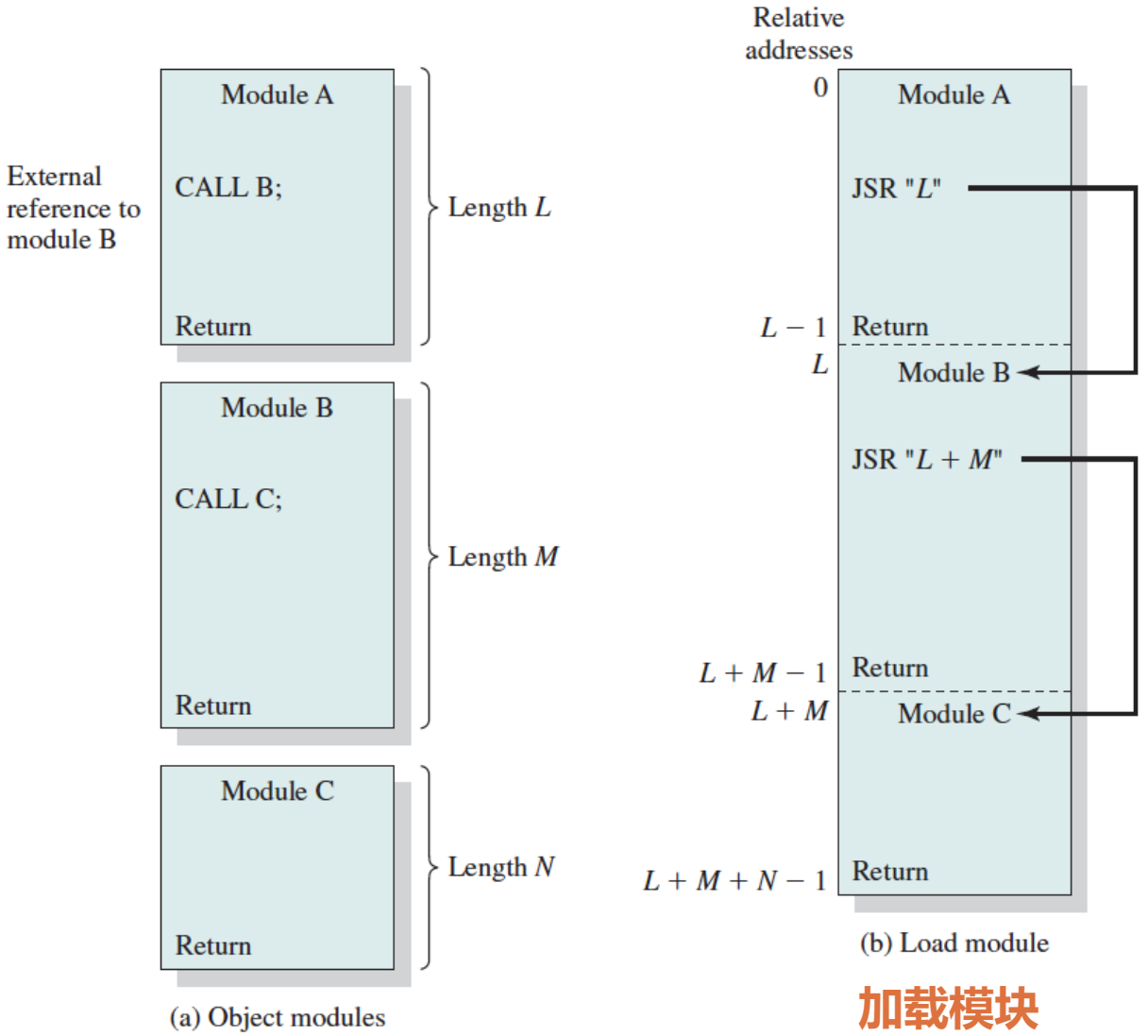


Figure 7.16 The Linking Function

目标模块

加载模块

6.1 存储管理的需求和作用

链接

- 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

6.1 存储管理的需求和作用

动态链接

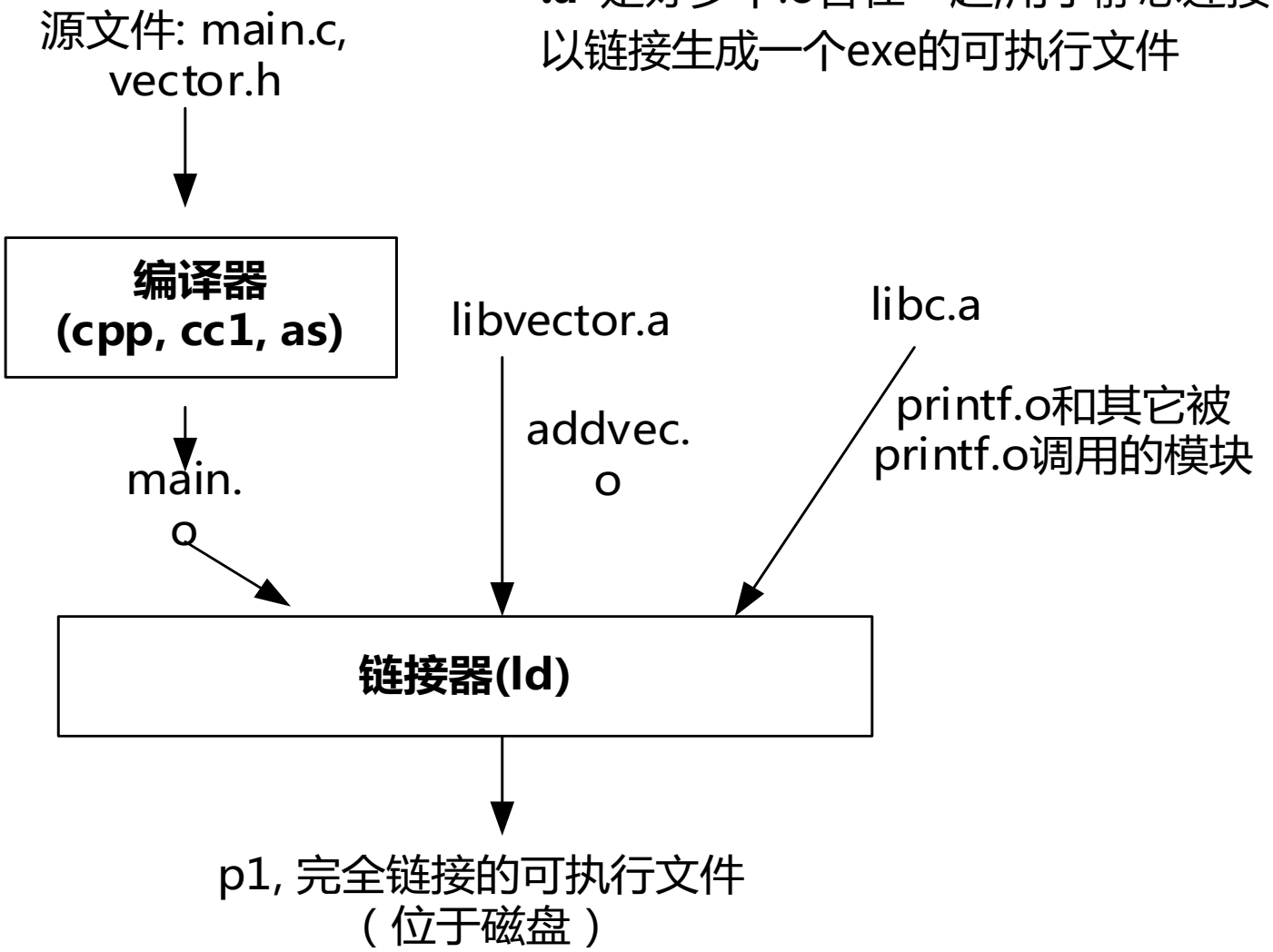
- 装入时动态链接
  - **装入时**如遇到对外部模块的引用，则**装入器找到并装入该模块，并将对模块的引用修改为该模块相对于整个应用模块起始地址的相对地址**
  - 更容易应对目标模块的改变
    - 若是静态链接，则目标模块的每次变化都要重新链接整个程序模块
- 运行时动态链接
  - 对**某些目标模块的外部引用在装入时并不进行转换**，而是保持在装入到内存的程序中
  - 当真正**运行时**执行到该模块时，**操作系统定位并装入模块，并与调用模块进行链接**
  - 这类模块通常是共享的，如Windows中的DLL

6.1 存储管理的需求和作用

静态编译和链接过程

.o 就相当于windows里的obj文件 ，一个.c或.cpp文件对应一个.o文件

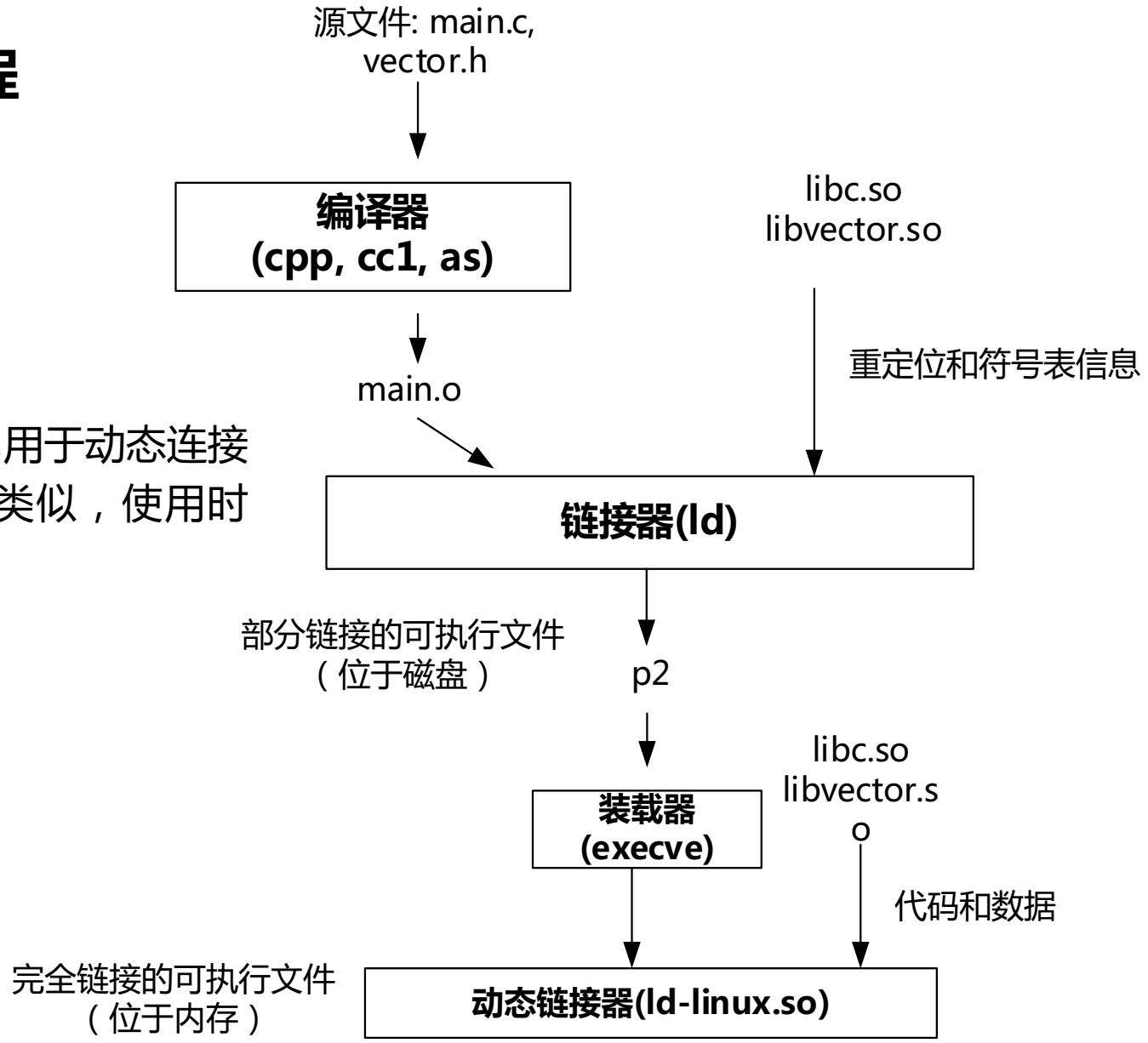
.a 是好多个.o合在一起,用于静态连接，多个.a可以链接生成一个exe的可执行文件



6.1 存储管理的需求和作用

动态链接过程

.so 是shared object,用于动态连接的，和windows的dll类似，使用时才载入。



6.1 存储管理的需求和作用

装入

- 在加载一个装载代码之前，存储管理程序将分配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，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）
  - 基于地址重定位的装入
    - 静态重定位--装入时绑定
    - **动态重定位--运行时绑定**

