第三部分 内存

- 第7章 内存管理
  - 基本机制
  - 分区技术
- 第8章 虚拟内存
  - 分页技术
  - 分段技术
  - 地址转换

第7章 内存管理

存储管理的要求

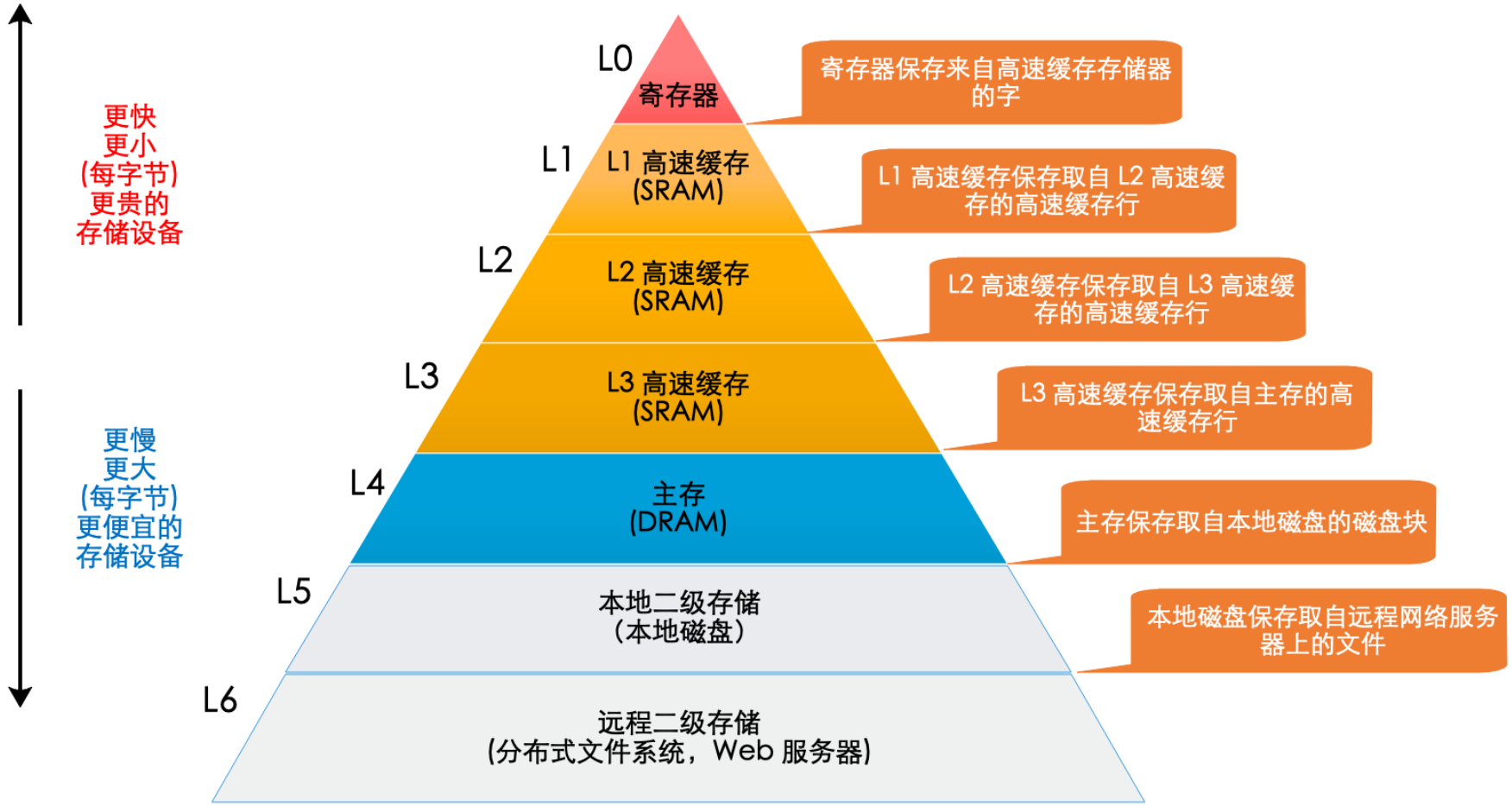
分区存储管理

分页存储管理

分段存储管理

存储体系

- 存储器的层次结构:



存储体系

- 存储器的层次结构:

  - 高速缓存Cache：KB~MB级、少量、高速、昂贵、易失
  - 内存：GB级、中等速度、中等价格、易失
  - SSD：数十GB级、较高速、价较高、非易失
  - 磁盘/外存：GB~TB级、低速、价廉、非易失

存储体系

- 内存
  - 系统区：存放操作系统
  - 用户区：存放用户程序和数据

存储管理的任务

- 主要工作：
  - 将程序载入内存以让CPU执行
- 目的：
  - 划分内存区域以容纳多个进程（多道程序设计）
  - 有效分配内存以容纳尽量多的进程

7.1 存储管理的要求

- 重定位
- 保护
- 共享
- 逻辑组织
- 物理组织

7.1.1 重定位

- 逻辑地址/相对地址
  - 逻辑地址：与内存内容无关的内存位置
  - 相对地址：相对于某一点（通常是程序起始位置，也可能是段基址）的逻辑地址（如偏移地址）
- 物理地址/绝对地址
  - 物理地址：内存的实际地址，也称绝对地址

7.1.1 重定位

- 多道程序和共享内存技术要求程序使用相对地址以支持重定位
  - 程序员不知道程序在运行时在内存中所处的位置
  - 处于内存某一块区域的程序代码，在运行中可能被交换出(swap out)到外存，后来又被交换进(swap in)到内存的另外一块区域
- 重定位一般要求硬件（CPU）支持

7.1.2 保护

- 多道程序和共享内存技术要求一个进程不能对其他进程进行有意或无意的非授权访问
- 程序的内存引用只能在运行时检查
  - 重定位导致无法在编译时检查绝对地址
  - 多数程序设计语言允许地址的动态计算（如数组下标、指向某种数据结构的指针等）
  - 通常整合在重定位硬件机制中(从软件上，难以预计所有非法情况，开销较高)

7.1.3 共享

- 支持不同进程访问内存的同一区域
- 可能情形
  - 同一程序的不同进程实例共享程序区
  - 合作进程间共享某些数据结构
- 重定位机制通常也支持共享

7.1.4 逻辑组织

- 物理上，主存和辅存通常是一维线性结构
- 采用模块化形式组织用户程序及数据，反映程序组织的逻辑性
  - 模块化有利于设计期间的编程
  - 模块化有利于运行时刻的保护/共享
- 分段存储管理技术最符合用户（程序员）组织程序的观点

7.1.5 物理组织

- 主存与辅存间的信息交换有利于实现系统目标
- 要求程序员关心物理组织不切实际
  - 主存有限需采用覆盖技术，消耗程序员精力和时间
  - 程序员无法预知可用主存的数量和位置
- 物理组织是操作系统的责任（资源管理）

甩锅

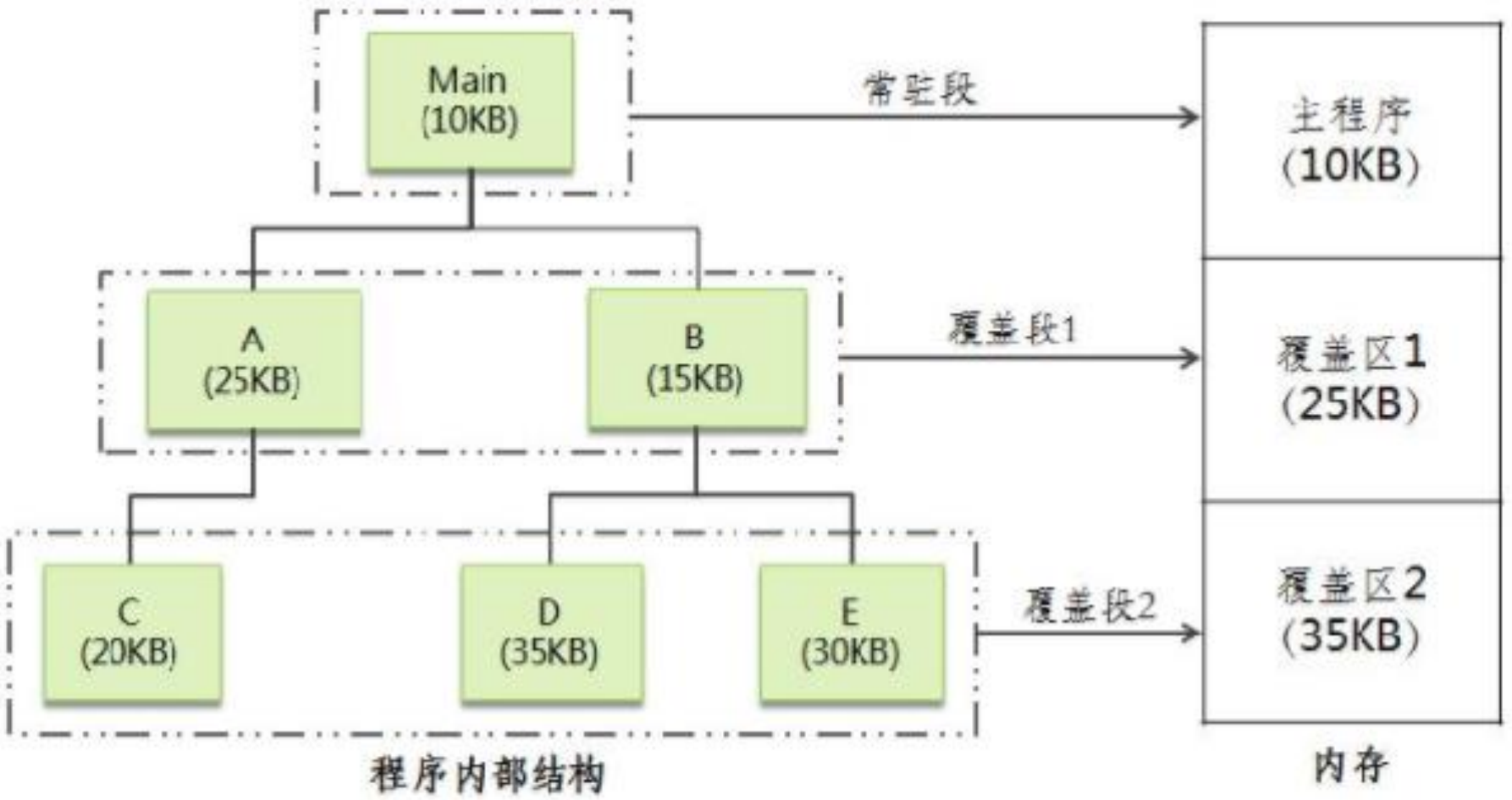
覆盖技术和交换技术

- 多道环境下扩充内存
- 覆盖技术主要用在早期的操作系统中
- 交换技术被广泛用于小型分时系统中，导致了虚存技术的出现
- 共同点：
  - 进程的程序和数据主要放在外存
  - 只有当前需要执行的部分才放在内存
  - 内外存之间进行信息交换

