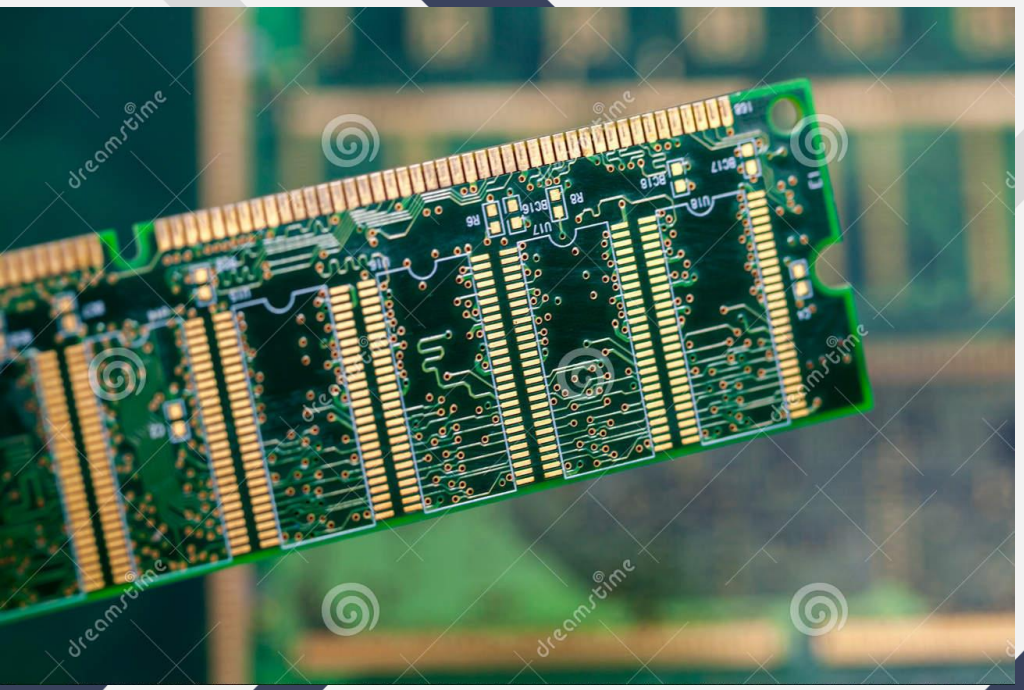


2022

操作系统A

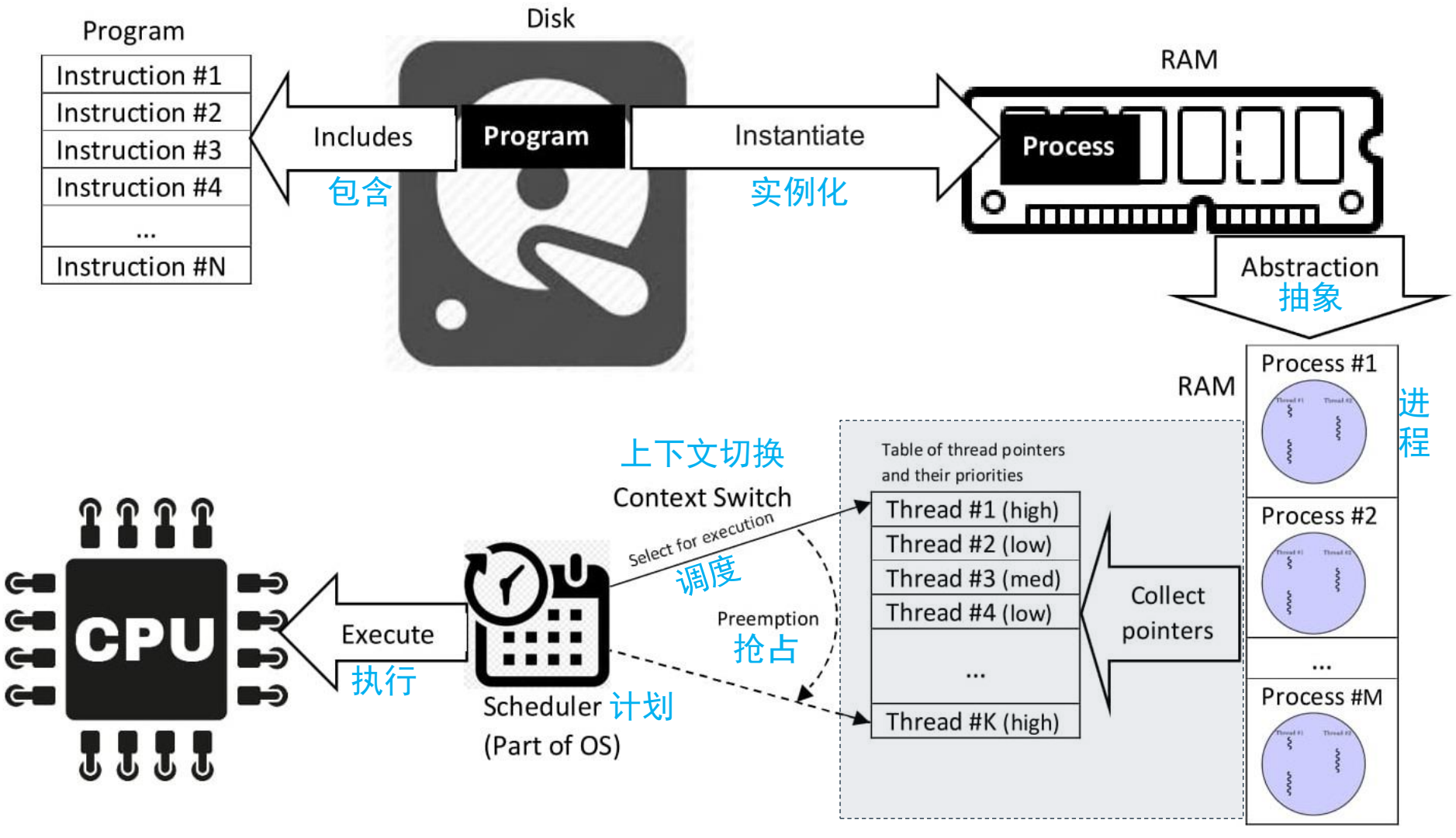
第四章
存储器管理

赵谷雨\ 信息科学与工程学院

Email: gaiazhao@ysu.edu.cn

程序的执行

计算机执行时，几乎每一条指令都涉及对存储器的访问。



目录

CONTENTS

第四章 存储器管理

4.1 存储器的层次结构

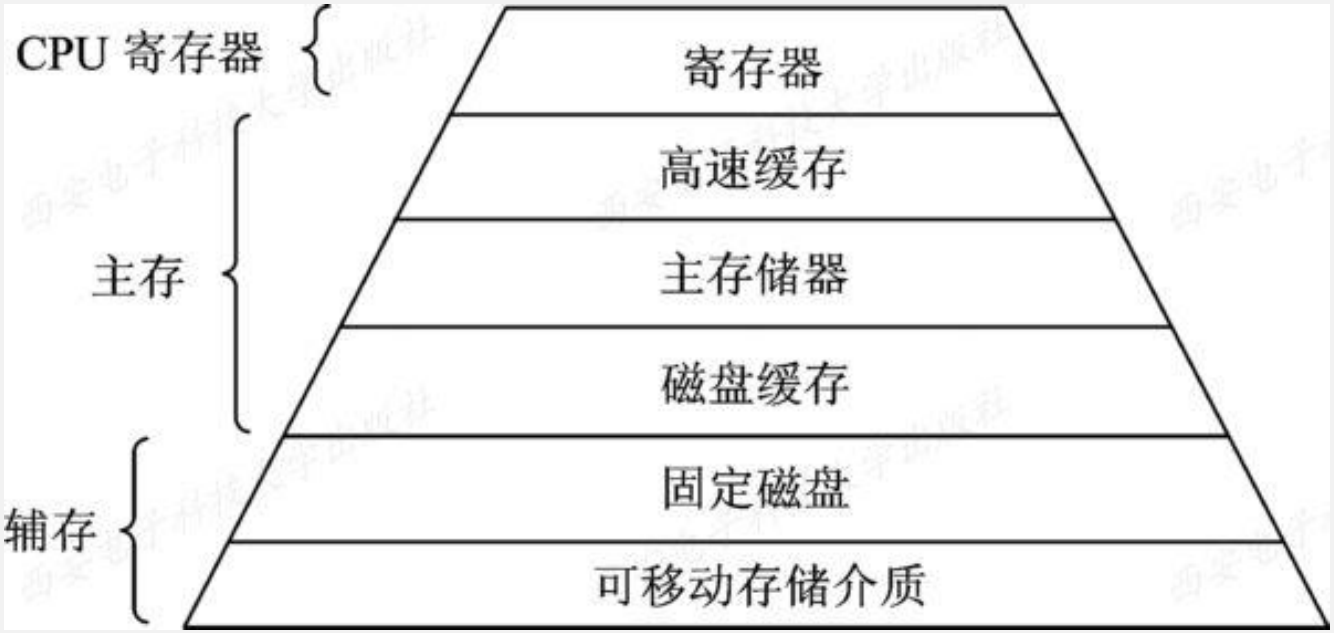
4.1 存储器的层次结构

在计算机执行时，几乎每一条指令都涉及对存储器的访问，因此要求对存储器的访问速度能跟得上处理机的运行速度。或者说，存储器的速度必须非常快，能与处理机的速度相匹配，否则会明显地影响到处理机的运行。此外还要求存储器具有非常大的容量，而且存储器的价格还应很便宜。

4.1.1 多层结构的存储器系统

1. 存储器的多层结构

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。在较高档的计算机中，还可以根据具体的功能细分为寄存器、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。



4.1.1 多层结构的存储器系统

2. 可执行存储器

寄存器和主存储器被称为可执行存储器。

进程可以在很少的时钟周期内使用一条load或store指令对可执行存储器进行访问。但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

主存储器简称内存或主存，是计算机系统中的主要部件，用于保存进程运行时的程序和数据，也称可执行存储器。

处理机从主存中取指令和数据，把指令放入指令寄存器，把数据放入数据寄存器。访问速度远低于CPU的指令执行速度。

2. 寄存器

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，完全能与CPU协调工作，但价格却十分昂贵，因此容量不可能做得很大。现在，随着成本的降低，逐渐增多。

4.1.3 高速缓存和磁盘缓存

1. 高速缓存

高速缓存介于寄存器和存储器之间的存储器，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，这样可大幅度地提高程序执行速度。高速缓存容量远大于寄存器，而比内存约小两到三个数量级左右，从几十KB到几MB，访问速度快于主存储器。

重要的指标是命中率。

4.1.3 高速缓存和磁盘缓存

2. 磁盘缓存

由于目前磁盘的I/O速度远低于对主存的访问速度，为了缓和两者之间在速度上的不匹配，而设置了磁盘缓存，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。主存也可以看作是辅存的高速缓存，因为，辅存中的数据必须复制到主存方能使用，反之，数据也必须先存在主存中，才能输出到辅存。

第四章 存储器管理

4.2 程序的装入和链接

4.2 程序的装入和链接

用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，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**，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示出了这样的三步过程。本节将扼要阐述程序(含数据)的链接和装入过程。

4.2 程序的装入和链接

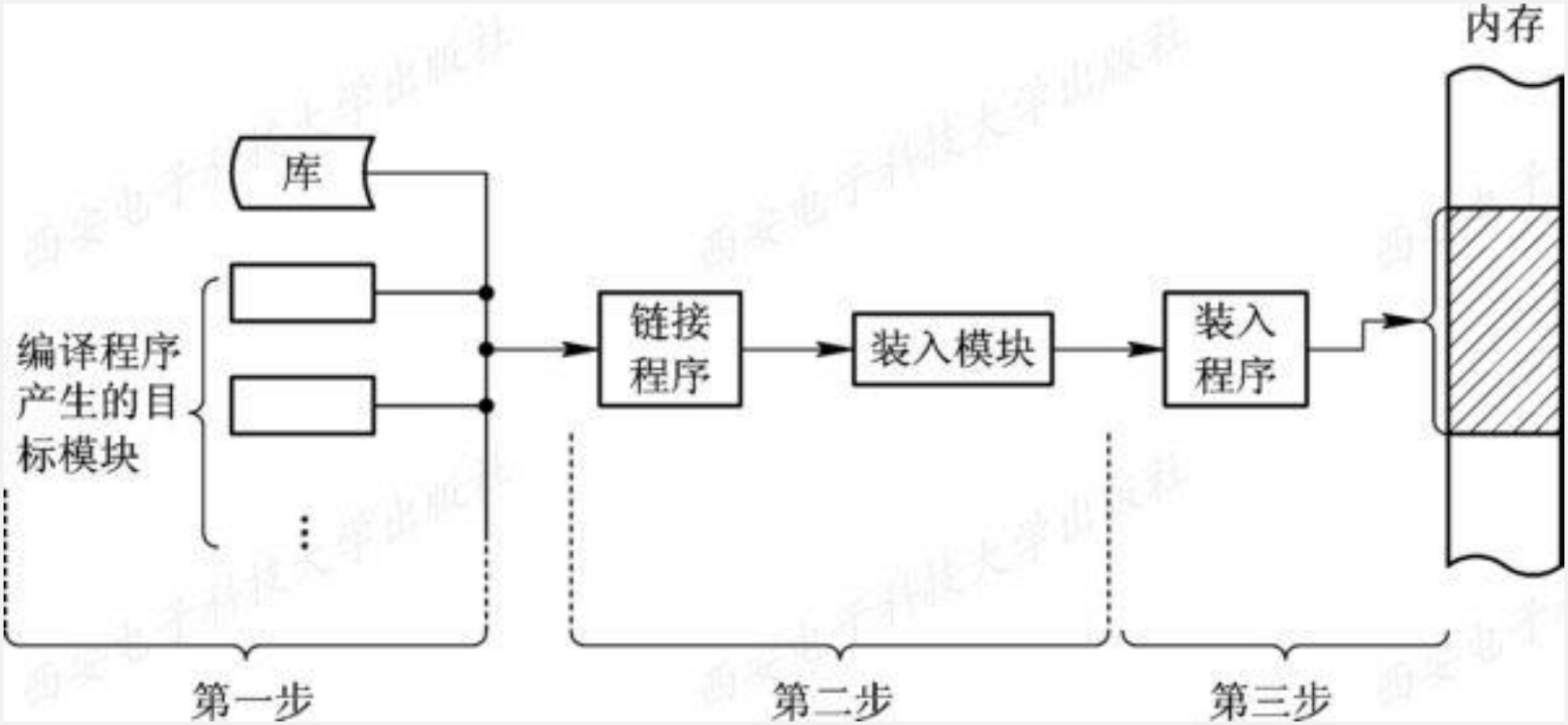


图4-2 对用户程序的处理步骤

4.2.1 程序的装入

在将一个装入模块装入内存时，可以有如下三种装入方式：

1. 绝对装入方式(Absolute Loading Mode)

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。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。

4.2.1 程序的装入

2. 可重定位装入方式(Relocation Loading Mode)

在多道程序环境下，编译程序不可能预知经编译后所得到的目标模块应放在内存的何处。因此，对于用户程序编译所形成的若干个目标模块，它们的起始地址通常都是从0开始的，程序中的其它地址也都是相对于起始地址计算的。

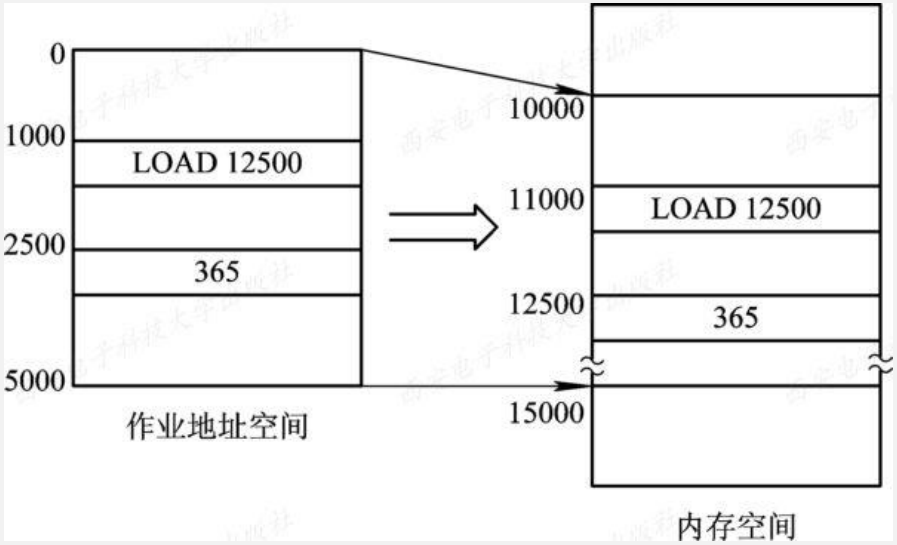


图4-3 作业装入内存时的情况

4.2.1 程序的装入

3. 动态运行时的装入方式(Dynamic Run-time Loading)

装入程序把装入模块装入内存后，并不立即把装入模块中的逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行的时候，因此，需要有重定位寄存器的支持。

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。在图4-4(a)中示出了经过编译后所得到的三个目标模块A、B、C，它们的长度分别为L、M和N。在模块A中有一条语句CALL B，用于调用模块B。在模块B中有一条语句CALL C，用于调用模块C。B和C都属于外部调用符号，在将这几个目标模块装配成一个装入模块时，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。

4.2.2 程序的链接

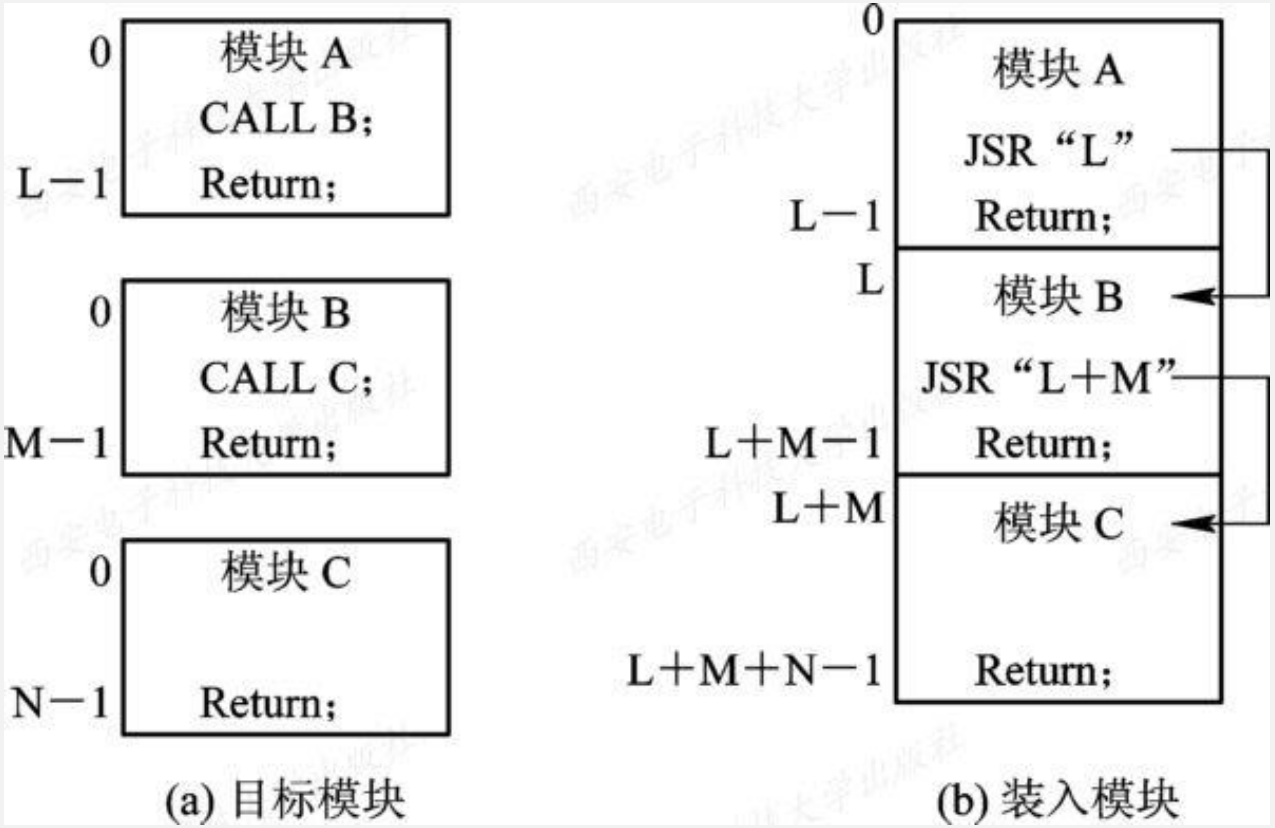


图4-4 程序链接示意图

4.2.2 程序的链接

2. 装入时动态链接(Load-time Dynamic Linking)

将用户源程序编译后所得到的一组目标模块，在装入内存时，采用边装入边链接的链接方式。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。装入时动态链接方式有以下优点：