6.1 存储管理的需求和作用

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

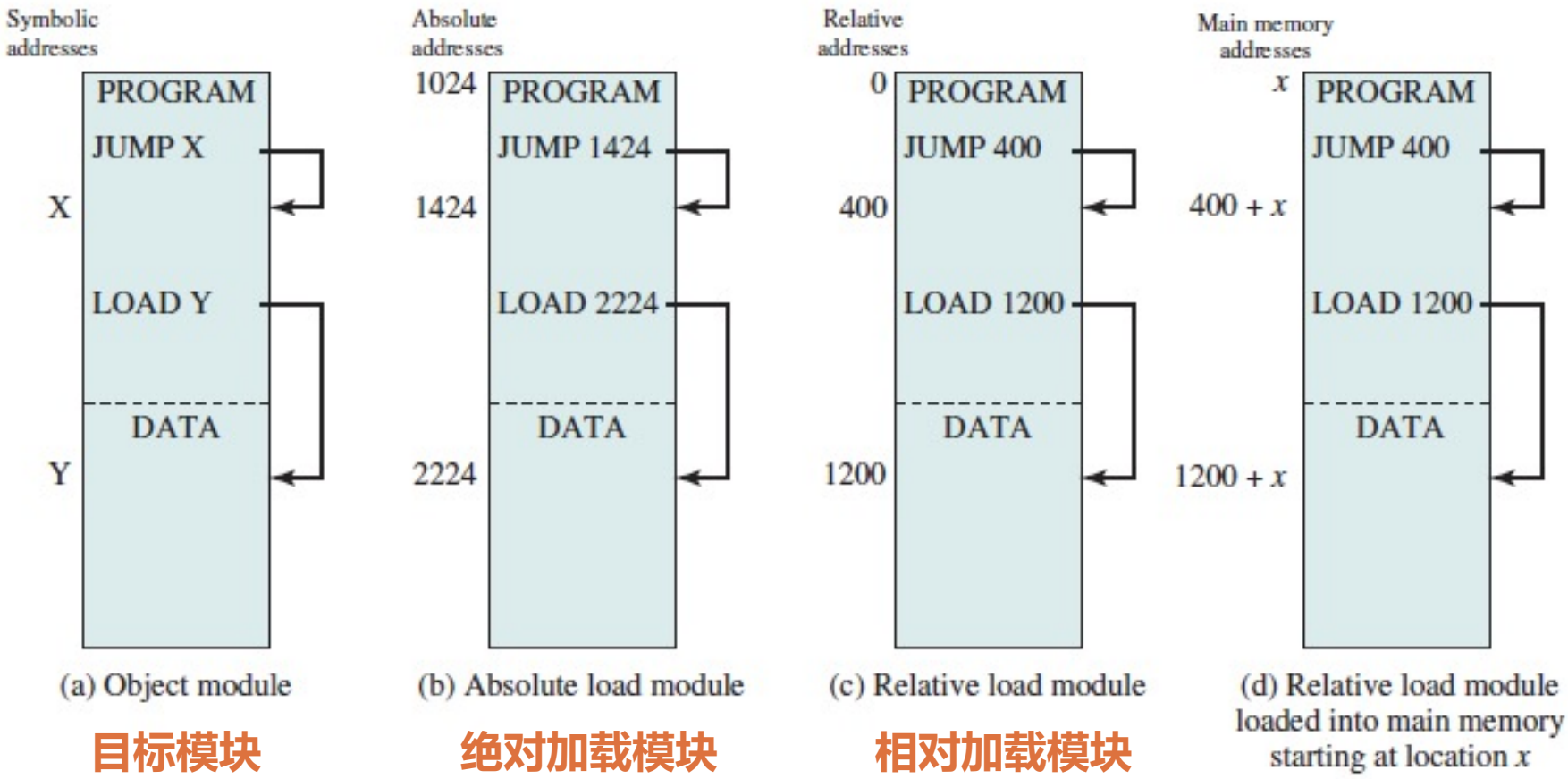
6.1 存储管理的需求和作用

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**

6.1 存储管理的需求和作用

基于地址重定位装入

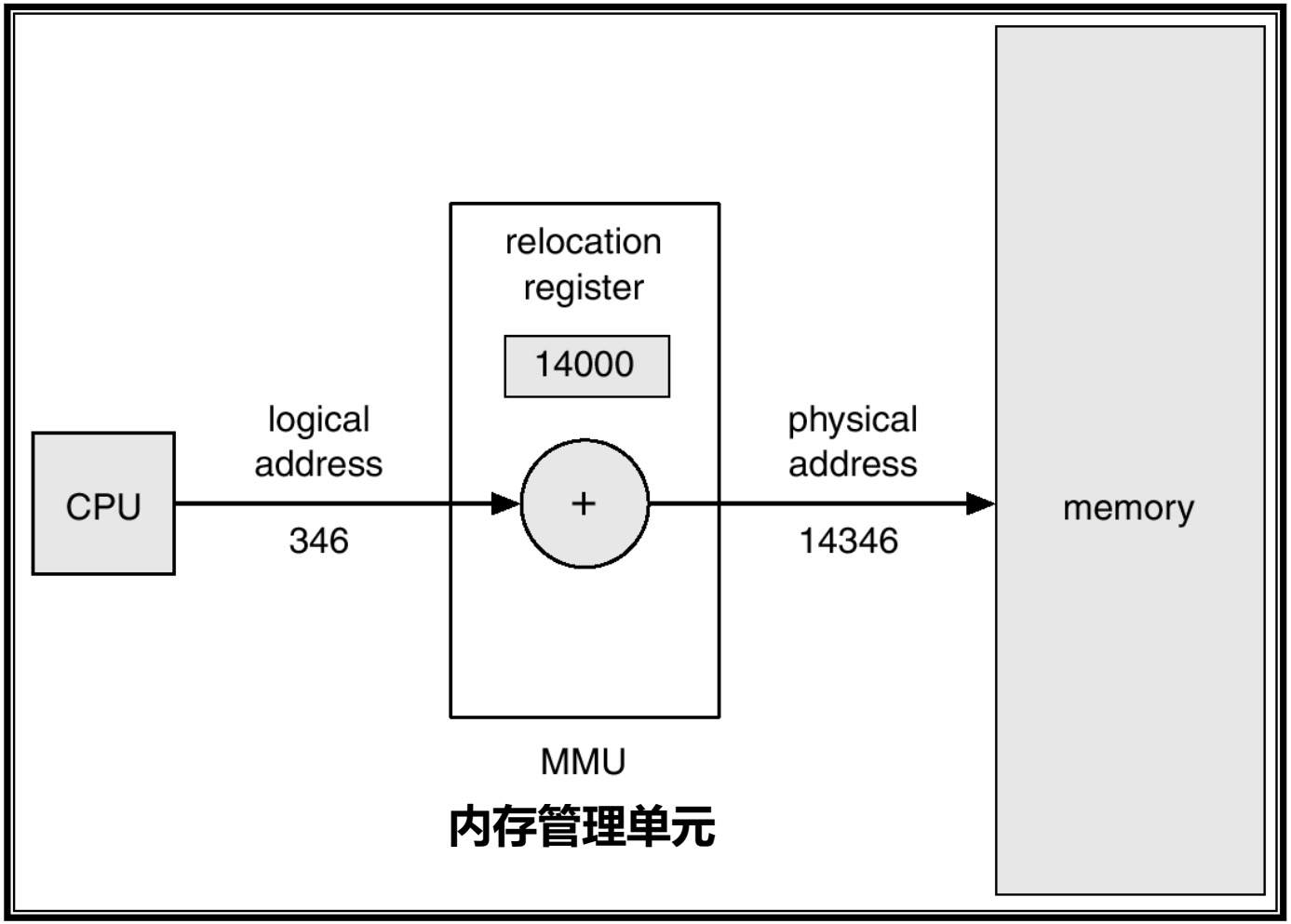


目标模块　　绝对加载模块　　相对加载模块

Figure 7.15 Absolute and Relocatable Load Modules

相对加载模块被加载到其实地址为x的内存中

6.1 存储管理的需求和作用



动态地址重定位示意

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

大纲

- 6.1 存储管理的需求和作用
- **6.2 连续空闲存储管理**
- 6.3 分页存储管理
- 6.4 分段存储管理

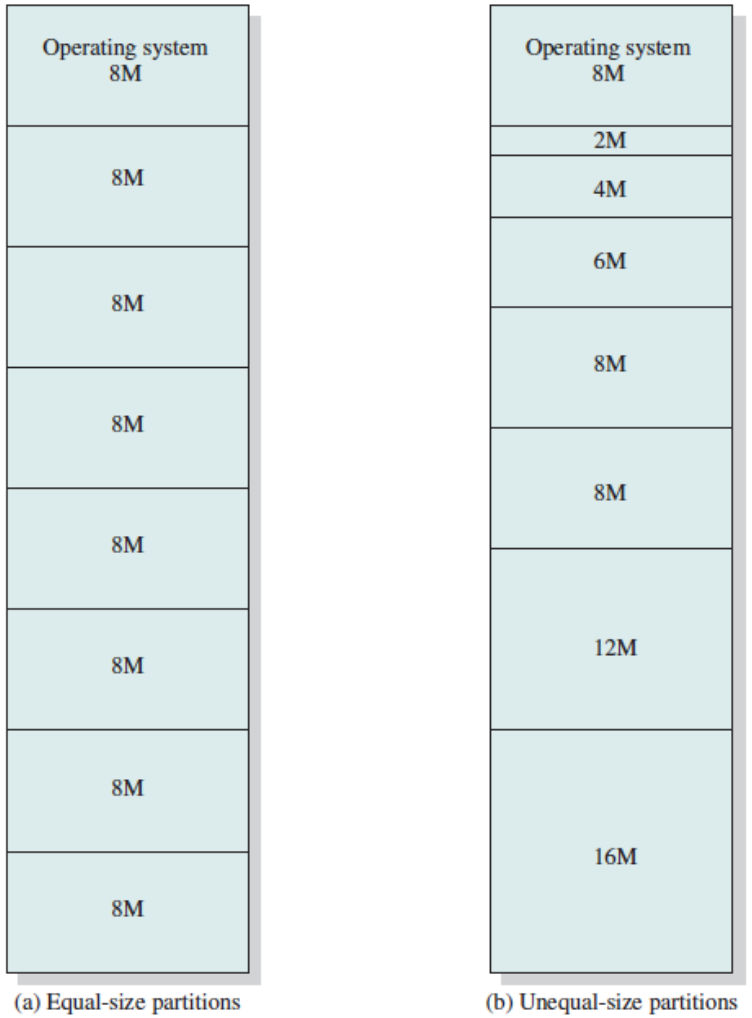
6.2 连续空闲存储管理

6.2.1 固定分区管理

- 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。

6.2 连续空闲存储管理

6.2.1 固定分区管理



(a) Equal-size partitions (b) Unequal-size partitions

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

6.2 连续空闲存储管理

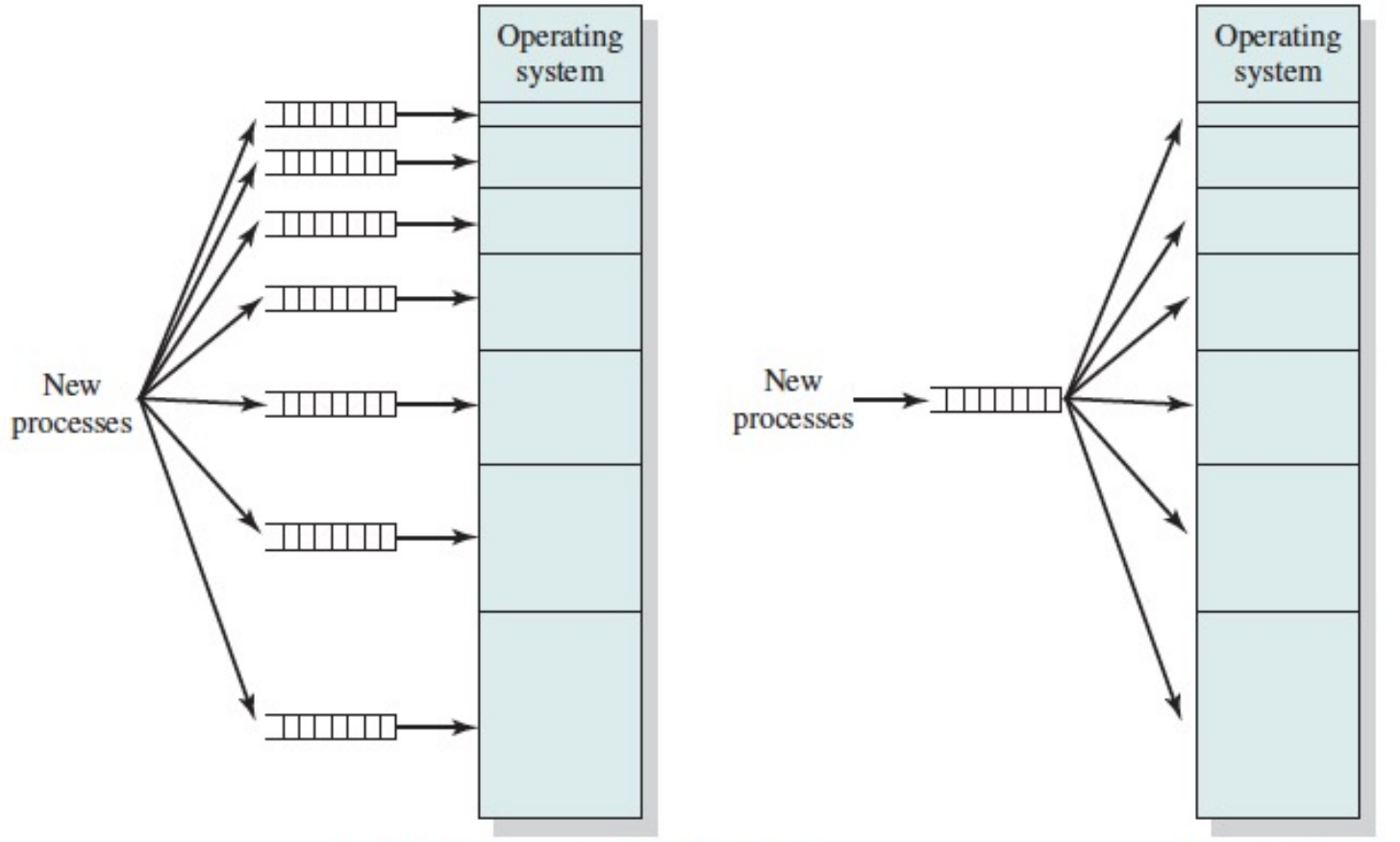
6.2.1 固定分区管理

分区分配算法

- 分区大小相同
  - 无所谓
- 分区大小不同
  - 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
    - 通常对应于每个分区都有一个单独的排队队列；
  - 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

6.2 连续空闲存储管理

6.2.1 固定分区管理



(a) One process queue per partition (b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning

6.2 连续空闲存储管理

6.2.1 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

6.2 连续空闲存储管理

6.2.1 固定分区管理

- 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**。
  - 运行过程中扩充主存困难
  - 限制了多道程序的道数

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

- **按照进程的需求为其分配恰好的内存空间**
  - 初始时，整个用户区是一个大空闲分区
  - 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**。
  - 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。

6.2 连续空闲存储管理

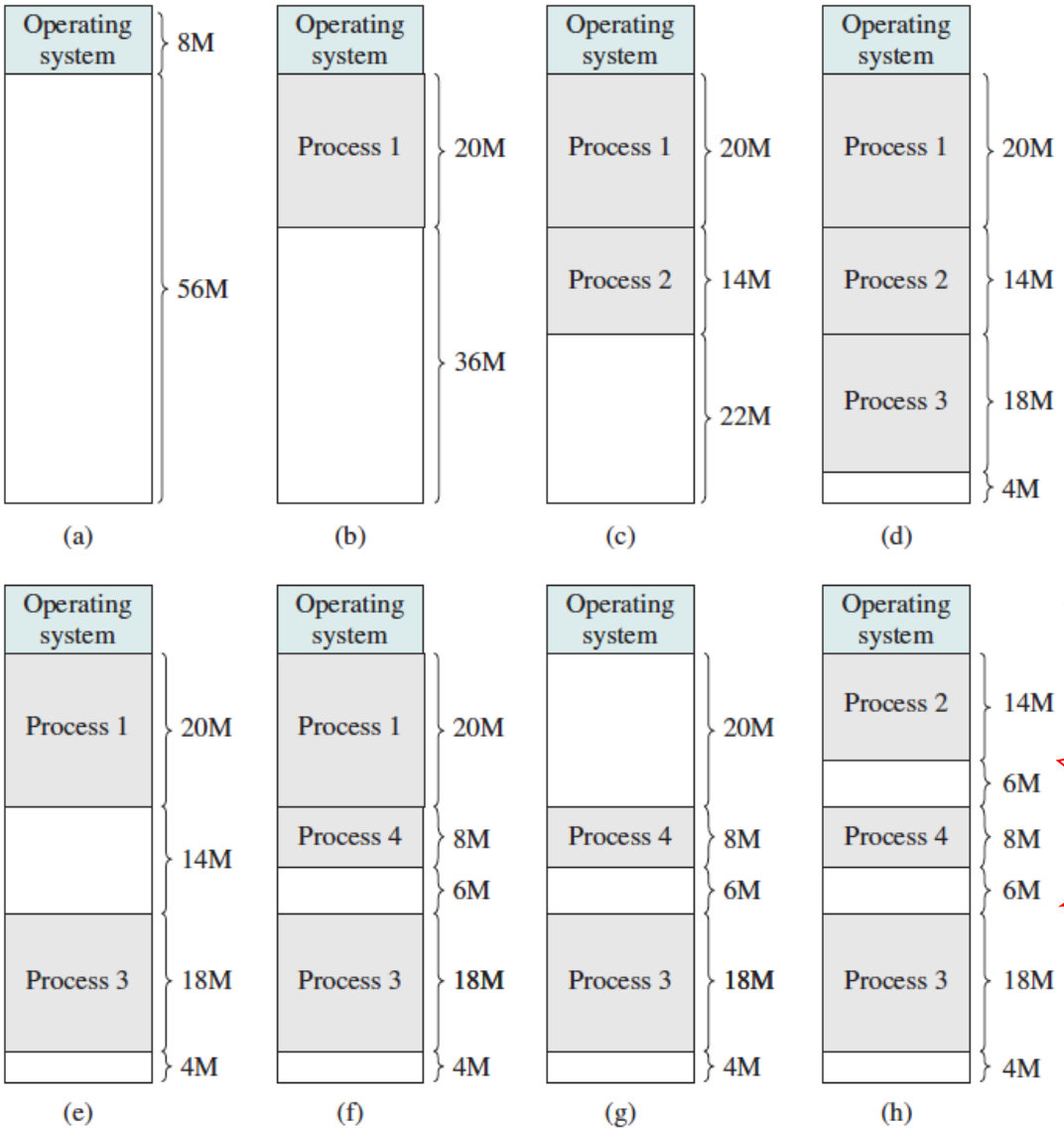


Figure 7.4 The Effect of Dynamic Partitioning

若此时进程P5进入
要求10M内存?