覆盖技术（overlaying）

- 把程序划分为若干个功能上相对独立的程序段，按照其自身的逻辑结构，让不会同时执行的程序段共享同一块内存区域
- 程序员向系统提供覆盖结构，由操作系统完成程序段之间的覆盖

覆盖技术（overlaying）



交换技术（swapping）

- 交换技术
  - 内存空间紧张时，系统将内存中某些进程暂时移到外存(swap out)，把外存中某些进程换进内存(swap in)，占据前者原来所占用的区域
  - 交换技术即进程在内存与外存之间的动态调度

交换技术（swapping）

- 与覆盖技术比较：
  - 交换技术不要求用户给出程序段之间的逻辑覆盖结构
  - 交换发生在不同的进程或作业之间，而覆盖发生在同一进程或作业之内

	覆 盖	交 换
逻辑覆盖结构	需要	不需要
作用对象	进程内部	进程之间

7.2 分区存储管理技术

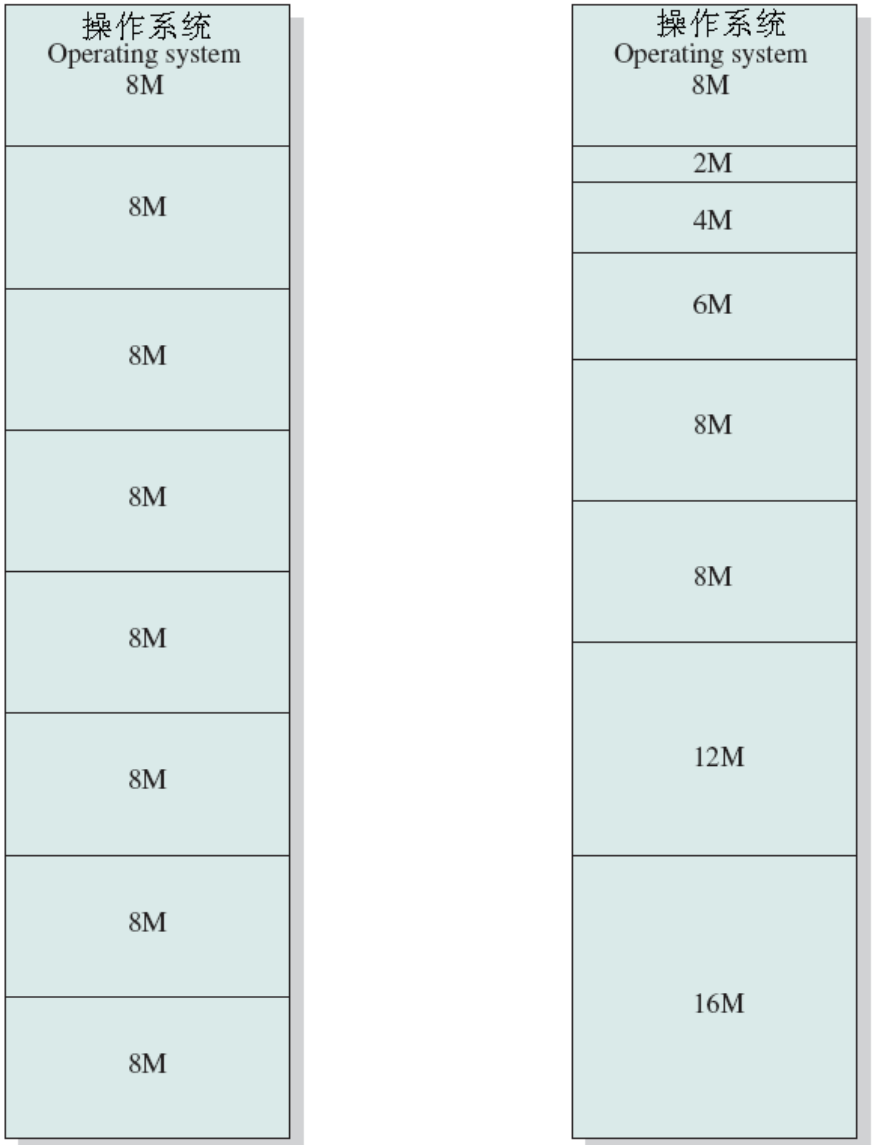
- 分区（partitioning）
  - 包括固定分区和动态分区
  - 只在一些特殊场合使用（如内核存储管理）
  - 在一些已过时的操作系统中采用

7.2 分区存储管理技术

- 简单内存分页（不单独使用）——大小相等，需一次装入一个进程的所有页
- 简单进程分段（不单独使用）——需一次装入一个进程的所有段
- 虚拟内存分页——不需一次装入一进程的所有页
- 虚拟内存分段—不需一次装入一进程的所有段

7.2.1 固定分区

- 将内存划分成若干固定大小的区域：
  - 等长
  - 不等长



(a) Equal-size partitions
大小相等的分区

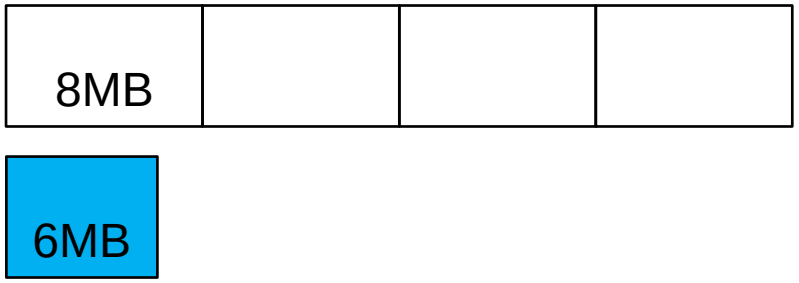
(b) Unequal-size partitions
大小不等的分区

Example of Fixed Partitioning of a 64-Mbyte Memory

64MB内存的固定分区例

固定分区

- 等长分区：任何小于或等于分区大小的进程都可以全部载入某一可用分区中（无可用分区时，可以应用交换技术）



- 问题：
  - 若进程大于分区，则只能部分载入，要应用覆盖技术
  - “小”进程将产生内碎片(internal fragmentation)，导致内存利用率降低

固定分区

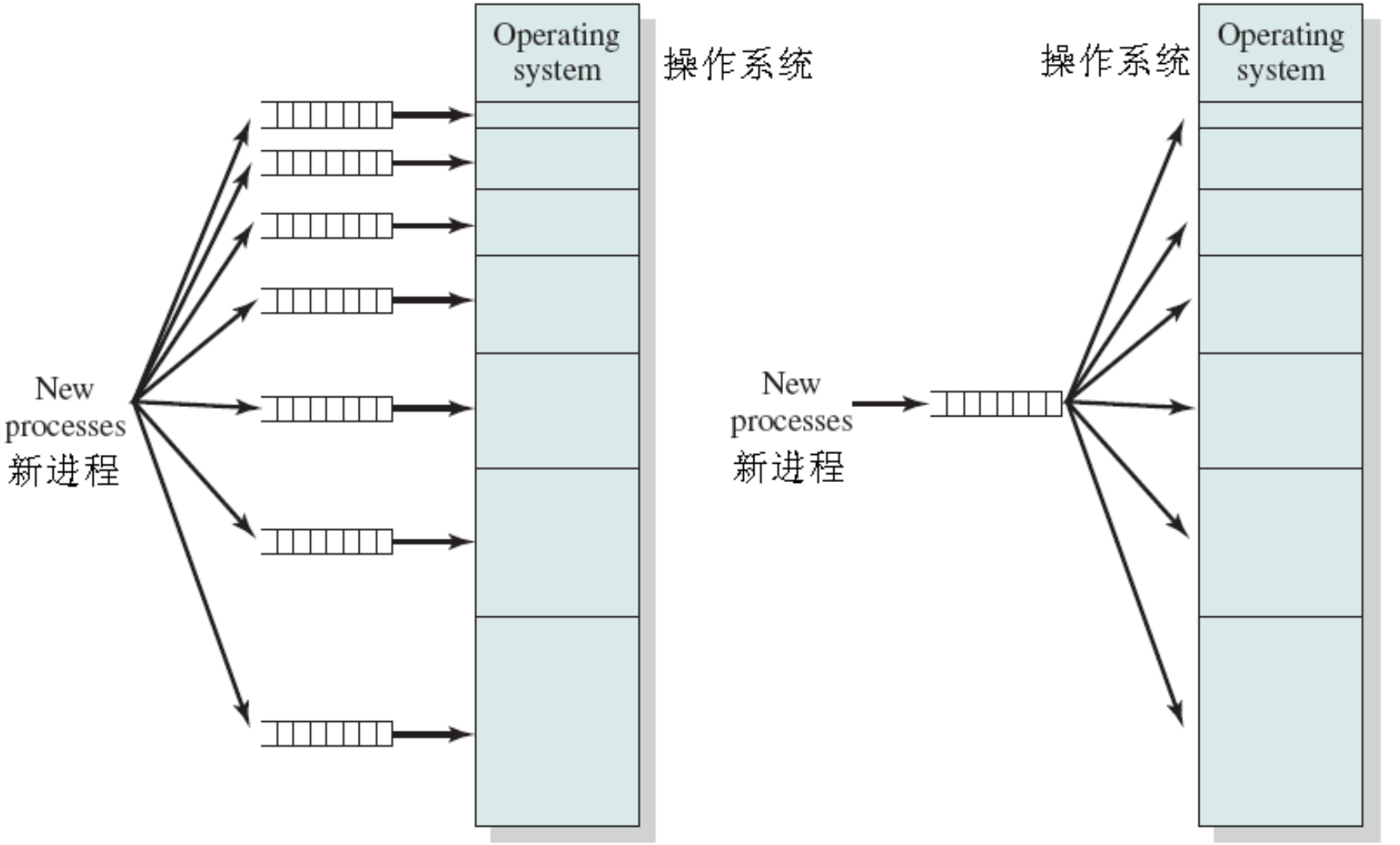
- 等长分区：任何小于或等于分区大小的进程都可以全部载入某一可用分区中（无可用分区时，可以应用交换技术）