(1) 便于修改和更新。

(2) 便于实现对目标模块的共享。

4.2.2 程序的链接

3. 运行时动态链接(Run-time Dynamic Linking)

由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有部分目标模块根本就不运行。

将对某些模块的链接推迟到程序执行时进行。

第四章 存储器管理

4.3 连续分配存储管理方式

本节问题和目标

问题：如何为用户装入程序分配内存空间？

目标：

- 了解系统管理内存的方法
- 掌握单一连续分配、固定分区分配方式
- 重点掌握动态分区分配和动态可重定位分区分配方法

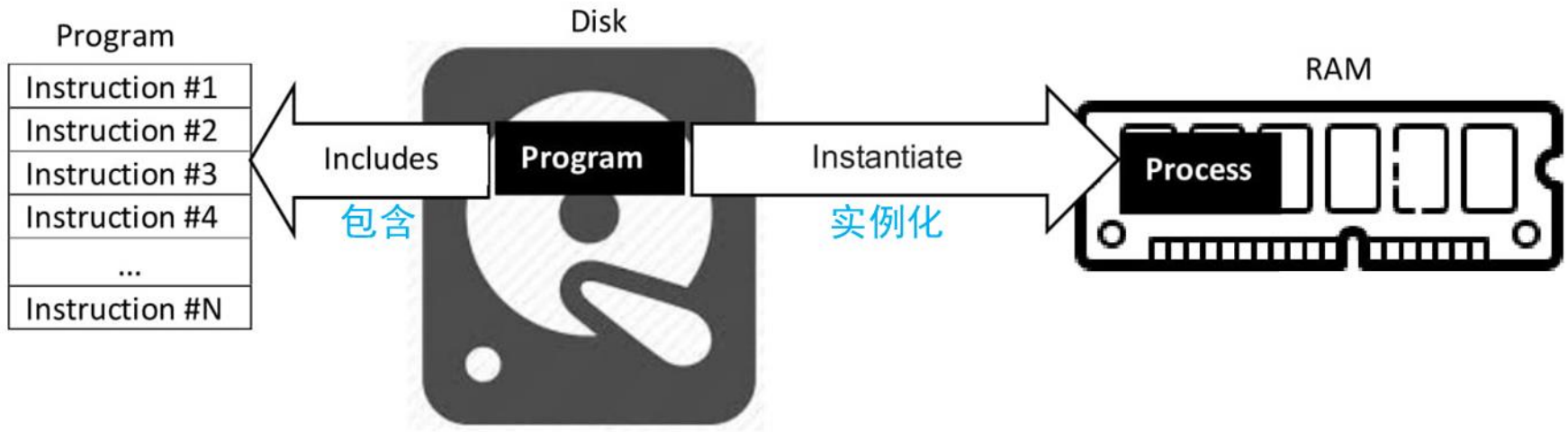
4.3 连续分配存储管理方式

用户程序 → 内存分配 → 内存/进程 → 运行

代码或数据的
逻辑地址相邻
↓
物理地址相邻

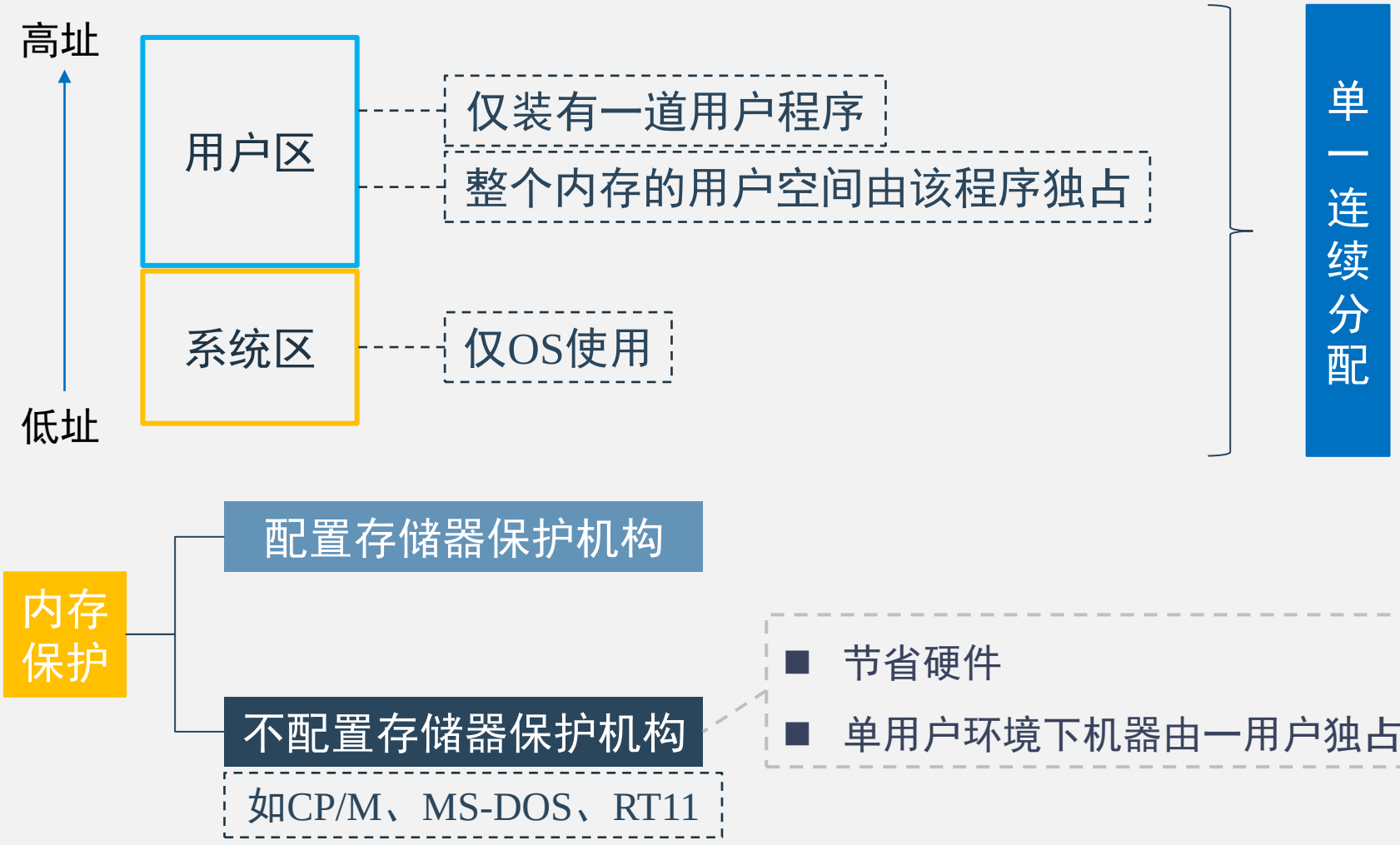
连续分配方式：为用户程序分配一个连续的内存空间

- 单一连续分配
- 固定分区分配
- 动态分区分配
- 动态可重定位分区分配



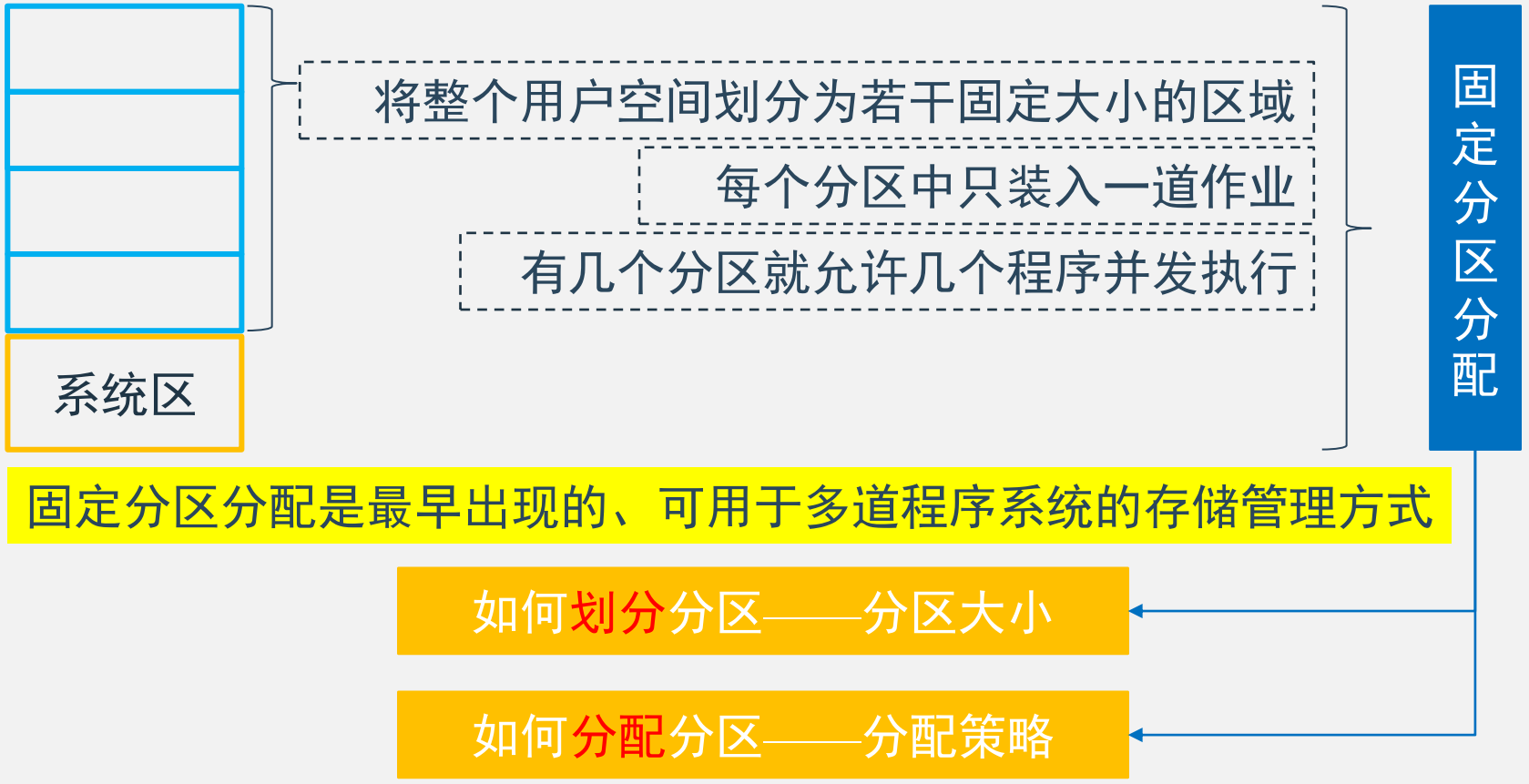
4.3.1 单一连续分配

背景 单道程序环境下，把内存分为系统区和用户区。



4.3.2 固定分区分配

背景 多道程序系统出现，内存中需载入多道程序且不发生干扰。



4.3.2 固定分区分配

1. 划分分区的方法

分区大小相等	VS	分区大小不等
缺乏灵活性		灵活性增加

对使用一台计算机控制多个相同对象的场合比较适用：

- 对象所需内存空间大小相同
- 分配方便实用

如：炉温群控系统

可调查系统中运行作业的大小，根据用户需求来划分；

通常把内存划分为：

- 多个较小的分区
- 适量的中等分区
- 少量的大分区

4.3.2 固定分区分配

2. 内存分配

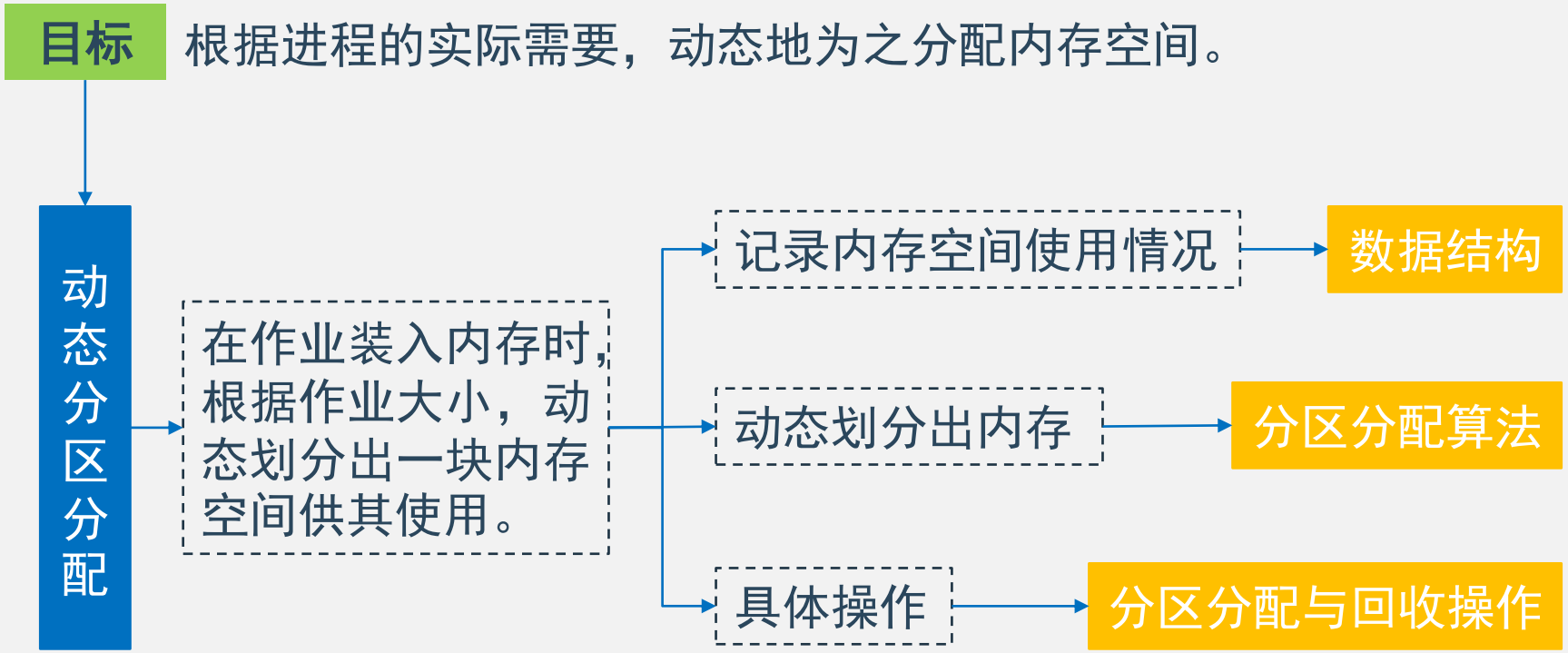
为便于内存分配，需要记录内存分区的使用情况、大小、起始地址。

建立分区使用表

分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

空间	操作系统
24 KB	作业 A
32 KB	作业 B
64 KB	作业 C
128 KB	
…	…
256 KB	

4.3.3 动态分区分配


目标
根据进程的实际需要，动态地为之分配内存空间。
动态分区分配
在作业装入内存时，根据作业大小，动态划分出一块内存空间供其使用。
记录内存空间使用情况
数据结构
动态划分出内存
分区分配算法
具体操作
分区分配与回收操作

4.3.3 动态分区分配

1. 动态分区分配中的数据结构—如何管理内存

为实现动态分区分配，需要描述空闲分区和已分配分区的情况。

数据结构：空闲分区表 | 空闲分区链

每个空闲分区占一个表目，包括分区号、分区大小、分区始址和状态等信息。

分区号	分区大小(KB)	分区始址(K)	状态
1	50	85	空闲
2	32	155	空闲
3	70	275	空闲
4	60	532	空闲
5	…	…	…

图4-6 空闲分区表

将所有空闲分区链接成一个双向链

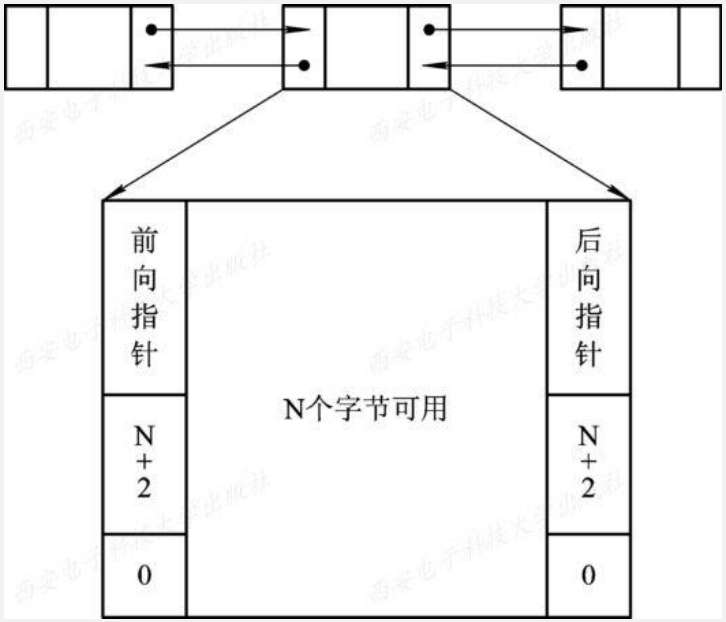


图4-7 空闲链结构

4.3.3 动态分区分配

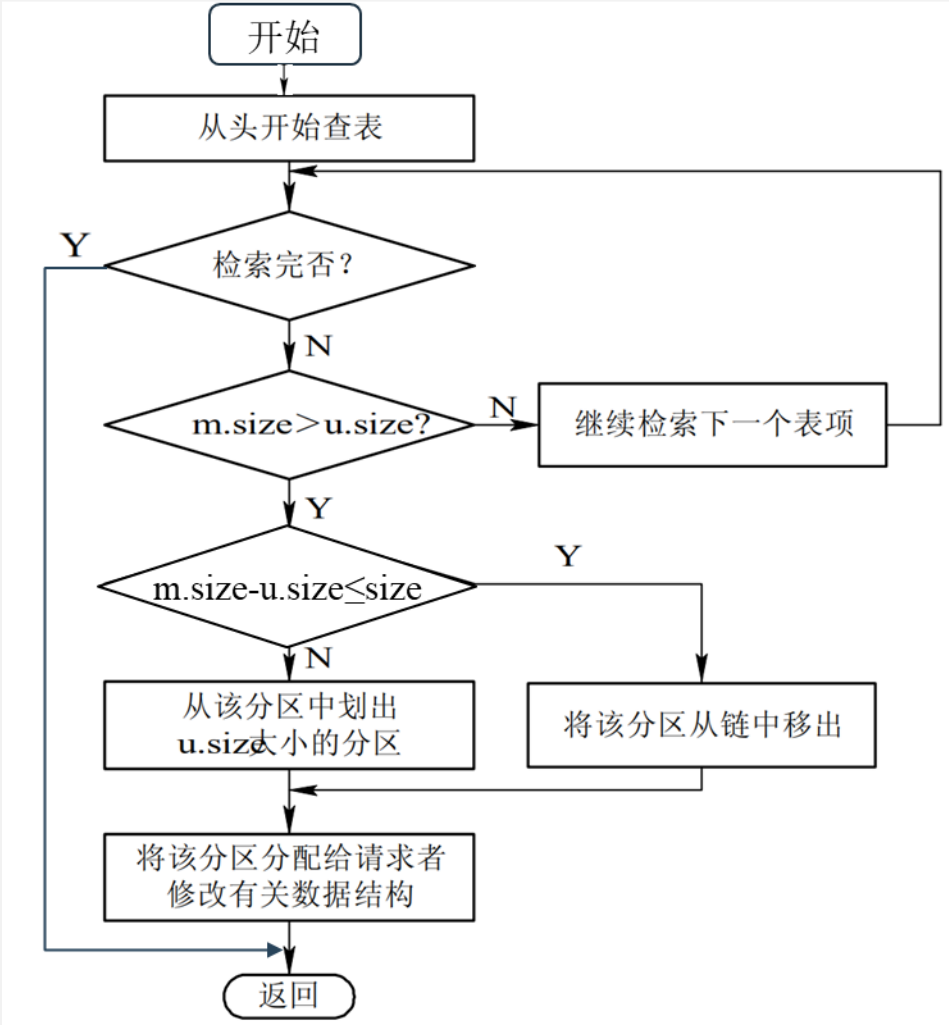
3. 分区分配操作—分配内存

分区号	大小/KB	起址/KB	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

系统应利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为u.size，表中每个空闲分区的大小可表示为m.size。size为规定好的不可再分割分区大小。

① 如果m.size – u.size <= size，将整个分区分配给请求者。

② 否则从该分区中划出u.size分配出去，余下部分仍旧保留在空闲分区链表中。



4.3.3 动态分区分配

3. 分区分配操作—回收内存

当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链(表)中找到相应的插入点，此时可能出现四种情况。

(1) 回收区与插入点的前一个空闲分区F1相邻接

此时应将回收区与插入点的前一分区合并，不必为回收分区分配新表项，而只需修改其前一分区F1的大小。

(a)

(2) 回收分区与插入点的后一空闲分区F2相邻接

此时也可将两分区合并，形成新的空闲分区，但用回收区的首址作为新空闲区的首址，大小为两者之和。

(b)