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

可变分区内存管理的数据结构

- 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
  - 空闲区表格管理法
  - 空闲区链表管理法

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

- **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
  - **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**

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

空闲区表格管理法

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

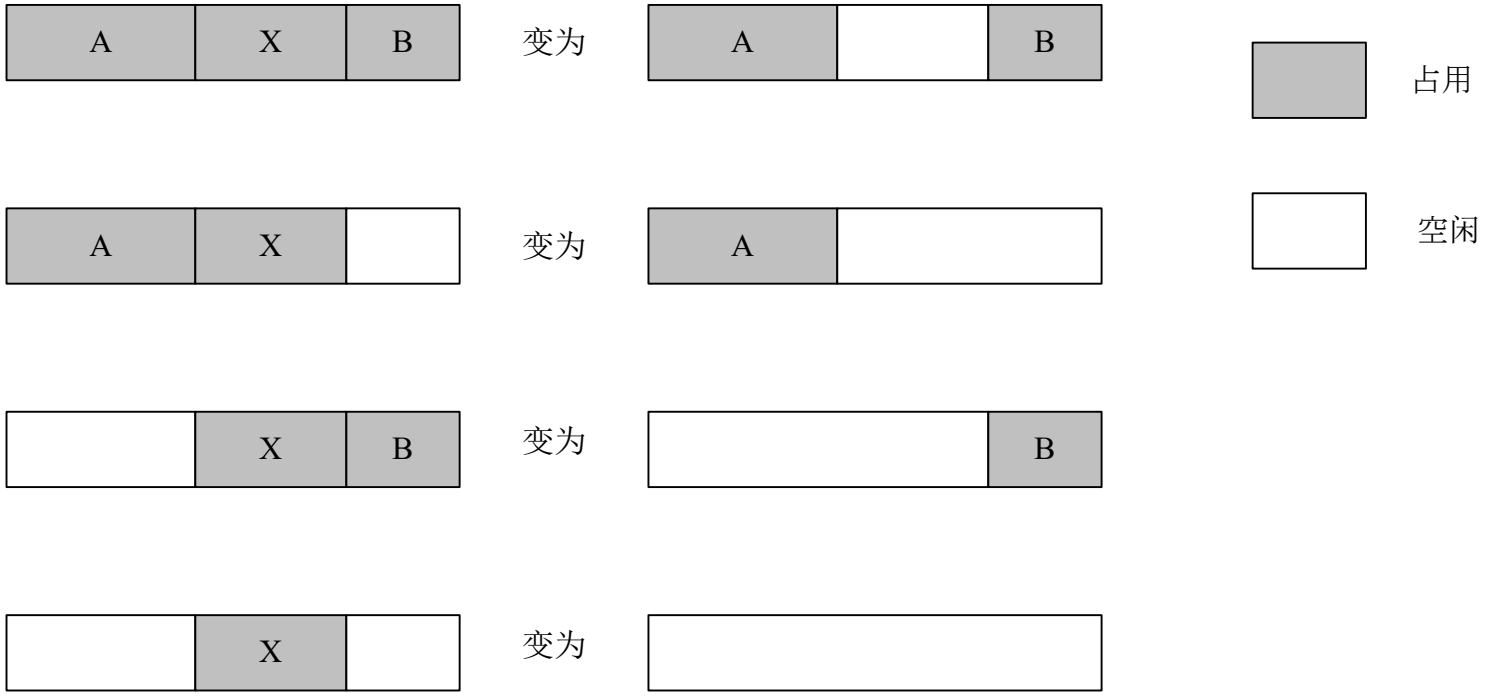
占用区表

分区号	起始地址	长度	占用标志
1	B	L2	Job1

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

空闲区表格管理法



作业X结束时，可变分区回收情况

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

空闲区链表管理法

- 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

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

- **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)

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

最先适应分配

- 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。**

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

下次适应分配算法

- 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）
- 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**。

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

最优适应分配算法

- 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

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

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**
- 可将空闲区按长度递减排列

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

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。
- 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

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

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)
  - 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。

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

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

存储保护：

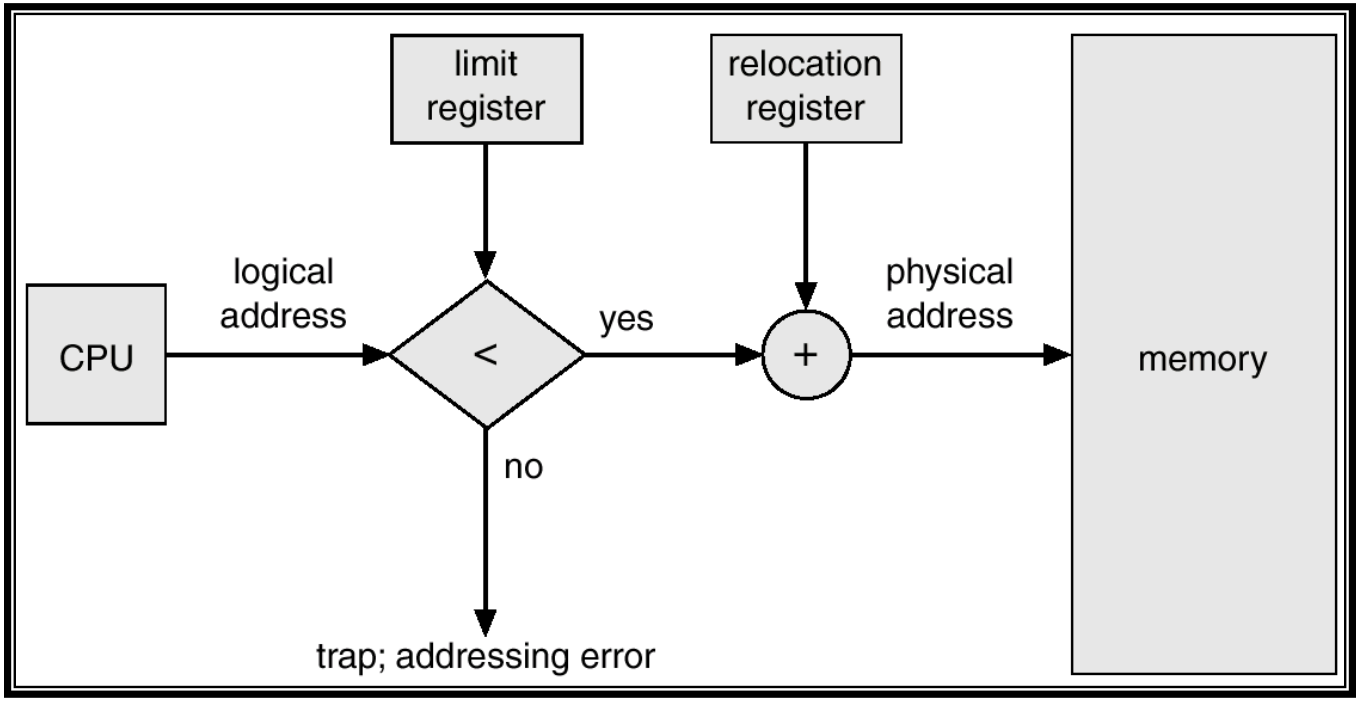
方法一：逻辑地址是否小于限长寄存器的值？

方法二：物理地址是否小于界限寄存器的值？

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

- 连续内存分配的地址转换和存储保护



基于基址/限长寄存器的存储保护

6.2 连续空闲存储管理

6.2.2 可变分区内存管理

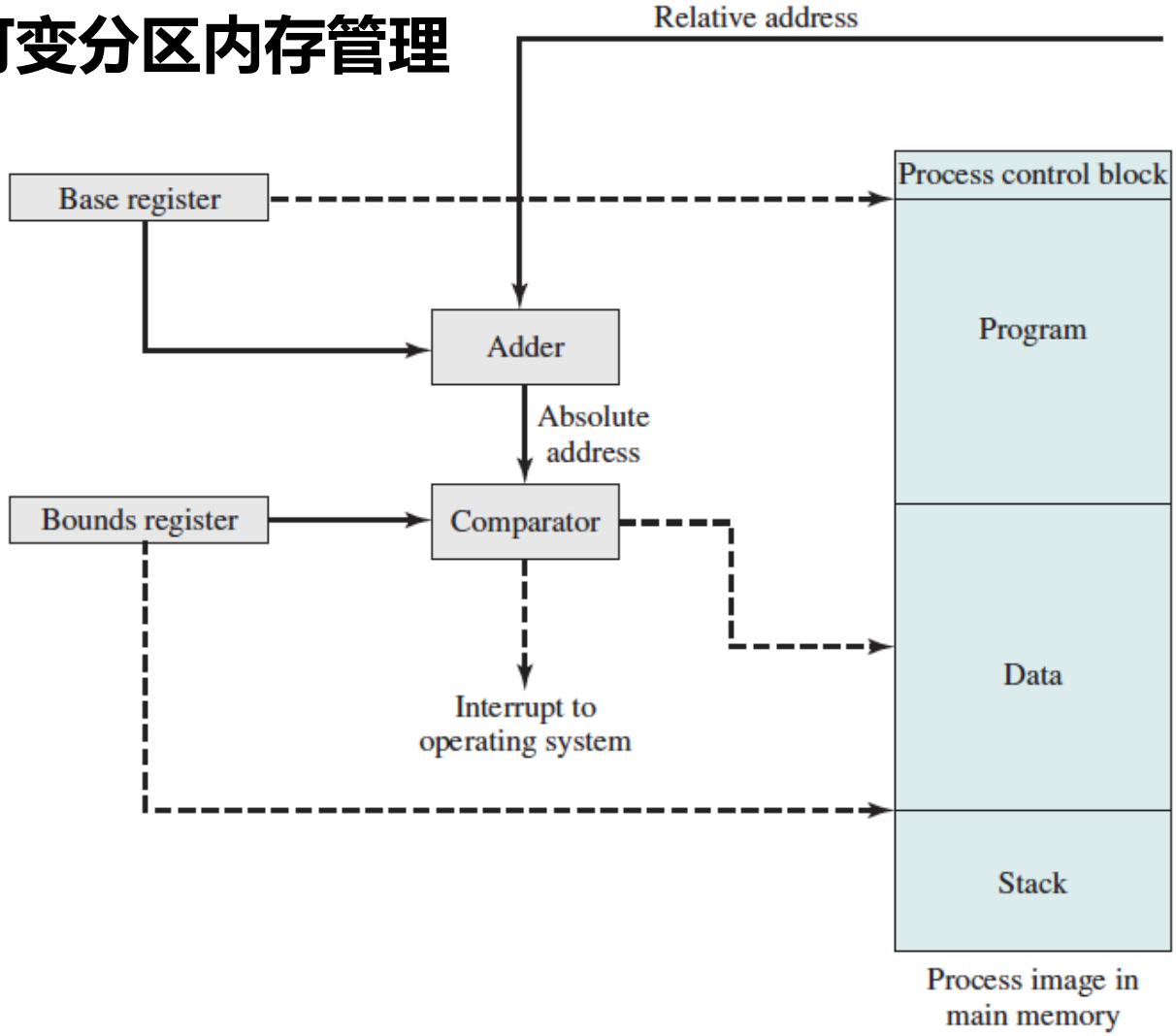


Figure 7.8 Hardware Support for Relocation

基于基址/界限寄存器的存储保护

6.2 连续空闲存储管理

6.2.3 主存不足的存储管理技术

- **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

6.2 连续空闲存储管理

6.2.3 主存不足的存储管理技术

- **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？
  - 通常系统把**时间片耗尽或优先级较低的**进程换出，因为短时间内他们不会投入运行

6.2 连续空闲存储管理

6.2.3 主存不足的存储管理技术

- **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。
  - 覆盖是指程序执行过程中程序的不同模块在内存中相互替代，以达到小内存执行大程序的目的。
  - **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。

大纲

- 6.1 存储管理的需求和作用
- 6.2 连续空闲存储管理
- **6.3 分页存储管理**
- 6.4 分段存储管理

6.3 分页存储管理

6.3.1 分页存储管理的原理

- 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

6.3 分页存储管理

6.3.1 分页存储管理的原理

- 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

6.3 分页存储管理

6.3.1 分页存储管理的原理

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

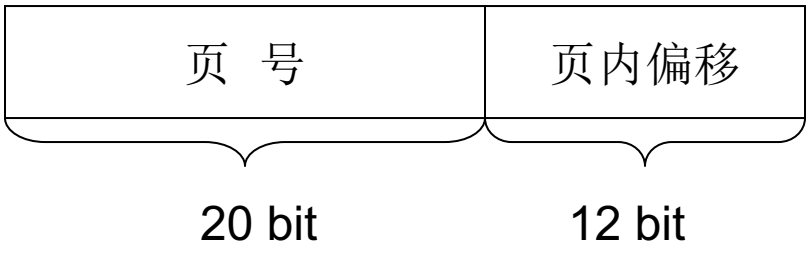
6.3 分页存储管理

6.3.1 分页存储管理的原理

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.