- 不等长分区：在一定程度上减缓了等长分区存在的问题（没有完全解决！）

固定分区的放置算法

- 等长分区的放置算法：
  - 进程可以放到**任意**一个可用分区中
- 不等长分区的放置算法：
  - 多队列：为每个分区设立一个输入队列，各个队列中的进程只能使用对应的分区——某些队列为空时会造成内存空间的浪费（如小分区队列满而大分区队列空）
  - 单队列：只有一个输入队列，进程使用可容纳它的最小空闲分区（无可用分区时可“交换”）

不等长固定分区的放置算法



(a) One process queue per partition
每个分区一个队列

(b) Single queue
单一队列

Memory Assignment for Fixed Partitioning

固定分区中的内存分配

固定分区

- 优缺点：
  - 相对简单，开销小
  - 分区数目预设（系统创建时），限制了活动进程数
  - 分区大小预设，小作业不能充分利用其占有空间
- 采用固定分区的操作系统
  - MFT（Multiprogramming with a Fixed Number of Tasks，IBM 1966年推出）

7.2.2 动态分区

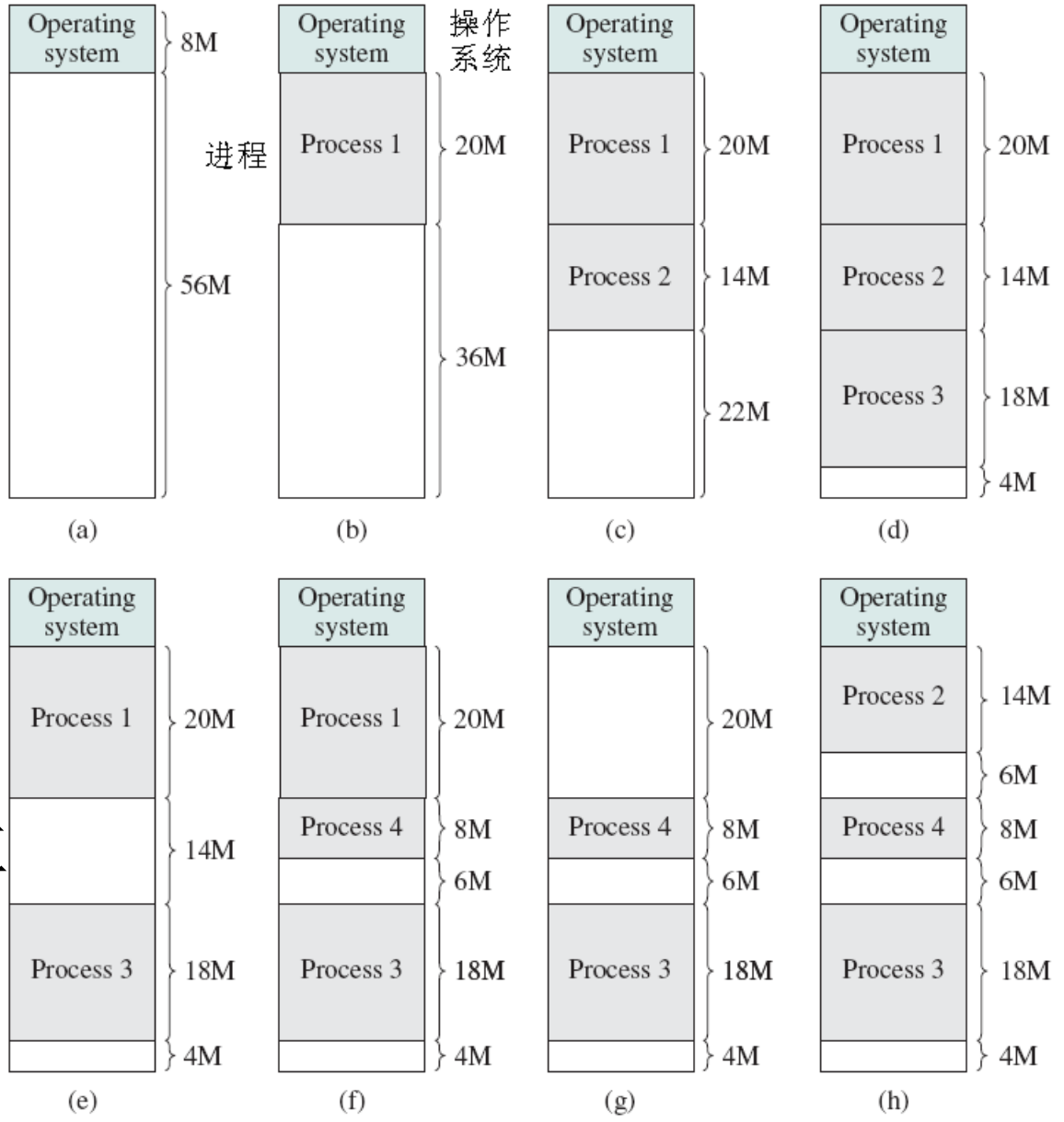
- 实例：IBM OS/MVT (Multiprogramming with a Variable Number of Tasks，具有可变任务数的多道程序设计，1967年)
- 系统运行中分区数目和大小均可以改变
  - 最初分区数为0，进程需要多少就给它分配多大分区（**按需分配**）
- 存在问题：外碎片 (external fragmentation)

动态分区

- 消除碎片：压缩(compaction)移动进程使相互紧靠

耗时，

且需要进行动态重定位



The Effect of Dynamic Partitioning
动态分区的效果

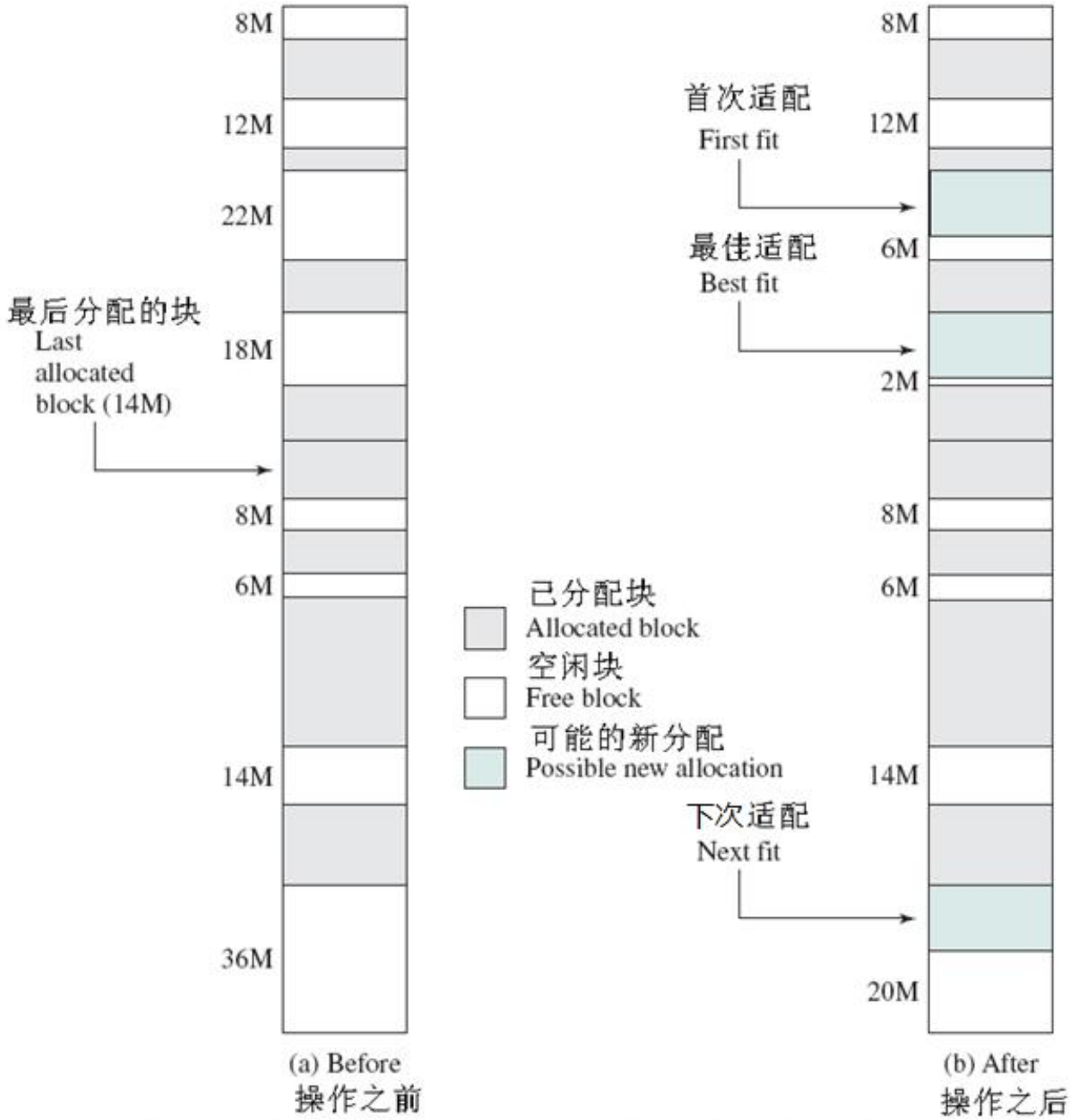
动态分区的放置算法

- 首次适配(first fit)算法
  - 从前端开始扫描内存，直到找到一个足够大的空闲区
- 下次适配(next fit，邻近适配)算法
  - 从上次分配结束的地方开始扫描内存，直到找到一个足够大的空闲区

动态分区的放置算法

- 最佳适配(best fit)算法
  - 扫描整个内存，找出一个足够大的最小的空闲区
- 最坏适配(worst fit)算法
  - 扫描整个内存，找出一个最大的空闲区

动态分区放置算法



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

分配16MB块前后的内存配置例

动态分区的放置算法性能比较

- 首次适配算法
  - 简单，且通常最好、最快
- 下次适配算法
  - 稍差于首次适配
  - 通常分配位于内存末端的空闲区，致使大块空闲区很快被分裂，因此经常要“压缩”

动态分区的放置算法性能比较

- 最佳适配算法
  - 通常最差
  - 产生大量无用的小碎片，导致需要更经常的“压缩”