4.3.3 动态分区分配

3. 分区分配操作—回收内存

当进程运行完毕释放内存时，系统根据回收区的首址，从空闲区链(表)中找到相应的插入点，此时可能出现四种情况。

(3) 回收区同时与插入点的前、后两个分区邻接

此时将三个分区合并，使用F1的表项和F1的首址，取消F2的表项，大小为三者之和。

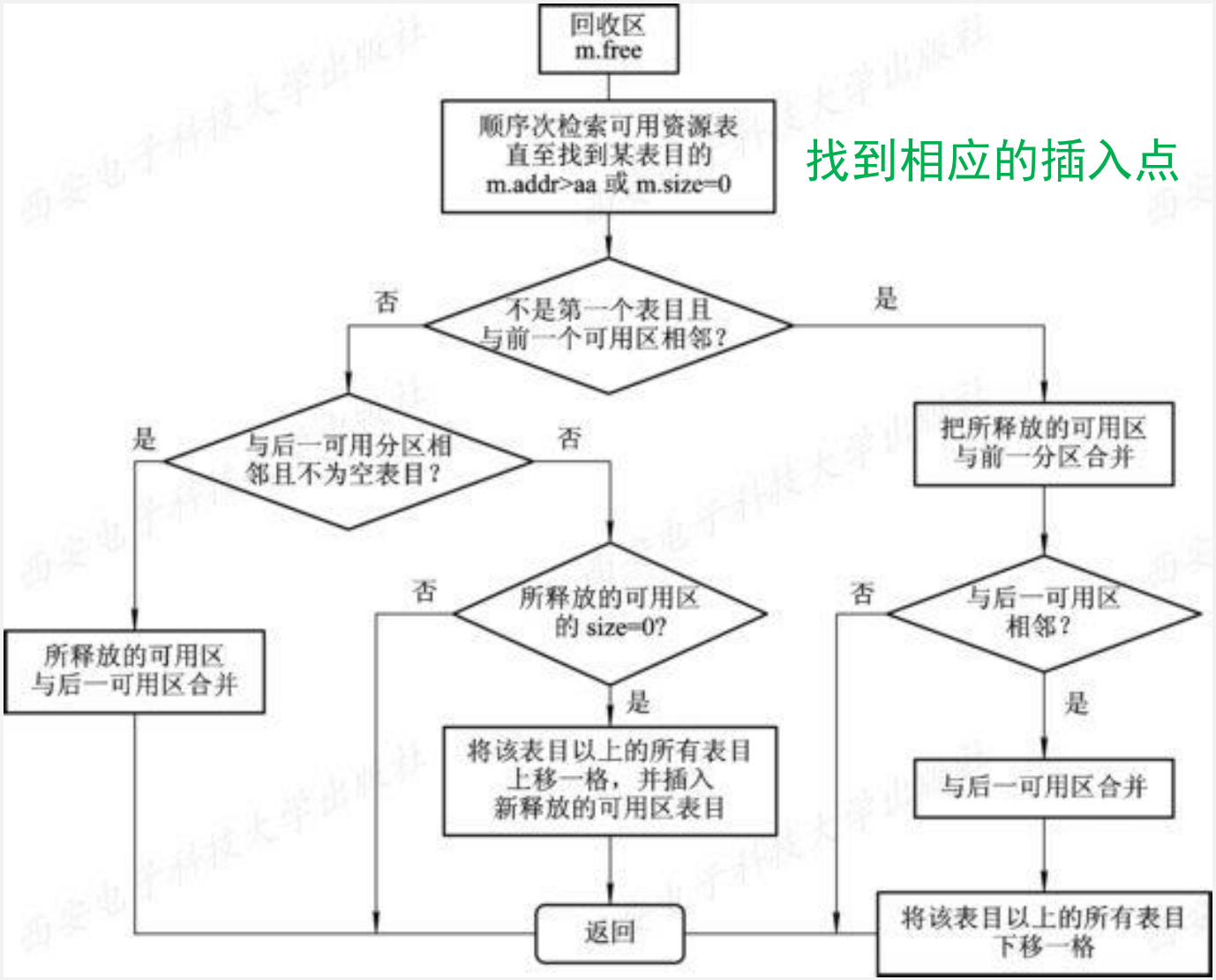
(c)

(4) 回收区既不与F1邻接，又不与F2邻接

应为回收区单独建立一个新表项，填写回收区的首址和大小，并根据其首址插入到空闲链中的适当位置。

4.3.3 动态分区分配

3. 分区分配操作—回收内存



4.3.3 动态分区分配

2. 分区分配算法—动态划分出内存块

为把一个新作业装入内存，须按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。由于内存分配算法对系统性能有很大的影响，故人们对它进行了较为广泛而深入的研究，于是产生了许多动态分区分配算法。

动态分区分配的分区分配算法，在4.3.4和4.3.5节详细介绍。

4.3.4 基于顺序搜索的动态分区分配算法

首次适应(first fit, FF)算法

循环首次适应(next fit, NF)算法

最佳适应(best fit, WF)算法

最坏适应(worst fit, WF)算法

适用于不太大的系统。将系统中的空闲分区连接成一个链。但对于大型系统，因为链太长，所以效率低下。

顺序搜索，是指依次搜索空闲分区链上的空闲分区，去寻找一个大小能满足要求的分区。

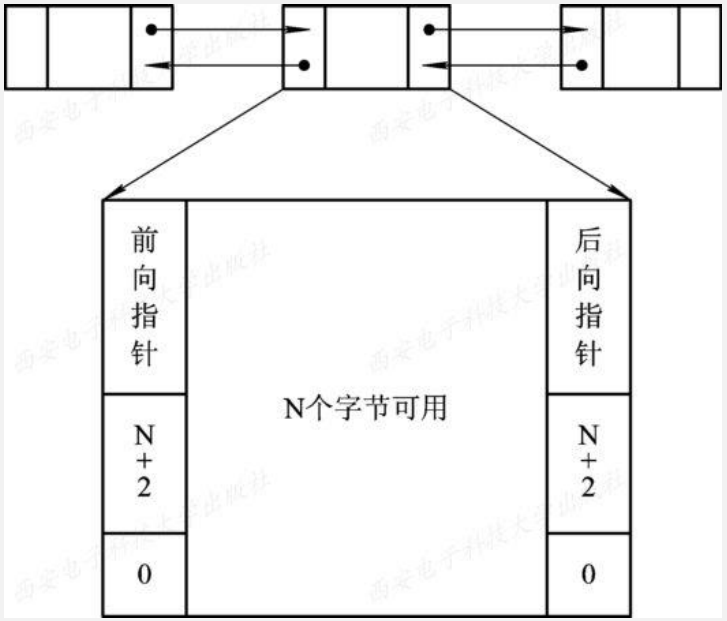


图4-7 空闲链结构

4.3.4 基于顺序搜索的动态分区分配算法

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。

(1) 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。

(2) 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。

(3) 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。

(1) 优先利用内存中低地址部分，保留了大空间，大作业可用。

(2) 低地址部分碎片太多，查找空闲耗费时间多。

4.3.4 基于顺序搜索的动态分区分配算法

循环首次适应(next fit, NF)算法

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。

(1) 使内存中空闲分区分布更均匀，避免低址部分留下许多很小的空闲分区。
(2) 减少查找可用空闲分区的开销。
(3) 但使内存缺少大的空闲分区。

4.3.4 基于顺序搜索的动态分区分配算法

最佳适应(best fit, WF)算法

总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。

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。

由于每次都是找到最小的，因此产生大量难以利用的细小碎片。

4.3.4 基于顺序搜索的动态分区分配算法

最坏适应(worst fit, WF)算法

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。

(1) 剩余空间不会太小，产生碎片的可能性最小。

(2) 分配效率高，因为可以形成一个容量从大到小的空闲分区链，一般第一个就能找到。

4.3.5 基于索引搜索的动态分区分配算法

为了提高搜索空闲分区的速度，在大中型系统中往往采用基于索引搜索的动态分区分配算法。

- 快速适应(quick fit, QF)算法
- 伙伴系统(buddy system)
- 哈希算法

4.3.5 基于索引搜索的动态分区分配算法

快速适应(quick fit, QF)算法

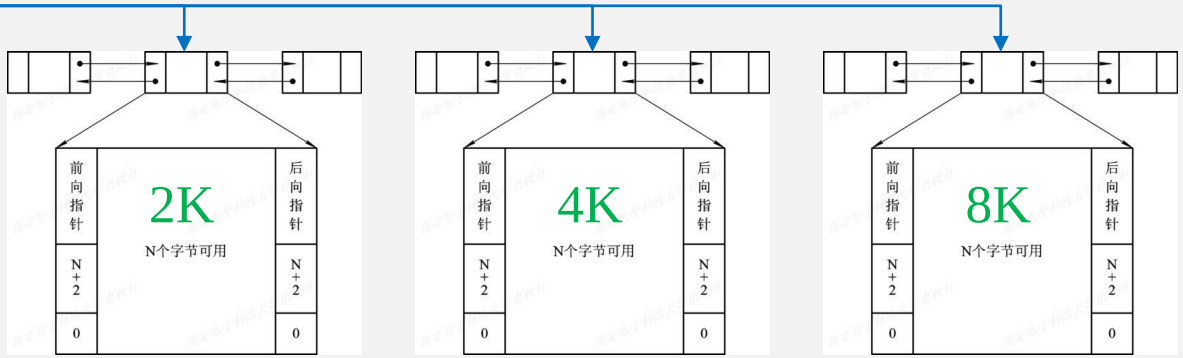
该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，在内存中设立一张管理索引表，其中的每一个索引表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针。

(1) 根据进程长度，从索引表中寻找最小空闲区链表；

(2) 从链表中取出第一块进行分配。

查找效率高，不对分割分区，不产生碎片，但分区归还算法复杂，系统开销大。

索引表



4.3.5 基于索引搜索的动态分区分配算法

伙伴系统(buddy system)

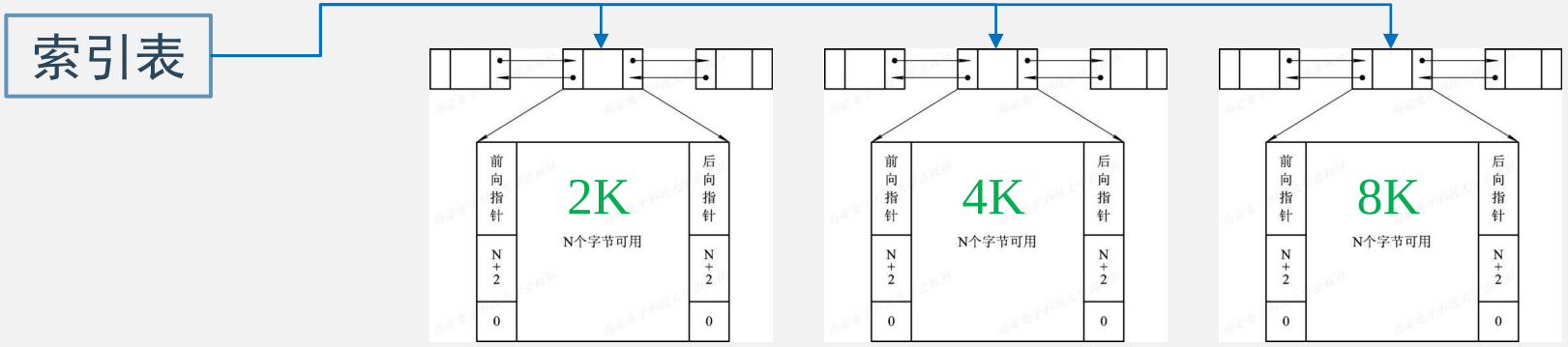
该算法规定，无论已分配分区或空闲分区，其大小均为2的k次幂(k为整数，l≤k≤m)。通常2^m是整个可分配内存的大小(也就是最大分区的大小)。

假设进程需要长度为n的存储空间：

(1) 计算i，使$2^{i-1} < n <= 2^i$，在空闲分区大小为2i的空闲分区链表中查找，如果找到，则分配。

(2) 没找到，则在2^{i+1}的空闲分区链表中查找，如果找到，则将该区等分，一个分配，一个加入2^i的空闲分区链表。

(3) 如果仍没找到，则继续类似 (2) 的处理。



4.3.5 基于索引搜索的动态分区分配算法

伙伴系统(buddy system)

在伙伴系统中，对于一个大小为2k，地址为x的内存块，其伙伴块的地址则用buddyk(x)表示，其通式为：

$$buddy_k(x)=\begin{cases} x+2^k & (若\ xMOD\ \ 2^{k+1}=0) \\ x-2^k & (若\ xMOD\ \ 2^{k+1}=2^k) \end{cases}$$

时间性能比快速适应算法差，比顺序搜索算法好。

空间性能比快速适应算法好，比顺序搜索算法差。

4.3.5 基于索引搜索的动态分区分配算法

哈希算法

利用哈希快速查找的优点，和空闲分区在空利用空闲区表中的分布规律建立哈希函数，构造以空闲分区大小为关键字的哈希表。根据需要的空间大小，计算哈希函数，得到位置，从而实现最佳分配策略。

4.3.6 动态可重定位分区分配

1. 紧凑

连续分配方式的一个重要特点是，一个系统或用户程序必须被装入一片连续的内存空间中。当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区，而缺乏大的空闲空间。即使这些分散的许多小分区的容量总和大于要装入的程序，但由于这些分区不相邻接，也无法把该程序装入内存。

4.3.6 动态可重定位分区分配

1. 紧凑

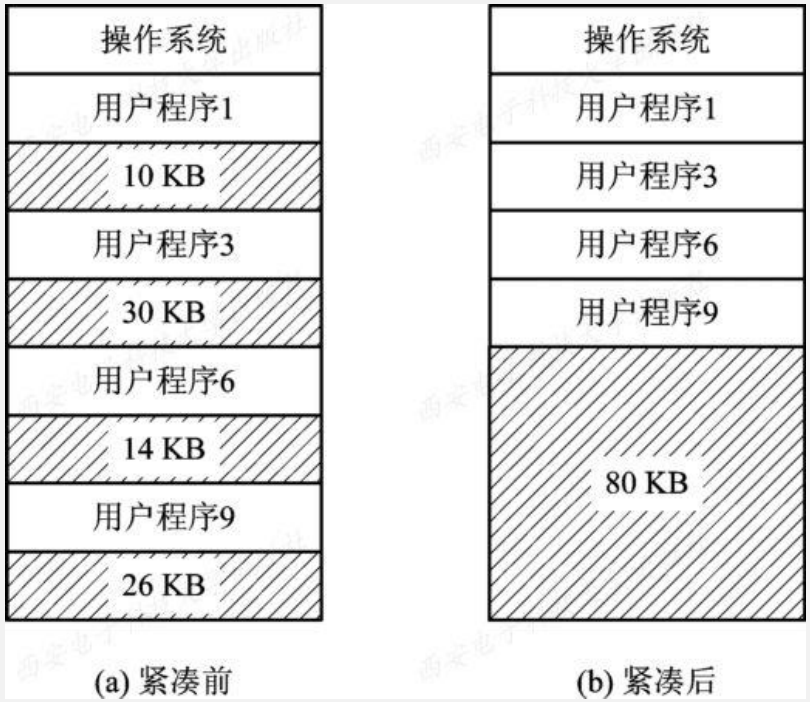


图4-11 紧凑的示意

但是，每次紧凑之后，都需要对程序和数据进行重定位。解决方法是：动态重定位。

4.3.6 动态可重定位分区分配

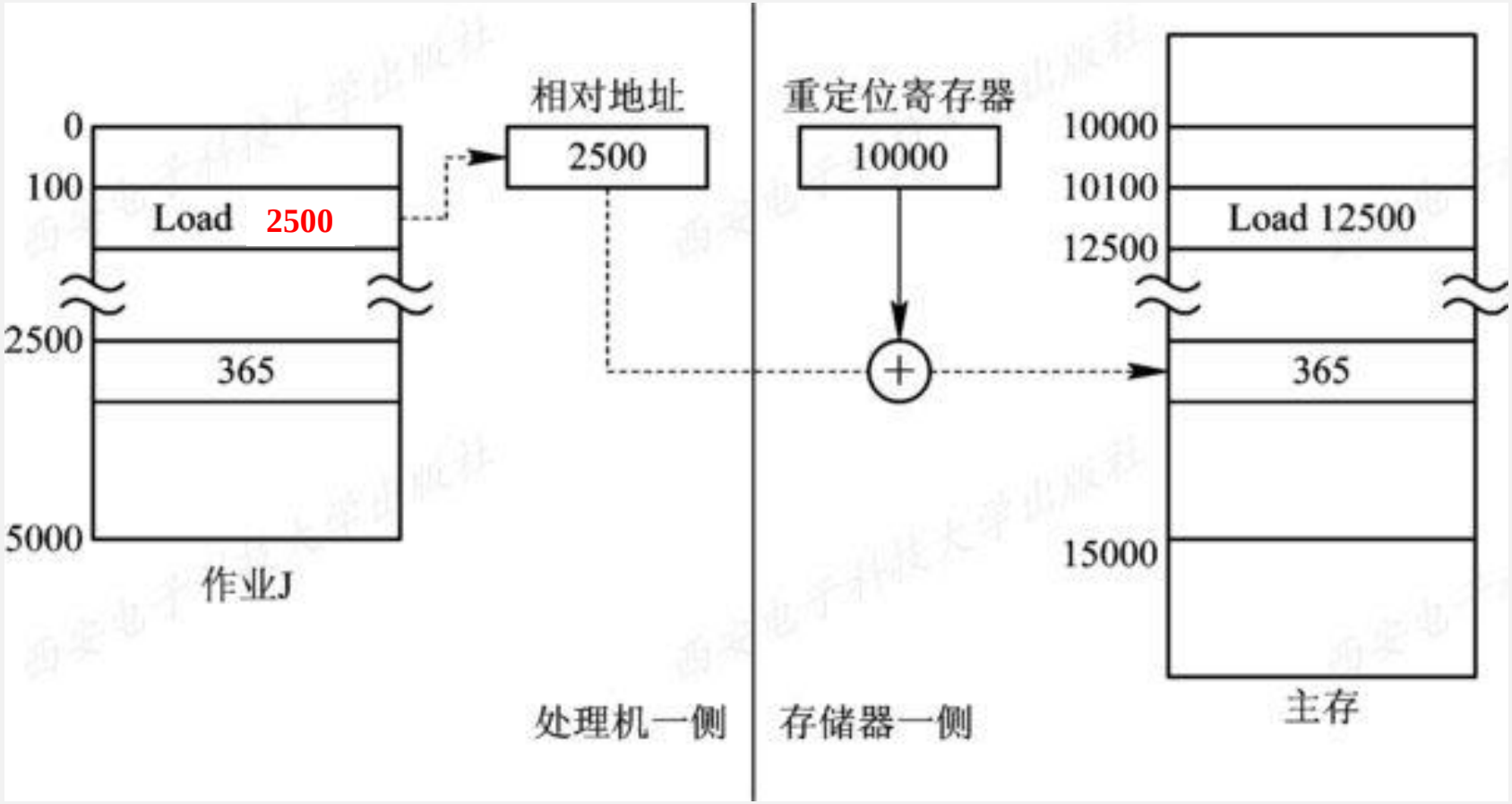
2. 动态重定位

在动态运行时装入方式中，作业装入内存后的所有地址仍然都是相对(逻辑)地址。而将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。

为使地址的转换不会影响到指令的执行速度，必须有硬件地址变换机构的支持，即须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。

