



## 第四章 存储器管理

- 4.1 存储器的层次结构
- 4.2 程序的装入和链接
- 4.3 连续分配存储管理方式
- 4.4 分页存储管理方式
- 4.5 分段存储管理方式

韩明峰      QQ 2022446359

## 4.1 存储器的层次结构

### 4.1.1 多层结构的存储器系统

#### 1. 存储器的多层结构

(1) 对于通用计算机而言，存储层次至少应具有三级：最高层为CPU寄存器，中间为主存，最底层是辅存。

(2) 在较高档的计算机中，还可以根据具体的功能分工细划为寄存器、高速缓存、主存储器、磁盘缓存、固定磁盘、可移动存储介质等6层。

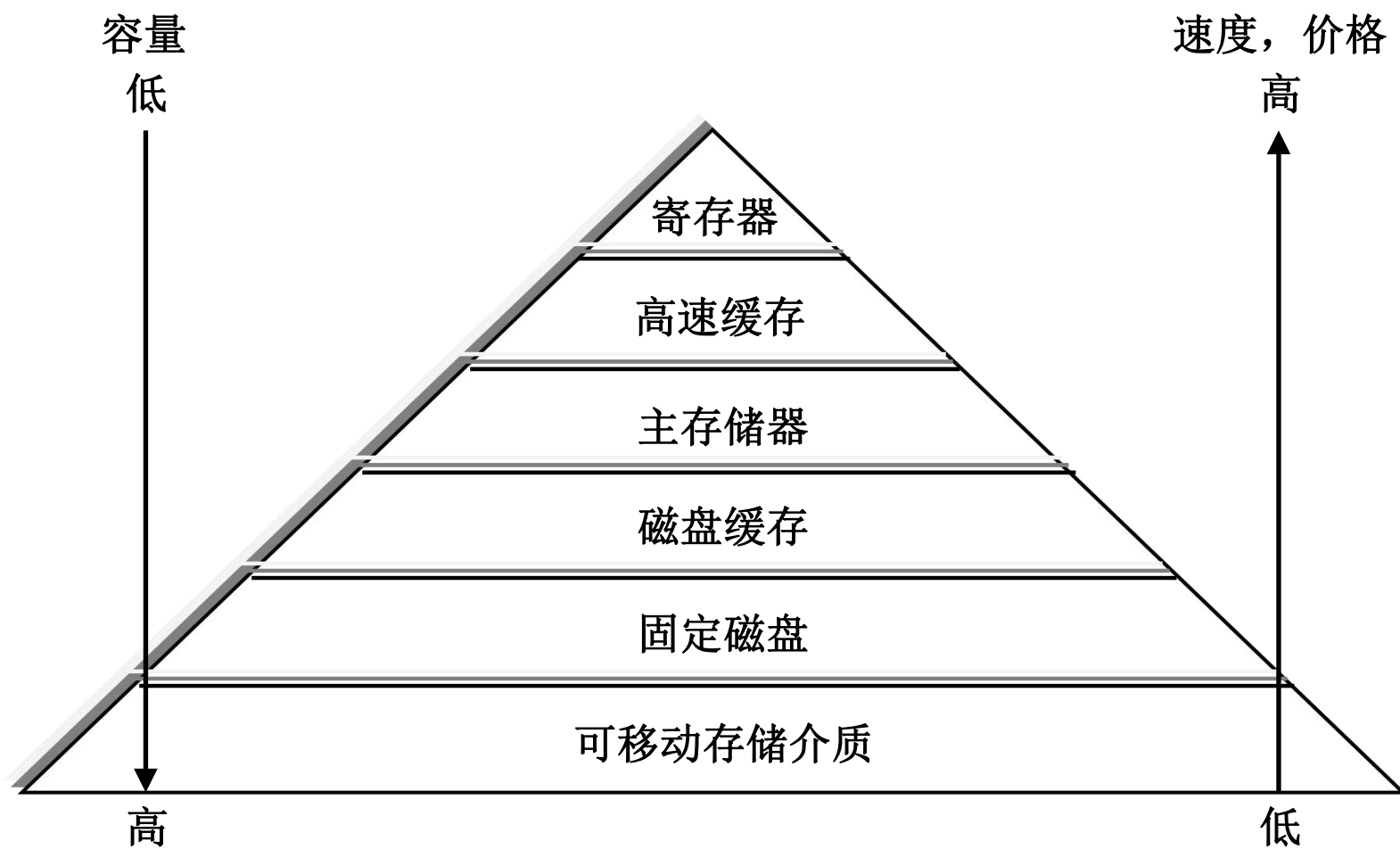


图4-1 计算机系统存储层次示意



## 2. 可执行存储器

(1) 寄存器和主存储器又被称为**可执行存储器**，可被CPU直接访问。

(2) 对辅存的访问则需要通过I/O设备实现，因此，在访问中将涉及到中断、设备驱动程序以及物理设备的运行，所需耗费的时间远远高于访问可执行存储器的时间，一般相差3个数量级甚至更多。



## 4.1.2 主存储器与寄存器

### 1. 主存储器

用于保存进程运行时的程序与数据。

### 2. 寄存器

由于主存储器访问速度远低于CPU执行指令的速度，为缓和这一矛盾，在计算机系统中引入了寄存器和高速缓存。

寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，但价格却十分昂贵，因此容量不可能做得很大。



### 4.1.3 高速缓存和磁盘缓存

#### 1. 高速缓存

主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数。

#### 2. 磁盘缓存

主要用于暂时存放频繁使用的一部分磁盘数据和信息，以减少访问磁盘的次数。但磁盘缓存与高速缓存不同，它本身并不是一种实际存在的存储器，而是利用主存中的部分存储空间暂时存放从磁盘中读出(或写入)的信息。

## 4.2 程序的装入和链接

用户程序要在系统中运行，通常都要经过以下几个步骤：

- (1) **编译**，由编译程序(Compiler)对用户源程序进行编译，形成若干个目标模块(Object Module)；
- (2) **链接**（静态链接），由链接程序(Linker)将编译后形成的一组目标模块以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；
- (3) **装入**，由装入程序(Loader)将装入模块装入内存。