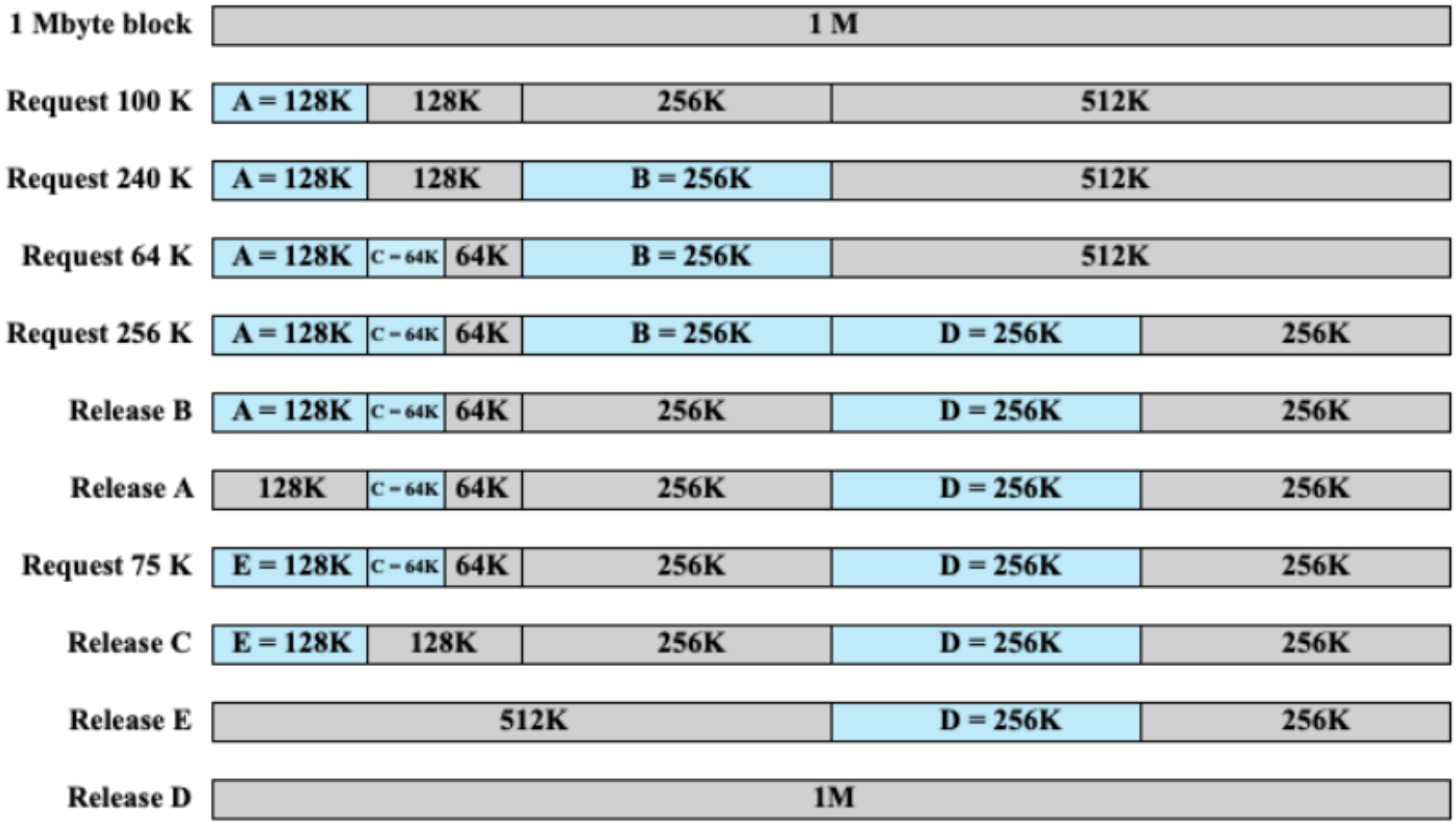
7.2.3 伙伴系统

- 固定分区限制了活动进程的数量，而且内存空间利用率低
- 动态分区维护复杂，且需要额外的“压缩”开销

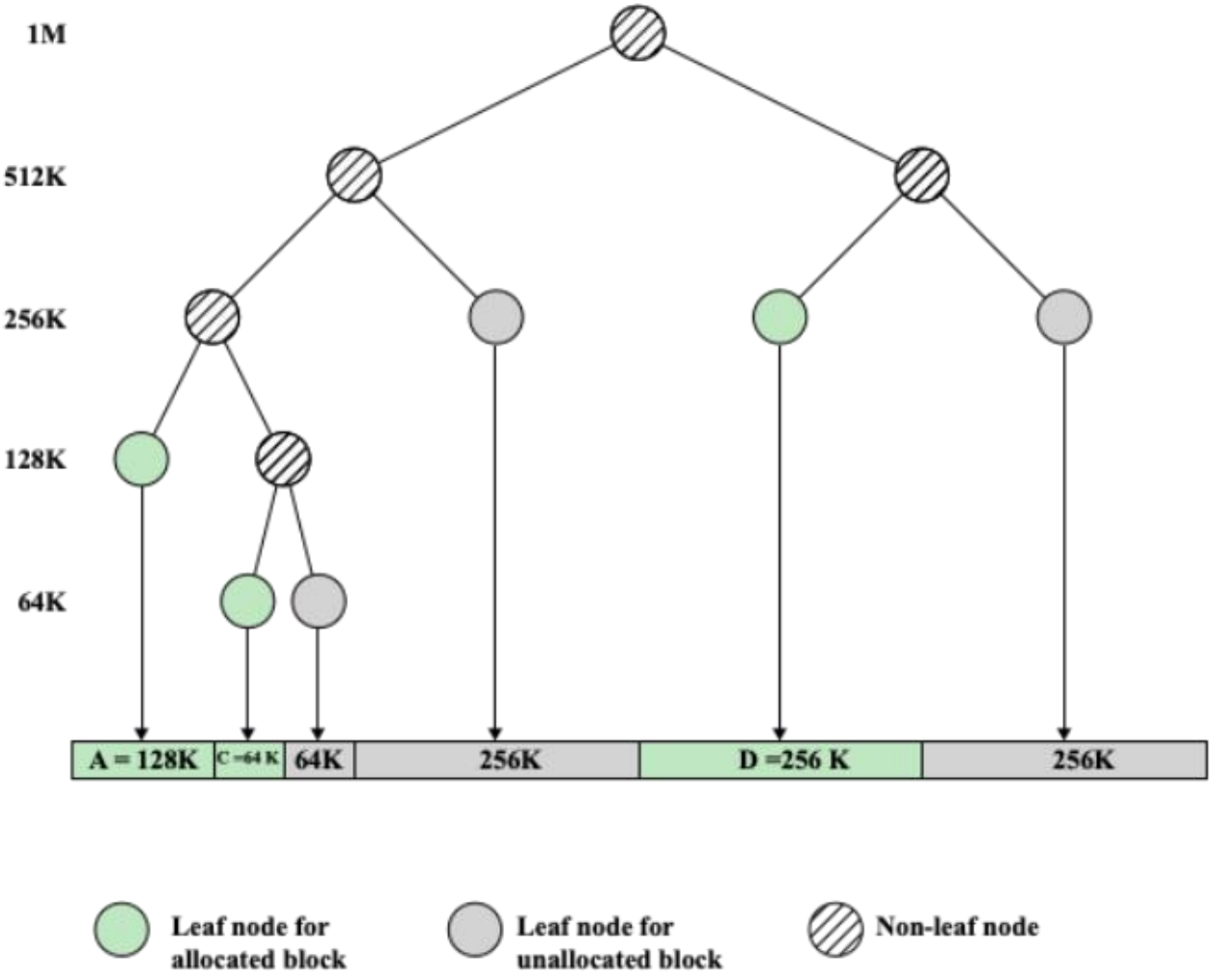
7.2.3 伙伴系统

- 折中方案——伙伴系统（buddy system）
  - 可用内存块的大小为2^K，$L \le K \le U$
    - 最小块的大小为2^L
    - 最大块的大小为2^U
  - 初始空间为大小为2^U的块
  - 若请求的空间大小$s<2^{U-1}$，则对分现有块
  - 维护大小为2^K的所有空闲块的列表

伙伴系统举例


1 Mbyte block
1 M
Request 100 K
A = 128K
128K
256K
512K
Request 240 K
A = 128K
128K
B = 256K
512K
Request 64 K
A = 128K
C - 64K
64K
B = 256K
512K
Request 256 K
A = 128K
C - 64K
64K
B = 256K
D = 256K
256K
Release B
A = 128K
C - 64K
64K
256K
D = 256K
256K
Release A
128K
C - 64K
64K
256K
D = 256K
256K
Request 75 K
E = 128K
C - 64K
64K
256K
D = 256K
256K
Release C
E = 128K
128K
256K
D = 256K
256K
Release E
512K
D = 256K
256K
Release D
1M