4.3.6 动态可重定位分区分配

2. 动态重定位



4.3.6 动态可重定位分区分配

3. 动态重定位分区分配算法

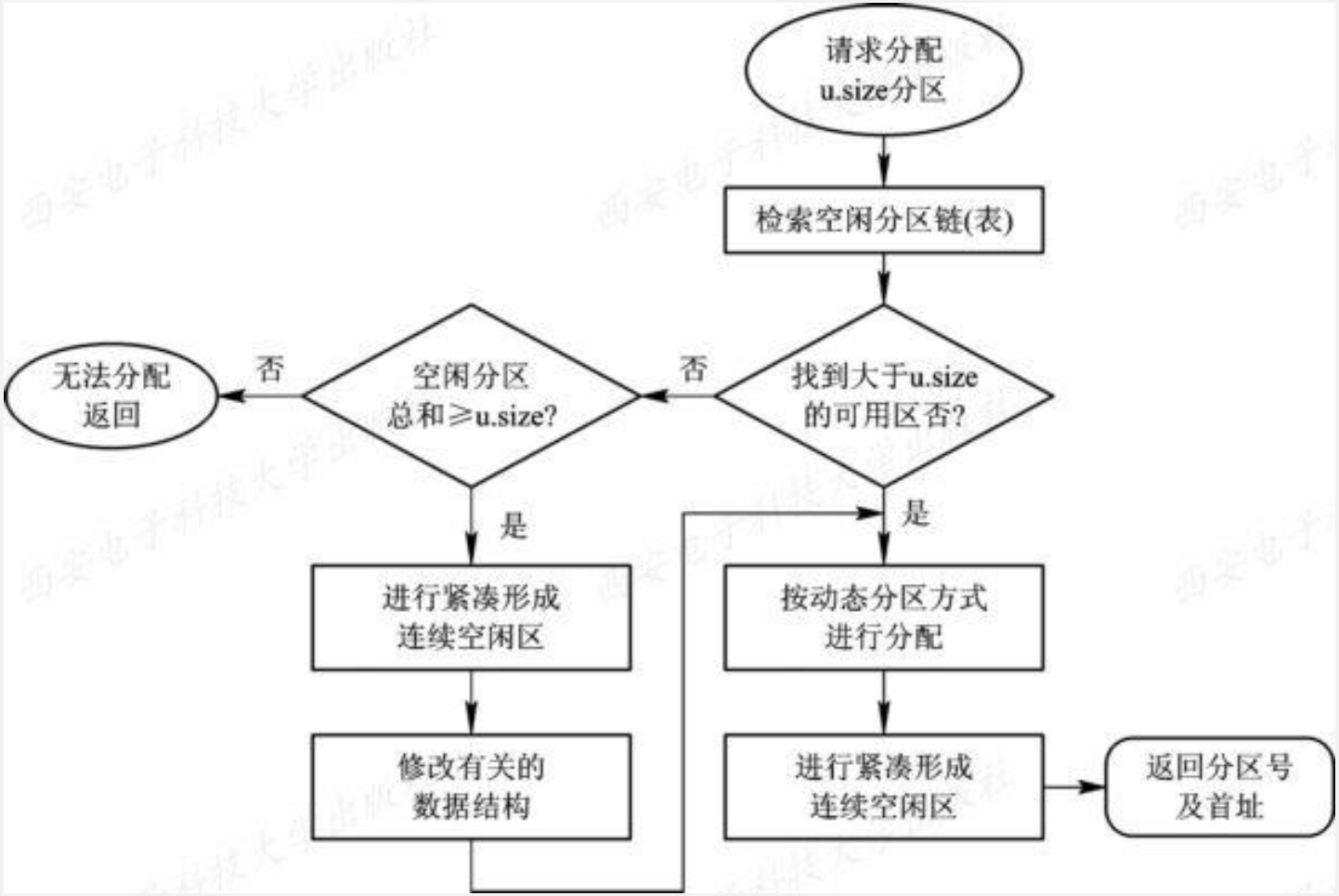
动态重定位分区分配算法与动态分区分配算法基本上相同，差别仅在于：在这种分配算法中，增加了紧凑的功能。

当该算法不能找到一个足够大的空闲分区以满足用户需求时，如果所有的小的空闲分区的容量总和大于用户的要求，这时便须对内存进行“紧凑”，将经“紧凑”后所得到的大空闲分区分配给用户。

如果所有的小的空闲分区的容量总和仍小于用户的要求，则返回分配失败信息。

4.3.6 动态可重定位分区分配

3. 动态重定位分区分配算法



第四章 存储器管理

4.4 对换(Swapping)

4.4 对换(Swapping)

对换技术的出现

最早用于麻省理工学院的单用户分析系统CTSS中。当时计算机的内存都非常小，为使系统能分时运行多个用户程序而引入了对换技术。

系统并所有用户的程序都存储在磁盘上，每次只能调入一个作业进入内存。当该作业的一个时间片用完时，将它调至外存的后备队列上等待，再从后备队列上将另一个作业调入内存。这就是最早出现的分时系统中所用的对换技术。现在已经很少使用。

要实现内外存的对换，需要有一台I/O速度较高的外存，容量也要足够大。

4.4.1 多道程序环境下的对换技术

1. 对换的引入

在多道程序环境下，一方面，在内存中的某些进程由于某事件尚未发生而被阻塞运行，但它却占用了大量的内存空间，甚至有时可能出现在内存中所有进程都被阻塞，而无可运行之进程，迫使CPU停止下来等待的情况；另一方面，却又有着许多作业，因内存空间不足，一直驻留在外存上，而不能进入内存运行。显然这对系统资源是一种严重的浪费，且使系统吞吐量下降。

4.4.1 多道程序环境下的对换技术

2. 对换的类型

在每次对换时，都是将一定数量的程序或数据换入或换出内存。根据每次对换时所对换的数量，可将对换分为如下两类：

(1) 整体对换，中级调度中的对换是以整个进程为单位的，故称为“进程对换”或“整体对换”。被广泛应用于多道系统中。

(2) 页面(分段)对换，以进程的一个页面或分段为单位进行，统称为“部分对换”。

为实现进程对换，系统必须实现：对换空间的管理、进程的换出、进程的换入。

4.4.2 对换空间的管理

1. 对换空间管理的主要目标

在具有对换功能OS中，通常把磁盘空间分为文件区和对换区两部分。

1) 对文件区管理的主要目标：提高文件存储空间的利用率。

2) 对对换空间管理的主要目标：提高进程换入和换出的速度。

4.4.2 对换空间的管理

2. 对换区空闲盘块管理中的数据结构

其数据结构的形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表的每个表目，应包含两项：对换区的首址及其大小，分别用盘块号和盘块数表示。

3. 对换空间的分配与回收

由于对换分区的分配采用的是连续分配方式，因而对换空间的分配与回收与动态分区方式时的内存分配与回收方法雷同。其分配算法可以是首次适应算法、循环首次适应算法或最佳适应算法等。回收方法主要是与相邻区连接在一起。

4.4.3 进程的换出与换入

1. 进程的换出

对换进程在实现进程换出时，是将内存中的某些进程调出至对换区，以便腾出内存空间。换出过程可分为以下两步：

(1) 选择被换出的进程

首先选择阻塞态或睡眠态的进程，当有多个时，选择优先级低的。还要考虑进程在内存的驻留时间，避免频繁换入或换出。

(2) 进程换出过程

只能换出非共享的程序和数据段。

首先申请对换空间，若申请成功，则启动磁盘，将其传送到磁盘的对换区上。如果没有出错，则系统可回收所占用的内存。

4.4.3 进程的换出与换入

2. 进程的换入

对换进程将定时执行换入操作：

首先查看PCB集合中所有进程的状态，从中找出“就绪”状态但已换出的进程。当有许多这样的进程时，它将选择其中已换出到磁盘上时间最久(必须大于规定时间，如2s)的进程作为换入进程，为它申请内存。（1）如果申请成功，可直接将进程从外存调入内存。（2）如果失败，则需先将内存中的某些进程换出，腾出足够的内存空间后，再将进程调入。

4.4.3 进程的换出与换入

2. 进程的换入

由前面的学习可知，对换不是一件简单的事情，它需要消耗一定的系统资源，因此，对换程序的运行应该慎重对待。

目前使用较多的对换方案：

处理机正常运行时，不启动对换程序。

发现有许多进程经常发生缺页或内存紧张时，再启动对换程序。

如果发现缺页率明显减少，系统的吞吐量下降时，暂停对换程序。

第四章 存储器管理

4.5 分页存储管理方式

4.5 分页存储管理方式

连续分配方式→离散分配方式

(1) 分页存储管理方式

- 用户程序地址空间分为若干固定大小的区域，称为“页”。
- 内存空间分为若干个“物理块”或“页框”。
- 页和块大小相同。

(2) 分段存储管理方式：用户程序地址空间分为若干大小不同的段。

(3) 段页式存储管理方式：上面两点结合，目前应用很广。

4.5.1 分页存储管理的基本方法

1. 页面和物理块

(1) 页面

进程的逻辑地址空间被分成若干个页，并加从0开始的编号；内存的物理地址空间分成若干物理块，并加从0开始的编号。最后会形成页内碎片。

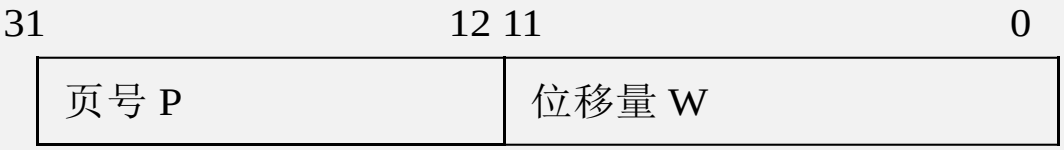
(2) 页面大小

不可过大过小。页面大小选择适中，应是2的幂，通常为1K-8K（也有资料建议512字节-1M）。

4.5.1 分页存储管理的基本方法

2. 地址结构

分页地址中的地址结构如下：



页内地址0-11位，所以每页大小4K。

页号12-31位，所以最多1M个页。

4.5.1 分页存储管理的基本方法

对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

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

例如：系统页面大小为1KB,设逻辑地址为2170B，则页号P和页内地址d为？

P = 2，d = 122。

例如： 系统页面大小为4KB，设逻辑地址为00008000H，则页号P和页内地址d分别为？

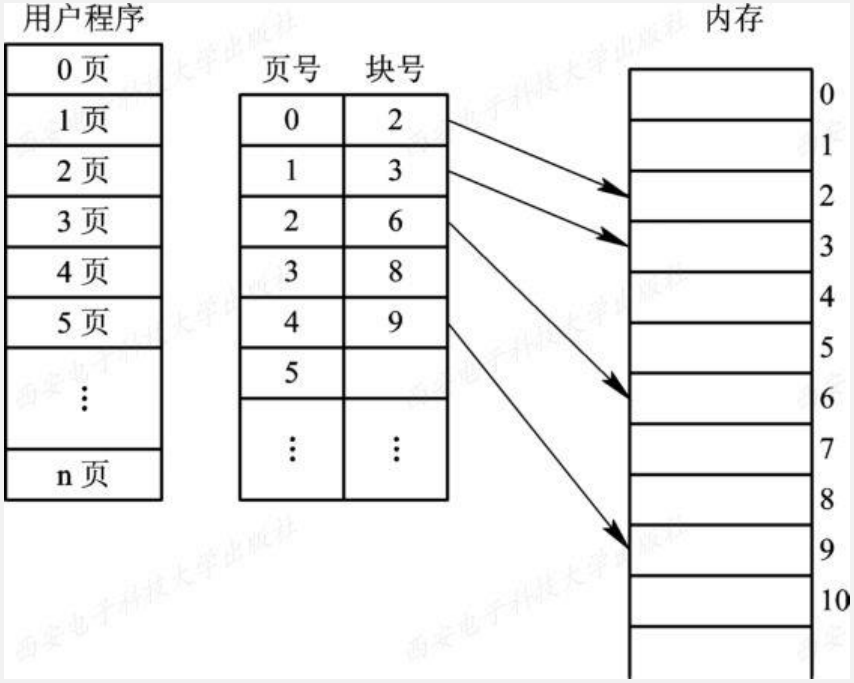
P = 8，d = 0。

4.5.1 分页存储管理的基本方法

3. 页表

在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表。

每个进程有一个页表。



4.5.2 地址变换机构

逻辑地址 → 物理地址。借助页表完成。

1. 基本的地址变换机构

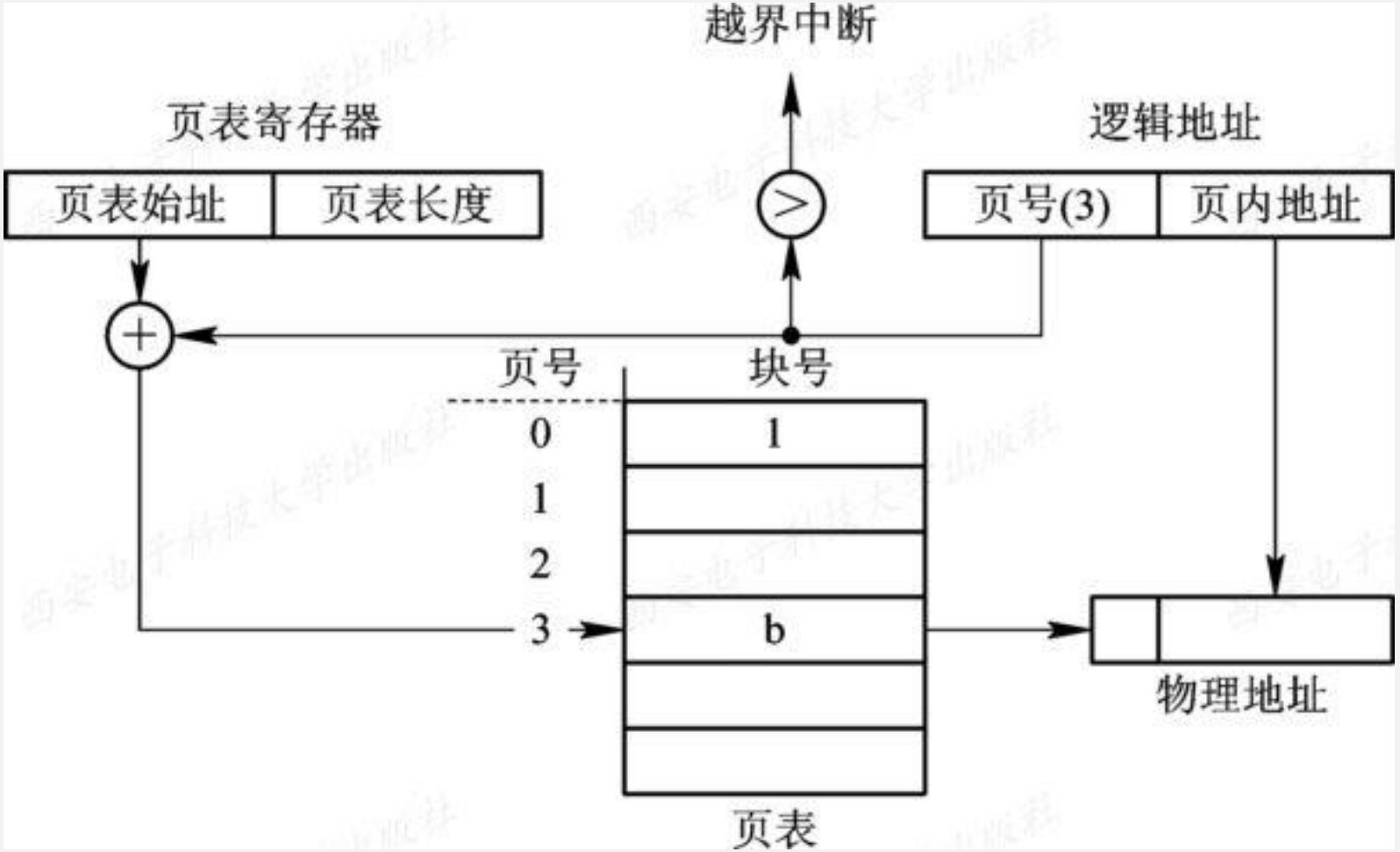
由于每条指令的地址都需要进行变换，因此需要采用硬件来实现。页表功能是由一组专门的寄存器来实现的。