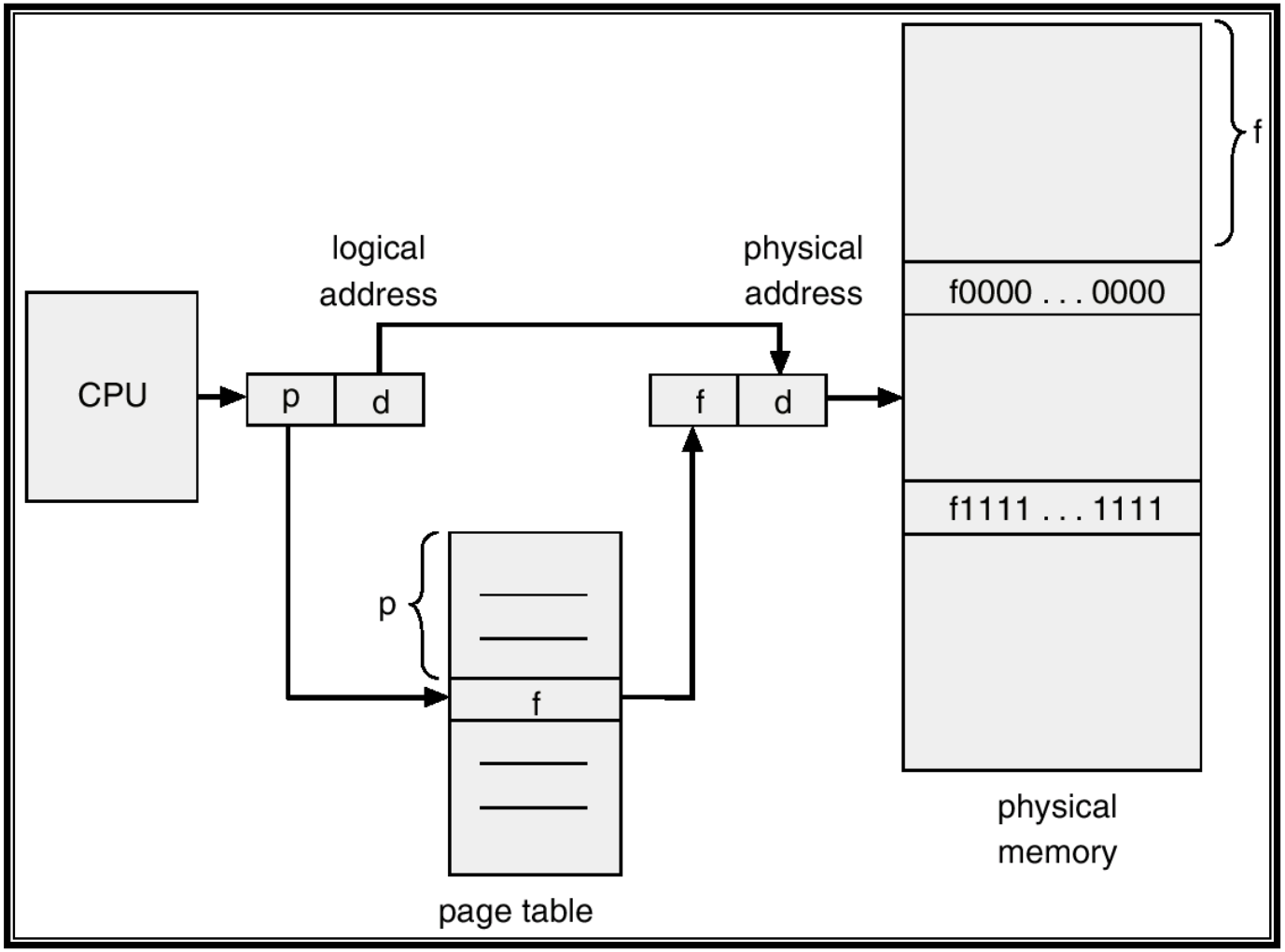
物理地址 = 页框号×页面大小+页内偏移



32位逻辑地址示意图

6.3 分页存储管理

6.3.1 分页存储管理的原理



6.3 分页存储管理

6.3.1 分页存储管理的原理

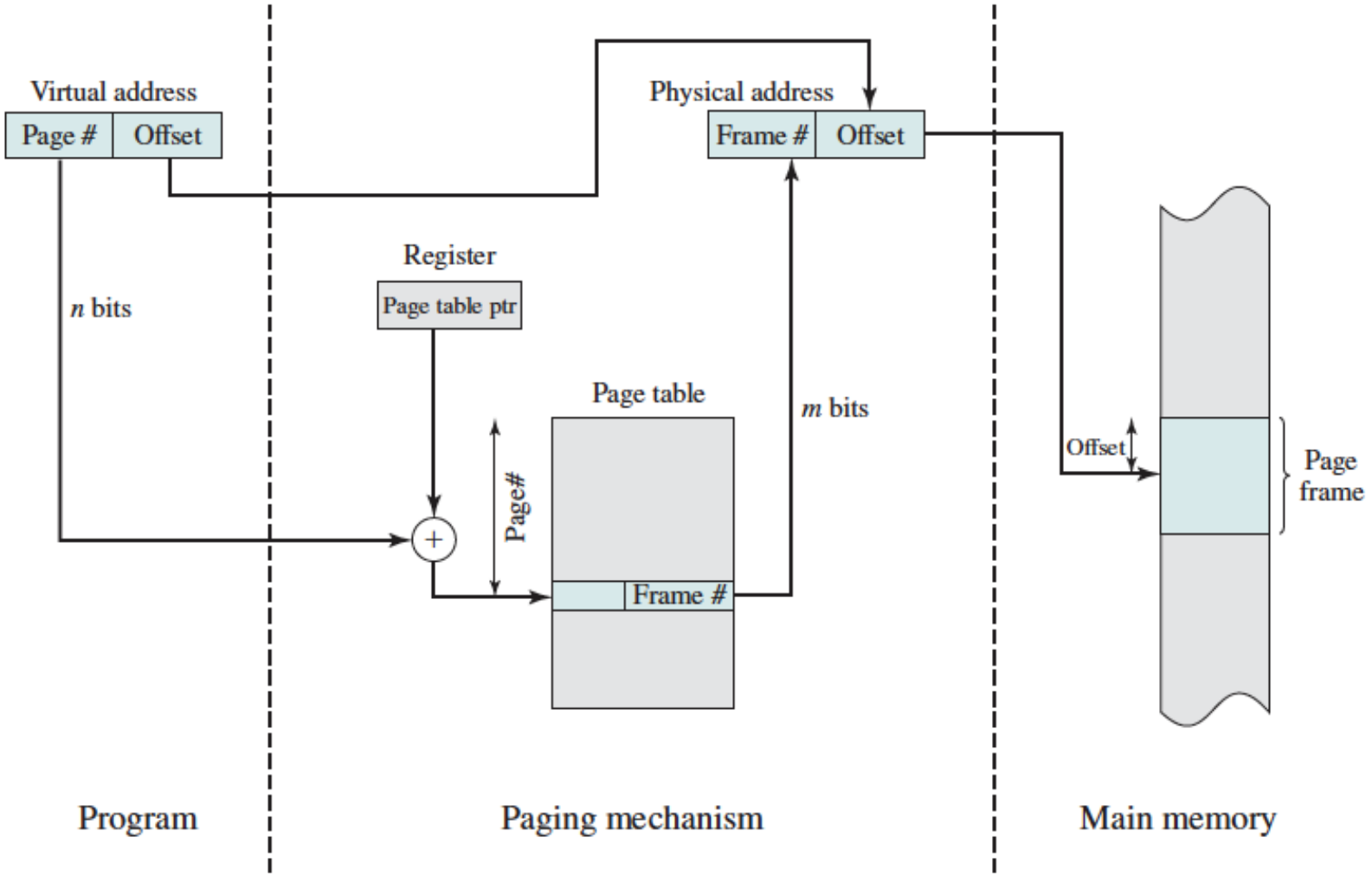


Figure 8.2 Address Translation in a Paging System

6.3 分页存储管理

6.3.1 分页存储管理的原理

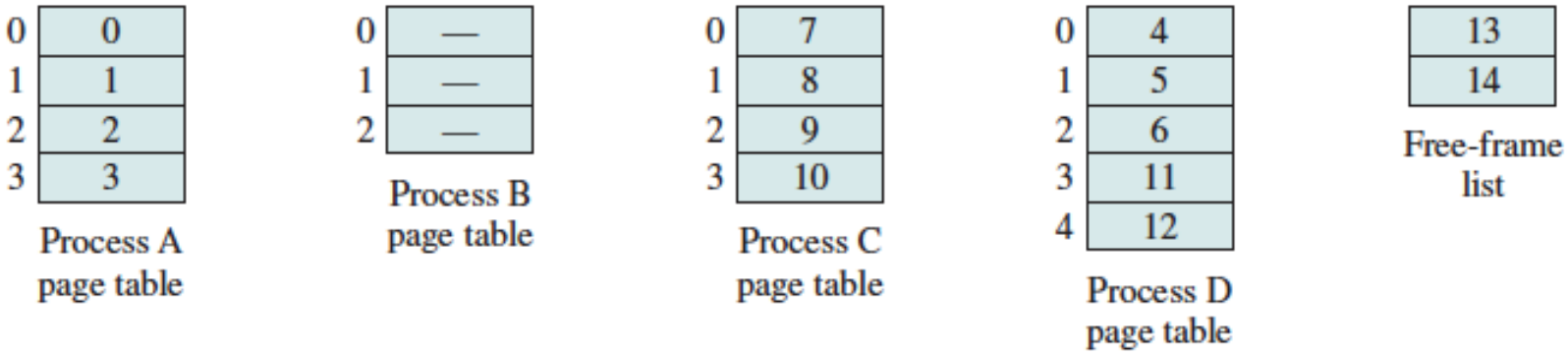
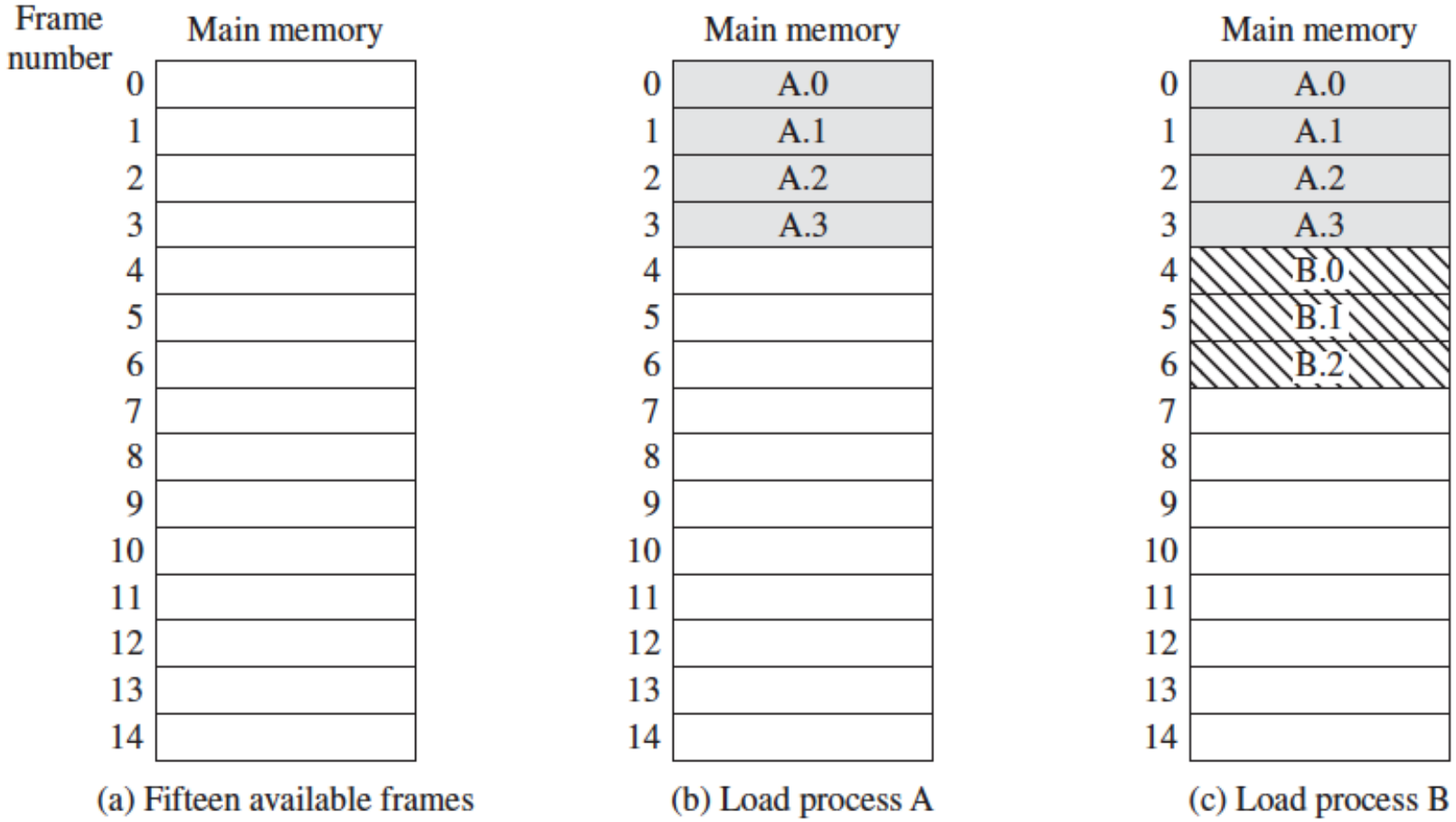


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

6.3 分页存储管理

6.3.1 分页存储管理的原理

为进程的页面分配空闲页框



(a) Fifteen available frames　(b) Load process A　(c) Load process B

6.3 分页存储管理

6.3.1 分页存储管理的原理

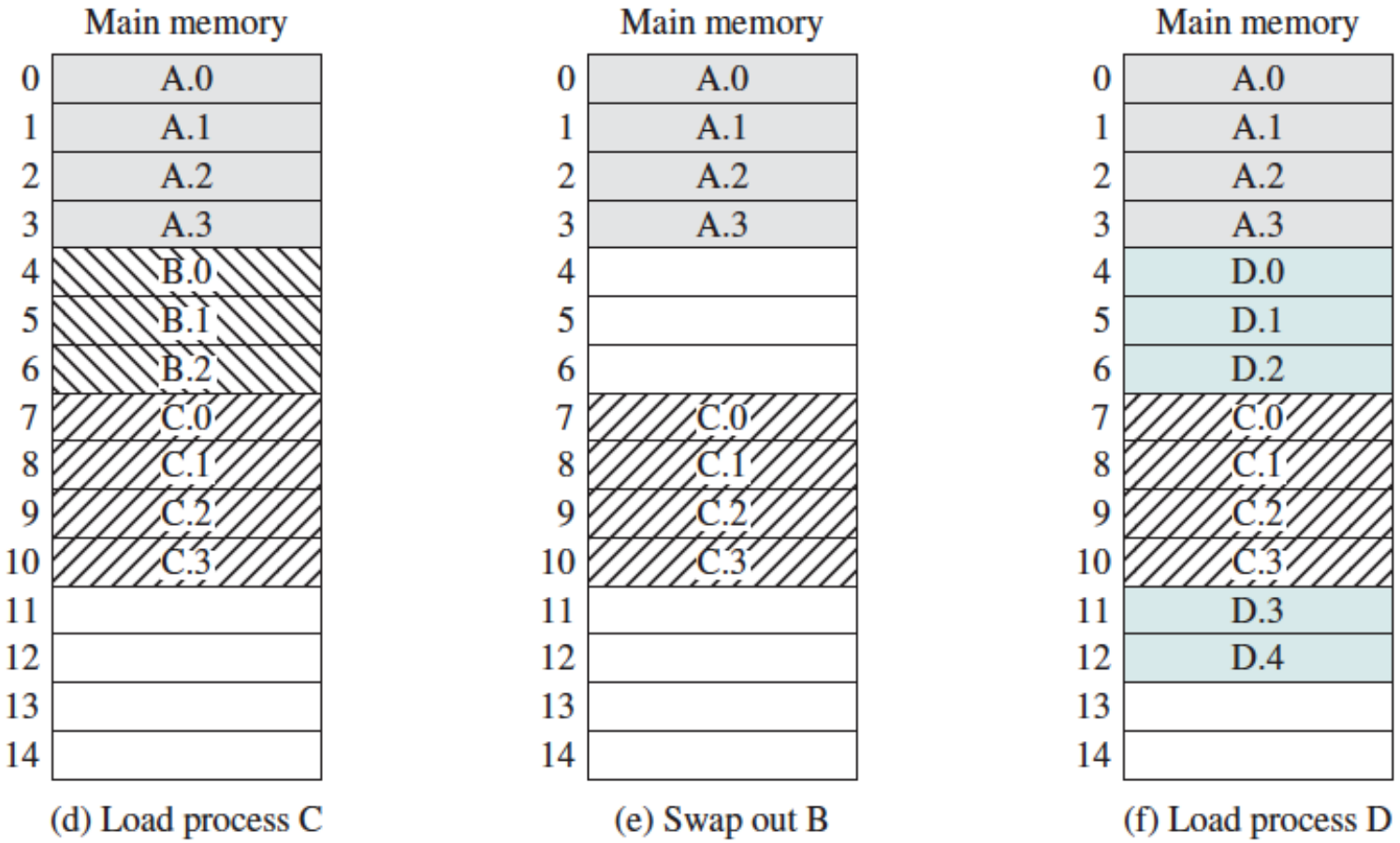


Figure 7.9 Assignment of Process to Free Frames

6.3 分页存储管理

6.3.1 分页存储管理的原理

页表的表现

- 页表一般保存在内存中
- 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
- 页表长度寄存器 (PRLR) 给出页表的大小
- 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