伙伴系统的树状表示

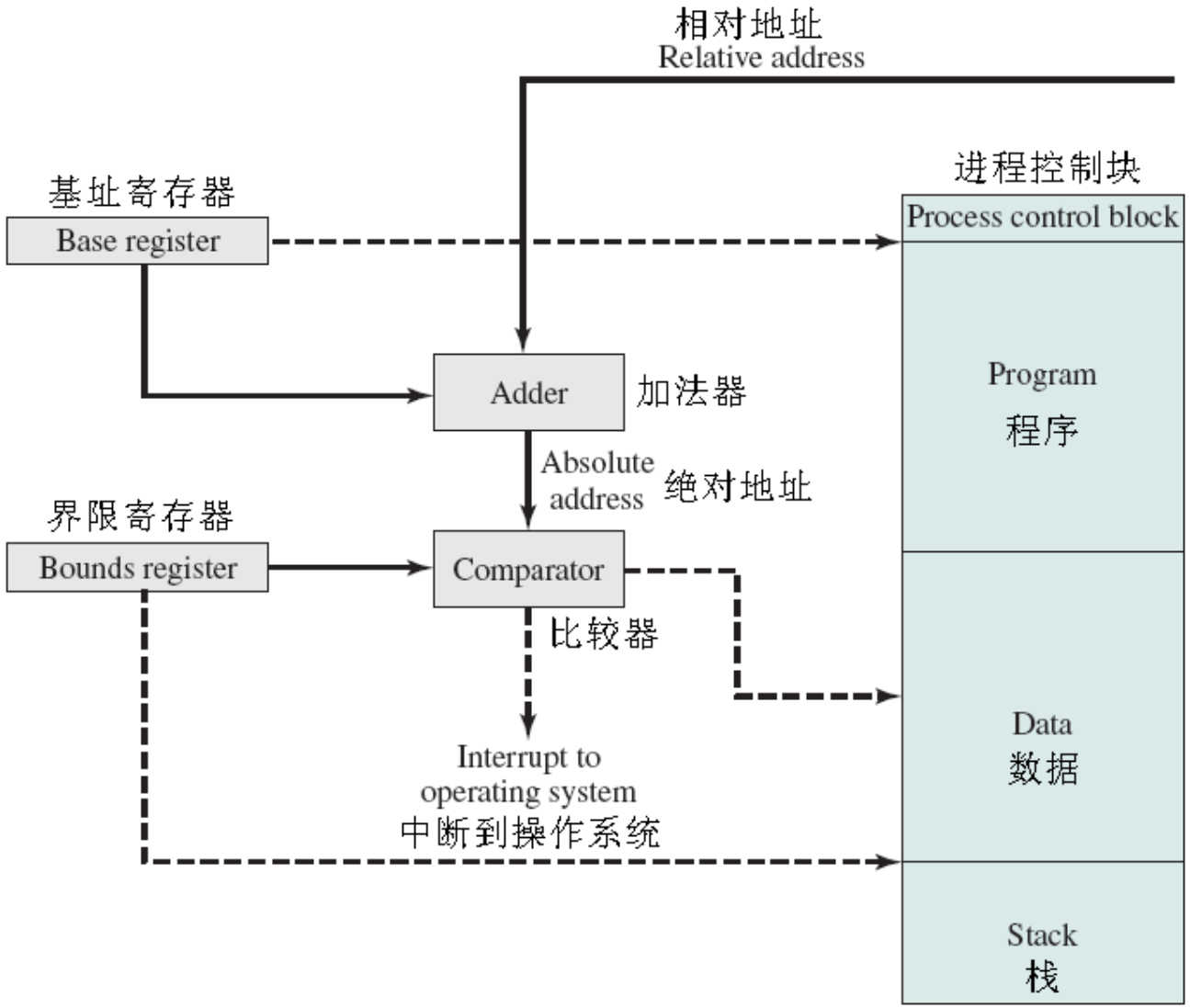

1M
512K
256K
128K
64K
A = 128K
C =64 K
64K
256K
D =256 K
256K
Leaf node for allocated block
Leaf node for unallocated block
Non-leaf node

7.2.4 分区管理中的重定位

- 静态重定位：
  - 仅进程第一次载入时把内存引用全换成绝对地址
- 动态重定位：
  - 动态分区只能采用动态重定位，因为消除碎片时的“压缩”操作，进程的内存位置会改变
  - 程序指令执行时，CPU内把逻辑地址才转换位内存地址

重定位的硬件机制

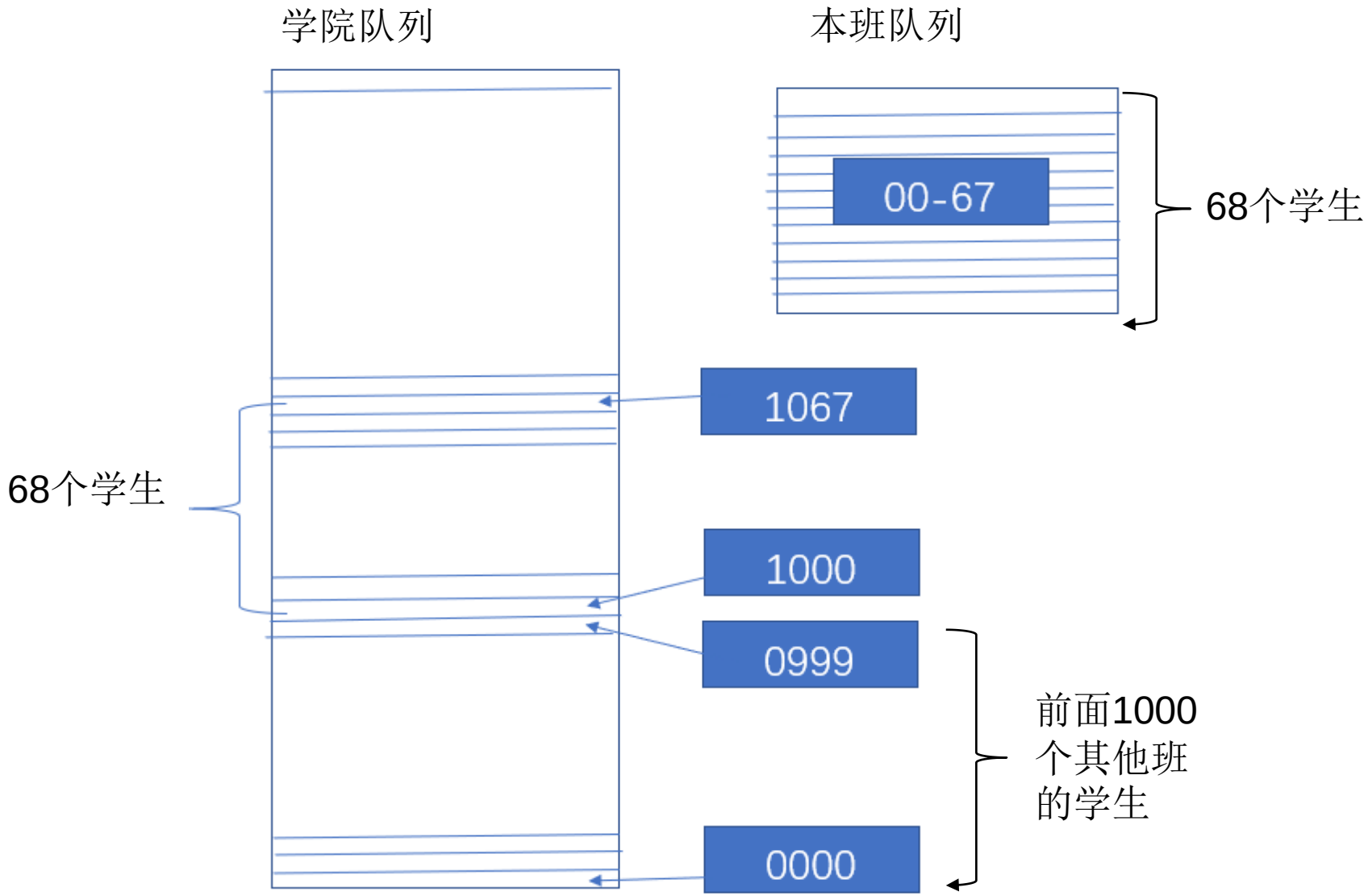
- 寄存器：
  - 基址寄存器
  - 界址寄存器
- 地址转换：
  - 计算物理地址
  - 安全检查



Hardware Support for Relocation

重定位的硬件支持

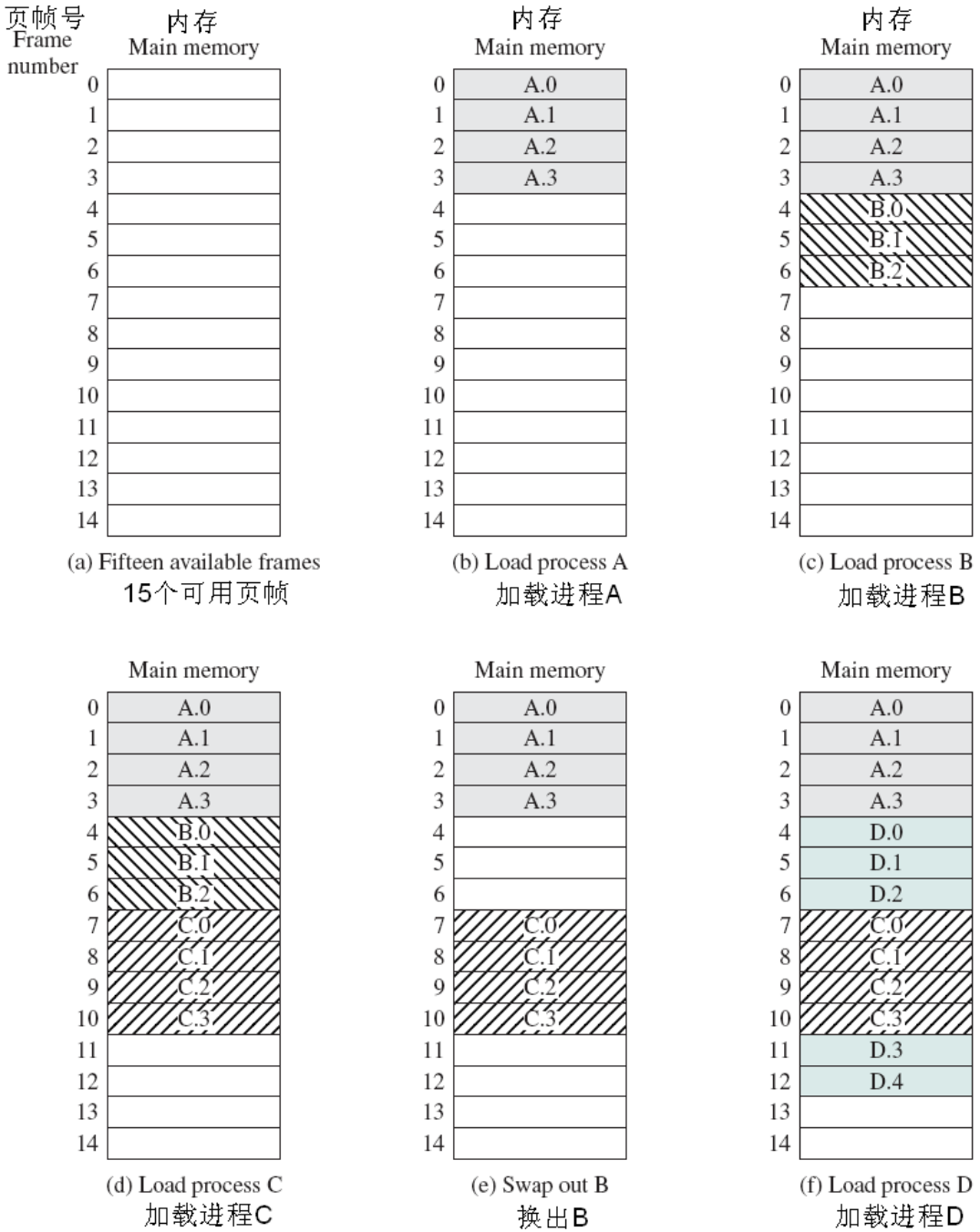
重定位类比：运动会班队序号对应院队序号



7.3 页式存储管理：分页 (paging)

- 将主存划分成许多等长的小块，称为页框(frame)
- 将进程划分成若干页(page)，一个页的大小与一个页框的大小相等
- 进程加载时，所有页被载入可用页框（不要求连续），同时建立页表(page table)——简单分页的主要数据结构