一个页表项用一个寄存器最好，但是成本太高，因此大部分页表是驻留内存的。

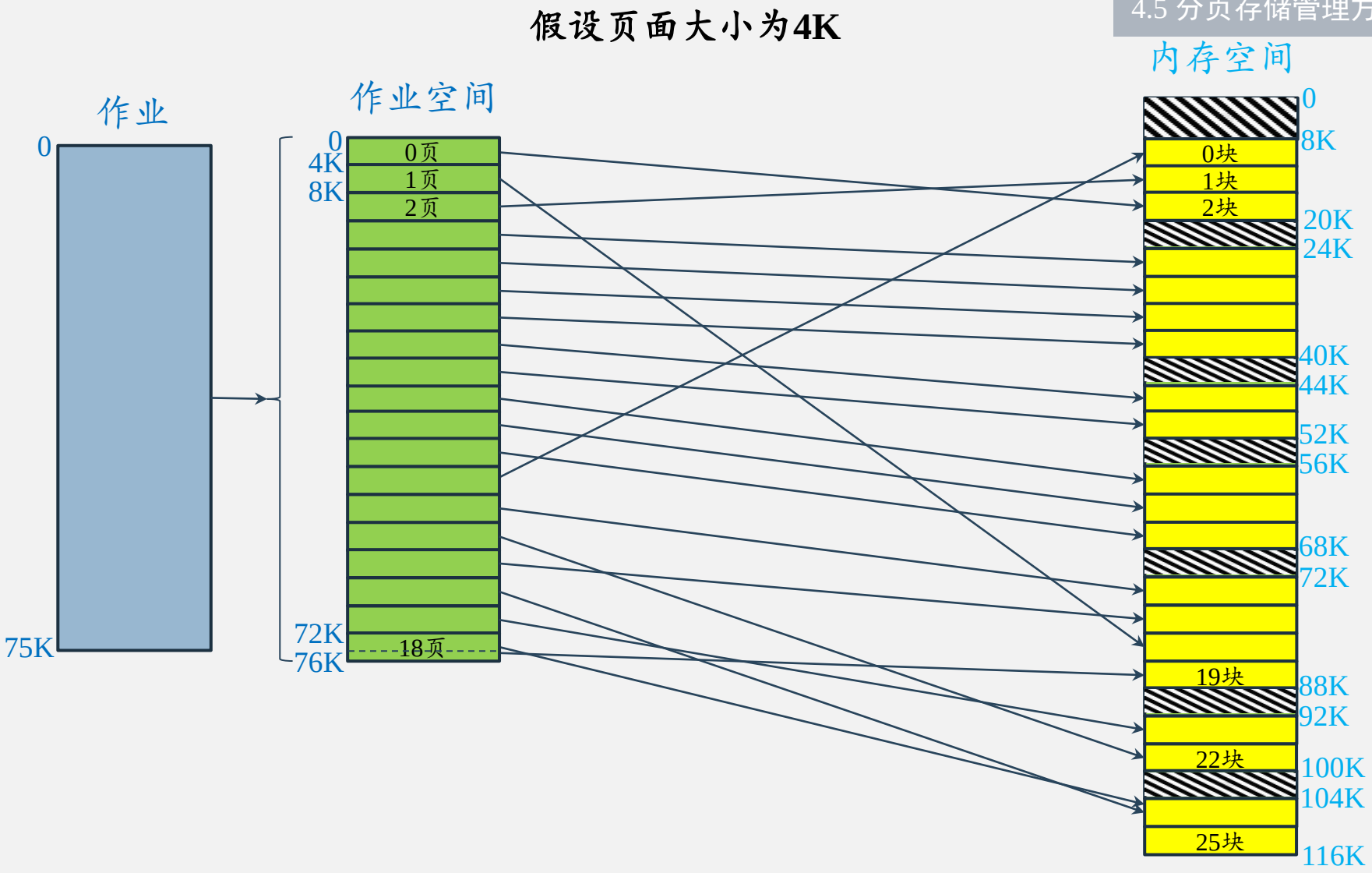
系统中仅设置一个页表寄存器（PTR），存放页表在内存中的起始地址和长度。

进程未执行时，页表起始地址和长度放在该进程的PCB中，执行时才装入页表寄存器。

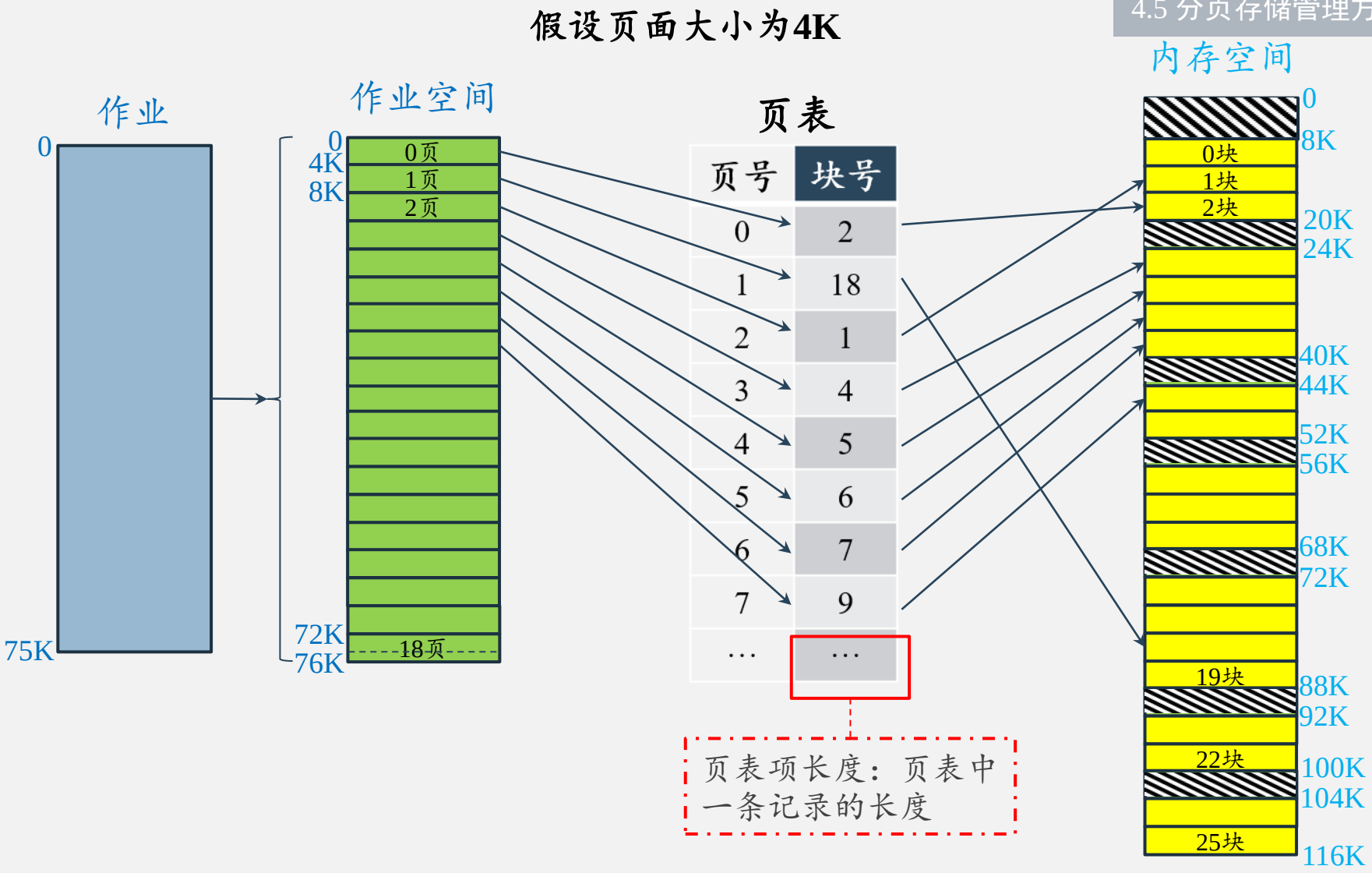
4.5.2 地址变换机构



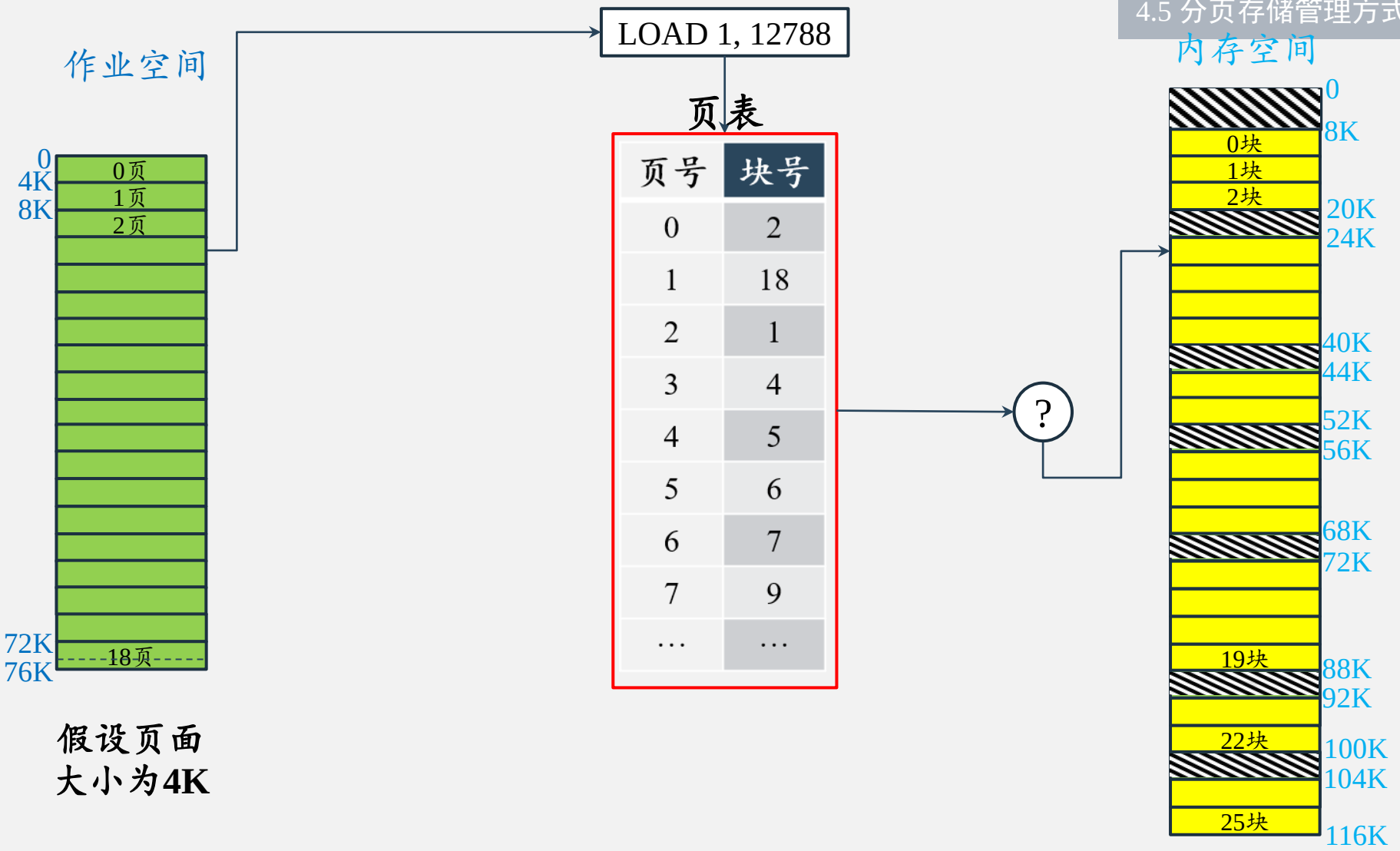
4.5.2 地址变换机构


假设页面大小为4K
作业
0
75K
作业空间
0
4K
8K
72K
76K
0页
1页
2页
18页
内存空间
0
8K
20K
24K
40K
44K
52K
56K
68K
72K
88K
92K
100K
104K
116K
0块
1块
2块
19块
22块
25块

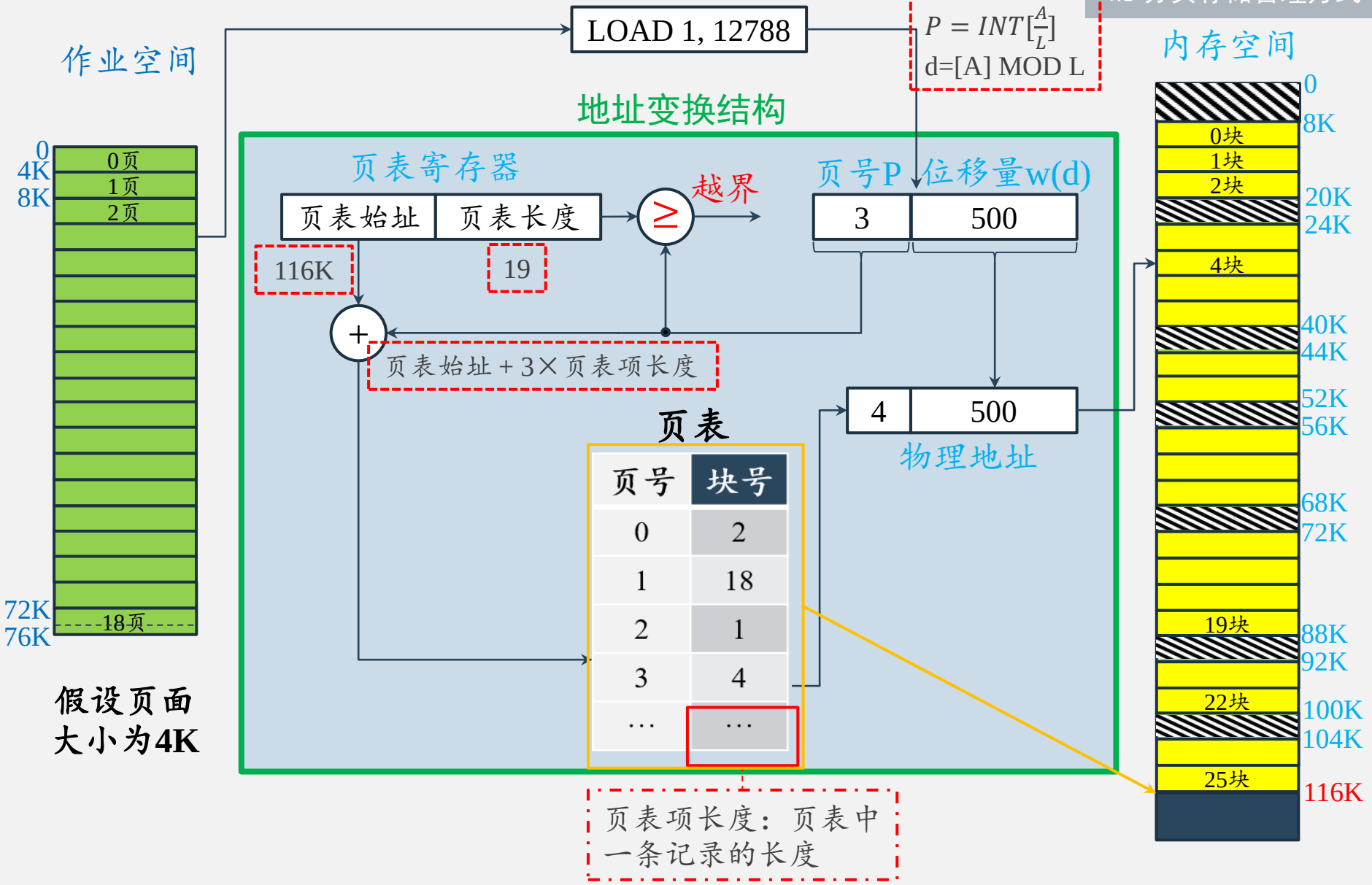
4.5.2 地址变换机构


假设页面大小为4K
作业
0
75K
作业空间
0
4K
8K
72K
76K
0页
1页
2页
18页
页表
页号
块号
0 2
1 18
2 1
3 4
4 5
5 6
6 7
7 9
… …
页表项长度：页表中一条记录的长度
内存空间
0
8K
20K
24K
40K
44K
52K
56K
68K
72K
88K
92K
100K
104K
116K
0块
1块
2块
19块
22块
25块

4.5.2 地址变换机构


作业空间
LOAD 1, 12788
页表
页号
块号
0
2
1
18
2
1
3
4
4
5
5
6
6
7
7
9
…
…
?
内存空间
0
8K
0块
1块
2块
20K
24K
40K
44K
52K
56K
68K
72K
19块
88K
92K
22块
100K
104K
25块
116K
0
4K
8K
0页
1页
2页
72K
76K
18页
假设页面
大小为4K

4.5.2 地址变换机构


作业空间
LOAD 1, 12788
$P = INT[\frac{A}{L}]$
d=[A] MOD L
内存空间
地址变换结构
页表寄存器
页表始址
页表长度
116K
19
≥
越界
页号P
位移量w(d)
3
500
+
页表始址+3×页表项长度
页表
页号
块号
0
2
1
18
2
1
3
4
…
…
4
500
物理地址
0页
1页
2页
18页
0
4K
8K
72K
76K
假设页面大小为4K
0
8K
20K
24K
40K
44K
52K
56K
68K
72K
88K
92K
100K
104K
116K
0块
1块
2块
4块
19块
22块
25块
页表项长度：页表中一条记录的长度

4.5.2 地址变换机构

2. 具有快表的地址变换机构

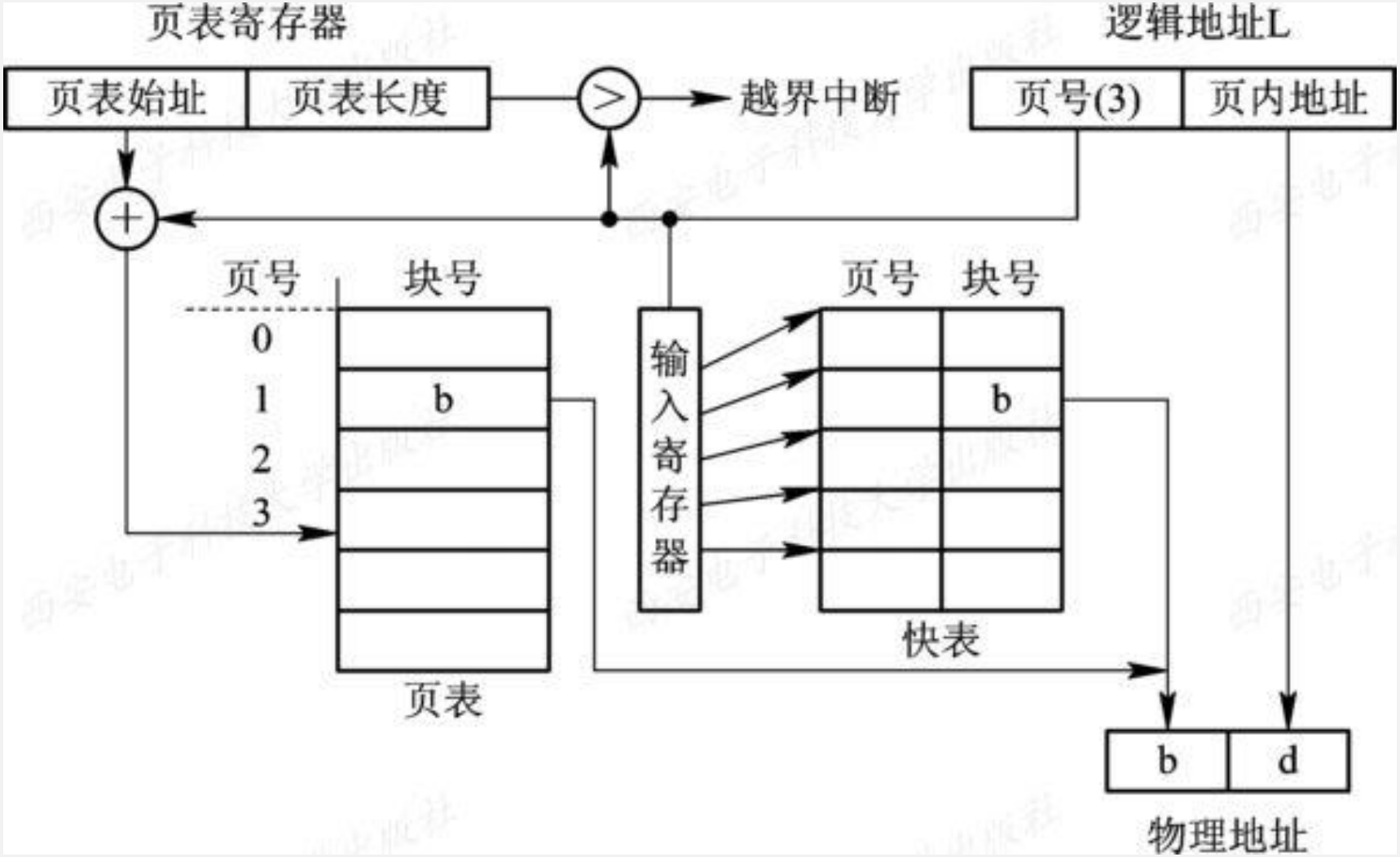
为提高地址变换速度，增设特殊高速缓冲寄存器，又称“联想寄存器”或“快表”。

CPU给出有效地址，在快表中查找页号。

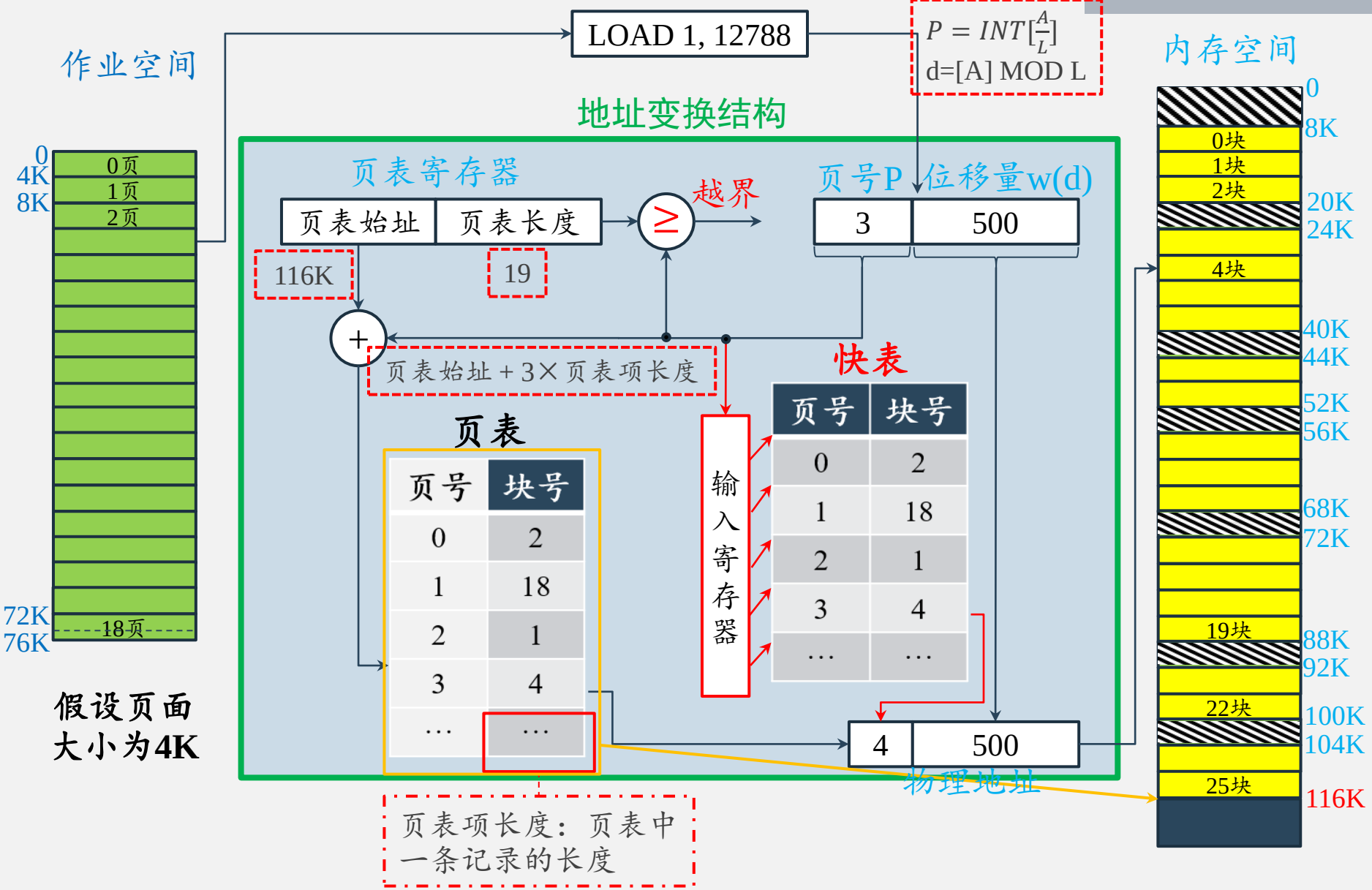
- 如果找到，直接可以读出物理地址；
- 如果找不到，再到内存中找，并重新修改快表（如果快表已满，则需要换出一个被认为是不再需要的页表项）。

一般快表只存放16-512个页表项。据统计，从快表中能找到所需页表项的概率可达到90%以上。

4.5.2 地址变换机构


页表寄存器
页表始址
页表长度
越界中断
逻辑地址L
页号(3)
页内地址
页号
块号
0
1
2
3
b
输入寄存器
页号
块号
b
快表
页表
b
d
物理地址

4.5.2 地址变换机构


LOAD 1, 12788
$P = INT[\frac{A}{L}]$
d=[A] MOD L
作业空间
地址变换结构
内存空间
页表寄存器
页表始址
页表长度
116K
19
越界
≥
页号P
位移量w(d)
3
500
+
页表始址+3×页表项长度
快表
页号
块号
0
2
1
18
2
1
3
4
…
…
输入寄存器
页表
页号
块号
0
2
1
18
2
1
3
4
…
…
4
500
物理地址
0页
1页
2页
18页
0
4K
8K
72K
76K
0块
1块
2块
4块
19块
22块
25块
0
8K
20K
24K
40K
44K
52K
56K
68K
72K
88K
92K
100K
104K
116K
假设页面大小为4K
页表项长度：页表中一条记录的长度

4.5.3 访问内存的有效时间

从进程发出指定逻辑地址的访问请求，经过地址变换，到在内存中找到对应的实际物理地址单元并取出数据，所需要花费的总时间，称为内存的有效访问时间(Effective Access Time，EAT)。假设访问一次内存的时间为t，在基本分页存储管理方式中，有效访问时间分为第一次访问内存时间(即查找页表对应的页表项所耗费的时间t)与第二次访问内存时间(即将需要的指令或数据取出所耗费的时间t)之和：

$$EAT = t + t = 2t$$

4.5.3 访问内存的有效时间

在引入快表的分页存储管理方式中，通过快表查询，可以直接得到逻辑页所对应的物理块号，由此拼接形成实际物理地址，减少了一次内存访问，缩短了进程访问内存的有效时间。

但是，由于快表的容量限制，不可能将一个进程的整个页表全部装入快表，所以在快表中查找到所需表项存在着命中率的问题。所谓命中率，是指使用快表并在其中成功查找到所需页面的表项的比率。这样，在引入快表的分页存储管理方式中，有效访问时间的计算公式即为：

$$EAT=a\times\lambda+(t+\lambda)(1—a)+t=2t+\lambda—t\times a$$

上式中，λ表示查找快表所需要的时间，a表示命中率，t表示访问一次内存所需要的时间。

4.5.3 访问内存的有效时间

可见，引入快表后的内存有效访问时间分为查找到逻辑页对应的页表项的平均时间$a \times \lambda + (t + \lambda)(1 - a)$，以及对应实际物理地址的内存访问时间t。假设对快表的访问时间λ为20 ns(纳秒)，对内存的访问时间t为100 ns，则下表中列出了不同的命中率a与有效访问时间的关系：