6.3 分页存储管理

6.3.2 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 #

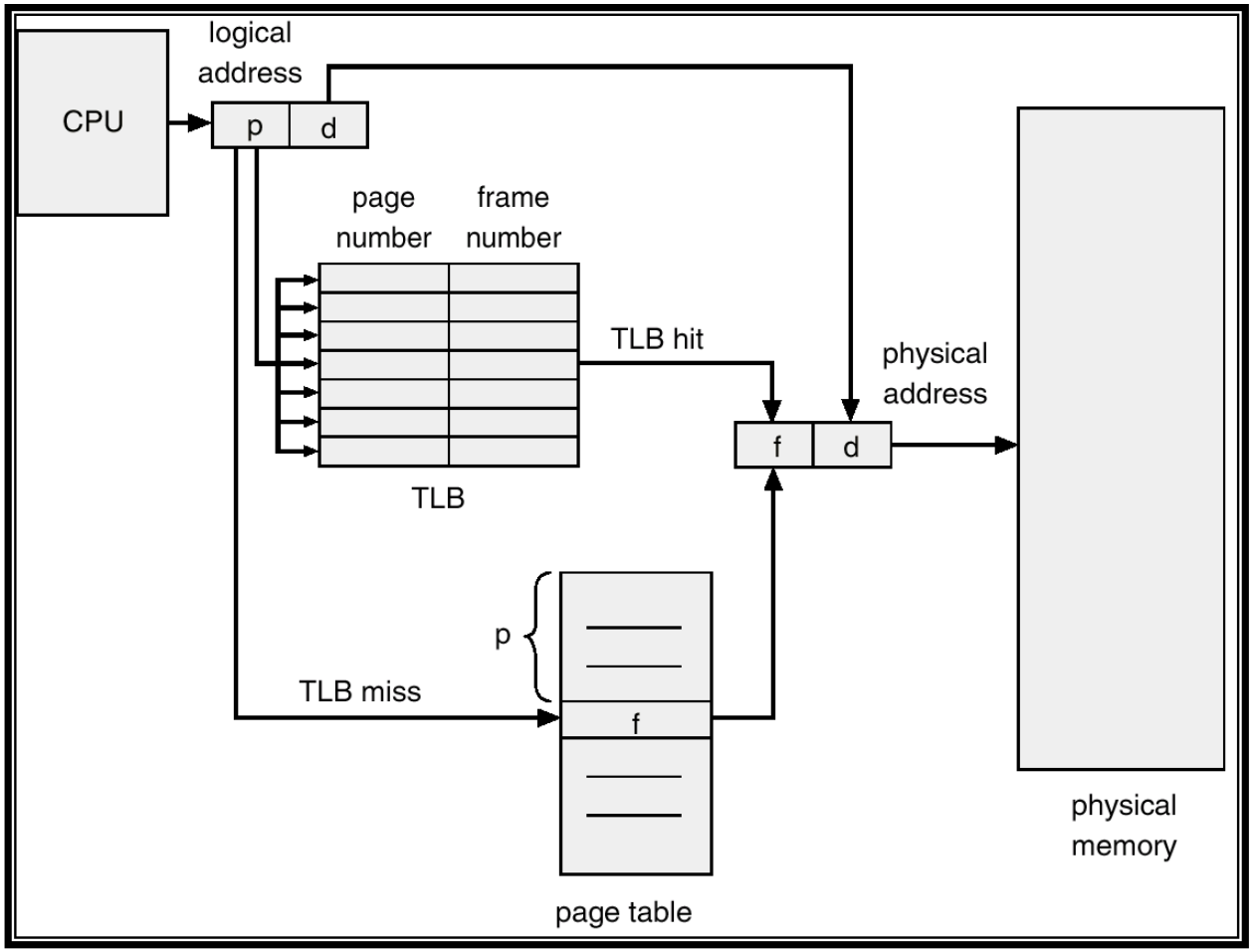
地址转换过程 (A´, A´´)

- 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 #

6.3 分页存储管理

6.3.2 相联存储器/快表(TLB)

具有TLB的页面转换机制



6.3 分页存储管理

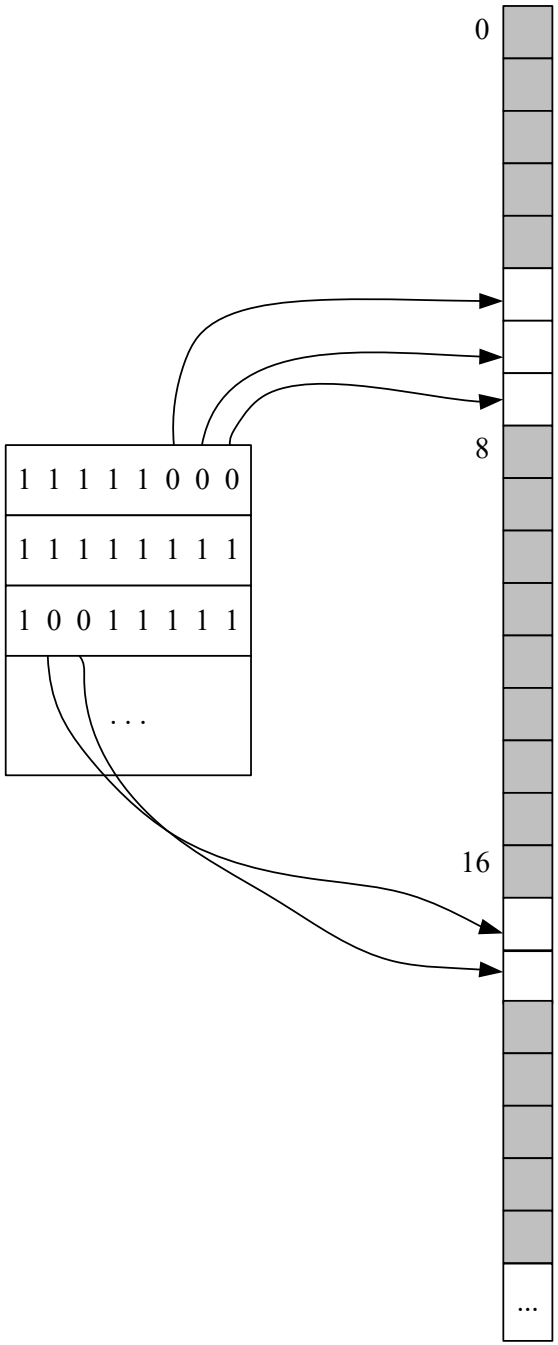
6.3.2 相联存储器/快表(TLB)

- 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。

6.3 分页存储管理

6.3.3 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
- 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。



6.3 分页存储管理

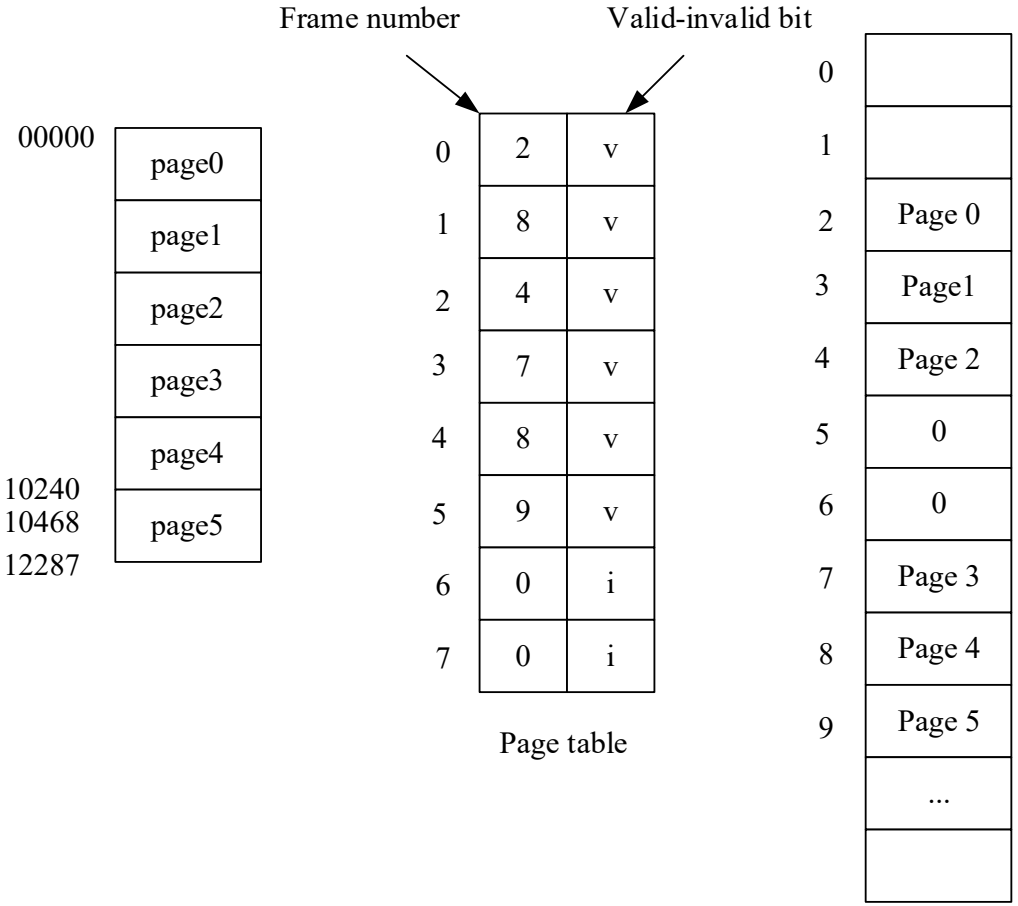
6.3.4 分页存储的存储保护

- 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。
- 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。