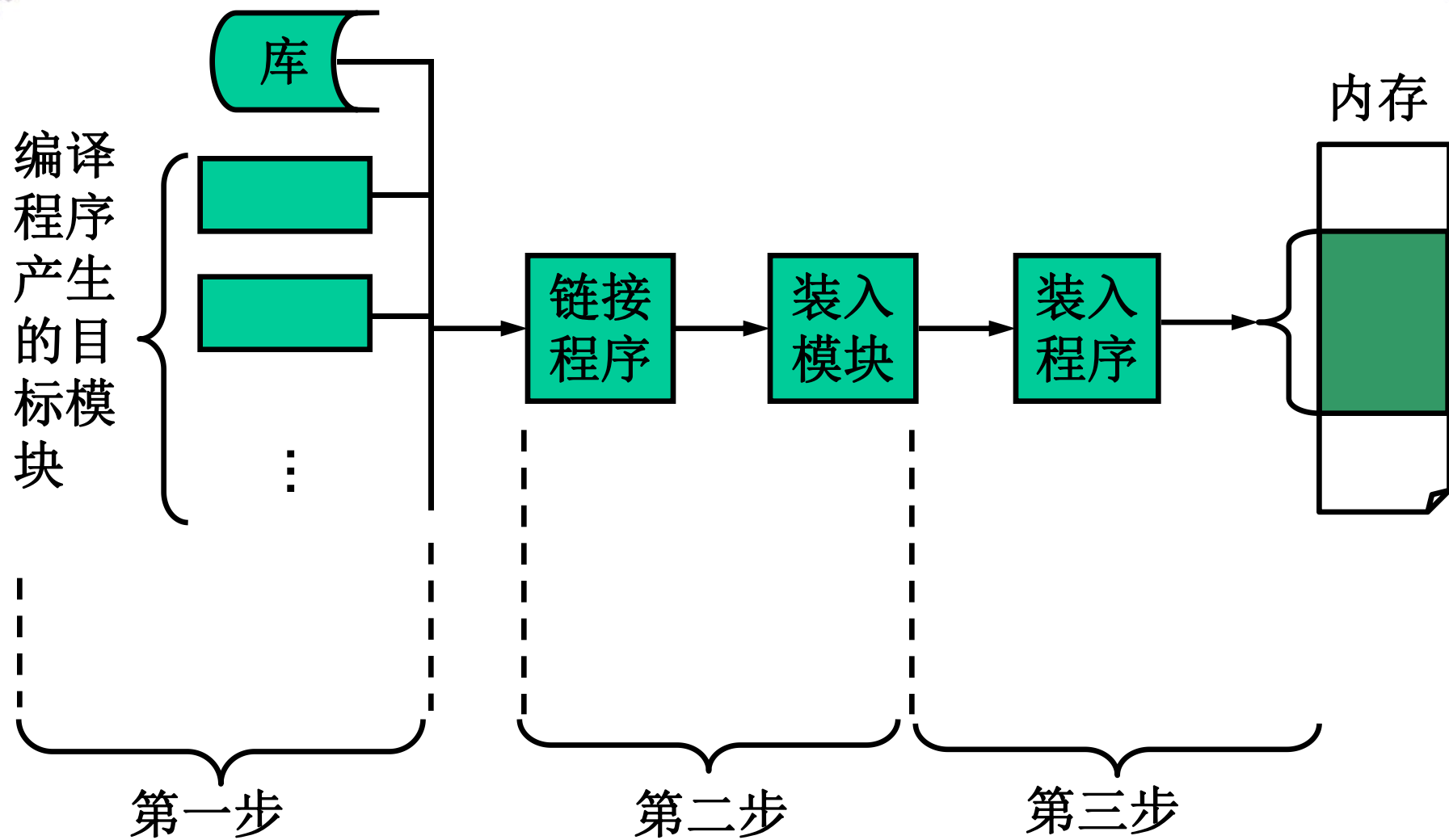


图4-2 对用户程序的处理步骤





## 4.2.1 程序的装入

可以有如下三种装入方式：

### 1. 绝对装入方式

用户程序经编译后，将产生**绝对地址**（即物理地址）的目标代码。

### 2. 静态重定位装入方式

地址变换在进程装入内存时一次完成。

### 3. 动态重定位的装入方式

地址变换推迟到**程序运行时**进行。

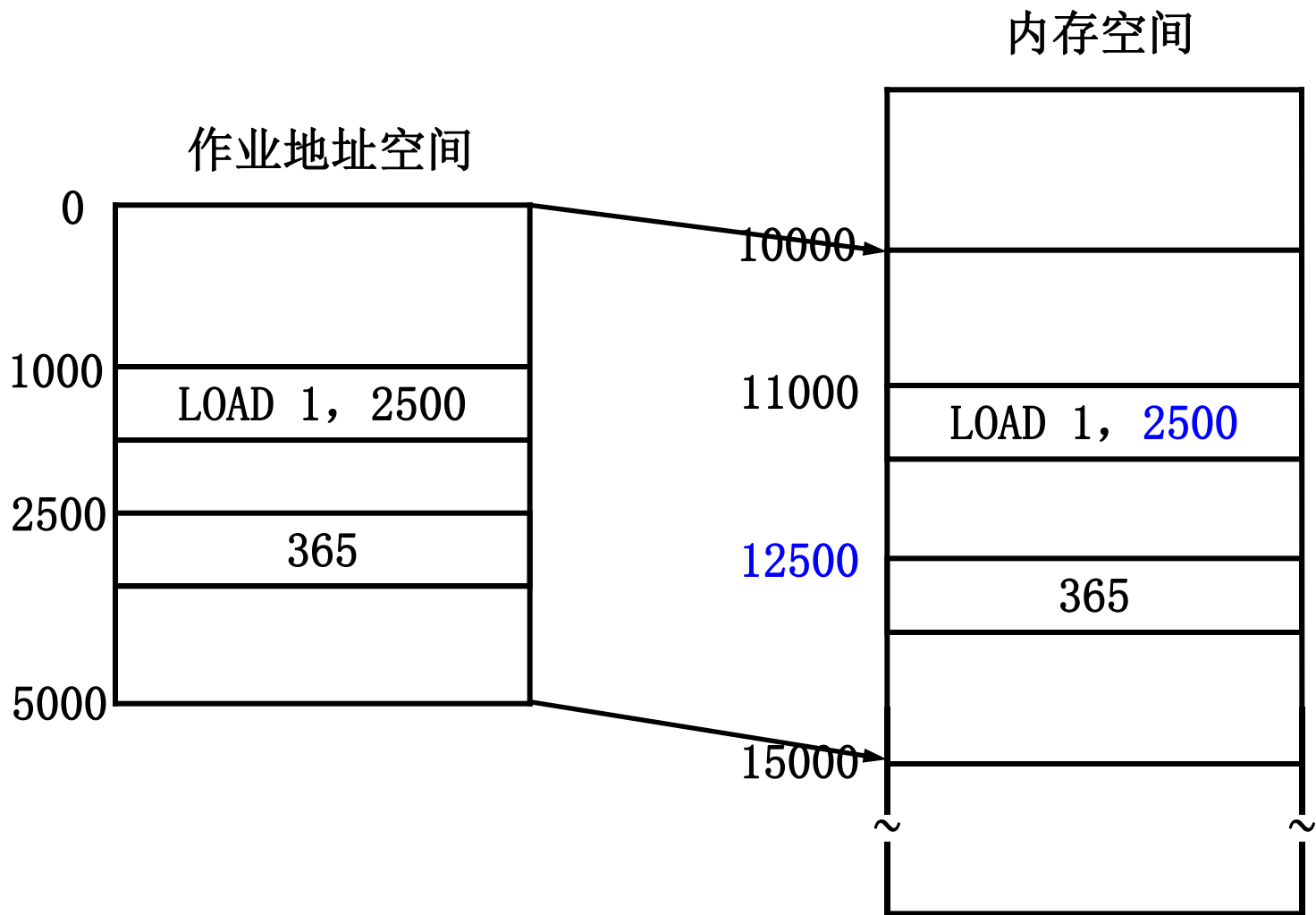


图4-3 作业装入内存时的情况（逻辑地址与物理地址不同）



## 4.2.2 程序的链接

### 1. 静态链接(Static Linking)方式

在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。须解决以下两个问题：

- (1) 对相对地址进行修改。
- (2) 变换外部调用符号。

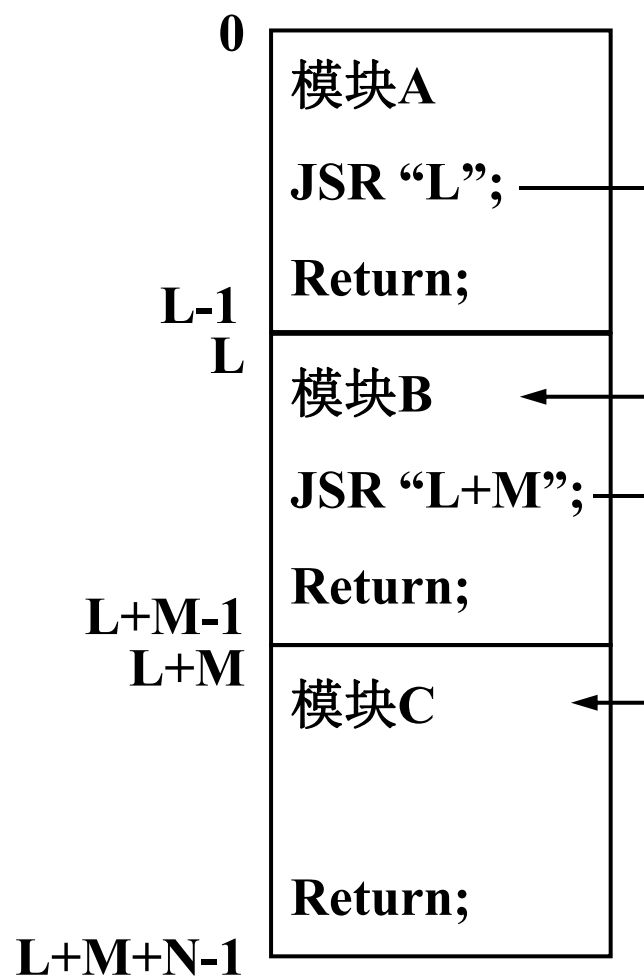
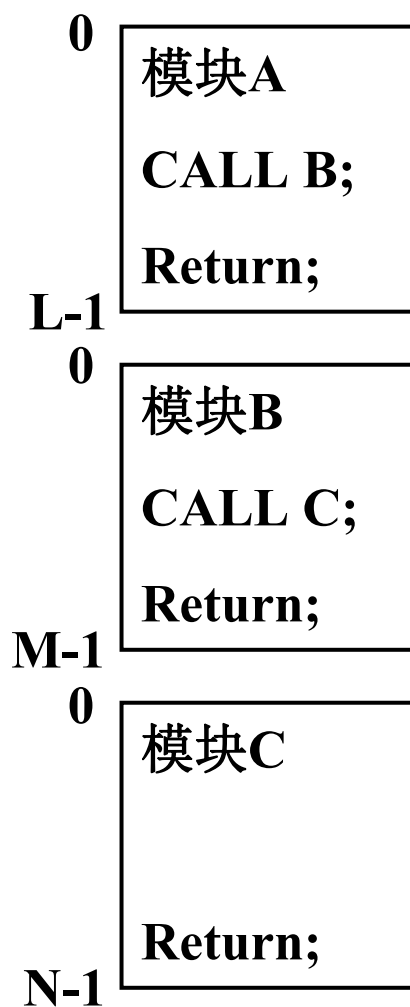


图4-4 程序链接示意图



## 2. 装入时动态链接 (Load-time Dynamic Linking)

即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照图4-4所示的方式修改目标模块中的相对地址。

## 3. 运行时动态链接 (Run-time Dynamic Linking)

对某些模块的链接推迟到[程序执行时](#)才进行。

## 4.3 连续分配存储管理方式

连续分配方式，是指为一个用户程序分配一个连续的内存空间。包括：

- (1) 单一连续分配
- (2) 固定分区分配
- (3) 可变分区分配
- (4) 动态重定位可变分区分配

### 4.3.1 单一连续分配



仅装有一道用户程序



## 4.3.2 固定分区分配

划分分区的方法：

(1) 分区大小相等。只适合于多个相同程序的并发执行，缺乏灵活性。

(2) 分区大小不等。多个小分区，适量的中等分区、少量的大分区。





### 4.3.3 基于顺序搜索的可变分区分配算法

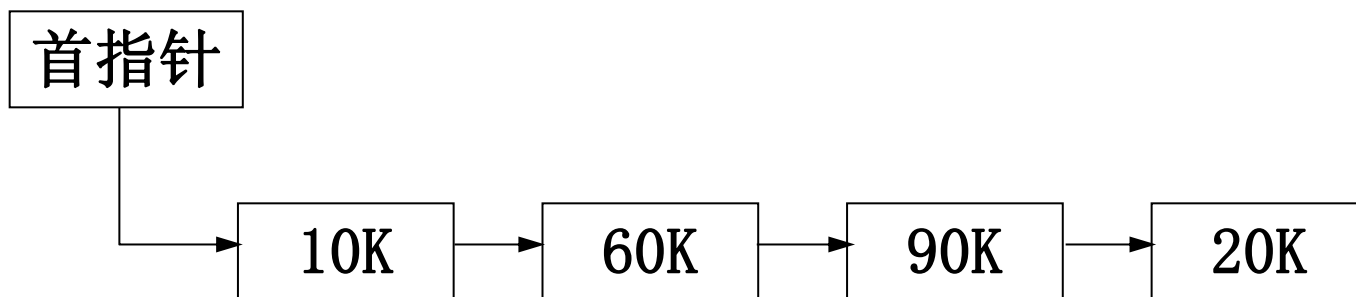
#### 1. 首次适应(first fit, FF)算法

FF算法要求空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止。

然后再按照作业的大小，从该分区中划出一块内存空间，分配给请求者，余下的空闲分区仍留在空闲链中。

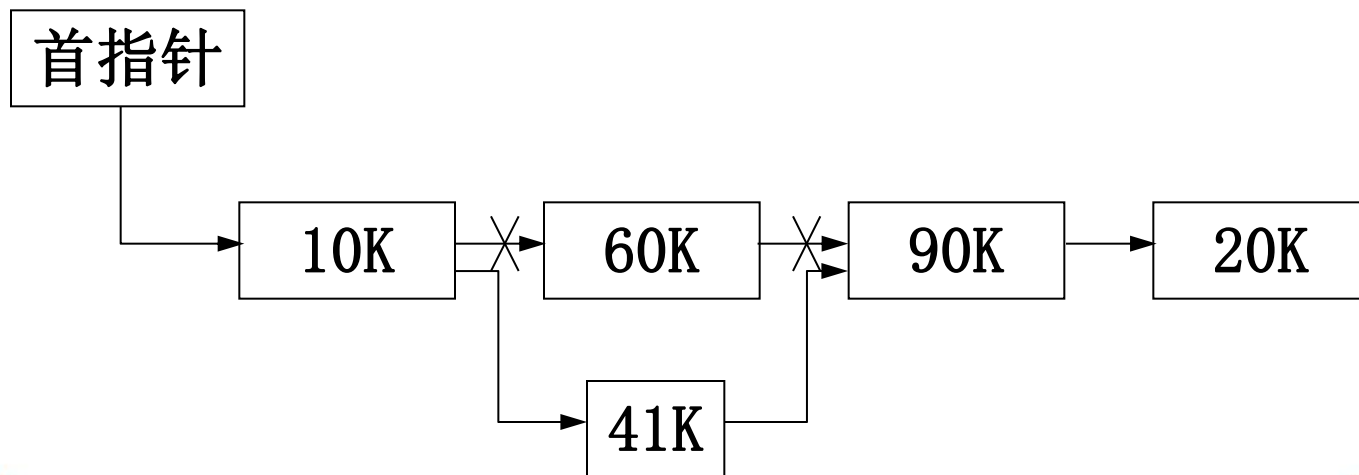
若从链首直至链尾都不能找到一个能满足要求的分区，则表明系统中已没有足够大的内存分配给该进程，内存分配失败，返回。

例：



有四块空白区 (从低地址到高地址)，来了一个作业需分配19k内存。

解：





## 2. 循环首次适应(next fit, NF)算法

为避免低址部分留下许多很小的空闲分区，以及减少查找可用空闲分区的开销，循环首次适应算法在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。