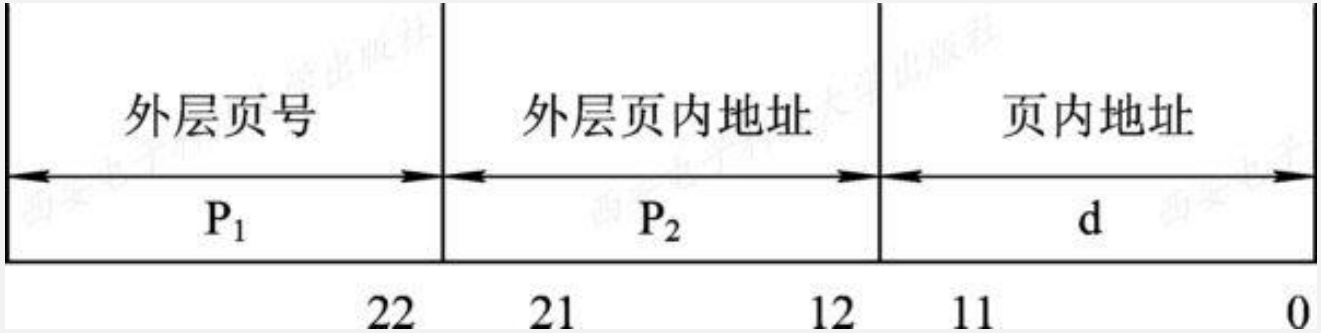
命中率 (%) a	有效访问时间 EAT
0	220
50	170
80	140
90	130
98	122

4.5.4 两级和多级页表

现代大多数计算机系统逻辑地址空间非常大，页表也随之变得非常大。如果需要连续空间，不太现实。

1. 两级页表(Two-Level Page Table)

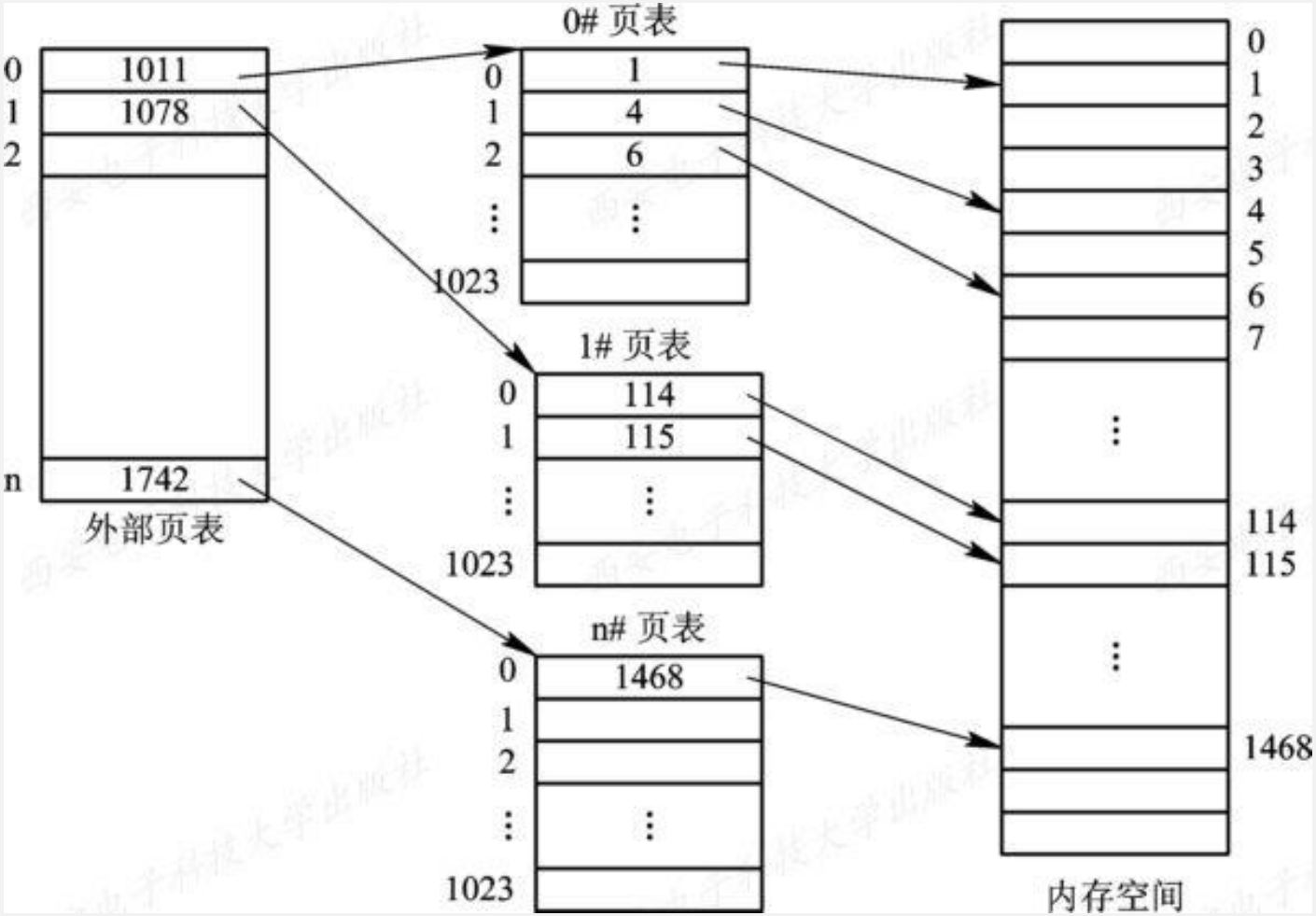
针对难于找到大的连续的内存空间来存放页表的问题，可利用将页表进行分页的方法，使每个页面的大小与内存物理块的大小相同，并为它们进行编号，即依次为0# 页、1# 页，…，n# 页，然后离散地将各个页面分别存放在不同的物理块中。同样，也要为离散分配的页表再建立一张页表，称为外层页表(Outer Page Table)，在每个页表项中记录了页表页面的物理块号。



4.5.4 两级和多级页表

为了方便实现地址变换，在地址变换机构中，同样需要增设一个外层页表寄存器，用于存放外层页表的始址，并利用逻辑地址中的外层页号作为外层页表的索引，从中找到指定页表分页的始址，再利用P2作为指定页表分页的索引，找到指定的页表项，其中即含有该页在内存的物理块号，用该块号P和页内地址d即可构成访问的内存物理地址。图4-18示出了两级页表时的地址变换机构。

4.5.4 两级和多级页表


0
1
2
n
1011
1078
1742
外部页表
0# 页表
0
1
2
1023
1
4
6
1# 页表
114
115
n# 页表
1468
0
1
2
3
4
5
6
7
114
115
1468
内存空间

4.5.4 两级和多级页表

2. 多级页表

对于64位计算机，能够支持1844744T的地址空间，三级页表结构也很难实现。因此，今年推出的64位操作系统，把可直接寻址的存储器空间减少为45位左右，然后利用三级页表。

4.5.5 反置页表(Inverted Page Table)

1. 反置页表的引入

现代计算机系统中，通常允许一个进程的逻辑地址空间非常大，因此就需要有许多的页表项，而因此也会占用大量的内存空间。

为减少页表占用的内存空间，引入反置页表，IBM的很多系统中都采用。

为每一个物理块设置一个页表项，并将它们按物理块号排序，内容是所属的进程和页号。每次进程来的时候，按照进程查找。

4.5.5 反置页表(Inverted Page Table)

2. 地址变换

根据进程标识符和页号，去检索反置页表。如果检索到与之匹配的页表项，则该页表项(中)的序号i便是该页所在的物理块号，若检索了整个反置页表仍未找到匹配的页表项，则表明此页尚未装入内存。

当内存容量很大时，对进程在表中位置的查找也相当困难。

第四章 存储器管理

4.6 分段存储管理方式

4.6 分段存储管理方式

单道→多道

使用不同大小的程序要求

提高内存利用率

满足程序员编程和使用方便

单一连续分配→固定分区分配

固定分区分配→动态分区分配

连续分配→离散分配（分页）

出现分段（需要语言编译程序支持）

4.6.1 分段存储管理方式的引入

1. 方便编程

2. 信息共享

页只是存放信息的物理单位(块)，无完整的逻辑意义， 段有。

3. 信息保护

信息保护同样是以信息的逻辑单位为基础的，而且经常是以一个过程、函数或文件为基本单位进行保护的。

4. 动态增长

5. 动态链接

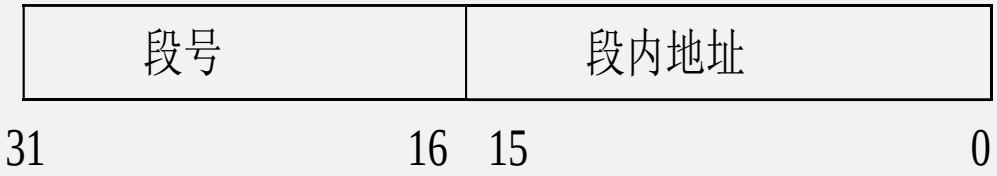
在程序运行过程中，当需要调用某个目标程序时，才将该段(目标程序)调入内存并进行链接。 动态链接要求的是以目标程序（段）作为链接的基本单位。

4.6.2 分段系统的基本原理

1. 分段

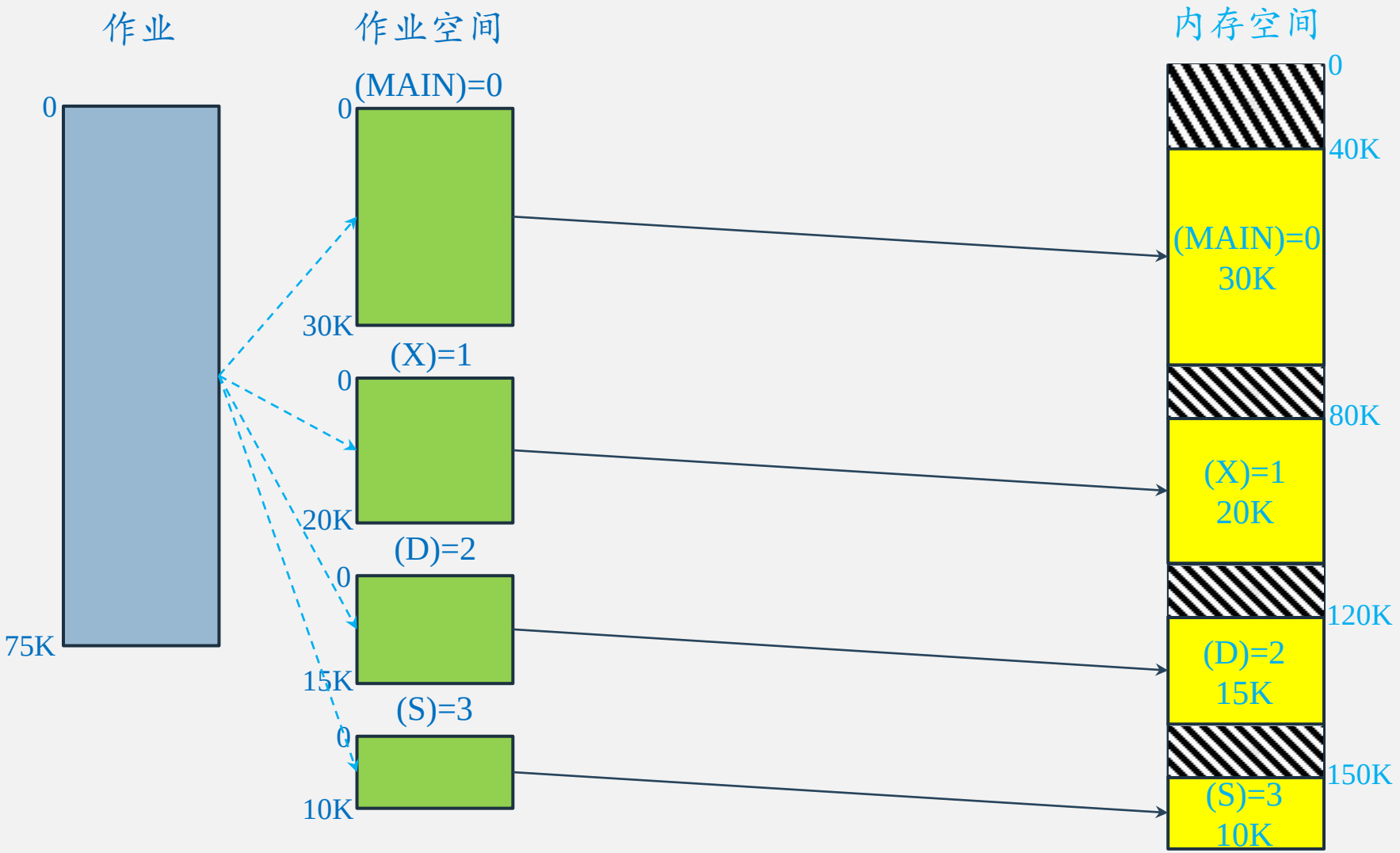
在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及栈段S等，如图4-19所示。