6.3 分页存储管理

6.3.4 分页存储的存储保护

- 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。



6.3 分页存储管理

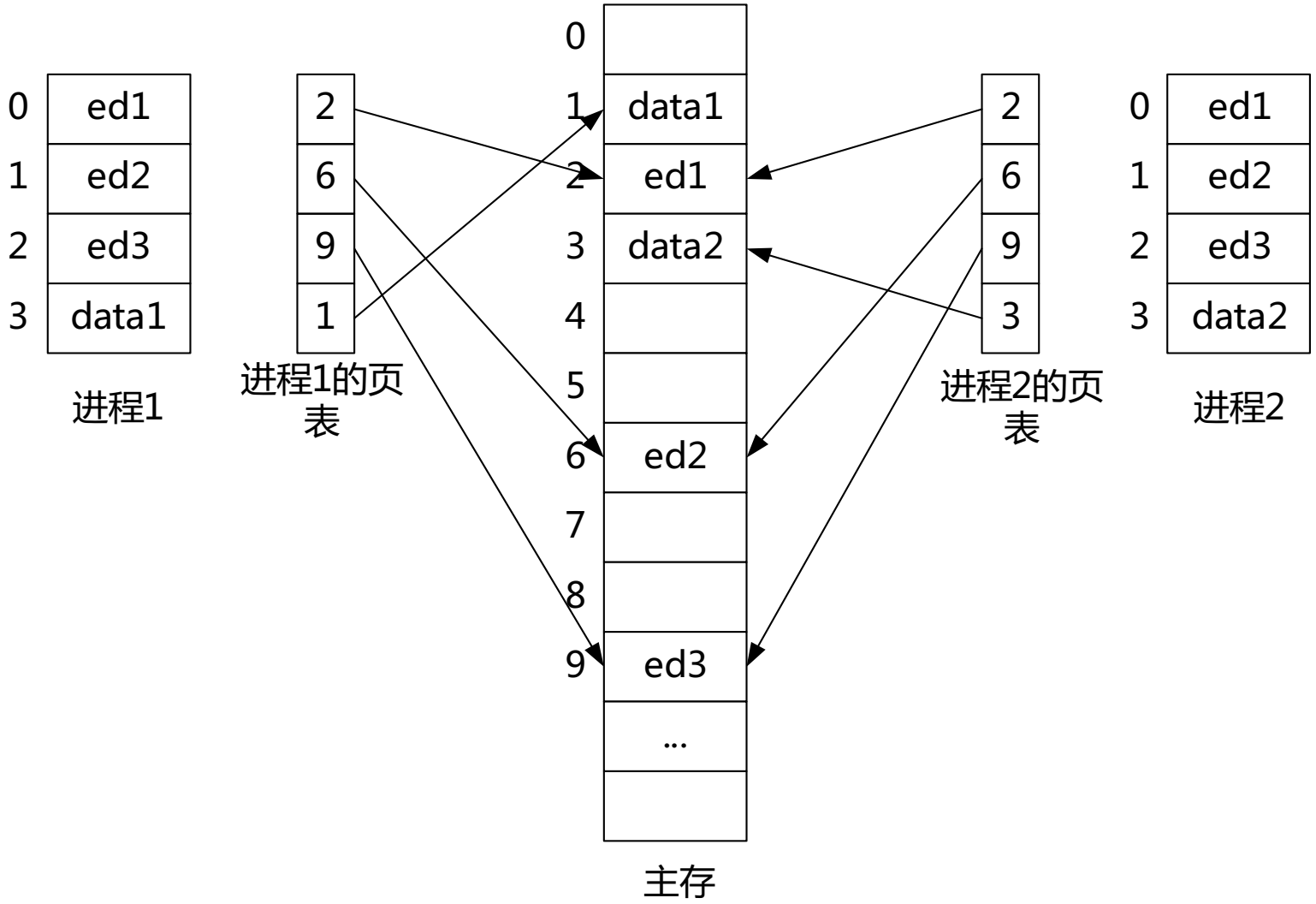
6.3.4 分页存储的存储共享

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

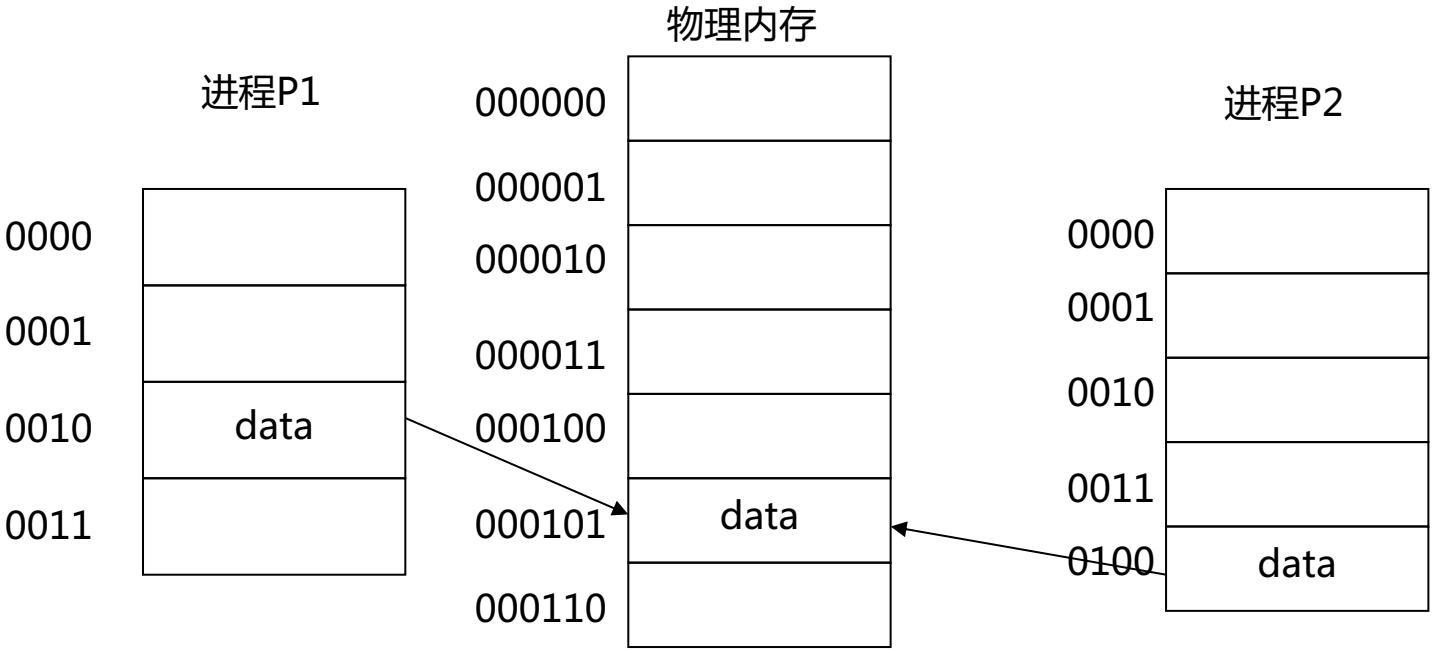
6.3 分页存储管理

6.3.4 分页存储的存储共享



6.3 分页存储管理

6.3.4 分页存储的存储共享



数据页共享及地址转换

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 0000000000-0010 1111111111

进程P2的数据页逻辑地址为0100 0000000000-0100 1111111111

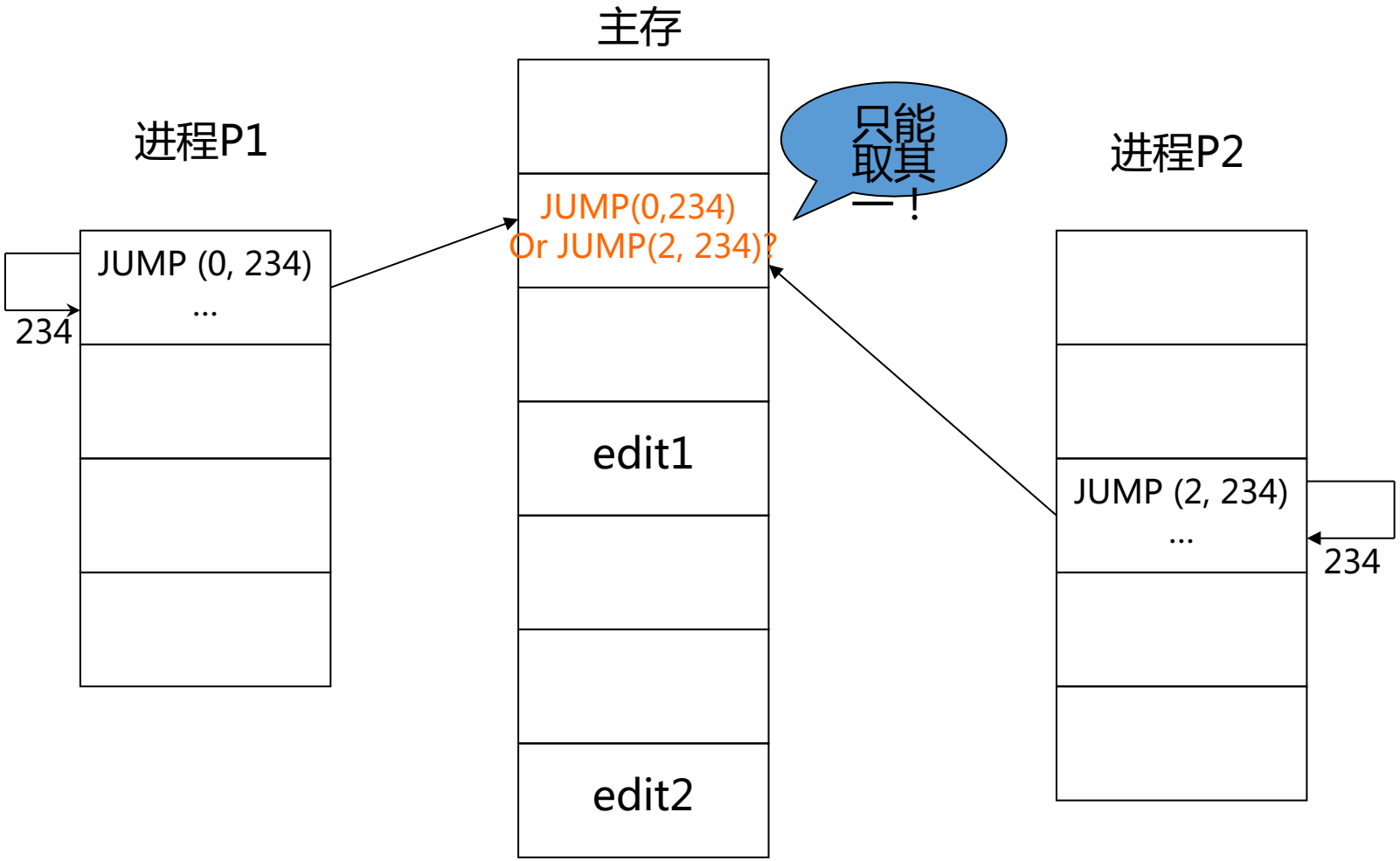
地址转换：

进程P1 **0010** 0101011000 → **000101** 0101011000

进程P2 **0100** 0101011000 → **000101** 0101011000

6.3 分页存储管理

6.3.4 分页存储的存储共享



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

6.3 分页存储管理

6.3.4 页表的组织

- 层次化页表/多级页表
- 哈希页表
- 反置页表

6.3 分页存储管理

6.3.4 页表的组织

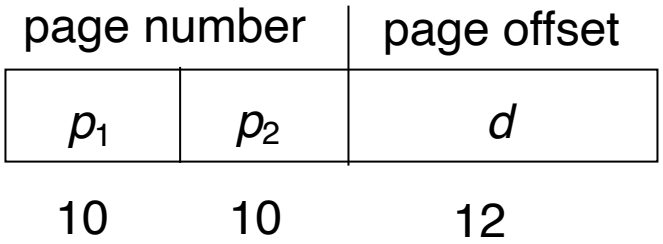
层次化多级页表

- 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

6.3 分页存储管理

6.3.4 页表的组织

- **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）
- 因此，逻辑地址被表示为:

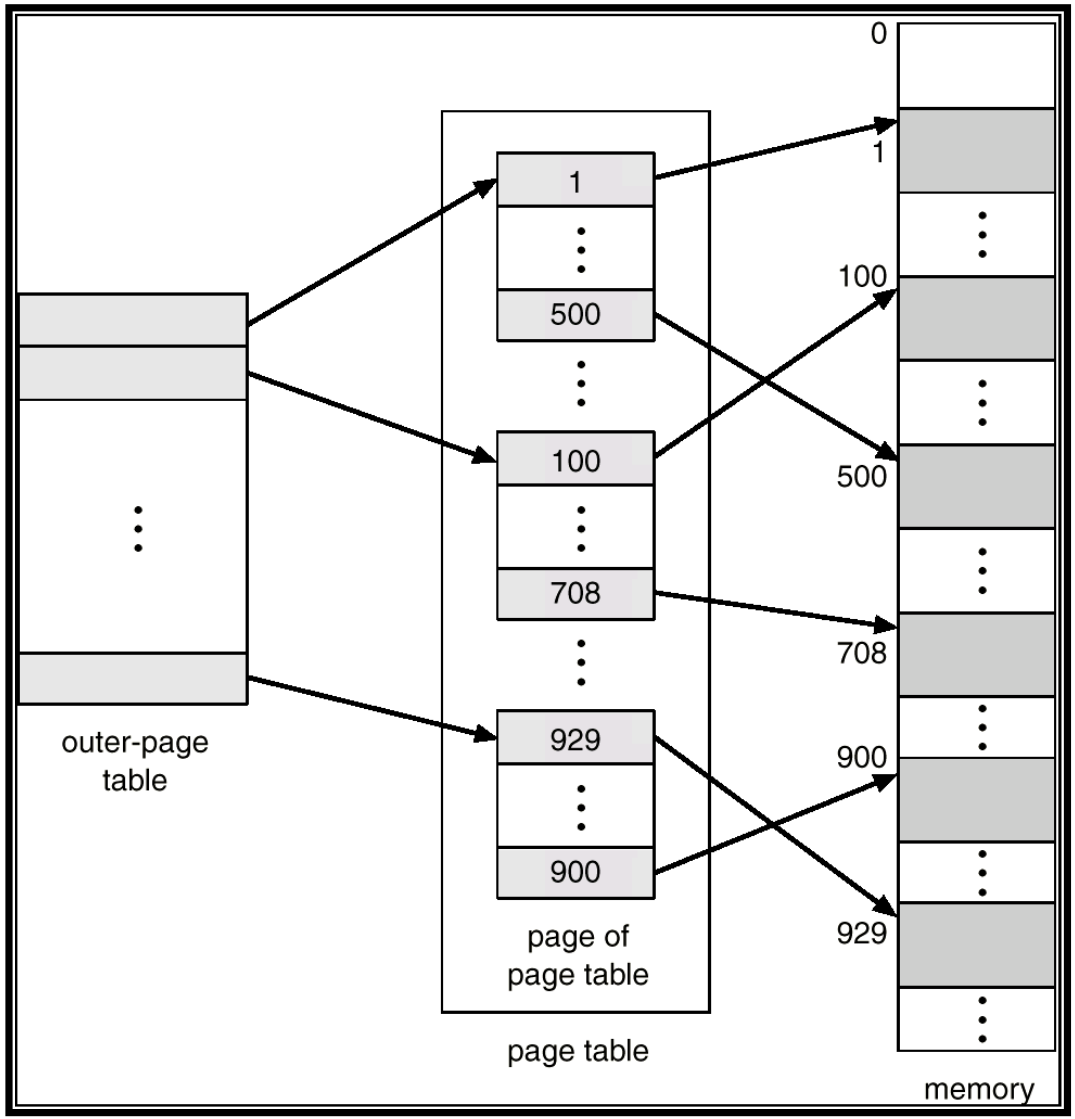


其中，p_1 用于查找一级页表, p_2 用于查找二级页表.

6.3 分页存储管理

6.3.4 页表的组织

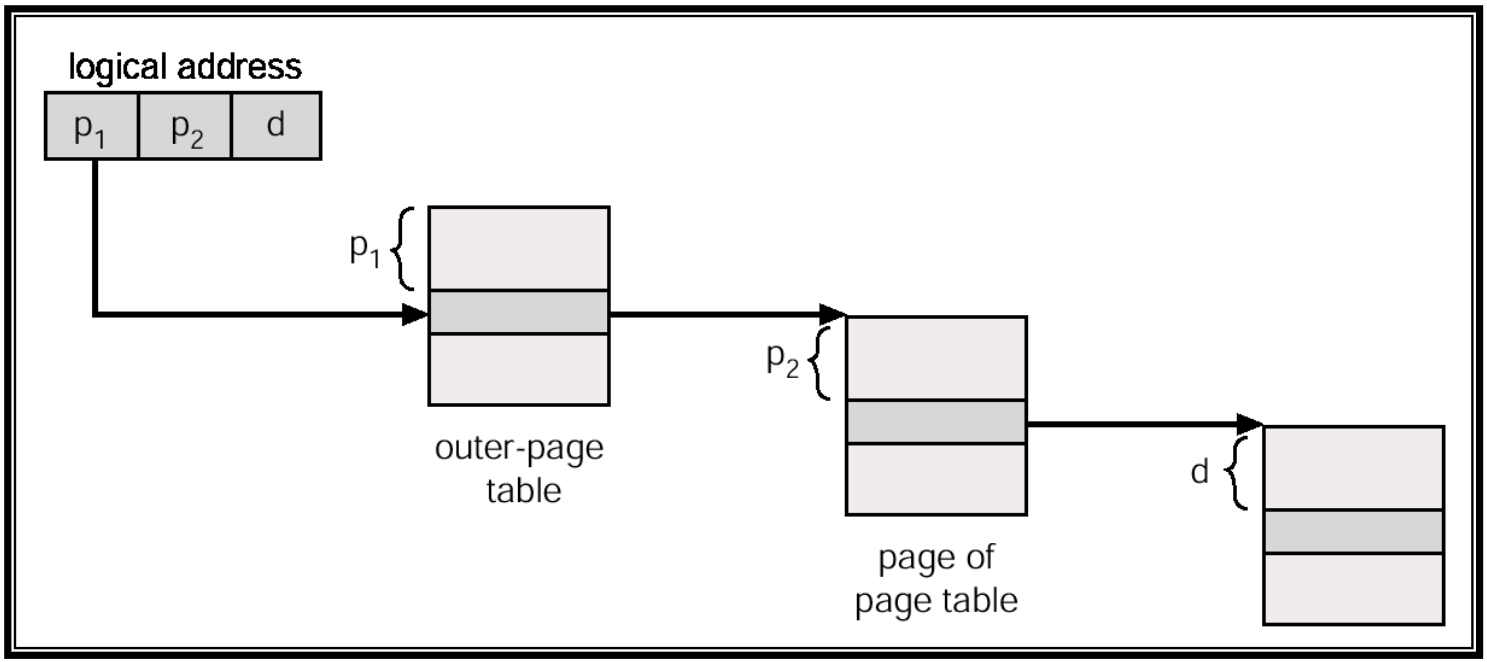
二级页表



6.3 分页存储管理

6.3.4 页表的组织

二级页表的地址转换



访问数据或指令需要三次内存访问

6.3 分页存储管理

6.3.4 页表的组织

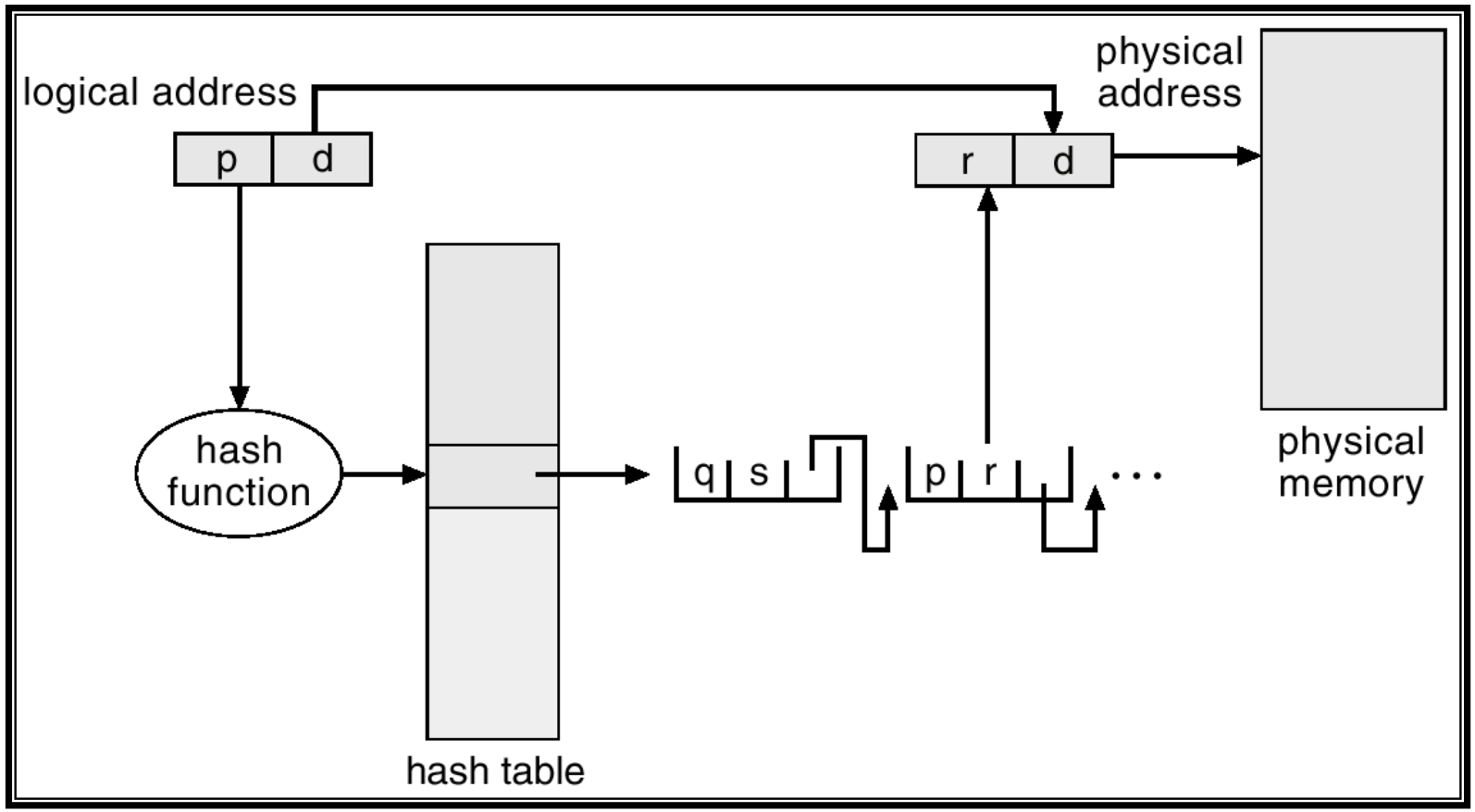
哈希页表

- 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
  - 对应的页框号
  - 指向下一个链表元素的指针