### 3. 最佳适应(best fit, BF)算法

所谓“最佳”是指，每次为作业分配内存时，总是把既能满足要求，又是最小的空闲分区分配给作业，避免“大材小用”。

为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。

例：

首指针

10K

20K

60K

90K

有四块空白区(从小容量到大容量)，来了一个作业需分配19k内存。

解：

首指针

10K

20K

60K

90K

1K

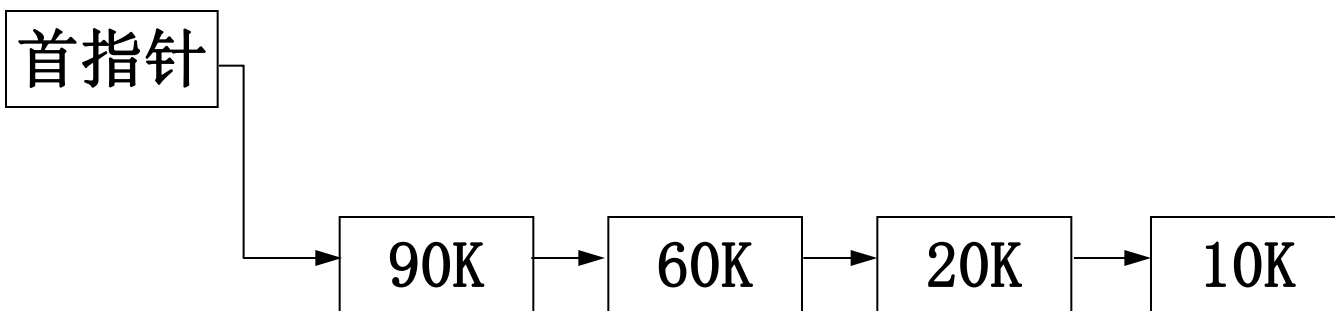
再排序

#### 4. 最坏适应(worst fit, WF)算法

在扫描整个空闲分区表或链表时，总是挑选一个最大的空闲区，从中分割一部分存储空间给作业使用，以至于存储器中缺乏大的空闲分区，故把它称为是最坏适应算法。

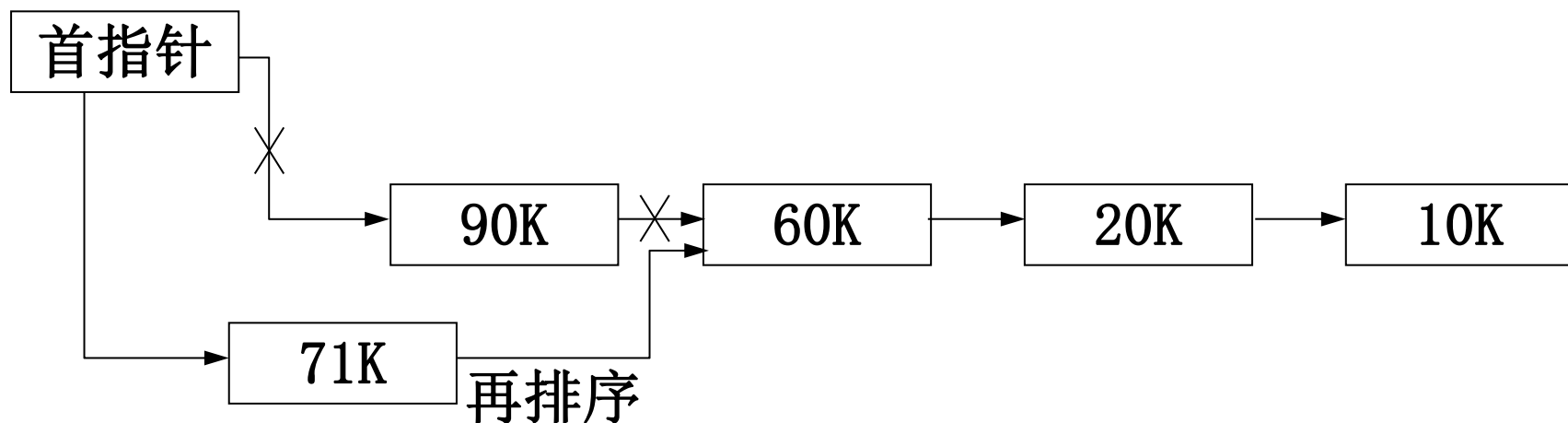
要求空闲分区链按其容量从大到小排列。

例：



有四块空闲区（从大容量到小容量），来了一个作业需分配19k内存。

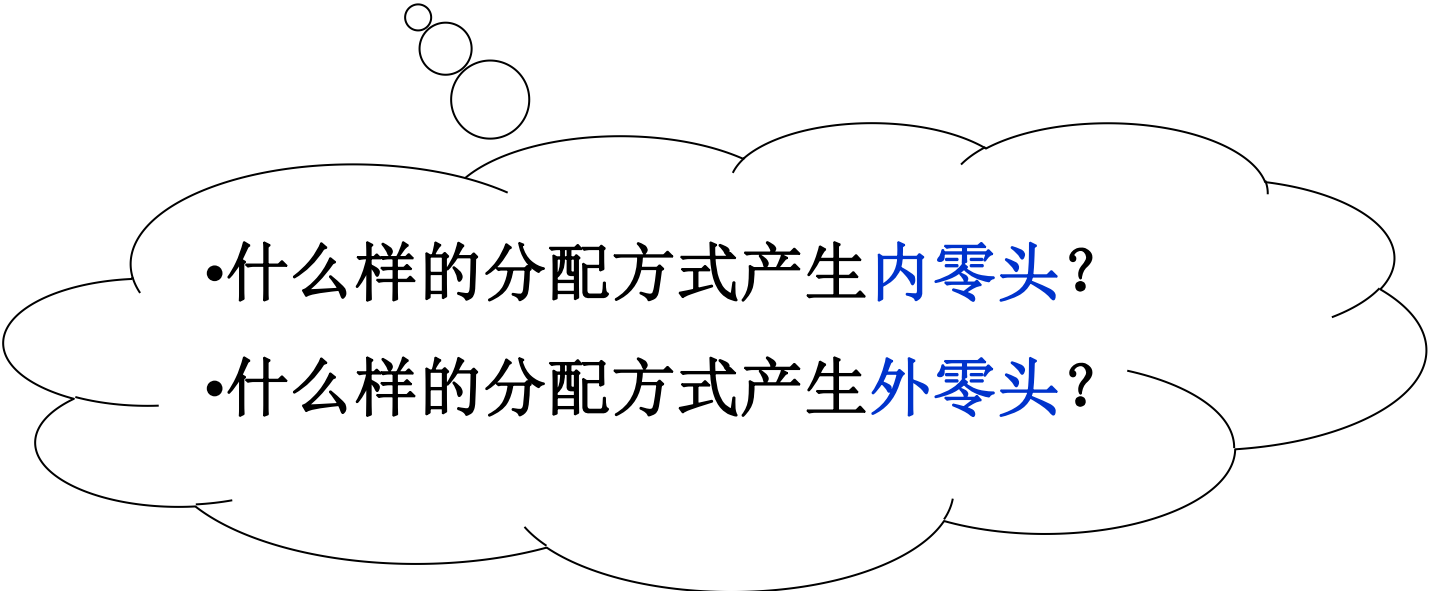
解：



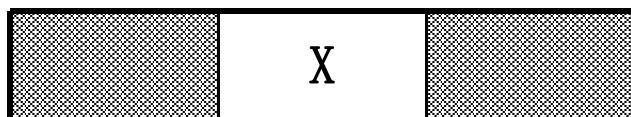
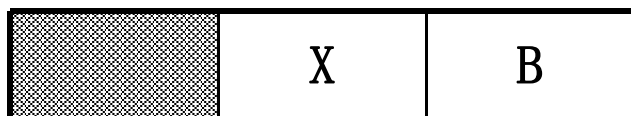
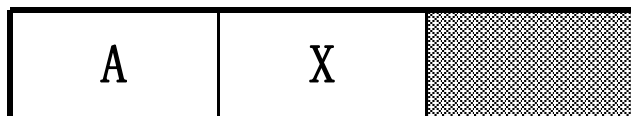
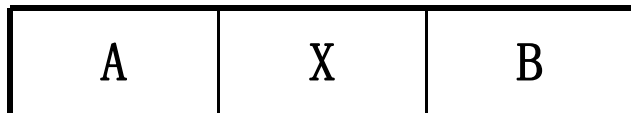


**内零头:**分配给用户而未被利用的部分，称作存储器的“内零头” (Internal Fragmentation)。

**外零头:**存储空间中小的无用的分区称为“外零头” (External Fragmentation)。

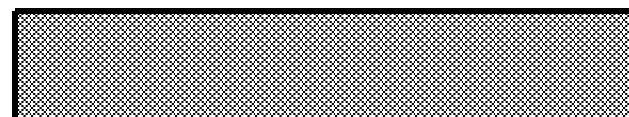
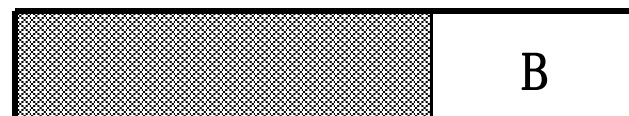
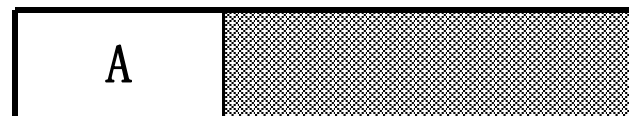
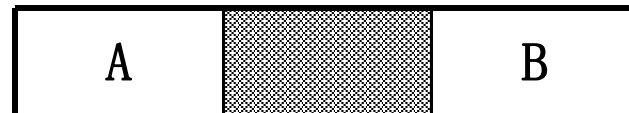
- 
- 什么样的分配方式产生**内零头**？
  - 什么样的分配方式产生**外零头**？





X释放前

空闲



X释放后

可变分区的回收算法图示



## 4.3.4 动态重定位分区分配

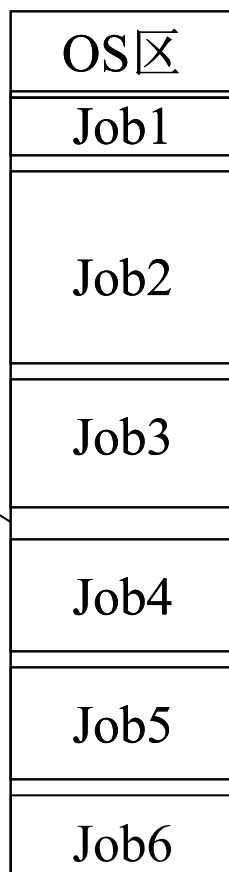
### 1. 紧凑

随着系统接收的作业的增加，内存中连续的大块分区将不存在，产生了大量的“碎片”。

**问题:**新的作业无法装入到每个“碎片”小分区上运行，但所有碎片的空间总和可能大于需求。

**解决方案:**通过移动内存中作业的位置，把原来多个分散的小分区“拼接”成一个大分区的方法，称为“拼接”或“紧凑”。

“零头”碎片



Job7

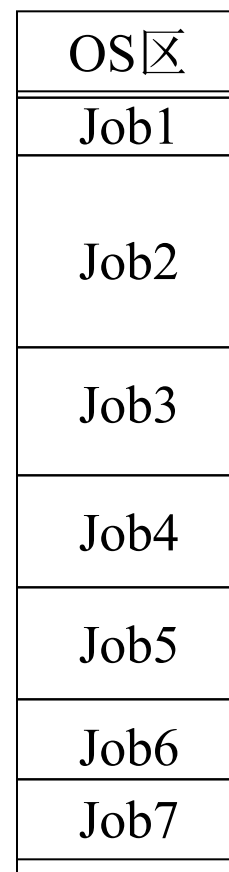


图4-11 紧凑的示意图



## 2. 动态重定位

(1) 动态重定位：相对地址转换为绝对(物理)地址的工作被**推迟**到程序指令要真正**执行时**进行。

(2) 实现：在系统中增设一个**重定位寄存器**，用它来存放程序(数据)在内存中的起始地址。

程序在执行时，真正访问的内存地址是相对地址与重定位寄存器中的地址相加而形成的。

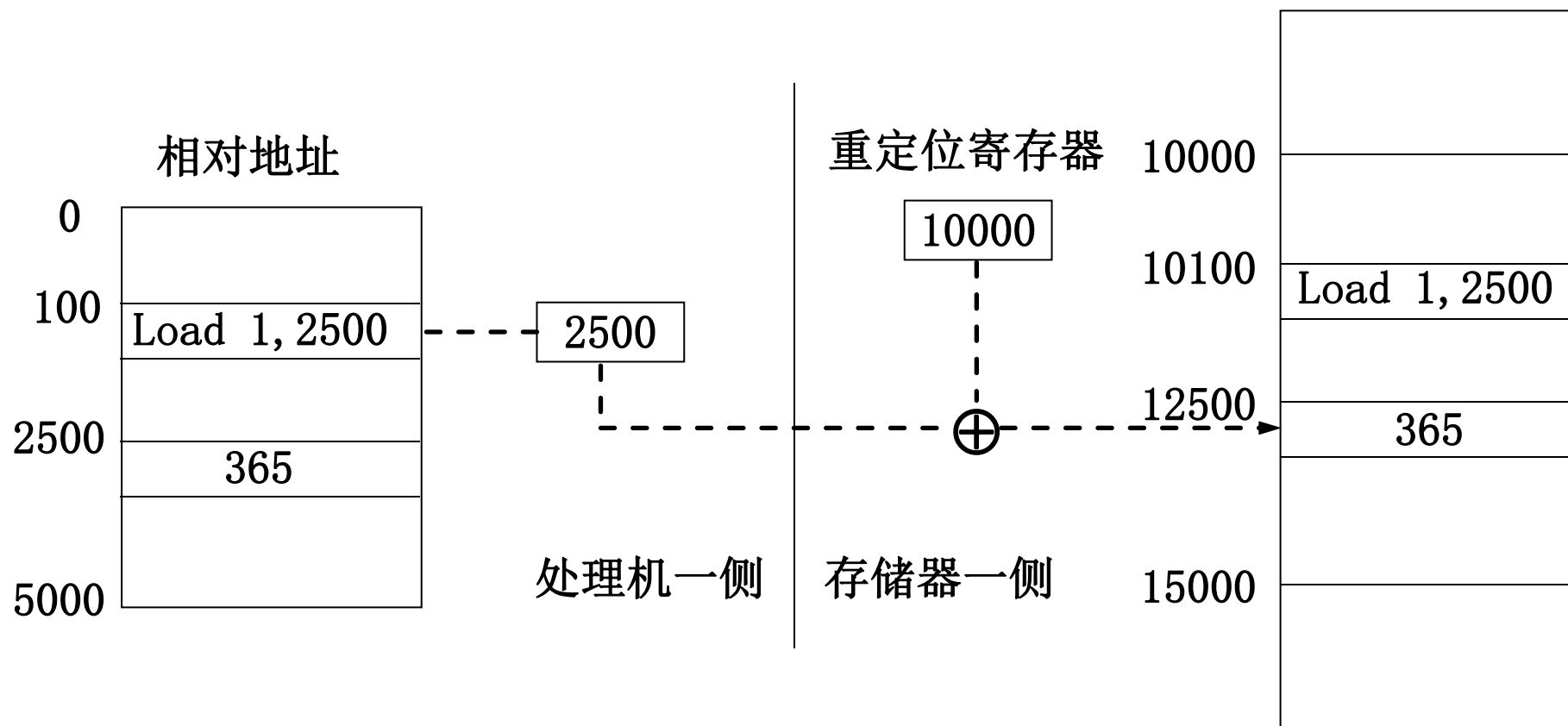


图4-12 动态重定位示意图



### 3. 动态重定位可变分区分配算法

动态重定位可变分区分配算法与可变分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了**紧凑**的功能。