分段地址中的地址具有如下结构：



该地址结构中，允许一个作业最长有64K个段，每个段最长64KB。

4.6.2 分段系统的基本原理



4.6.2 分段系统的基本原理

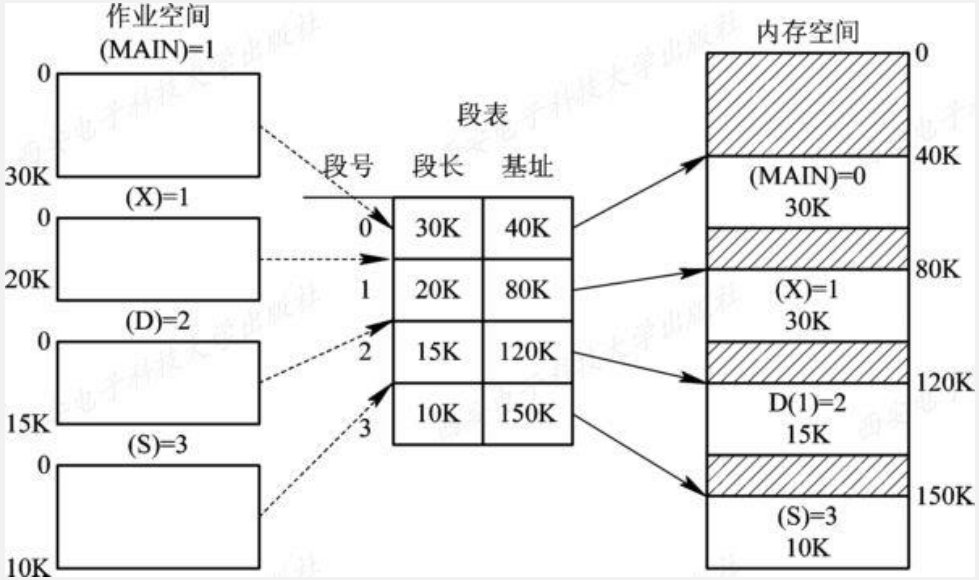
2. 段表

在分段式存储管理系统中，则是为每个分段分配一个连续的分区。进程中的各个段，可以离散地装入内存中不同的分区中。

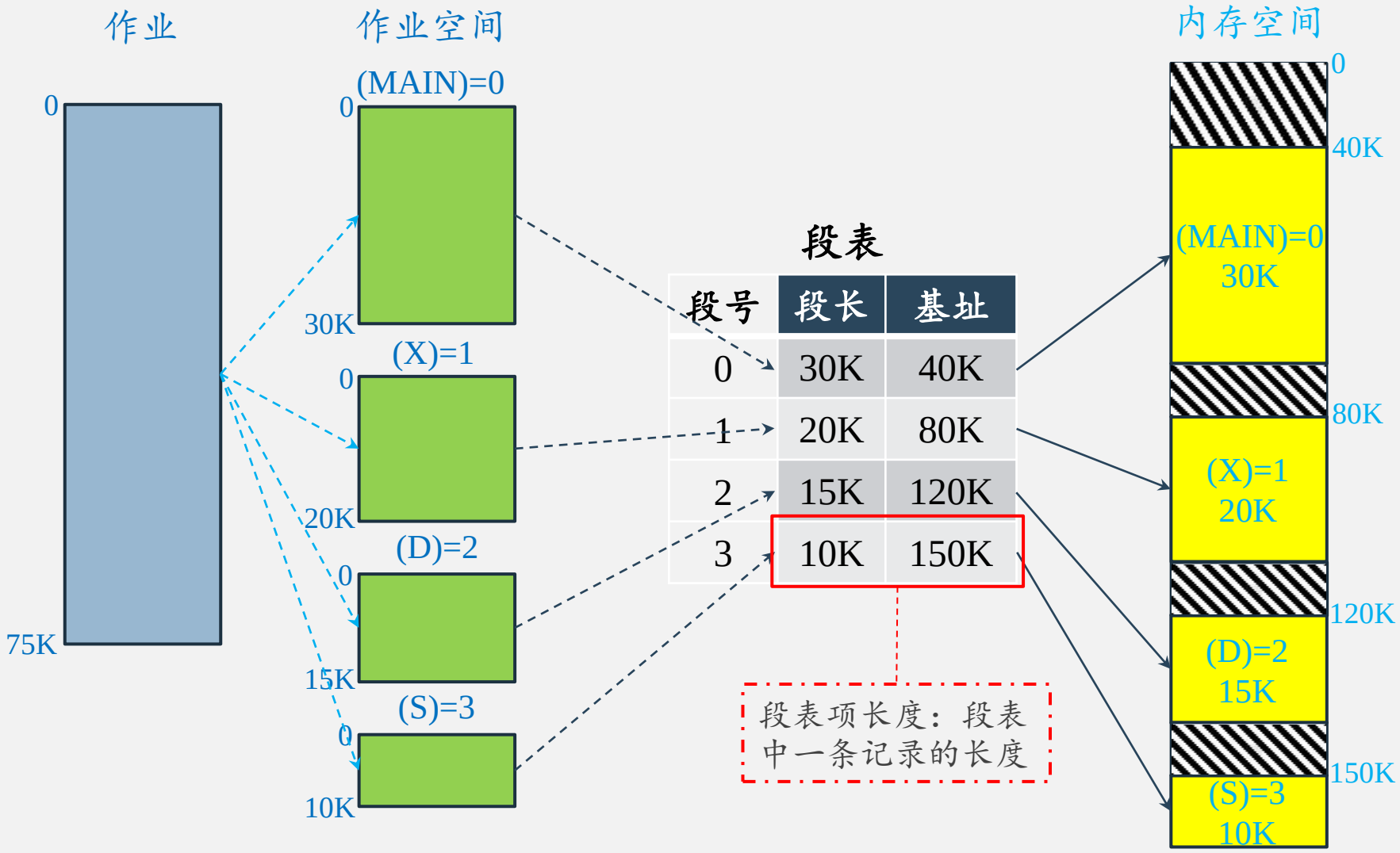
需要为每个进程建立一张段映射表，简称“段表”。

每个段占一个表项，记录该段在内存中的起始地址（基址）和段长。

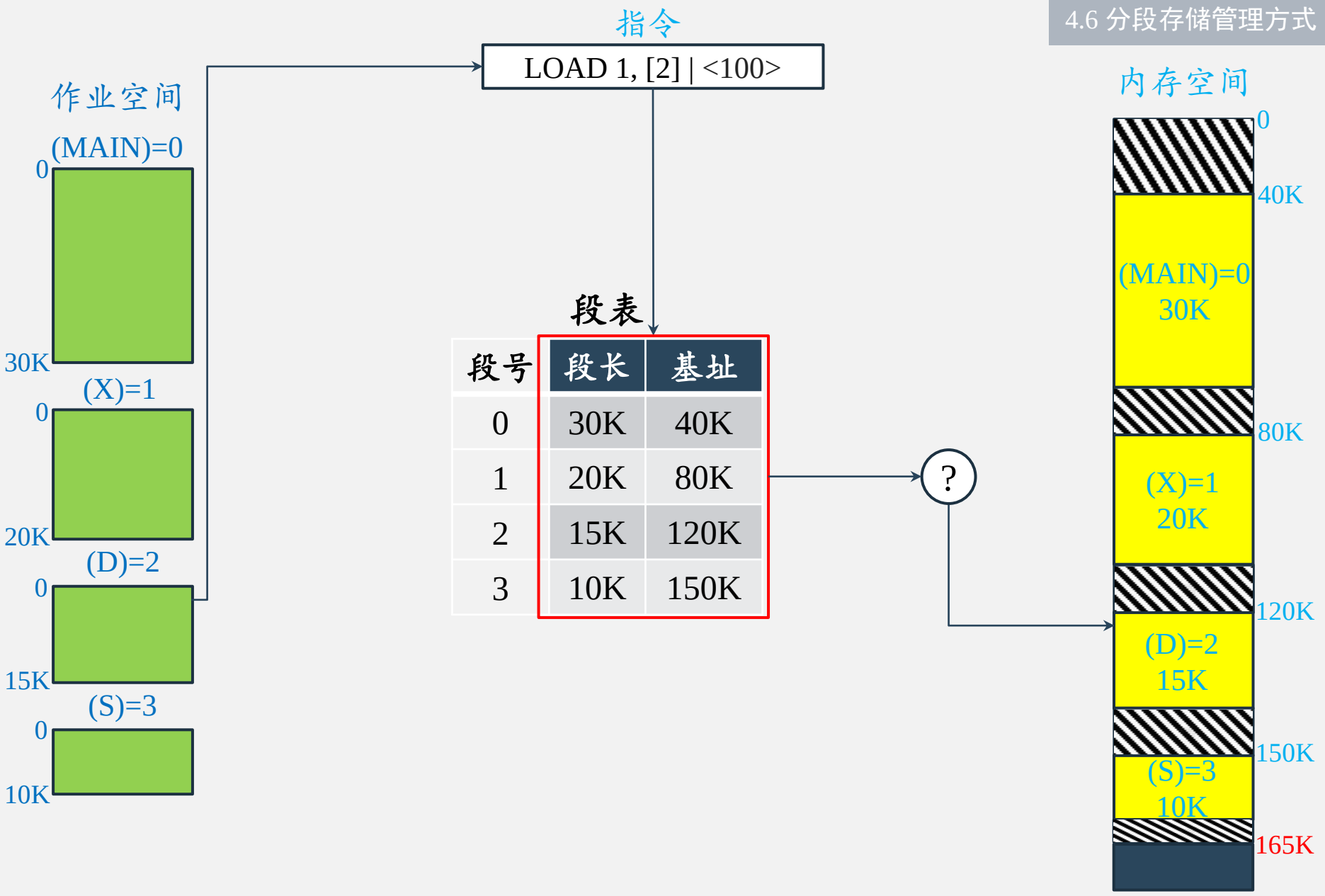
段表可以放在一组寄存器中，但常见的是将其放在内存中。



4.6.2 分段系统的基本原理



4.6.2 分段系统的基本原理



4.6.2 分段系统的基本原理

3. 地址变换机构

在系统中设置段表寄存器，用于存放段表始址和段表长度TL。在进行地址变换时：

系统将逻辑地址中的段号与段表长度TL进行比较。

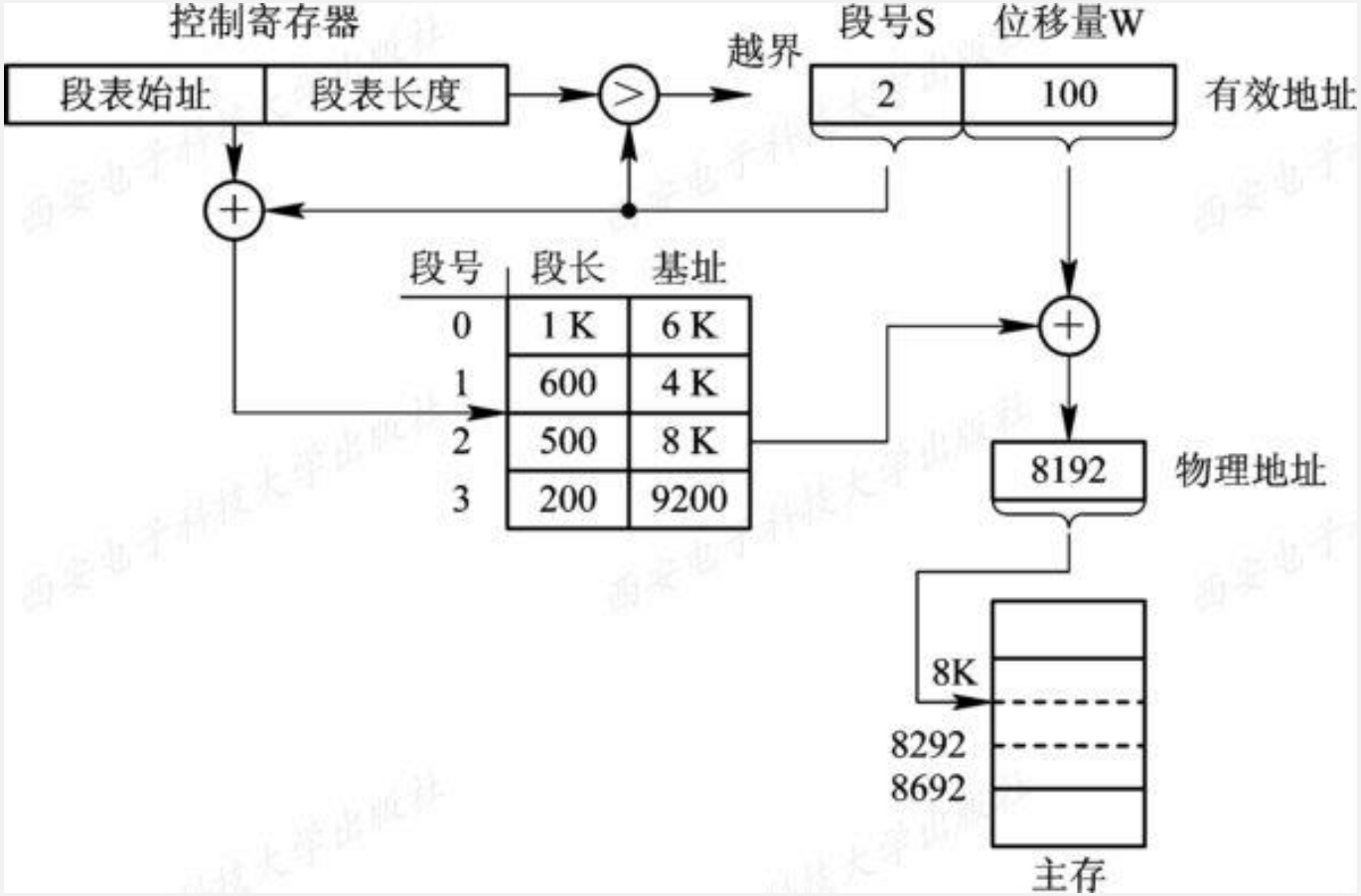
若S≥TL，产生越界中断信号。

若未越界，则读出该段在内存的起始地址。再检查段内地址d是否超过该段的段长SL。

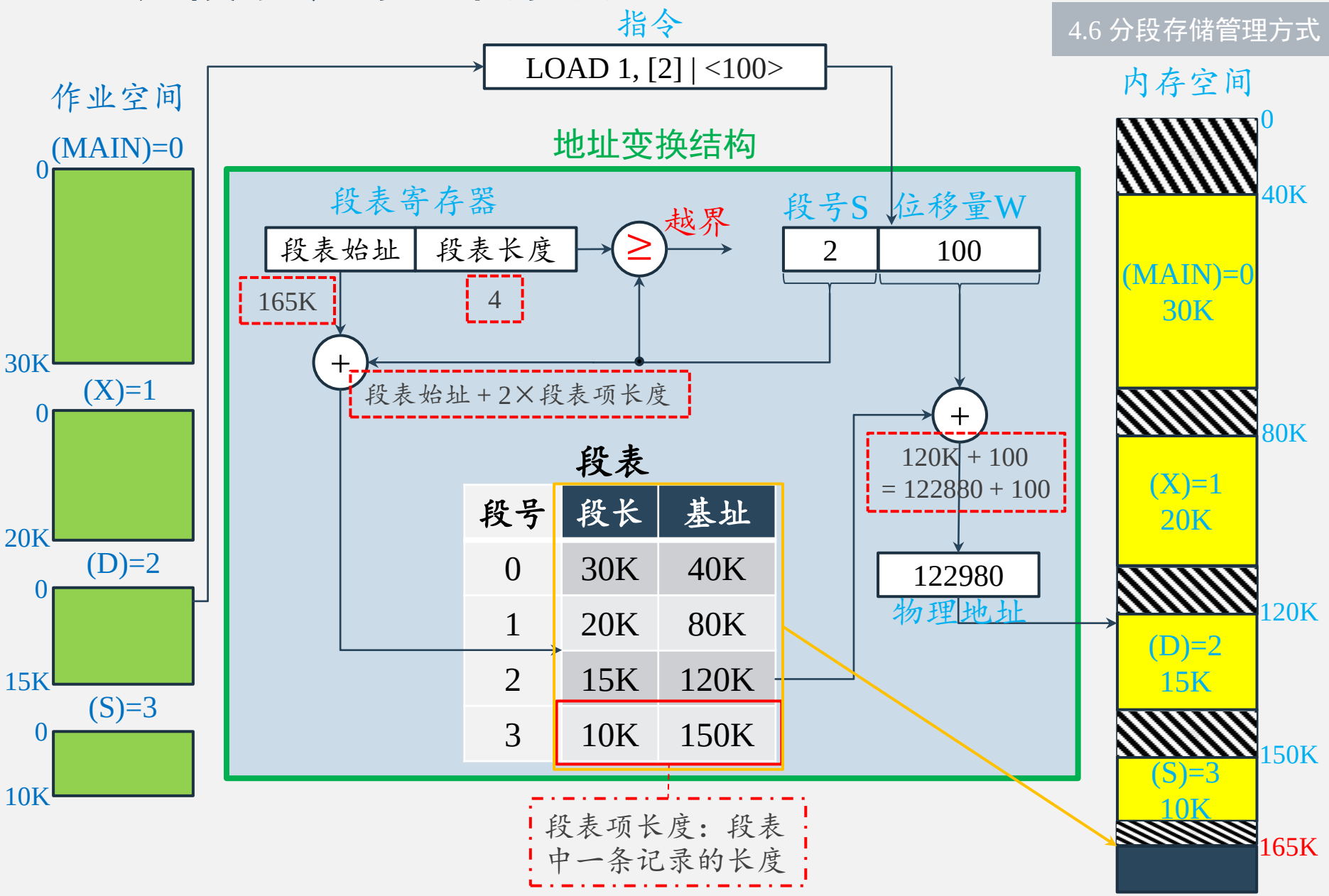
若超过，发出越界中断信号。

若未越界，则得到要访问的内存物理地址。

4.6.2 分段系统的基本原理



4.6.2 分段系统的基本原理



4.6.2 分段系统的基本原理

4. 分页和分段的主要区别

(1) 页是信息的物理单位，分页仅仅是系统管理上的需要，对用户不可见。段是逻辑单位，分段的目的是能更好地满足用户的需要。

(2) 页的大小固定且由系统决定，每个系统中只能有一种大小的页面。段的长度不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

4.6.3 信息共享

可重入代码（Reentrant Code）又称为“纯代码”（Pure Code），是一种允许多个进程同时访问的代码。

1. 分页系统中对程序和数据的共享

在分页系统中，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。

需要在每个需要访问共享程序的进程中维护很大的页表，而且各个进程中的内容相同。

4.6.3 信息共享

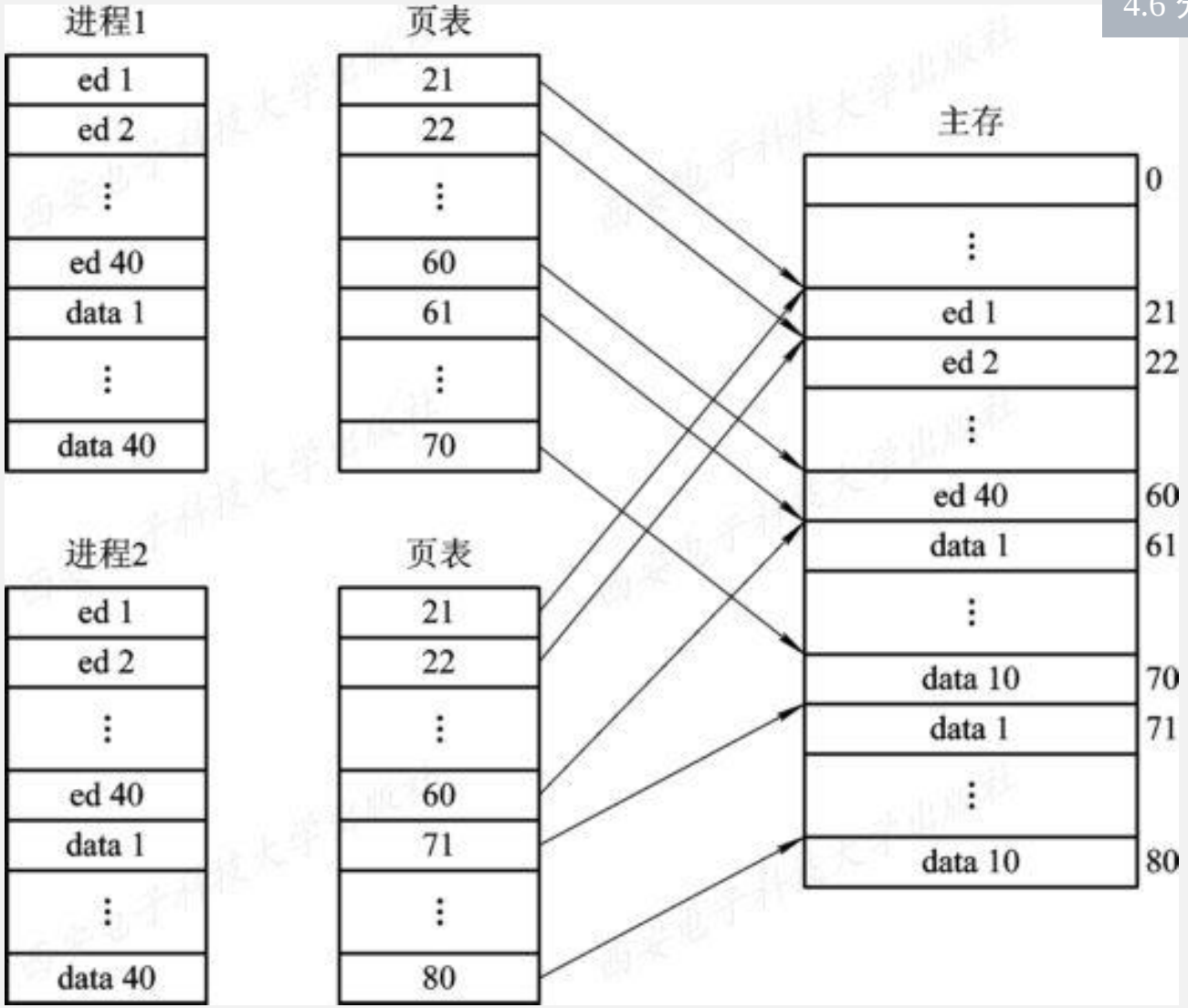


图4-21 分页系统中共享editor的示意图

4.6.3 信息共享

2. 分段系统中程序和数据的共享

在分段系统中，由于是以段为基本单位的，不管该段有多大，我们都只需为该段设置一个段表项，因此使实现共享变得非常容易。只需在(每个)进程中，为共享程序设置一个段表项即可。

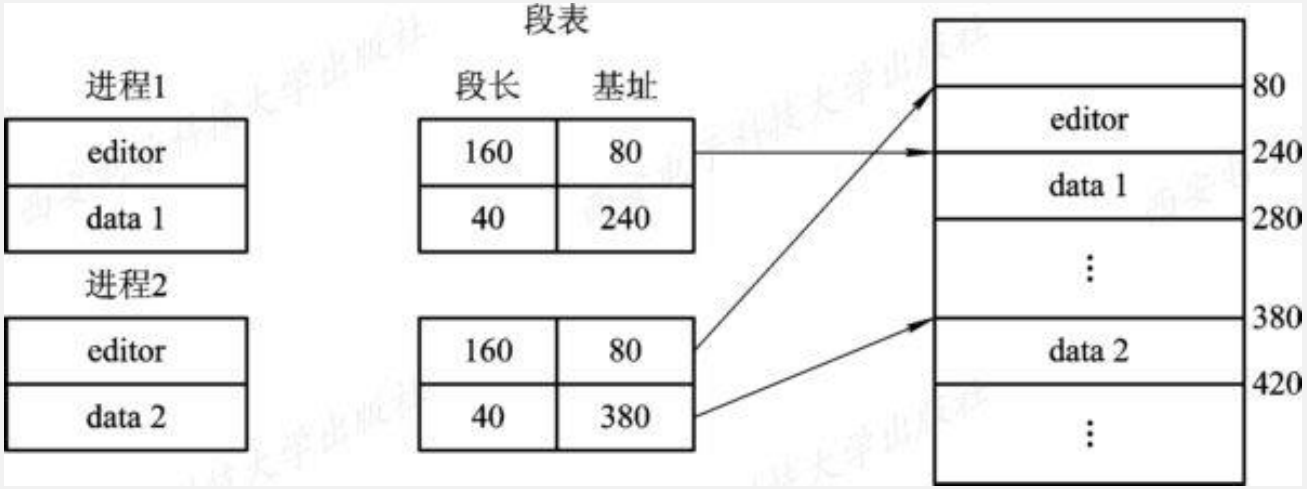


图4-22 分段系统中共享editor的示意图

4.6.4 段页式存储管理方式

1. 基本原理

先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。

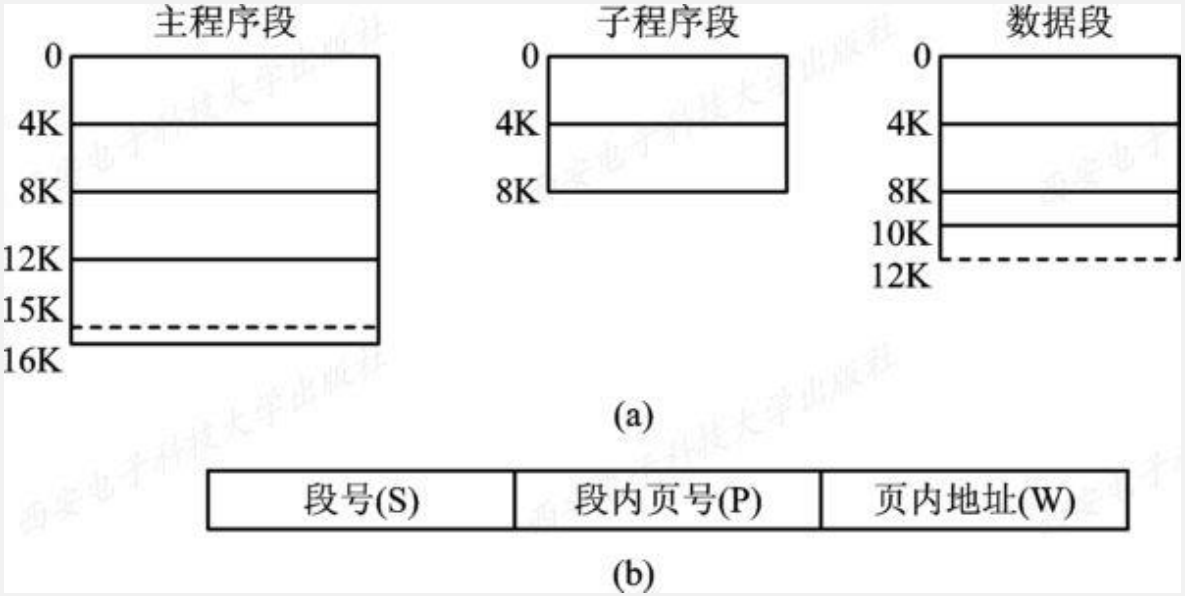
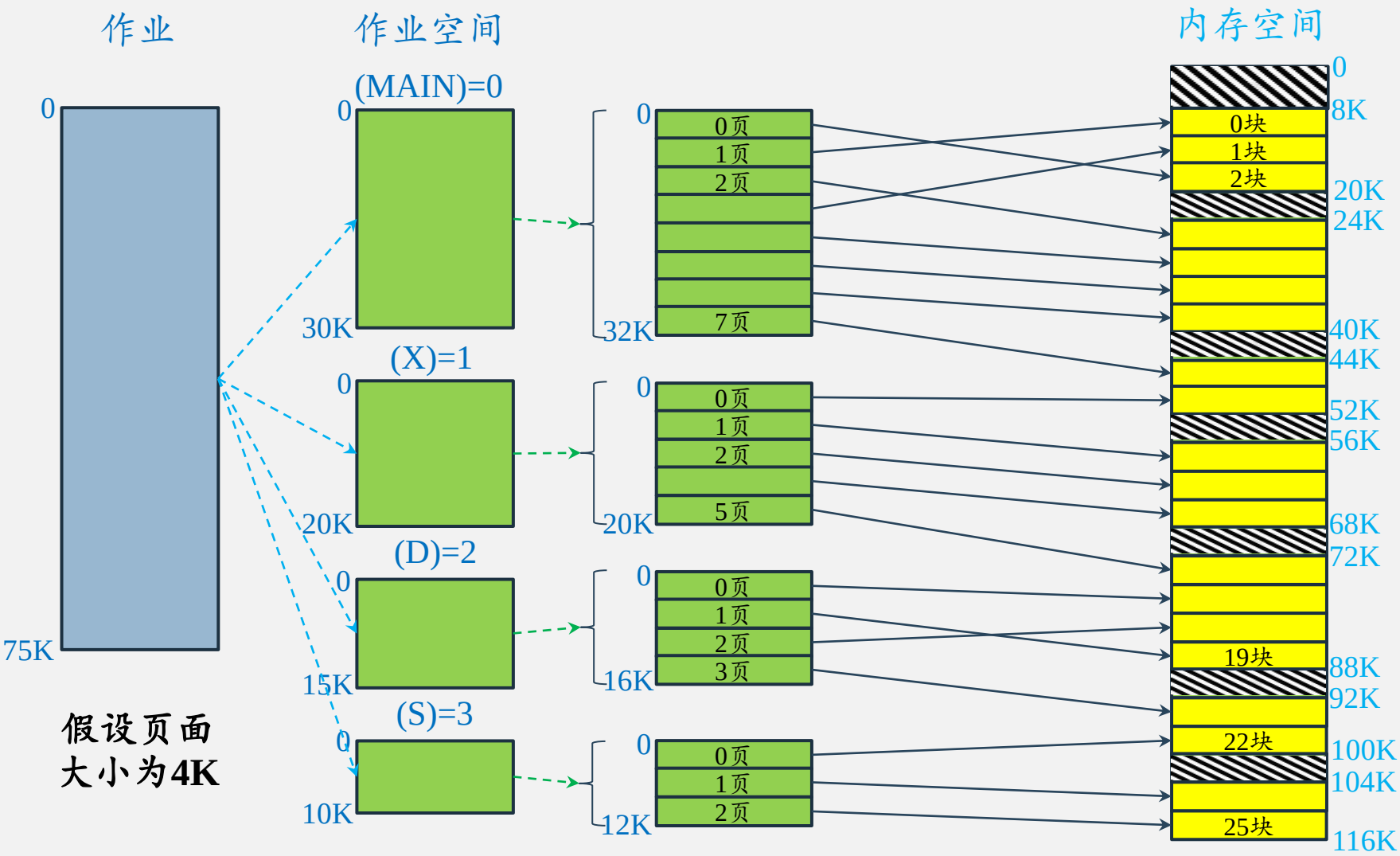