6.3 分页存储管理

6.3.4 页表的组织

哈希页表



6.3 分页存储管理

6.3.4 页表的组织

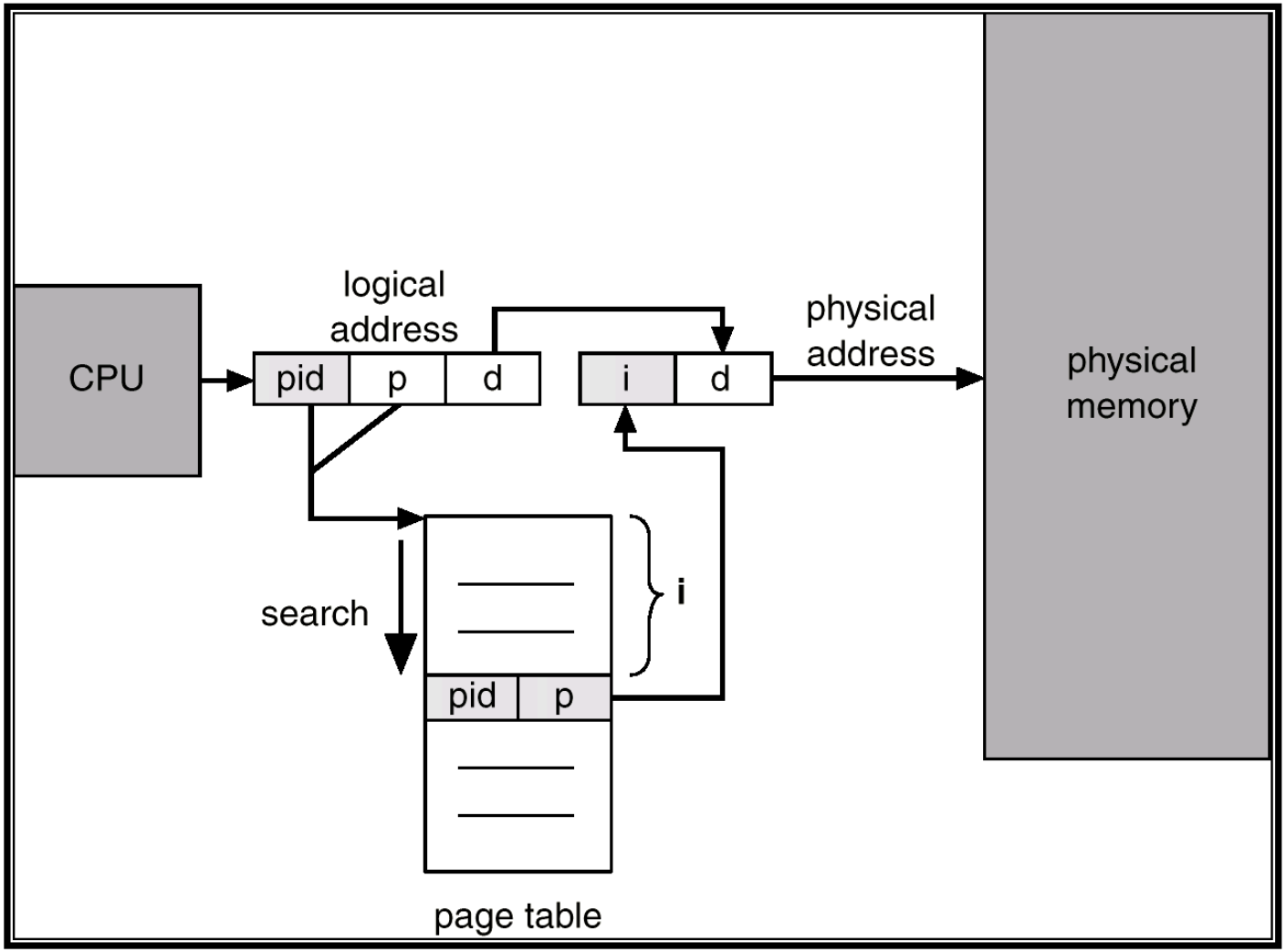
反置页表

- 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。

6.3 分页存储管理

6.3.4 页表的组织

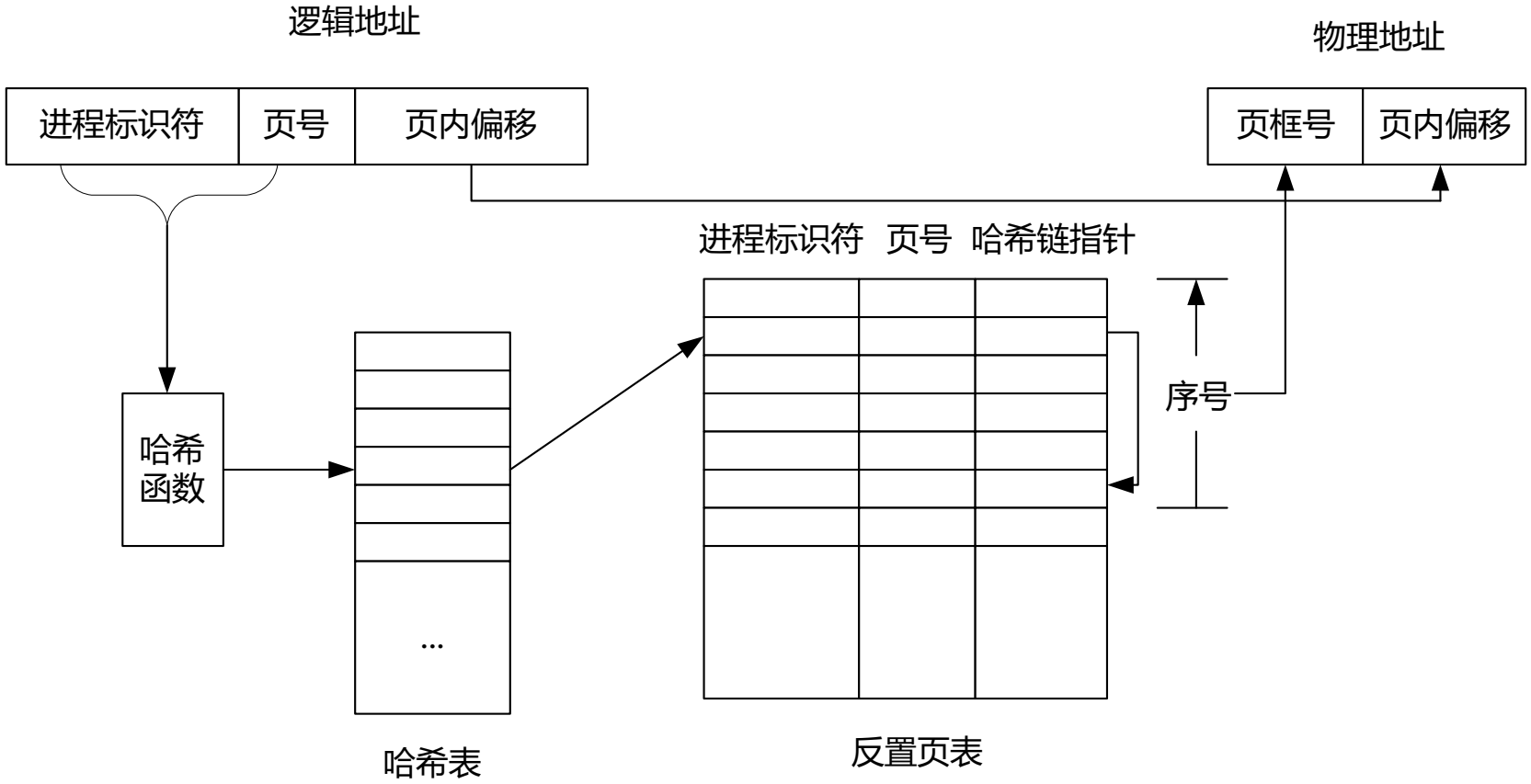
反置页表



6.3 分页存储管理

6.3.4 页表的组织

反置页表



基于哈希表的反置页表实现

哈希表和反置页表的哈希链指针更新较复杂

大纲

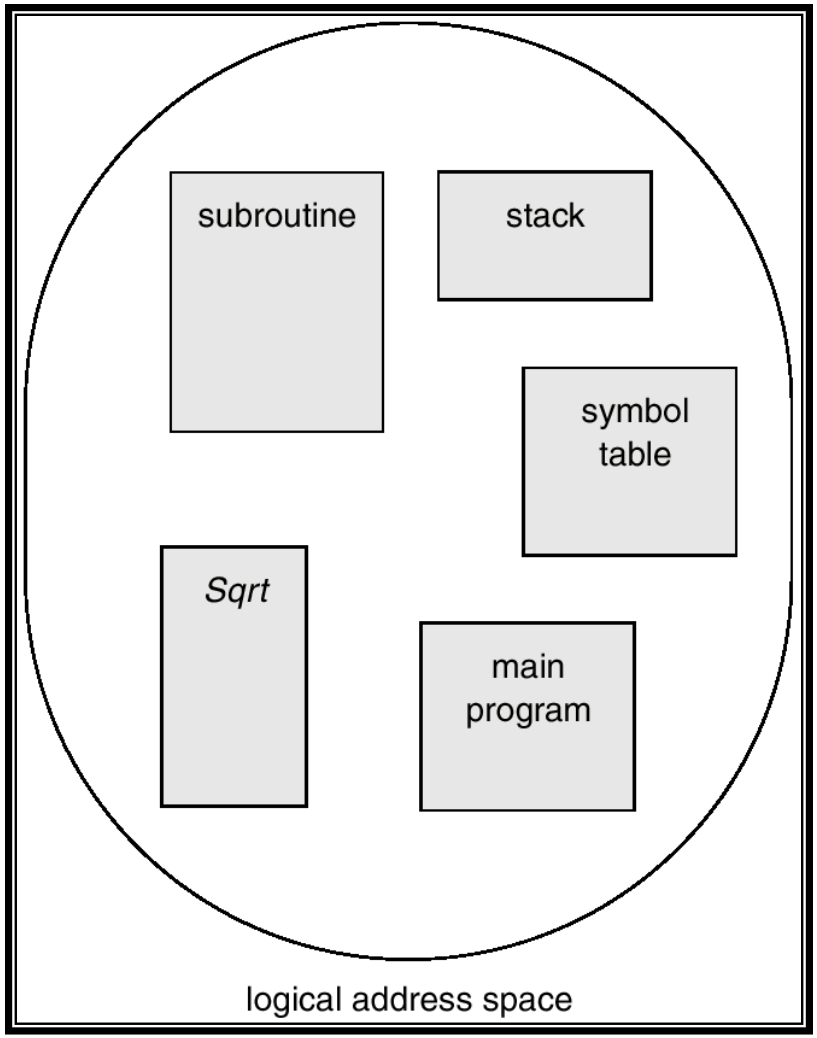
- 6.1 存储管理的需求和作用
- 6.2 连续空闲存储管理
- 6.3 分页存储管理
- **6.4 分段存储管理**