## 4.4 分页存储管理方式

连续分配方式会形成许多“碎片”，通过“紧凑”方法将碎片拼接成可用的大块空间，但须为此付出很大开销。

如果允许将一个进程直接分散地装入许多不相邻的分区中，则无需“紧凑”，由此产生离散分配方式：

- (1) 分页存储管理方式。
- (2) 分段存储管理方式。
- (3) 段页式存储管理方式。

## 4.4.1 分页存储管理的基本方法

### 1. 页面和物理块

**页面：** 将一个进程的逻辑地址空间分成若干个大小相等的页。

**物理块：** 把内存空间分成与页面相同大小的若干个块。

**页面大小：** 页面若太小，虽然可使页内碎片减小，但也会使每个进程占用较多的页面，影响页面换出换进的速度。

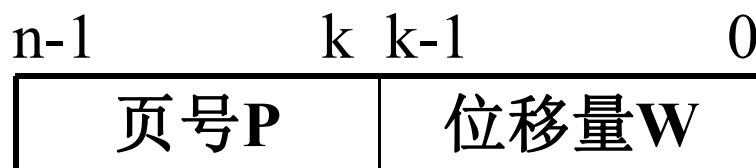
页面若太大，可以减少页表的长度，提高页面换出换进的速度，却会使页面内碎片增大。

因此，页面大小应是 $2^n$ ，通常为1KB—8KB。



## 2. 地址结构



逻辑地址



线性的逻辑地址长度为n位，包括两部分：

(1) **页号P**：n-k位，即：地址空间最多允许有 $2^{n-k}$ 页；

(2) **位移量W（页内位移量，即页内地址）**：k位，页内地址大小= $2^k$  Byte



若给定一个逻辑地址空间中的地址为A，页面大小为L，  
则可以由公式求出：

页号 $P=A/L$ ，

页内地址 $d=A \% L$ 。

例如：系统的页面大小L为1 KB， $A=2170$  B，

求出 $P=2170/1024=2$ ，

$d=2170\%1024=122$



### 3. 页表

页表实现逻辑页号到物理块号的映射。一个进程占用多少页，就在页表中有多少页表项。

用户程序

第0页
第1页
第2页
第3页
第4页
第5页
第6页

页号	块号
0	2
1	7
2	3
3	5
4	9
5	10
6	11

页表

内存

	0
	1
第0页	2
第2页	3
	4
第3页	5
	6
第1页	7
	8
第4页	9
第5页	10
第6页	11
	12

图4-14 页表的作用

## 4.4.2 地址变换流程

由于页内地址和物理地址一一对应，所以地址变换流程只是将逻辑地址中的**页号**，转换为内存中的物理地址的**物理块号**。

### 1. 基本的地址变换流程

(1) 进程切换时，将内存中的页表装入完全用硬件如一组专用的寄存器来实现的页表中。地址变换速度快，进程切换代价高，硬件成本高。

(2) 进程切换时，将PCB中的页表始址和页表长度装入**页表寄存器PTR**中。地址变换需访问位于内存中的页表，进程切换代价小，硬件成本低。

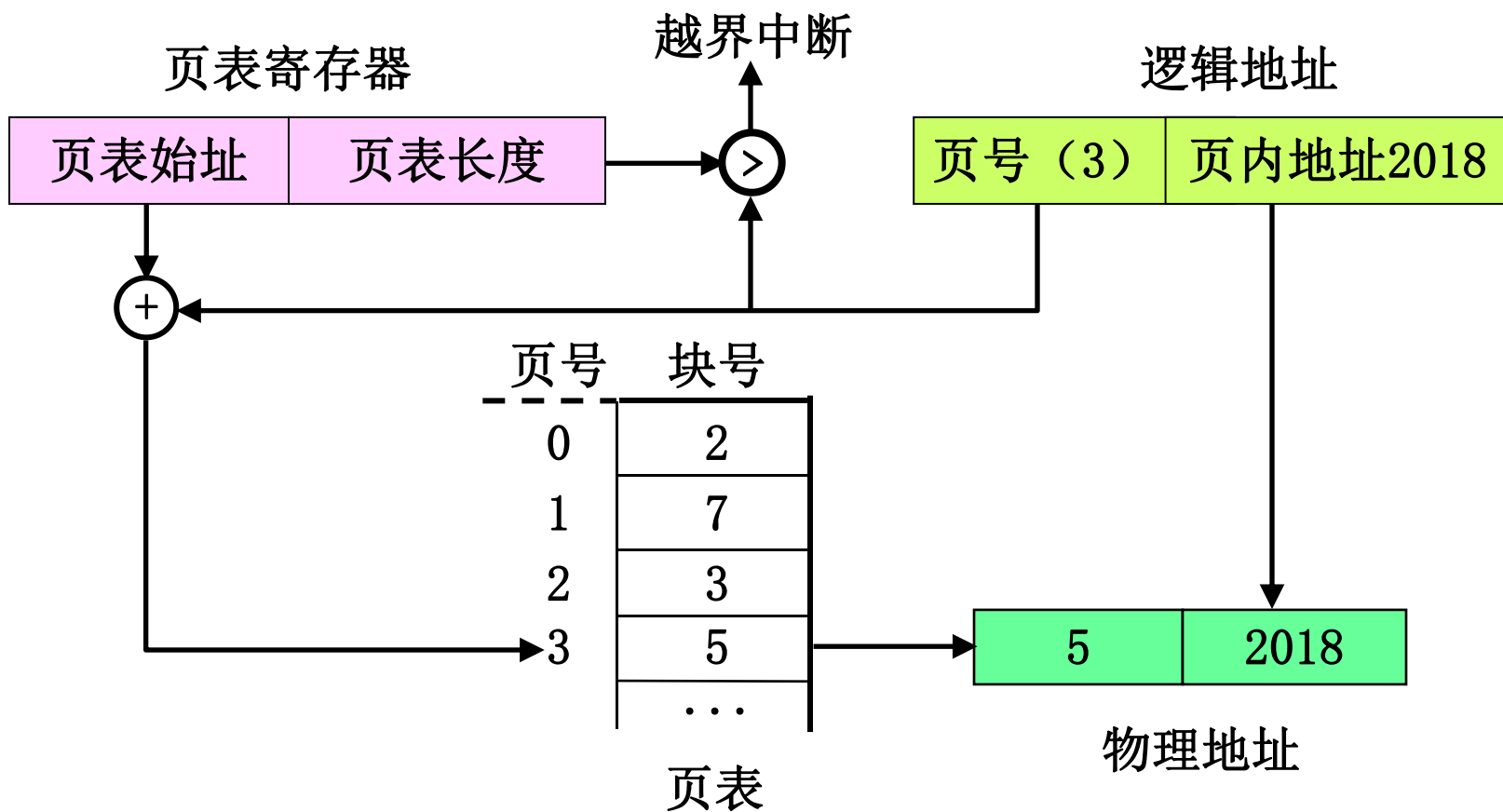




图4-15 分页系统的地址变换流程



【例4.1】某采用分页存储管理的系统中，物理地址占20位，逻辑地址中页号占6位，页面大小为1KB，请问：

(1) 该系统的内存空间大小为多少？

每个存储块的大小为多少？

逻辑地址共几位？

每个作业的最大长度为多少？

(2) 若第0、1、2页分别放在第3、7、9存储块中，则逻辑地址0420H对应的物理地址是多少？



解：

(1) 物理地址占20位，所以该系统的内存空间大小为：

$$2^{20}=1\text{MB} ;$$

存储块的大小与页面大小相同，而页面大小为1KB，因此存储块的大小为：1KB ；

由于页面大小为1KB，占10位，而页号占6位，因此逻辑地址共：10+6=16位；

逻辑地址共16位，从而该系统中的每个作业大小为：

$$2^{16}=64\text{KB} 。$$





(2) 逻辑地址到物理地址转换的计算方法：

十进制计算方法：

① 将逻辑地址转换成10进制：  $0420H=1056$

页号：  $P=1056/1024=1$

位移量：  $W=1056 \% 1024=32$

②查页表，1号页面对应7号物理块。

③每个物理块大小为1K，物理地址为：  $7 \times 1K + 32 = 7200$

## 二进制计算方法:

① 将逻辑地址转换为二进制:  $0420H = 0000\ 0100\ 0010\ 0000$

② 页面大小为1K, 所以位移量占10个二进制位, 将逻辑地址划分为页号和页内位移量两部分:  $0000\ 0100\ 0010\ 0000$

③ 把页号部分转换为十进制,  $(0000\ 01)_2 = 1$ ; 查找页表, 1号页对应7号物理块。

④ 将③中得到的块号转换为二进制, 并与②中划分得到的页内位移量拼接, 即得物理地址:  
 $0001\ 1100\ 0010\ 0000$ , 也可以转换为十六进制:  $1C20H$ , 或十进制7200。



## 2. 具有快表的地址变换流程

**提出改进的原因：** 计算机CPU存取数据需要访问内存两次，处理速度降低为1/2。

**改进方法：** 设置地址转换旁路缓冲器 (translation-lookaside buffer, TLB)，更好的名称应该是地址转换缓存 (address-translation **cache**)，也称快表，用相联存储器实现，并行查询。