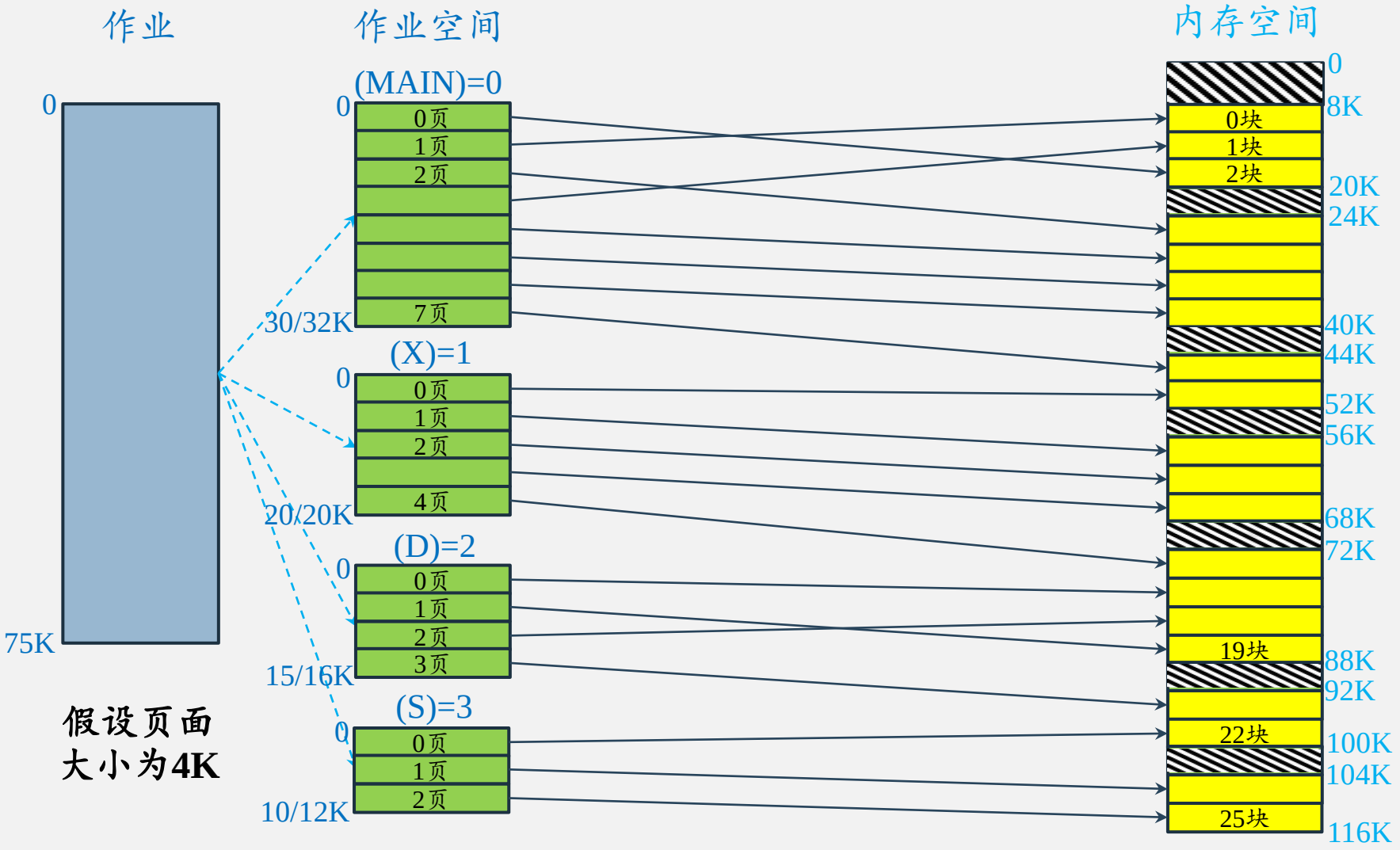
图4-23 作业地址空间和地址结构

4.6.4 段页式存储管理方式



假设页面大小为4K

4.6.4 段页式存储管理方式



4.6.4 段页式存储管理方式

在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置段表和页表。段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度。图4-24示出了利用段表和页表进行从用户地址空间到物理(内存)空间的映射。

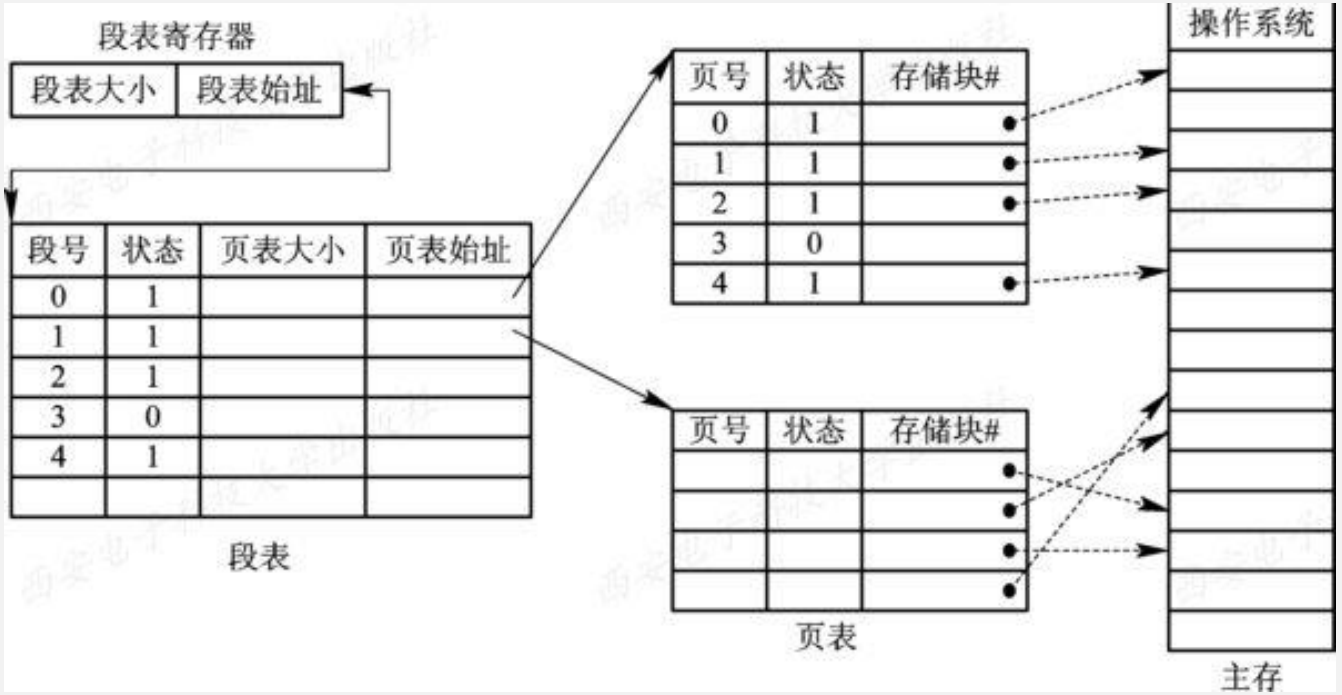
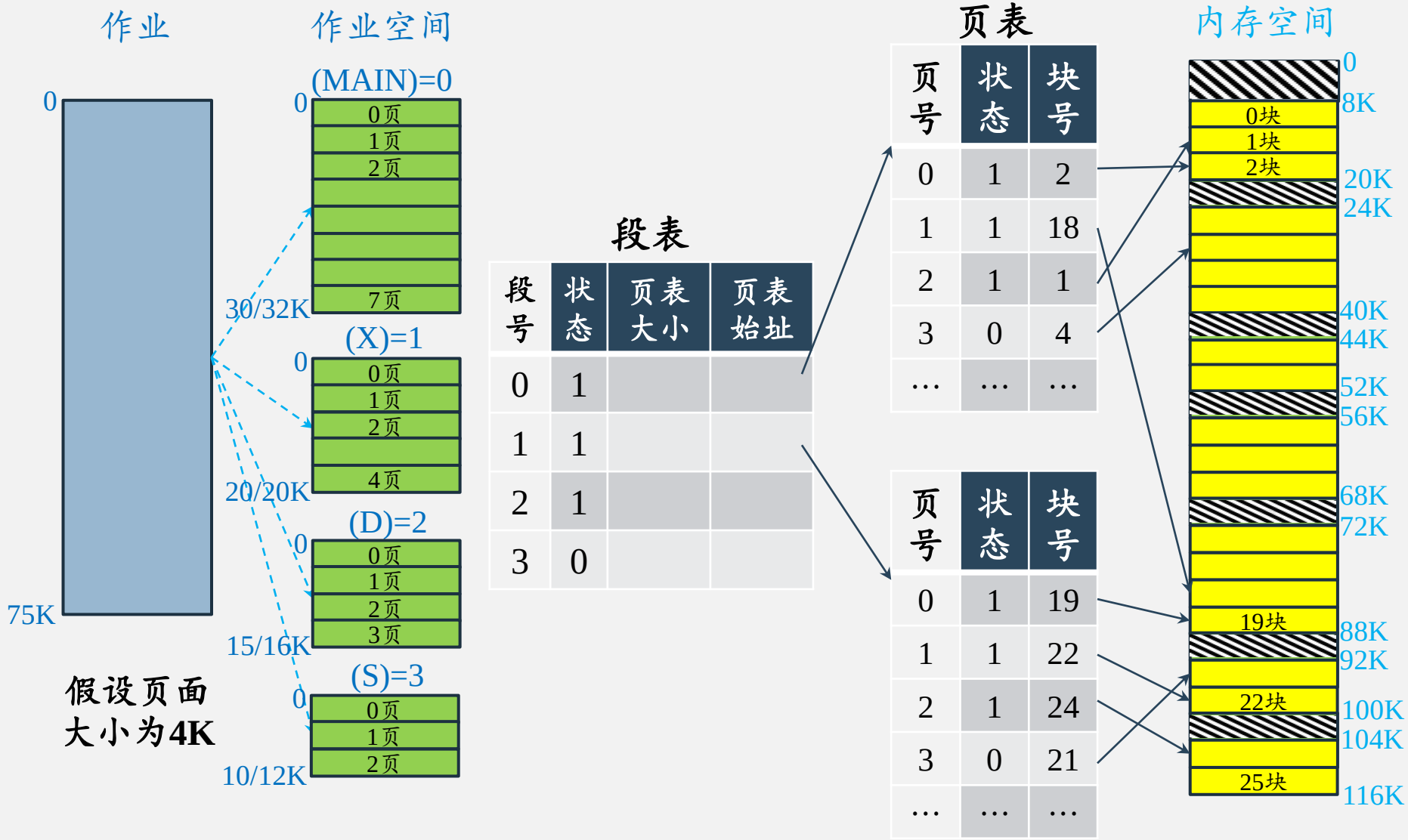


图4-24 利用段表和页表实现地址映射

4.6.4 段页式存储管理方式



4.6.4 段页式存储管理方式

2. 地址变换过程

配置一个段表寄存器，存放段表始址和段长TL。

进行地址变换时：

首先利用段号S，将它与段长TL进行比较，若未越界

利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，

利用逻辑地址中的段内页号P来获得对应页的页表项位置，读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。

4.6.4 段页式存储管理方式

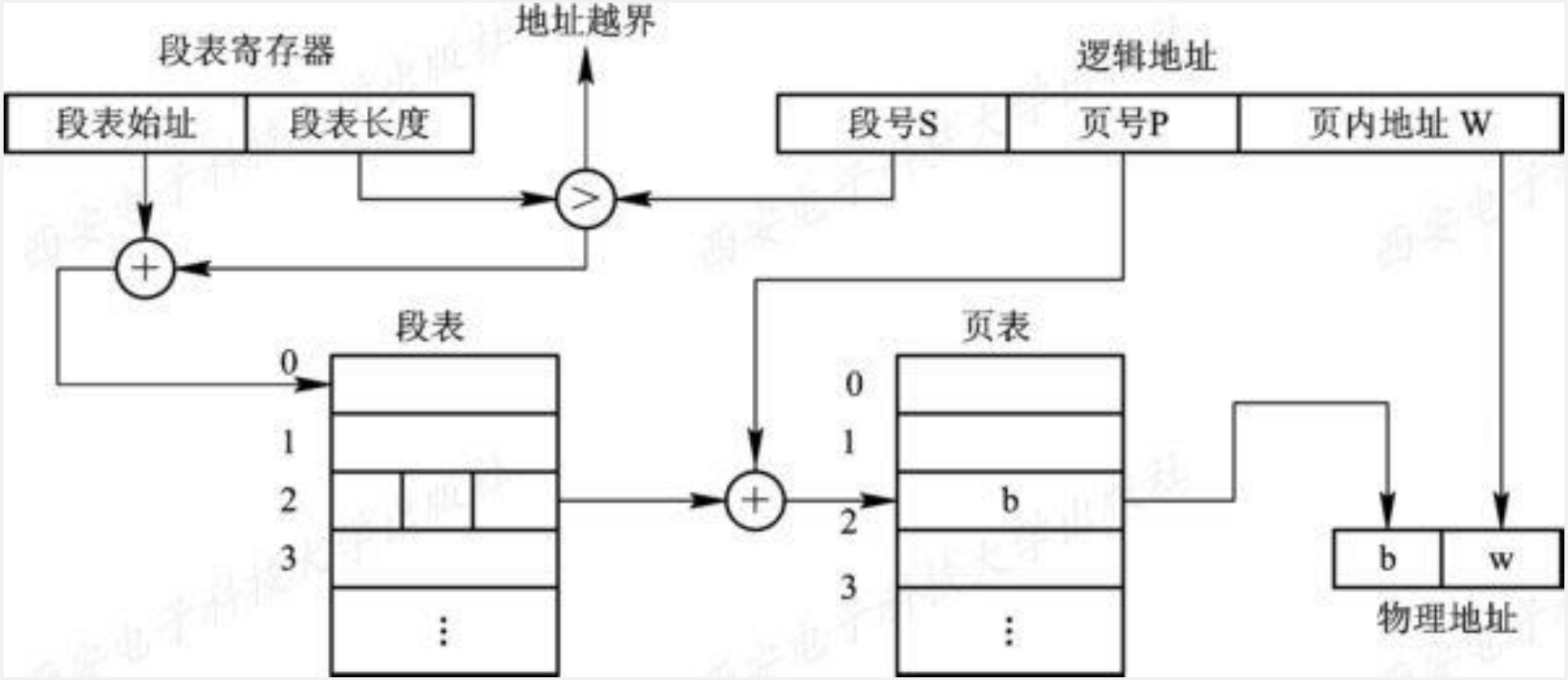
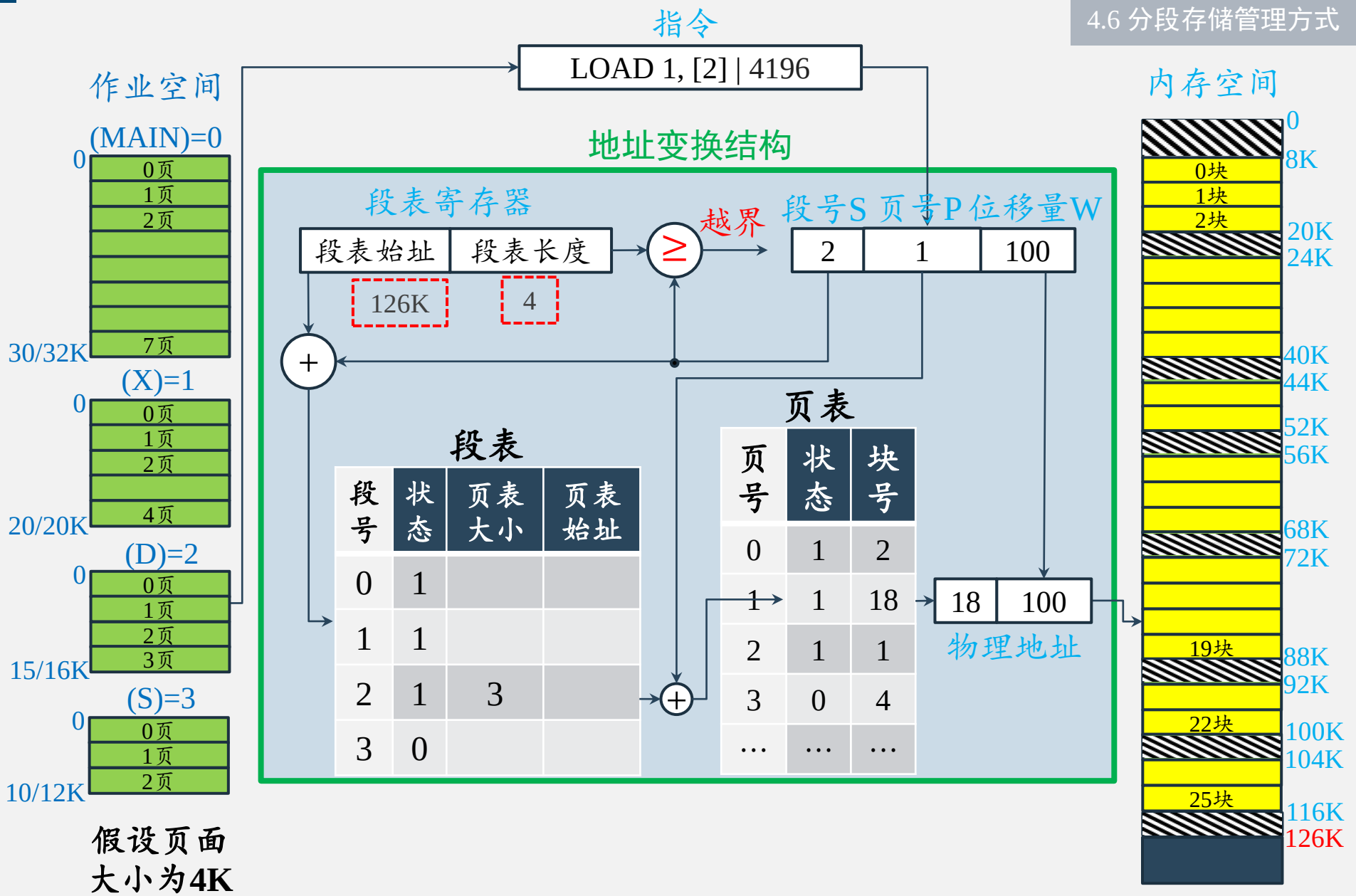


图4-25 段页式系统中的地址变换机构

4.6.4 段页式存储管理方式



4.6.4 段页式存储管理方式

为了获得一条指令或数据，需要三次访问内存。

1）访问内存中的段表，取得页表地址。

2）访问内存中的页表，取得物理块号，并与页内地址合并成物理地址。

3）取出内存中的指令或数据。

为了提高速度，可以增设一个高速缓冲寄存器，方法同前。

谢谢大家！ Q & A