简单分页的进程加载示例



Assignment of Process to Free Frames

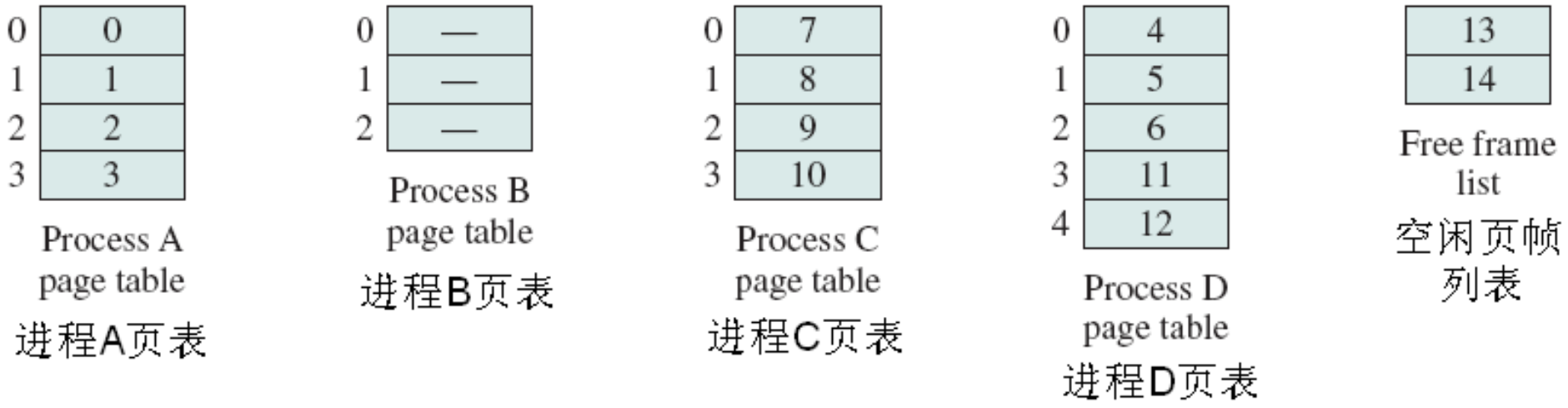
分配进程到空闲帧

页表（Page Table）

- 操作系统通过页表的建立和维护进行内存管理：
  - 操作系统为每个进程建立并维护一个页表
  - 页表的每个表项包含该页在内存中对应的页框号（还包含保护、共享等信息）
  - 页表以页号为索引
  - 操作系统另外维护一个空闲页框的列表

页表（Page Table）

- 操作系统通过页表的建立和维护进行内存管理：
  - 操作系统为每个进程建立并维护一个页表
  - 页表的每个表项包含该页在内存中对应的页框号（还包含保护、共享等信息）



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

前图例在时间点(f)时的数据结构

简单分页中的重定位

- 程序中的逻辑地址由两部分组成
  - 页号
  - 页内偏移
- CPU的一对寄存器记录当前运行进程的页表起始物理地址、页表长度
- (页号,偏移) → (页框号,偏移)
- 页(页框)的大小必须为2的整数m次幂

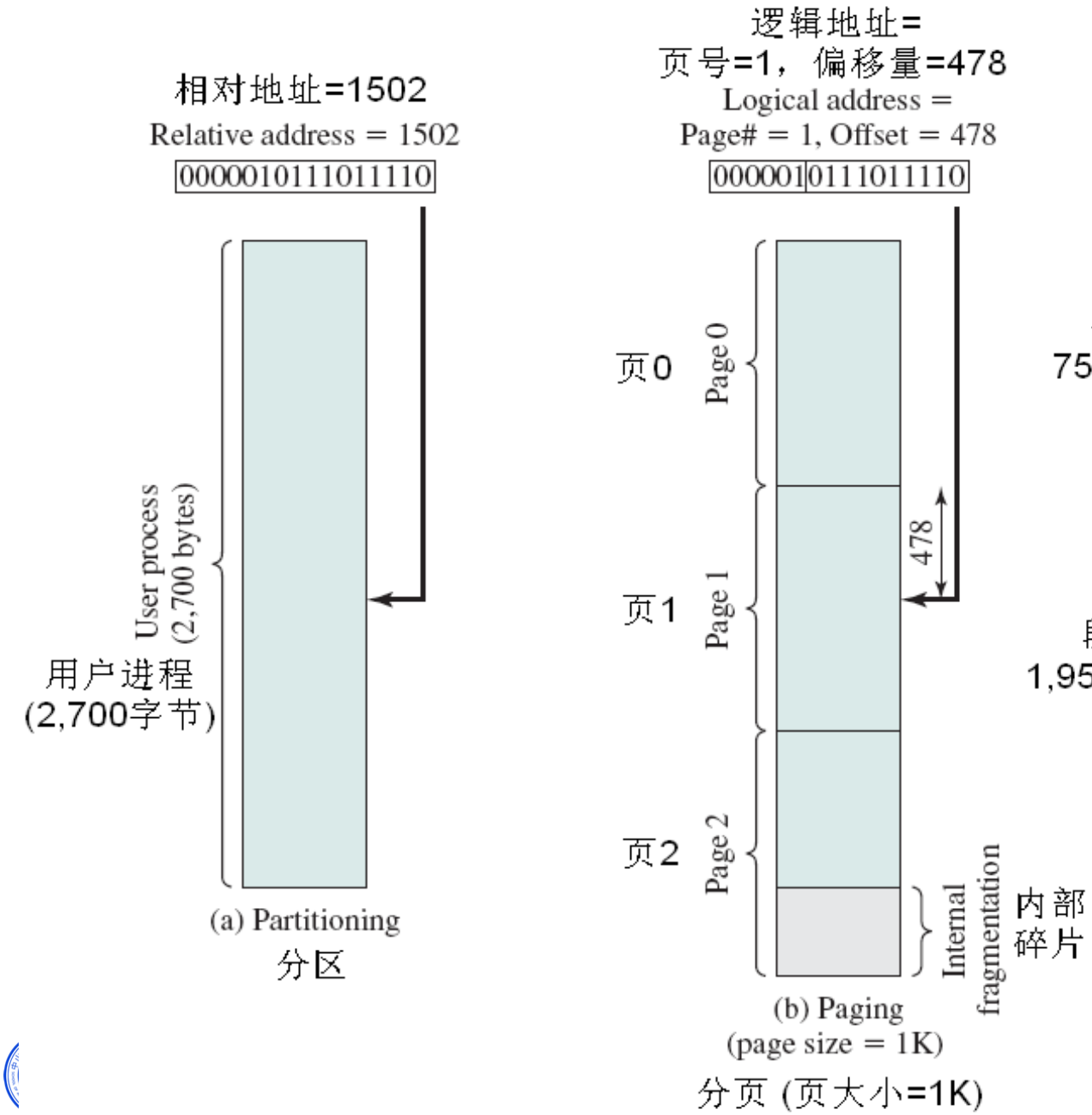
简单分页中的重定位

- 当页(页框)的大小为2的m次幂时，逻辑地址与相对地址一致
- 页面大小为2的m次幂时，页面（逻辑地址）对程序员、编译器/汇编程序、链接程序都是透明

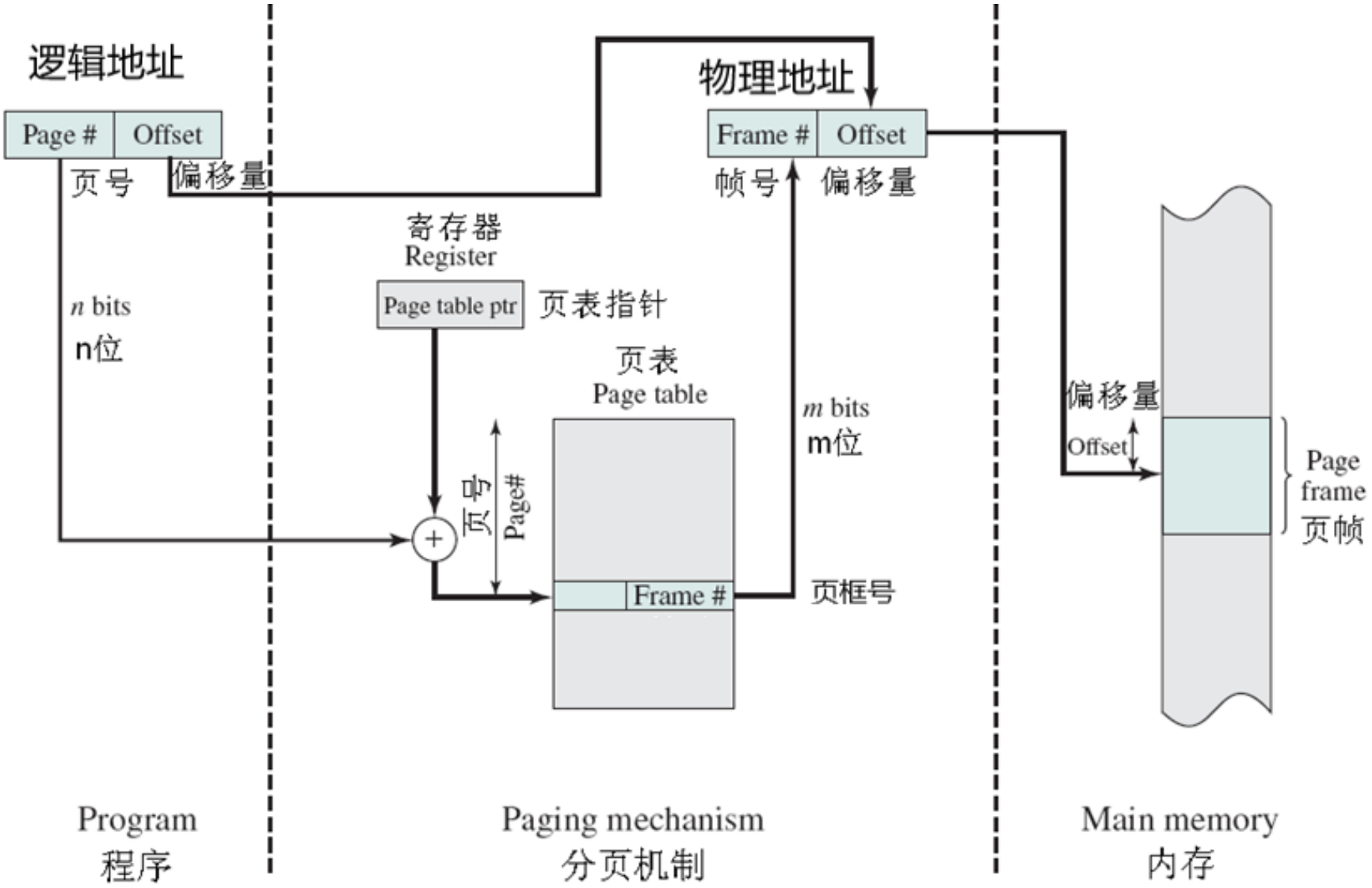
简单分页中的重定位

- 例：16位编址，若页面大小为1K（1024），则需（低）10位表示页内偏移，剩下（高）6位表示页号，则
  - 相对地址为1502的逻辑地址 = 1024 + 478 = (1, 478)
  - 逻辑地址为(1, 478)的相对地址 = 1*1024 + 478 = 1502