## 2. 具有快表的地址变换流程

### 实现原理:

(1) 如在快表中找到相匹配的页号，直接从快表中读出对应物理块号；

(2) 如在快表中没有找到相匹配的页号，还需查找页表，从页表中找出物理块号，再将此页表项存入快表中，更新快表。

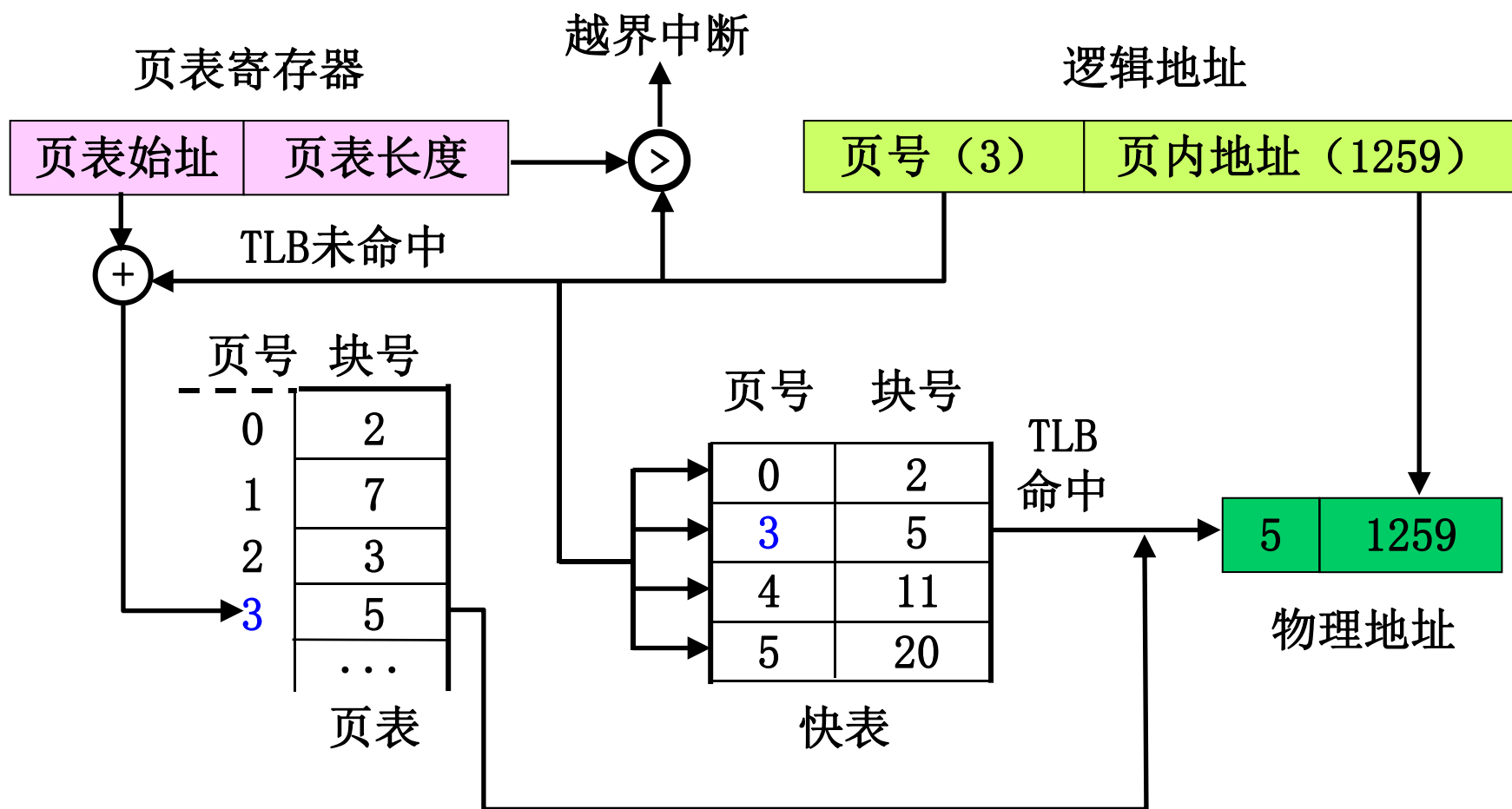


图4-16 具有快表的地址变换流程

【例4.2】某计算机主存地址空间大小为 256 MB，按字节编址。虚拟地址空间大小为 4 GB，采用页式存储管理，页面大小为 4 KB。TLB（快表）采用全相联映射，有 4 个页表项，内容如下表所示：

有效位	页号	块号	...
0	FF180H	0002H	...
1	3FFF1H	0035H	...
0	02FF3H	0351H	...
1	03FFFH	0153H	...

请问对虚拟地址 03FF F180H 进行虚实地址变换的结果是多少？

解：

(1) 页面大小为12位。

(2) 虚地址： $\boxed{03FF \ F}180H$

查快表

0153H

(3) 所查快表项有效，所以虚实地址变换的结果是  
**0153**180H

### 4.4.3 两级和多级页表

解决页表项非常大需连续存放的方法：

- (1) 采用离散方式。
- (2) 只将当前所需页表项调入内存，其余页表项驻留在外存，需要时再调入。

#### 1. 两级页表(Two-Level Page Table)

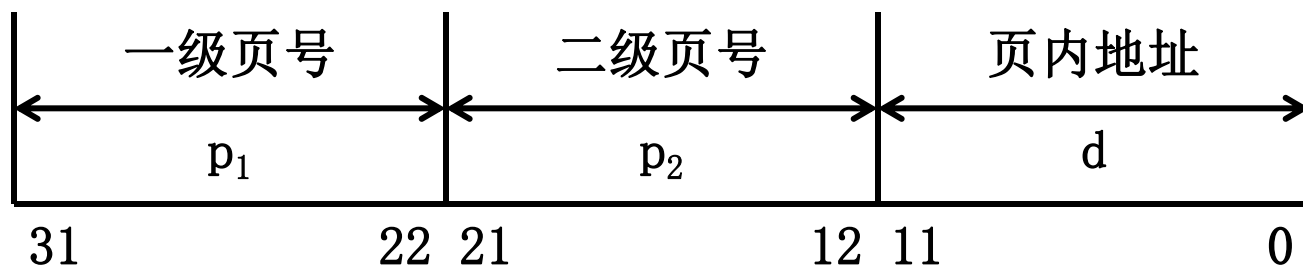
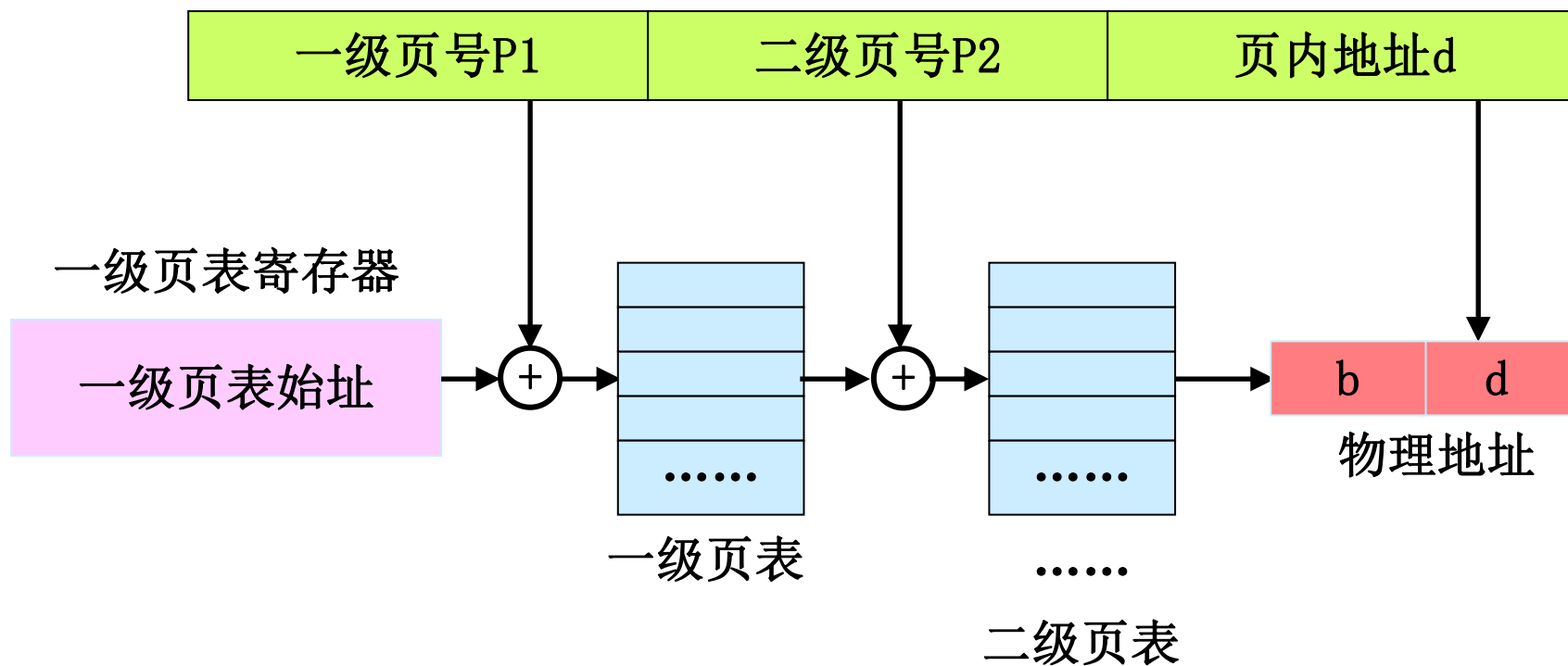


图4-17 两级页表结构



## 逻辑地址



具有两级页表的地址变换流程

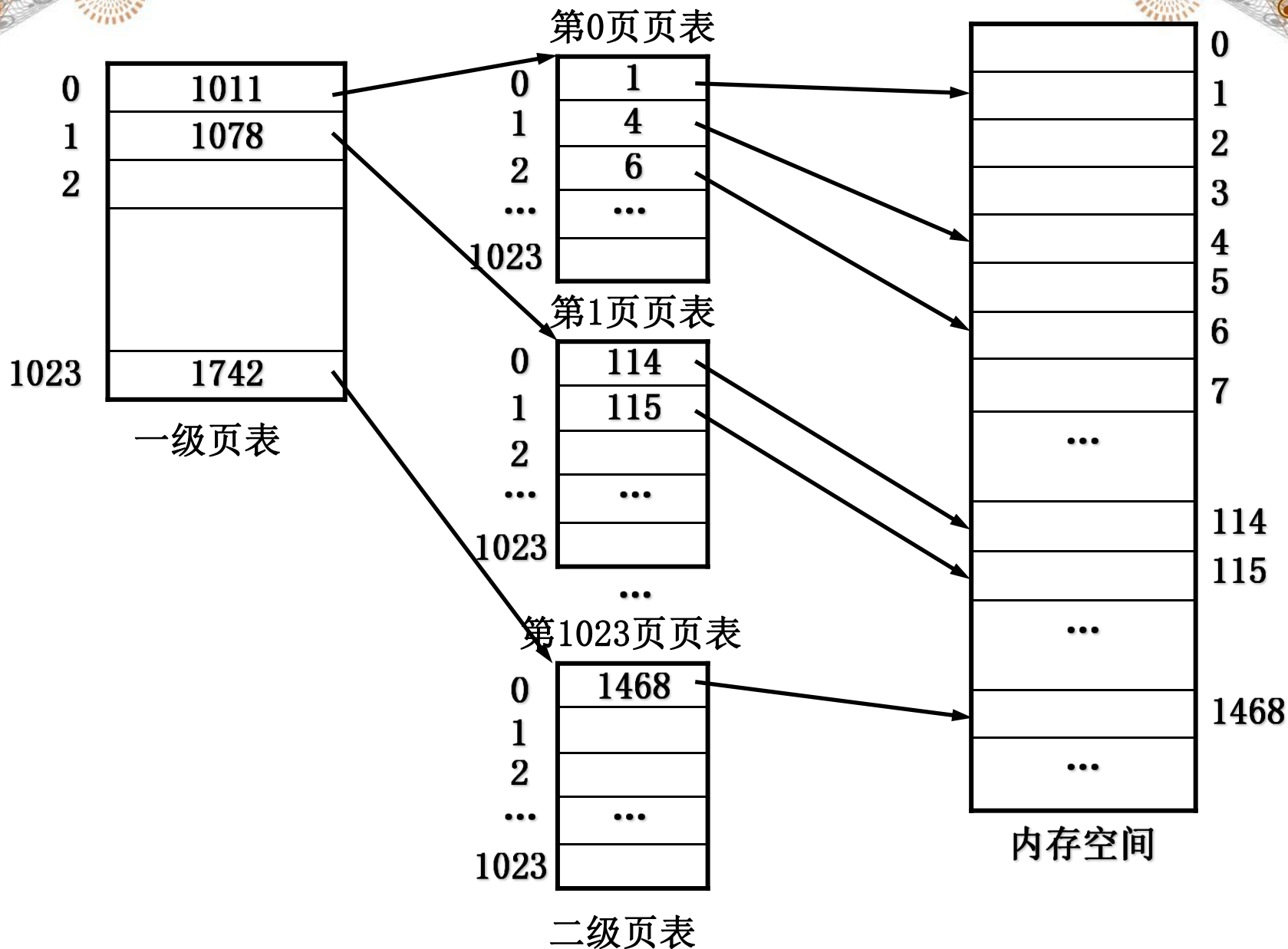



图4-18 两级页表



## 2. 多级页表

对于较大的逻辑地址空间，划分为一级页表、二级页表、三级页表等等。



【例4.3】考虑64位逻辑地址空间，8KB页面大小的内存系统，我们使用5级页表。每个进程第一级页表保存2K个页表项，剩下的四个级别的页表都会保存1K个页表项。

(1) 试给出系统中与这一多级页表结构相应的逻辑地址布局。

(2) 每个进程需要的总页表项数是多少？

(3) 如果采用单级页表，总页表项数是多少？



解：

(1) 每页的偏移地址位数是13，虚拟页表的位数是 $64 - 13 = 51$ ，第一级页表的位数是11，剩下四级页表的位数都是10。逻辑地址布局如下：

1级页表 11位	2级页表 10位	3级页表 10位	4级页表 10位	5级页表 10位	页内偏 移地址 13位
-------------	-------------	-------------	-------------	-------------	-------------------