6.4 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

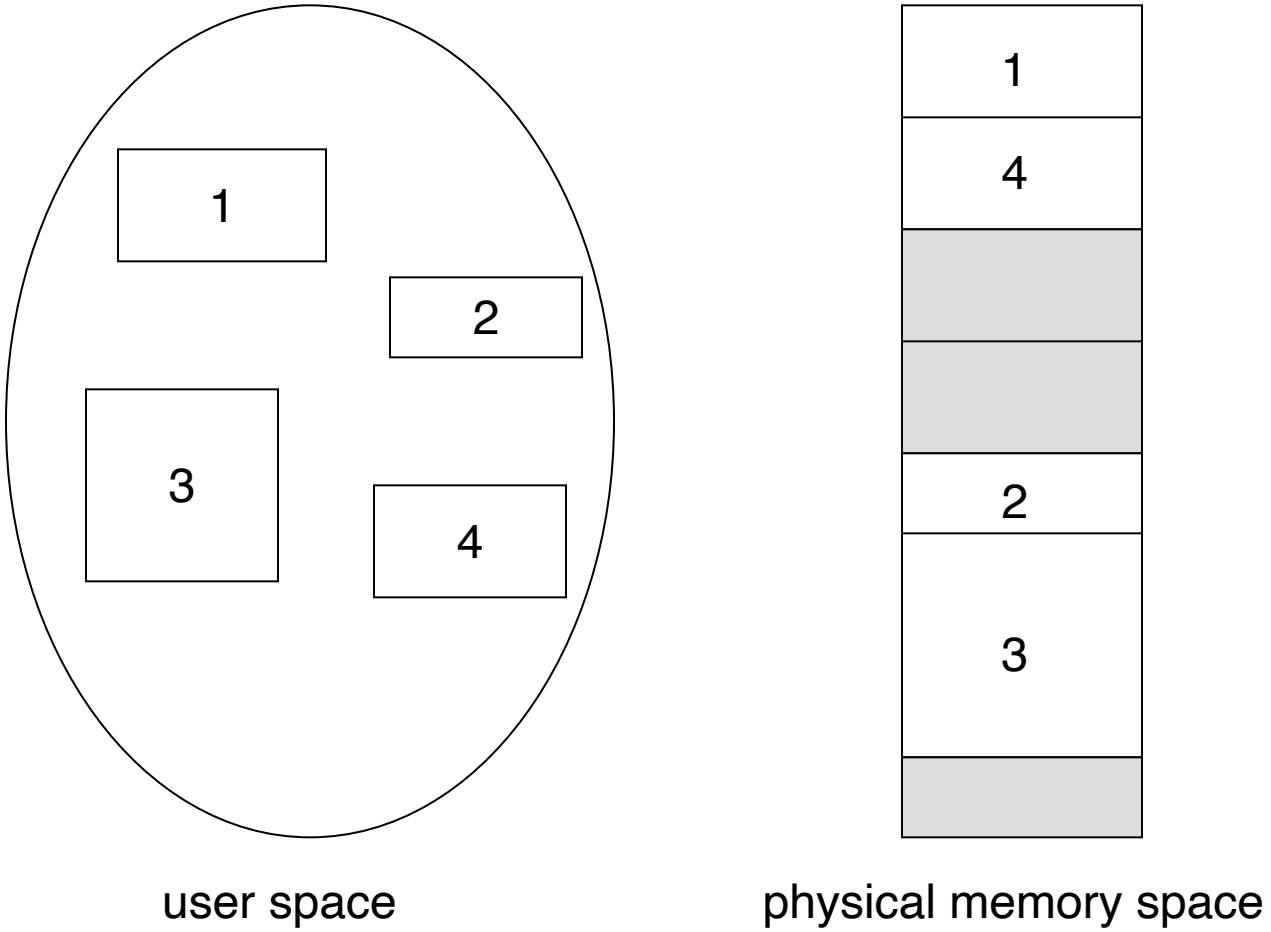
6.4 分段存储管理



程序的分段逻辑结构

6.4 分段存储管理

分段的逻辑视图

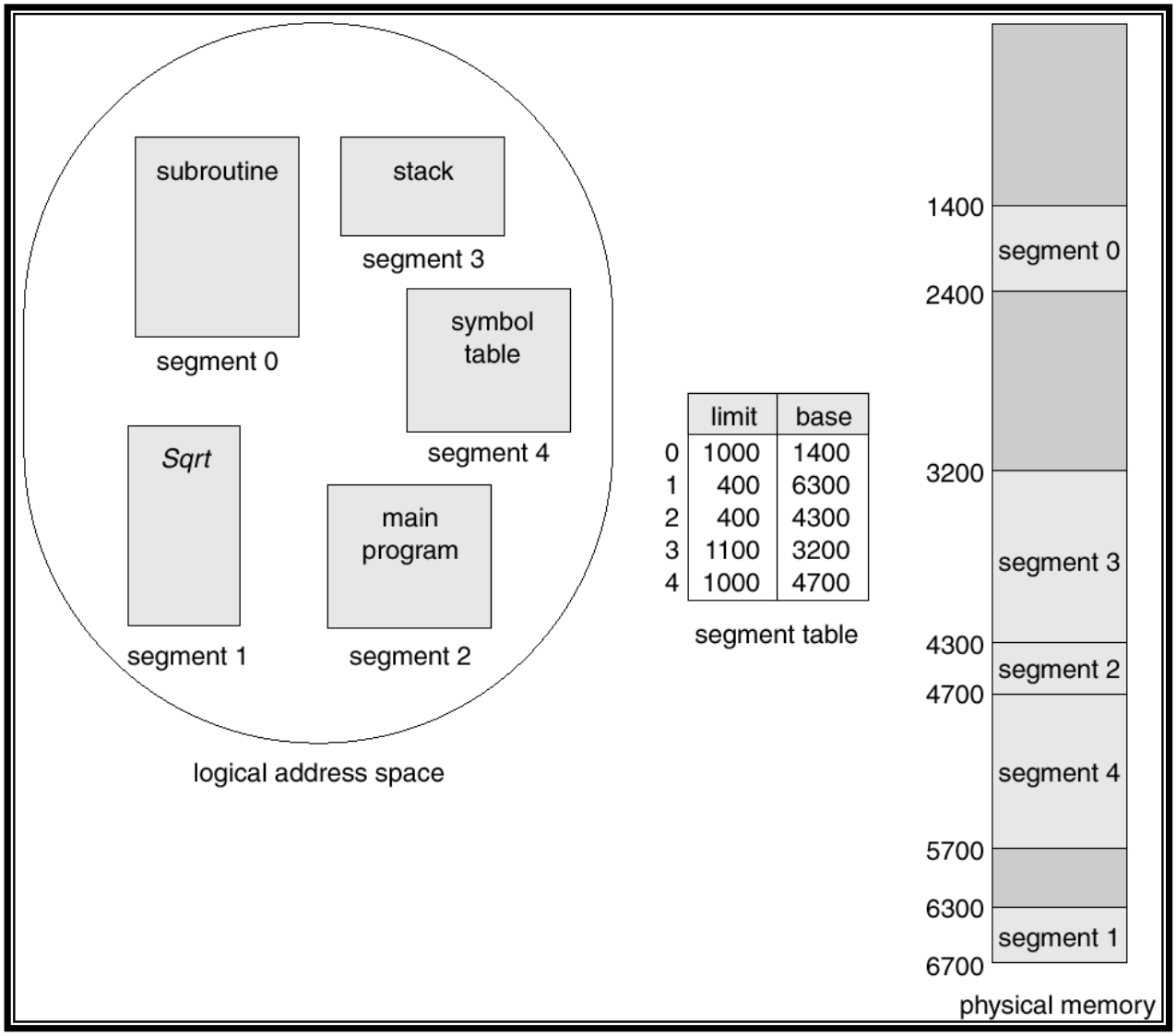


6.4 分段存储管理

段表

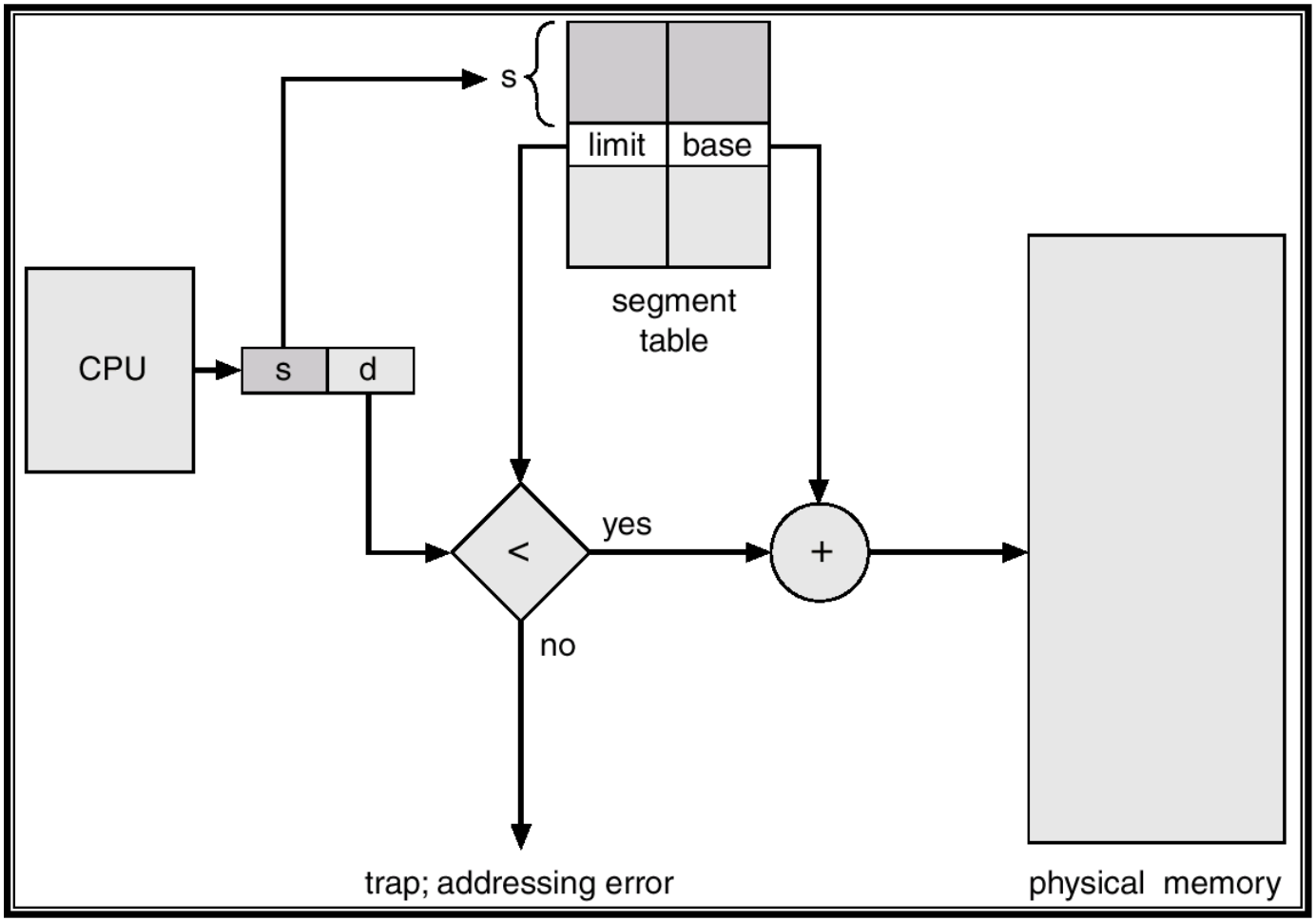
- 用于将二维的逻辑空间地址映射到一维的物理地址
- 段表中的每一个表项包含：
  - 段号
  - 段起始地址
  - 段长度

6.4 分段存储管理


subroutine
stack
segment 3
segment 0
symbol table
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory

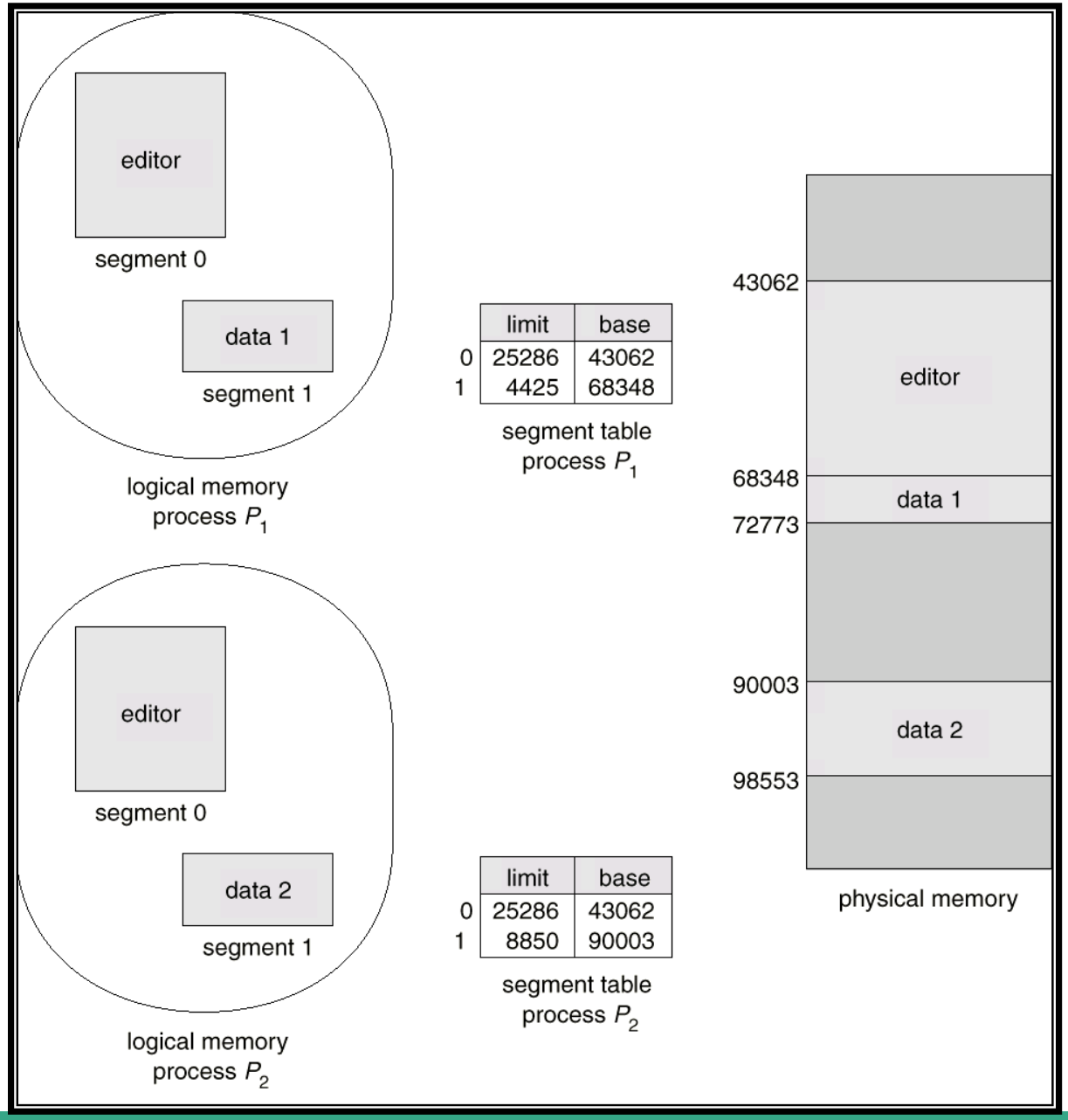
6.4 分段存储管理

分段存储管理的地址转换和存储保护



6.4 分段存储管理

段共享



6.4 分段存储管理

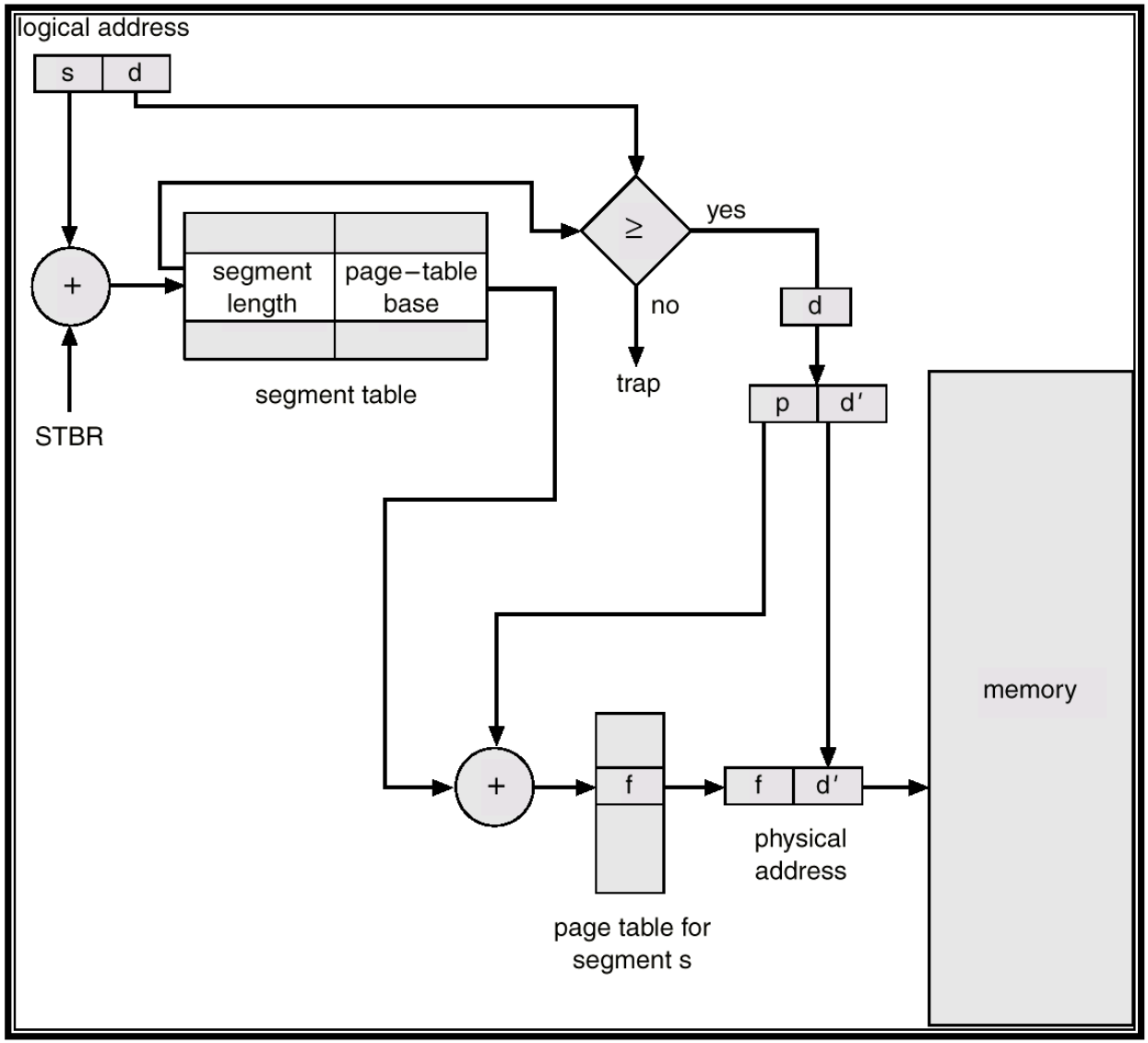
分段和分页的比较

- 信息组织：
  - 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
  - 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
- 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。

6.5 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

6.5 分段和分页相结合



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

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
- **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**
- 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

- **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