分区、分页、分段的逻辑地址



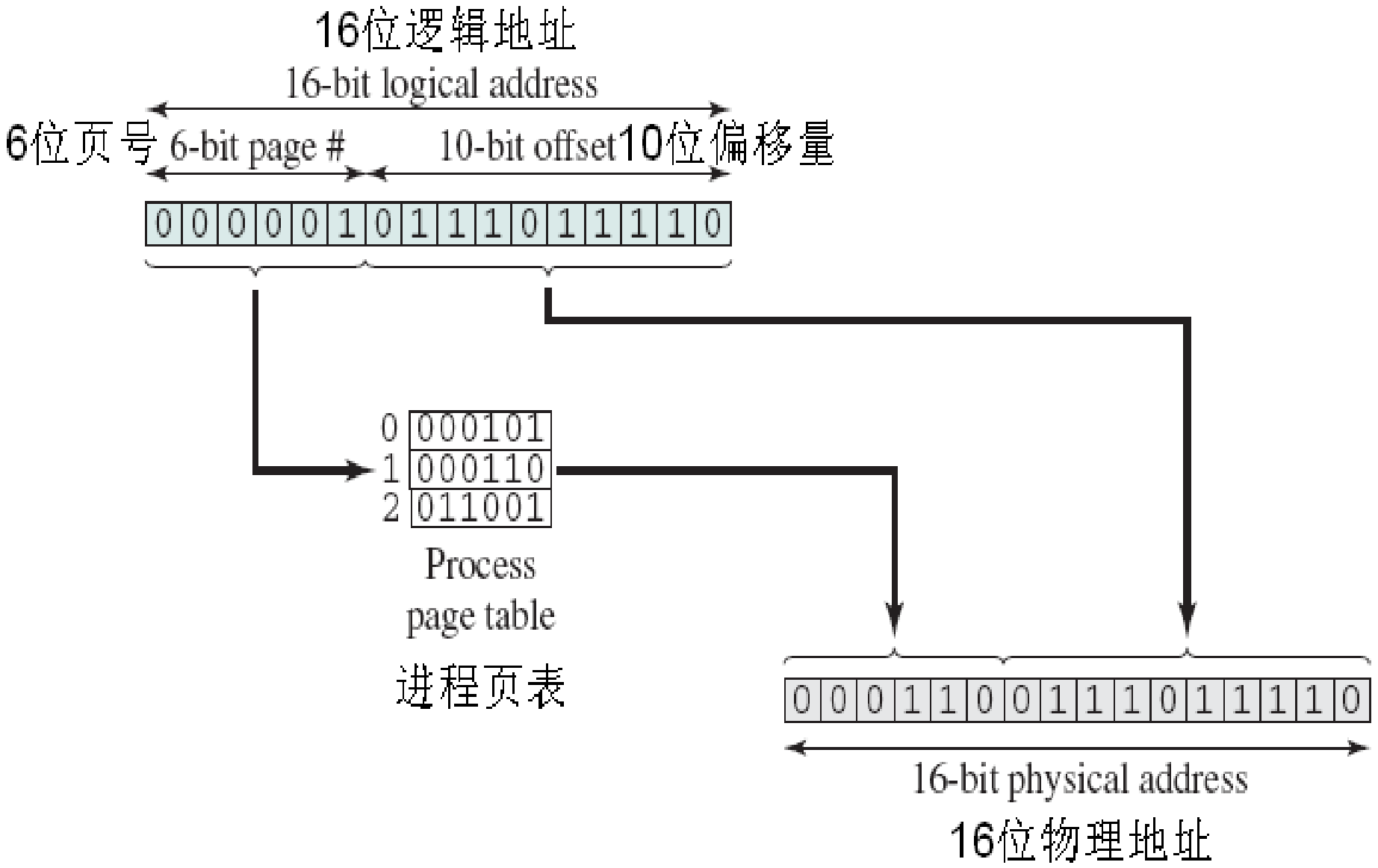
简单分页地址重定位



Address Translation in a Paging System

分页系统中的地址转换

简单分页中的重定位例子



简单分页的特点

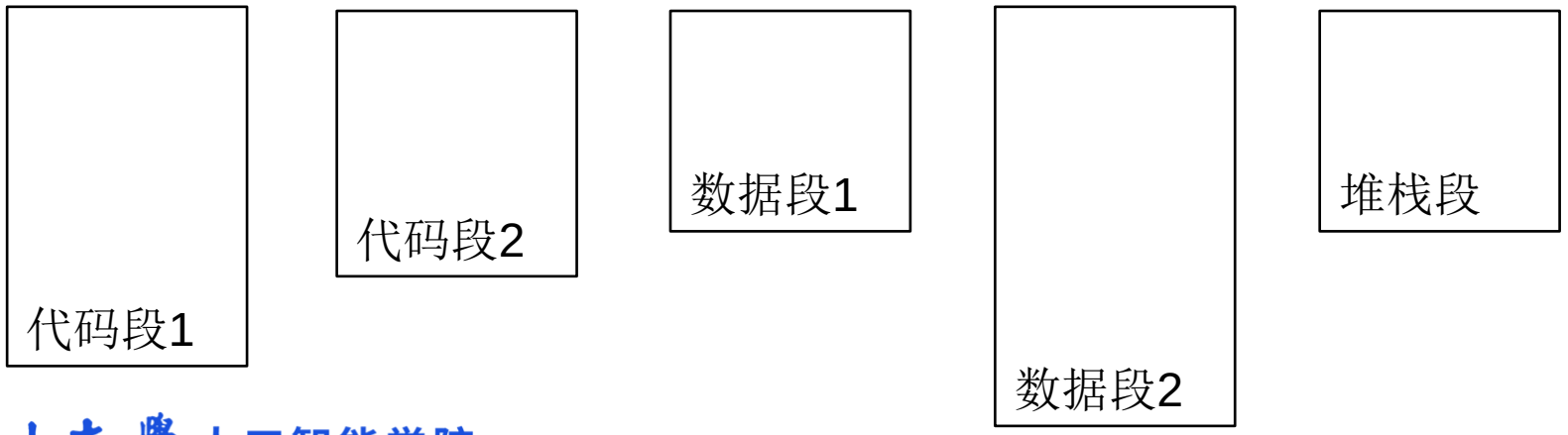
- 类似固定分区，不同在于：
  - 分页中的“分区”（页框）非常小（从而内碎片也小）
  - 分页中一个进程可占用多个“分区”（页框）（从而不需要覆盖）
  - 分页中不要求一个进程占用的多个“分区”（页框）连续（充分利用空闲“分区”）

简单分页的特点

- 存在问题：
  - 不易实现共享和保护（不反映程序的逻辑组织）
  - 不便于动态链接（线性地址空间）
  - 不易处理数据结构的动态增长（线性地址空间）

7.4 分段存储管理

- **基本原理**
  - 将程序及数据划分成若干段(segment)（不要求等长，但不能超过最大长度）
  - 进程加载时，所有段被载入内存可用区域（不要求连续），同时建立段表（segment table）



段表

- 操作系统通过段表管理内存

代码段1	内存区1起始地址	8K	E
代码段2	内存区2起始地址	6K	E
数据段1	内存区3起始地址	4K	R/W
数据段2	内存区4起始地址	12K	R/W
堆栈段	内存区5起始地址	2K	R/W

  - 操作系统为每个进程建立并维护一个段表
  - 段表的每个表项包含该段在内存中的起始物理地址、段长等
  - 段表以段号为索引
  - 操作系统另外还维护一个内存空闲块的表