(2) 每个进程需要的总页表项数是：

$$2^{11} + 2^{21} + 2^{31} + 2^{41} + 2^{51}$$

(3) 如果采用单级页表，总页表项数是 $2^{51}$ 。



#### 4.4.4 反向页表(Inverted Page Table)

##### 1. 反向页表的引入

为了减少页表占用的内存空间，为每一个物理块设置一个页表项，并按物理块的编号排序，表项内容是所隶属进程的标识符和页号。

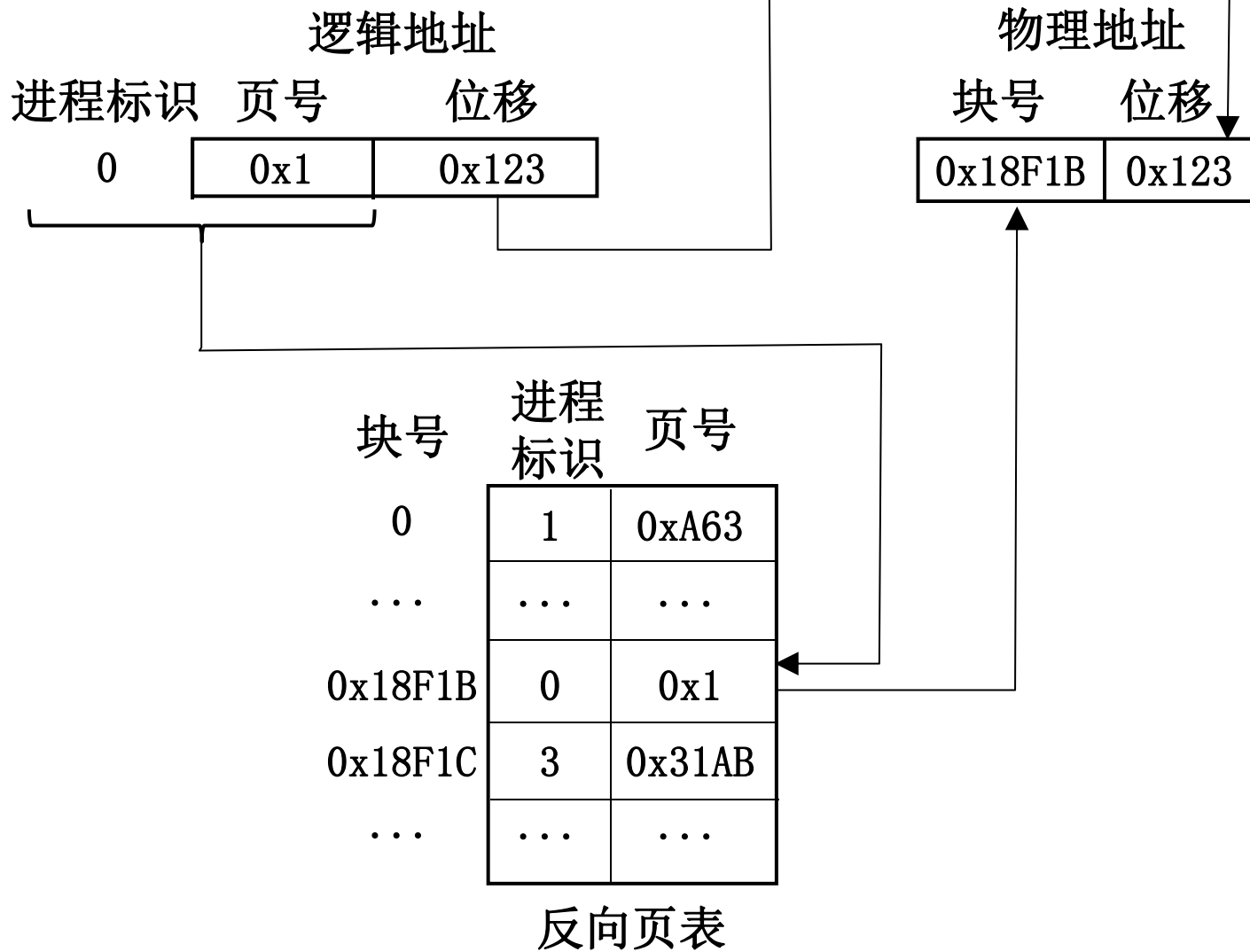
## 2. 地址变换流程

(1) 根据进程标识符和页号，去检索反向页表。如果检索到与之匹配的页表项，则该页表项的序号 $i$ 便是该页所在的物理块号，可用该块号与页内地址一起构成物理地址送内存地址寄存器。

(2) 若检索了整个反向页表仍未找到匹配的页表项，则表明此页尚未装入内存。

对于不具有请求调页功能的存储器管理系统，此时则表示地址出错；

对于具有请求调页功能的存储器管理系统，此时应产生请求调页中断，系统将把此页调入内存。



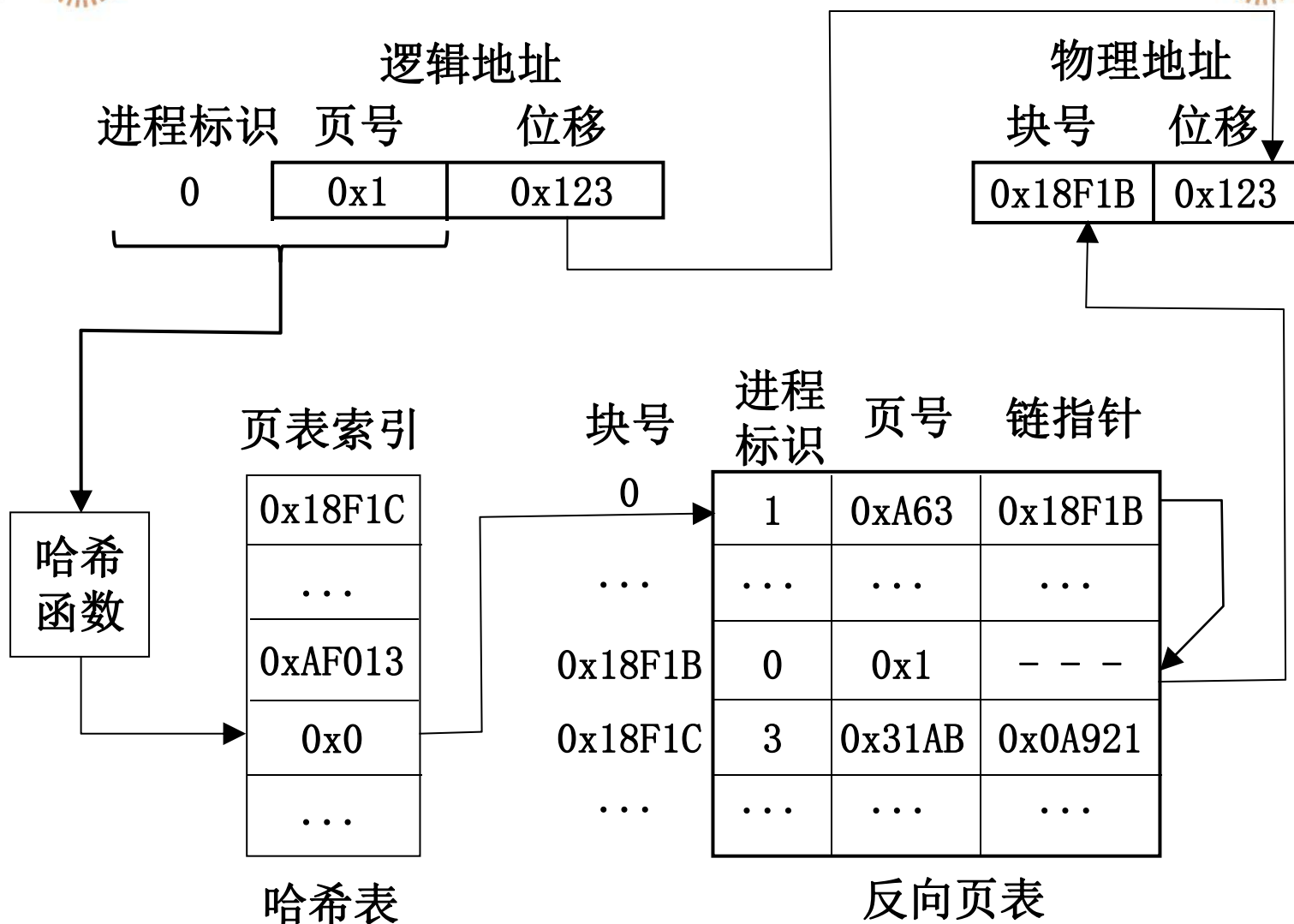
反向页表及其地址转换示意图





### 3. 哈希反向页表

可利用哈希函数来提高反向页表的查找速度。

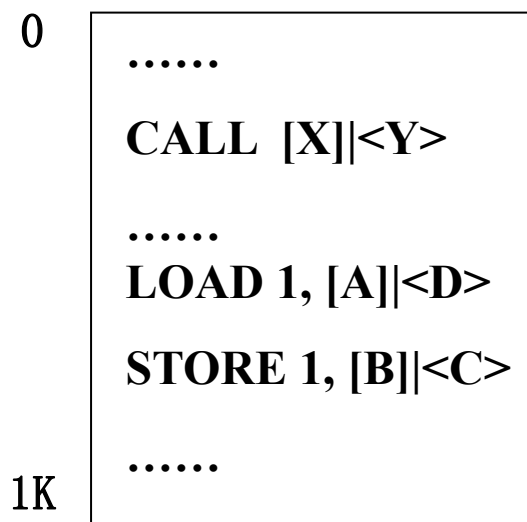


哈希反向页表及其地址转换示意图

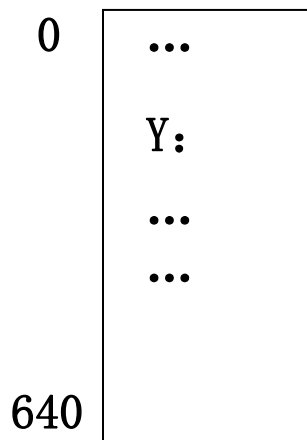
## 4.5 分段存储管理方式

### 4.5.1 分段存储管理方式的引入

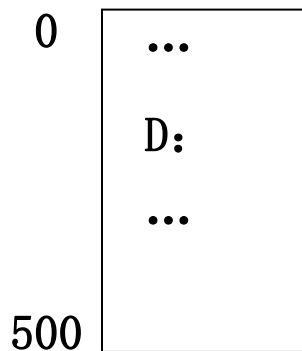
#### 1. 方便编程，整个作业的地址空间是二维的



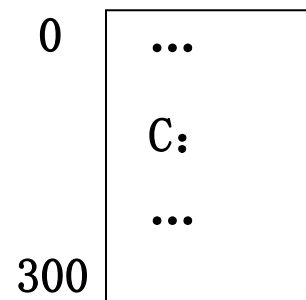
分段MAIN(主程序)



分段X(子程序)



分段A(数组)



分段B(工作区)



## 2. 信息共享

实现对程序和数据的共享是以信息的逻辑单位，如某个过程、函数或文件为基本单位的。

## 3. 信息保护



信息保护同样是以一个过程、函数或文件为基本单位的。

## 4. 动态增长

每个段都构成了一个独立的地址空间，可以独立地增长或减小而不会影响到其它的段。

## 5. 动态链接

在程序运行过程中，当需要调用某个目标程序(段)时，才将该段(目标程序)调入内存并进行链接。

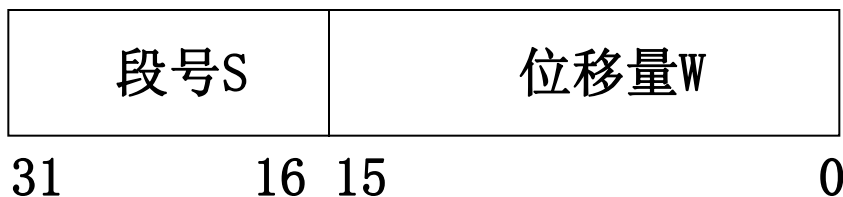


## 4.5.2 分段系统的基本原理

### 1. 分段

分段地址中的地址具有如下结构：

逻辑地址



### 2. 段表

每个段在表中占有一个表项，其中记录了该段在内存中的起始地址和段的长度。

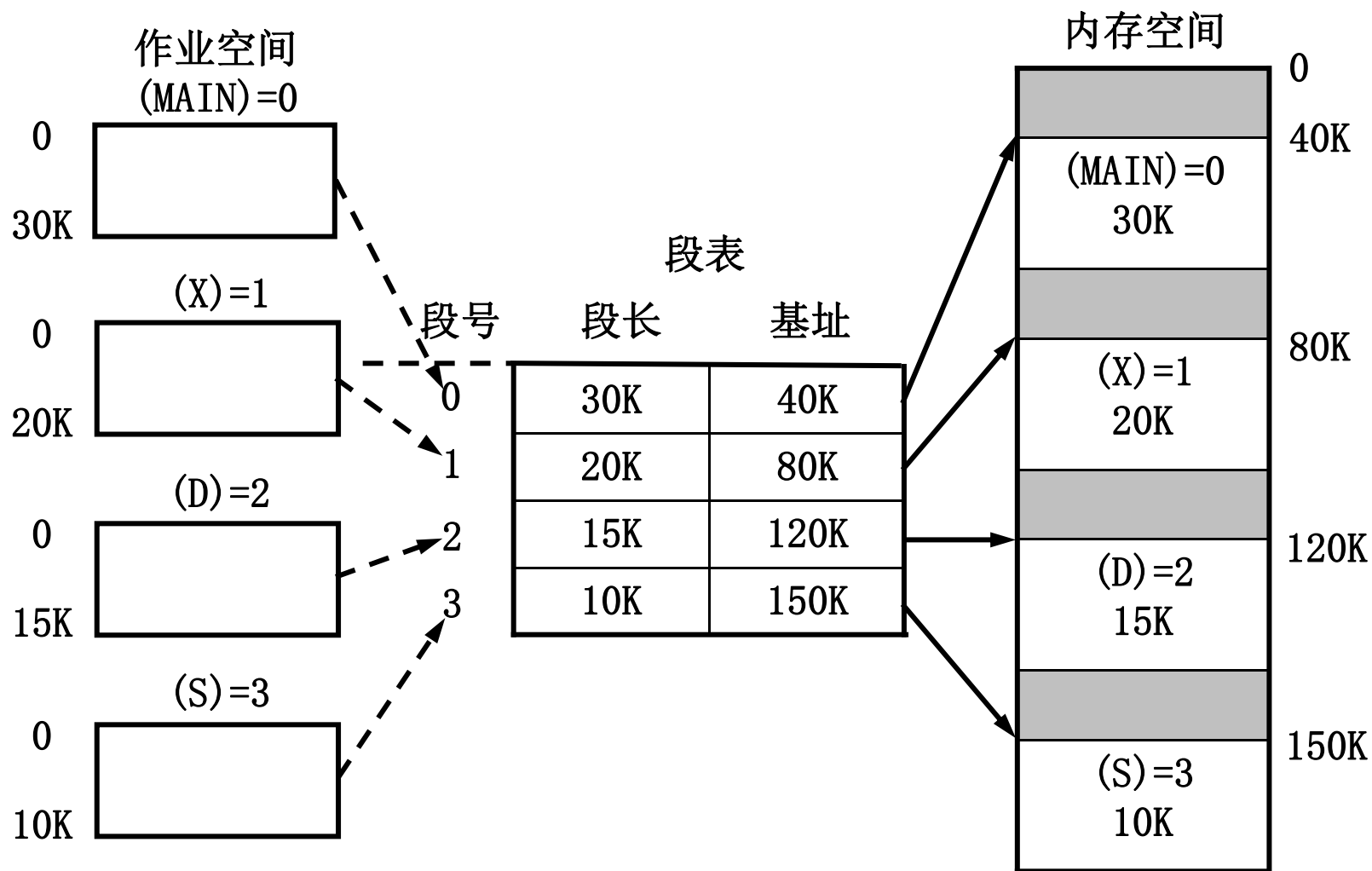


图4-19 利用段表实现地址映射



### 3. 地址变换流程

在系统中设置段表寄存器，用于存放段表始址和段表长度，以实现从进程的逻辑地址到物理地址的变换。

段表寄存器和段表还有存储保护的功能。

**问题：**当段表存放在内存中时，每访问一个数据，都需两次访问内存，降低了计算机的速率。

**解决方法：**设置TLB，用于保存最近常用的段表项。

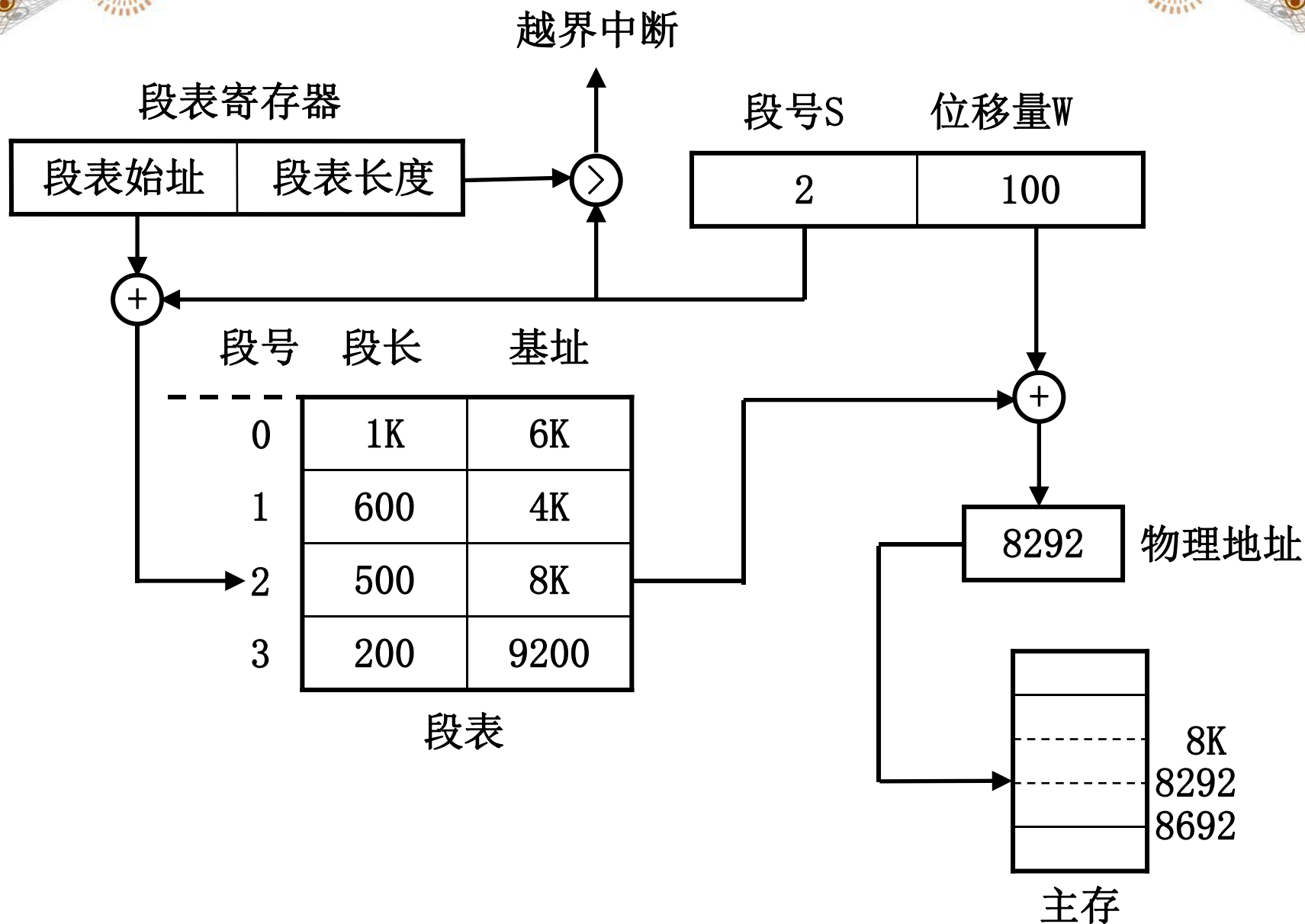


图4-20 分段系统的地址变换流程



#### 4. 分页和分段的主要区别

(1) 页是信息的**物理单位**，段是信息的**逻辑单位**，它含有一组其意义相对完整的信息。

(2) 页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面；

段的长度却不固定，决定于用户所编写的程序，通常由编辑程序在对源程序进行编辑时，根据信息的性质来划分。

(3) 分页的作业地址空间是一维的；分段的作业地址空间是二维的。