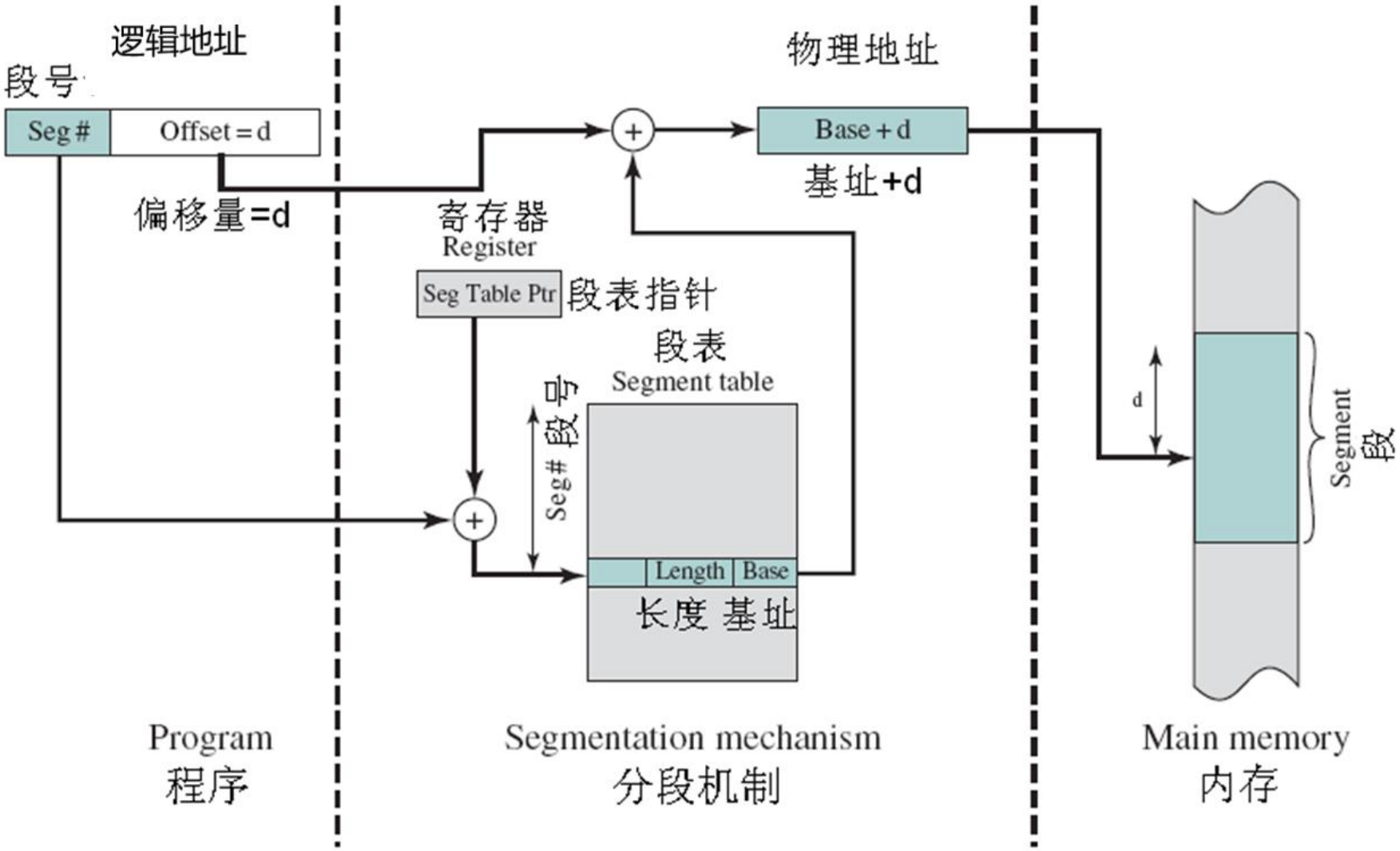
简单分段中的重定位

- 程序中的逻辑地址由两部分组成：
  - 段号
  - 段内偏移
- 进程进入运行态时，其段表地址被载入CPU专用寄存器

简单分段中的重定位

- 逻辑地址(n,m)－>物理地址的转换过程：
  - 根据n位计算段号
  - 以段号为索引到段表查找得到段起始物理地址
  - 比较偏移（m位）与段长（据段表），若前者大，则为非法地址
  - 物理地址＝段起始地址＋偏移

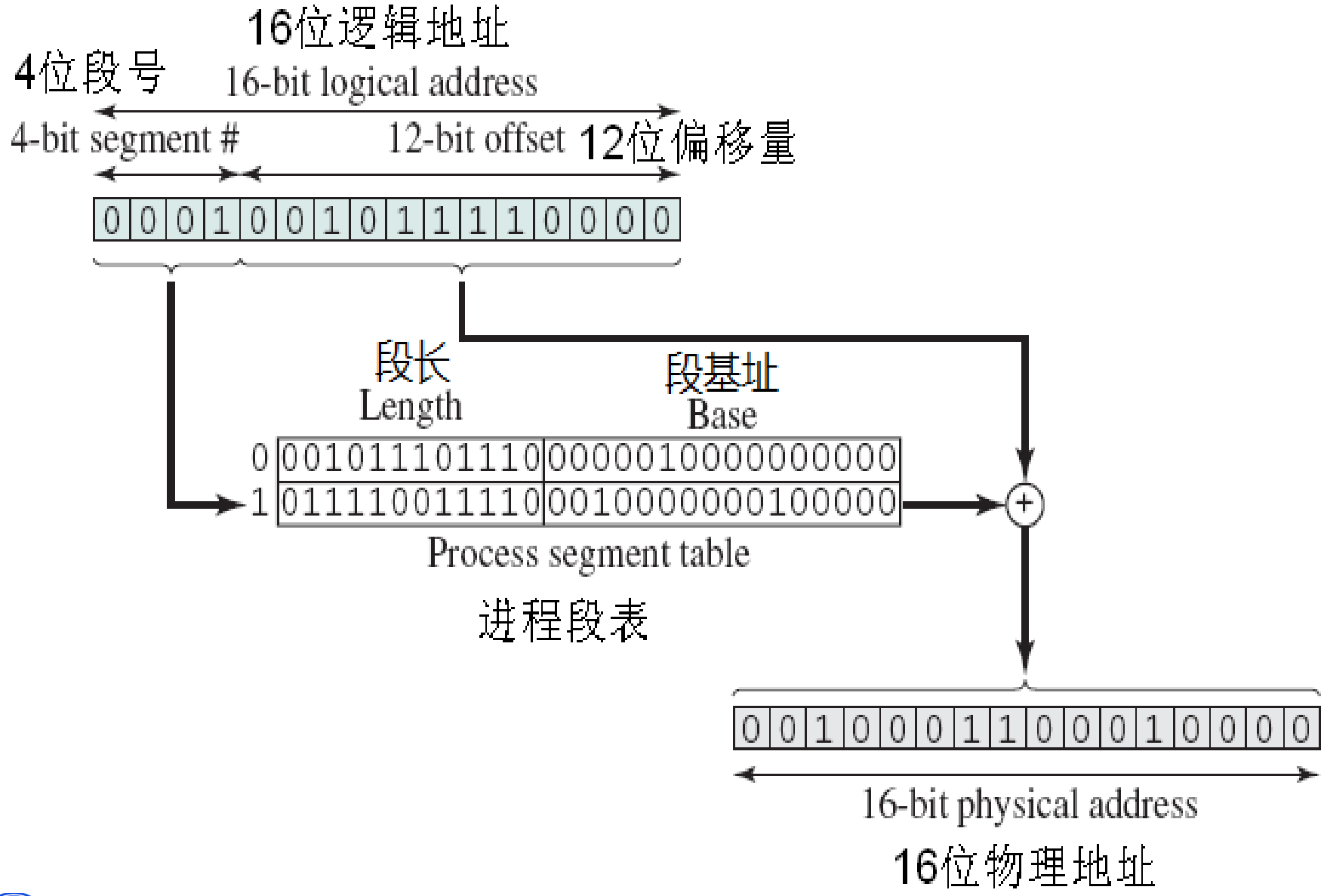
简单分段中的重定位



Address Translation in a Segmentation System

分段系统中的地址转换

简单分段中的重定位例子



页式管理和段式管理的比较

- 分页是出于系统管理的需要，分段是出于用户应用的需要
  - 一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处
- 页大小是系统固定的，而段大小则通常不固定

页式管理和段式管理的比较

- 逻辑地址表示
  - 分页是一维的，各个模块在链接时必须组织成同一个地址空间
  - 分段是二维的，各个模块在链接时可以每个段组织成一个地址空间
- 通常段比页大，因而段表比页表短，可以缩短查找时间，提高访问速度

页式管理和段式管理的比较

- 分段对程序员可见，从而可用来对程序和数据进行模块化组织
- 分段方便实现模块化共享和保护，如程序可执行、数据可读写（段表表项要有保护位）

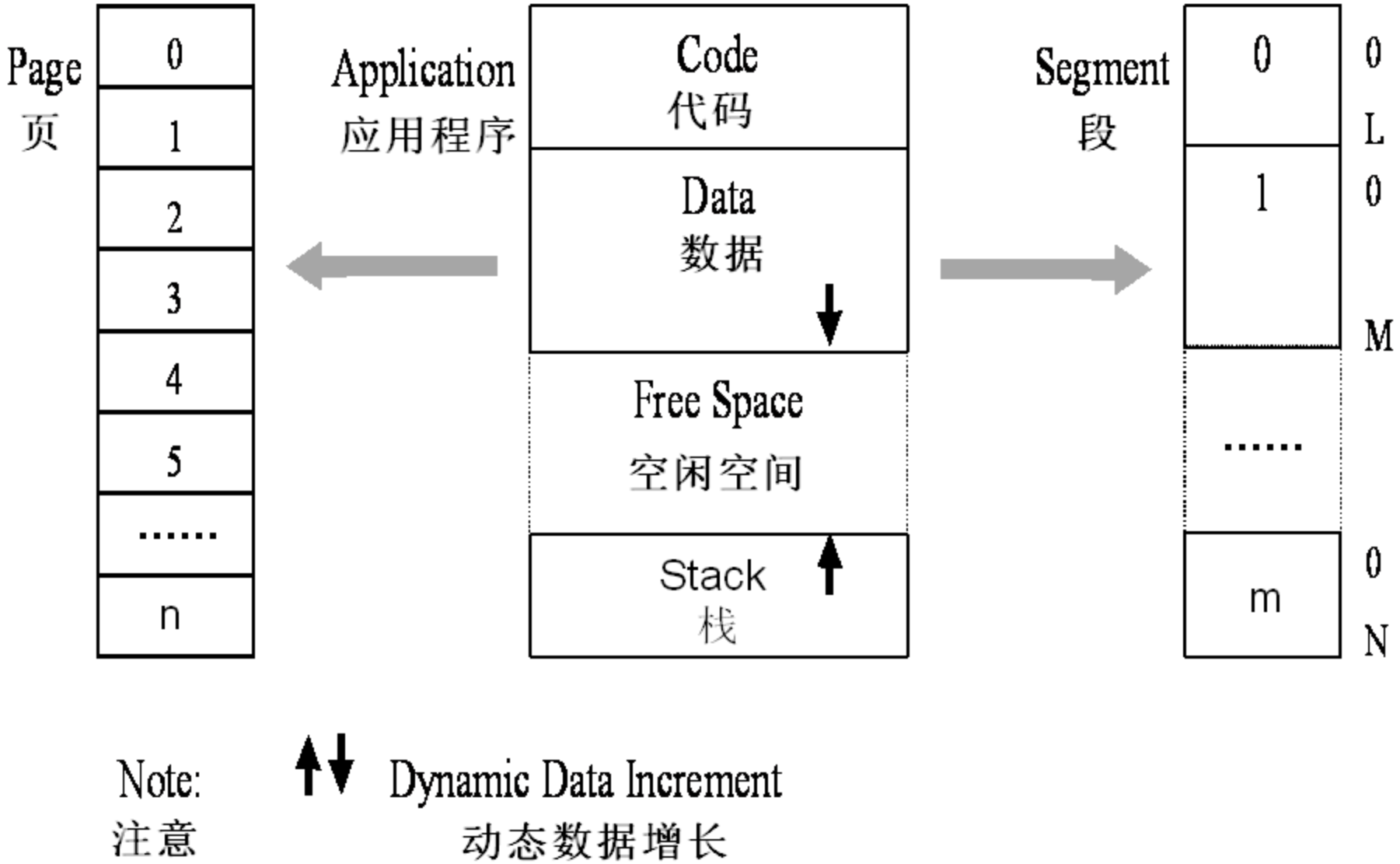
动态分区管理和段式管理的比较

- 都存在外碎片，但分段中可通过减少段长来减轻外碎片浪费程度
- 分段中一个进程可占用多个“分区”
- 分段中不要求一个进程占用的多个“分区”连续（但一般要求一个段所占用的多个“分区”连续）

动态分区管理和段式管理的比较

- 分段克服了分页存在的问题（数据结构的动态增长、动态链接、保护和共享）
- 分段存在外碎片，分页只有小的内碎片，分页内存利用率比分段高

页式管理和段式管理的比较



Note: 注意

Dynamic Data Increment 动态数据增长

问答