### 4.5.3 信息共享

#### 1. 分页系统中对程序和数据的共享

在分页系统中，虽然也能实现对程序和数据的共享，但远不如分段系统来得方便。

**举例：**假定在分页系统中，每个页面大小为4KB，160KB的代码将占用40个页面，数据区占10个页面。为了实现代码的共享，每个进程的页表中都建立50个页表项，比较浪费空间。

利用分页共享原理，上述多用户系统的存储分配如下（假定页面大小为4K，两个用户）：

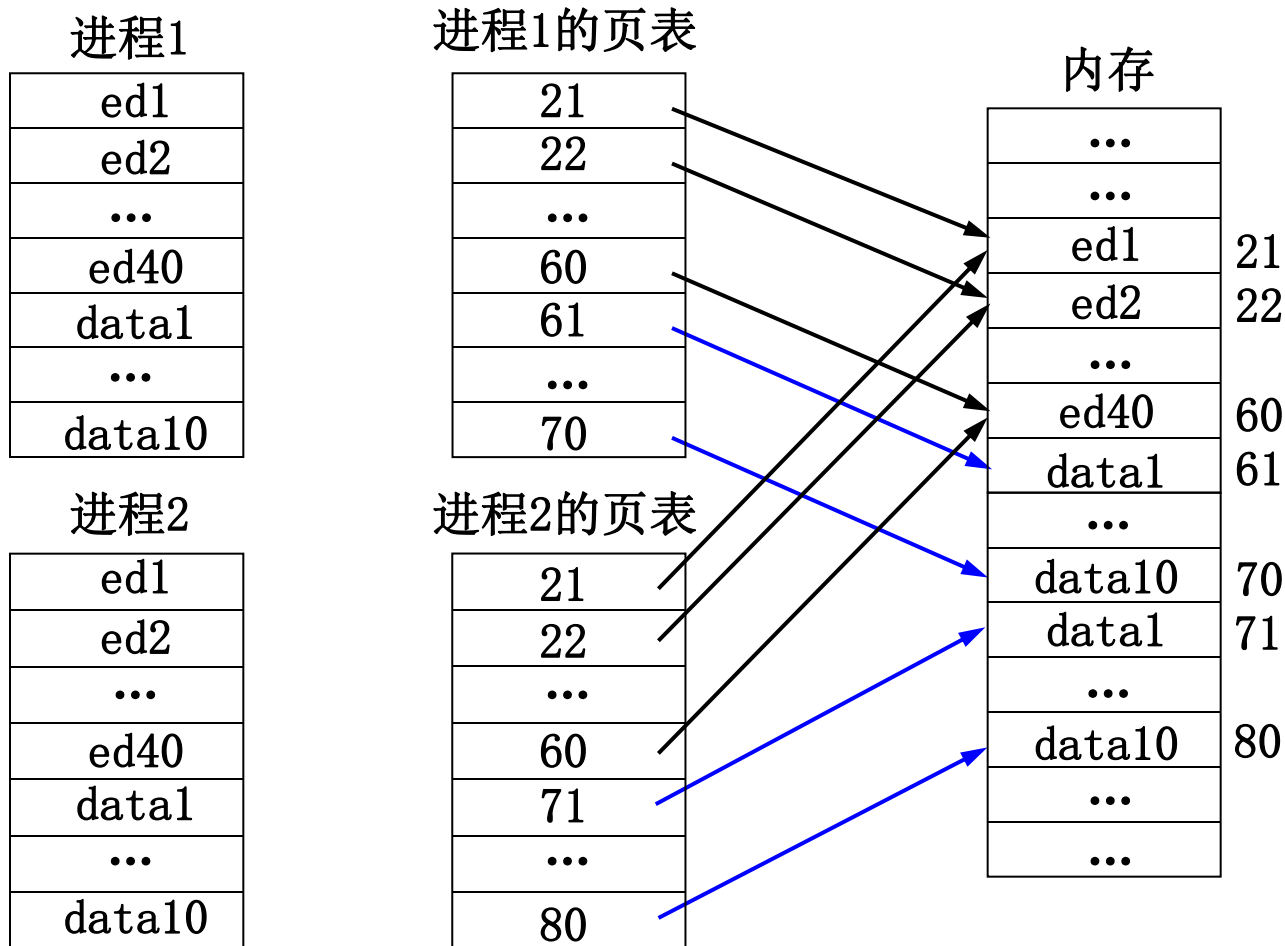


图4-21 分页系统中共享editor的示意图

## 2. 分段系统中程序和数据的共享

在分段系统中，实现共享十分容易，只需在每个进程的段表中为共享代码段设置一个段表项。

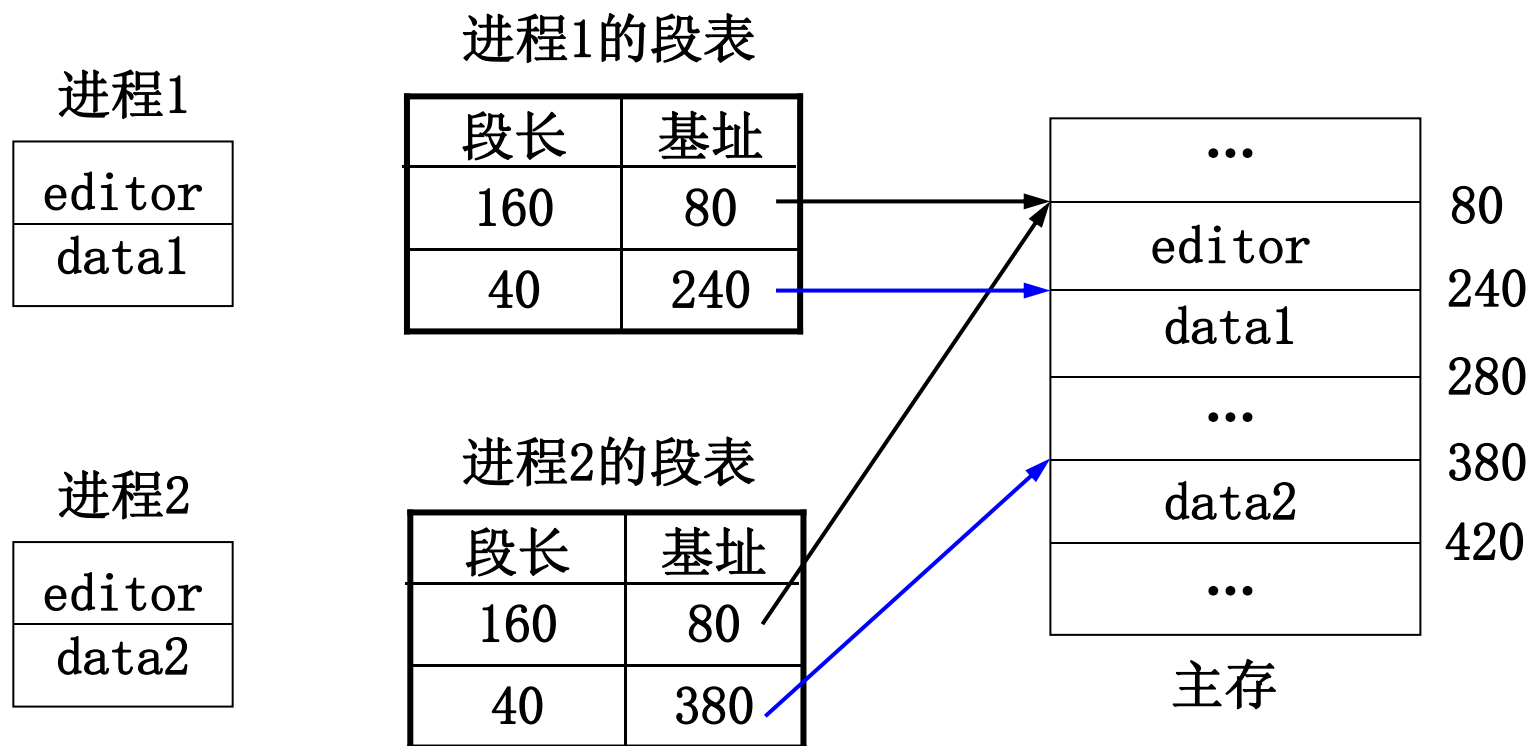


图4-22 分段系统中共享editor的示意图



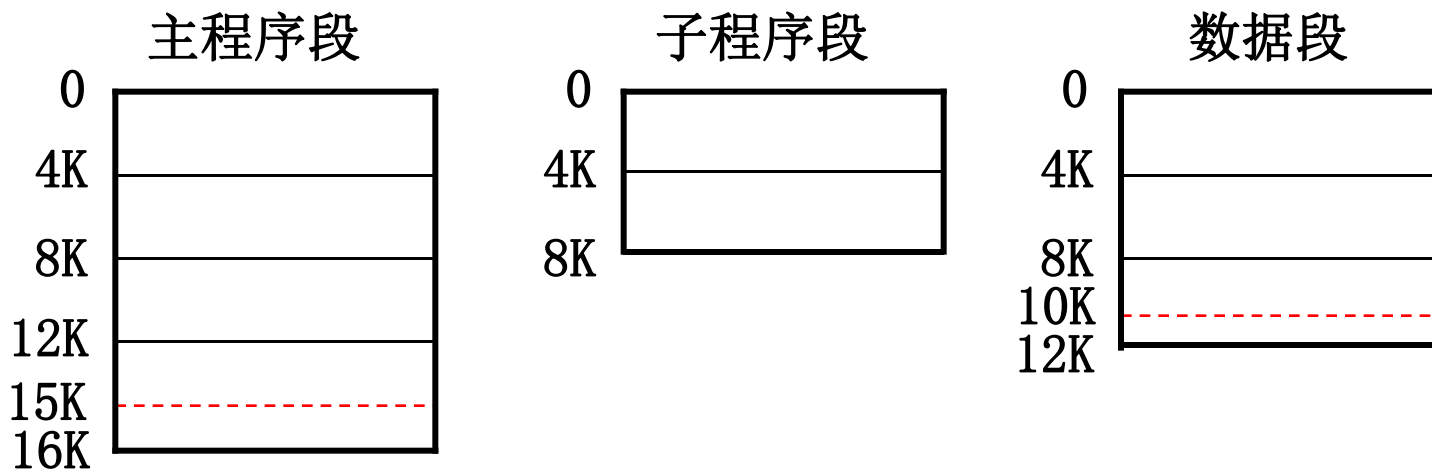
## 4.5.4 段页式存储管理方式

### 1. 基本原理

先将用户程序分成若干段，并为每个段赋予一个段名，再把每个段分成若干个固定大小的页面。

每个进程有一张段表，每个段有一张页表。

在段页式系统中，地址结构由段号、段内页号和页内地址三部分所组成。



地址结构

段号 (S)	段内页号 (P)	页内地址 (W)
--------	----------	----------

图4-23 作业地址空间和地址结构

## 2. 地址变换流程

在段页式系统中，为了获得一条指令或数据，需要访问三次内存：

第一次：访问内存中的段表，取得页表始址；

第二次：访问内存中的页表，取得该页所在的物理块号，将块号与页内地址形成物理地址；

第三次：访问第二次所得的地址，取出指令或数据。

**缺点：**访内存次数增加两倍

**解决方法：**设置TLB。

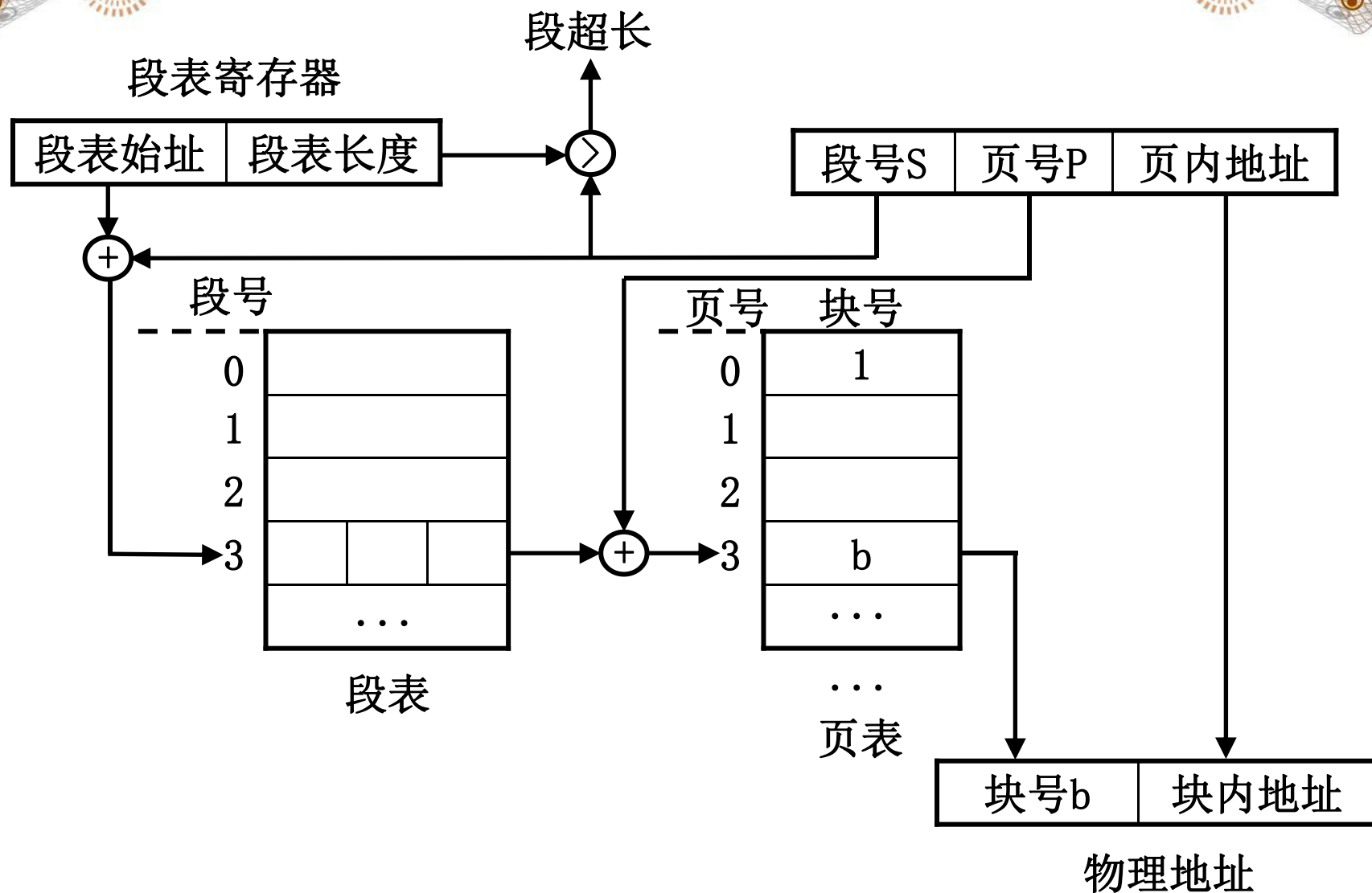


图4-25 段页式系统中的地址变换流程