



经典教材《计算机操作系统》**最新版**

第5章 存储器管理

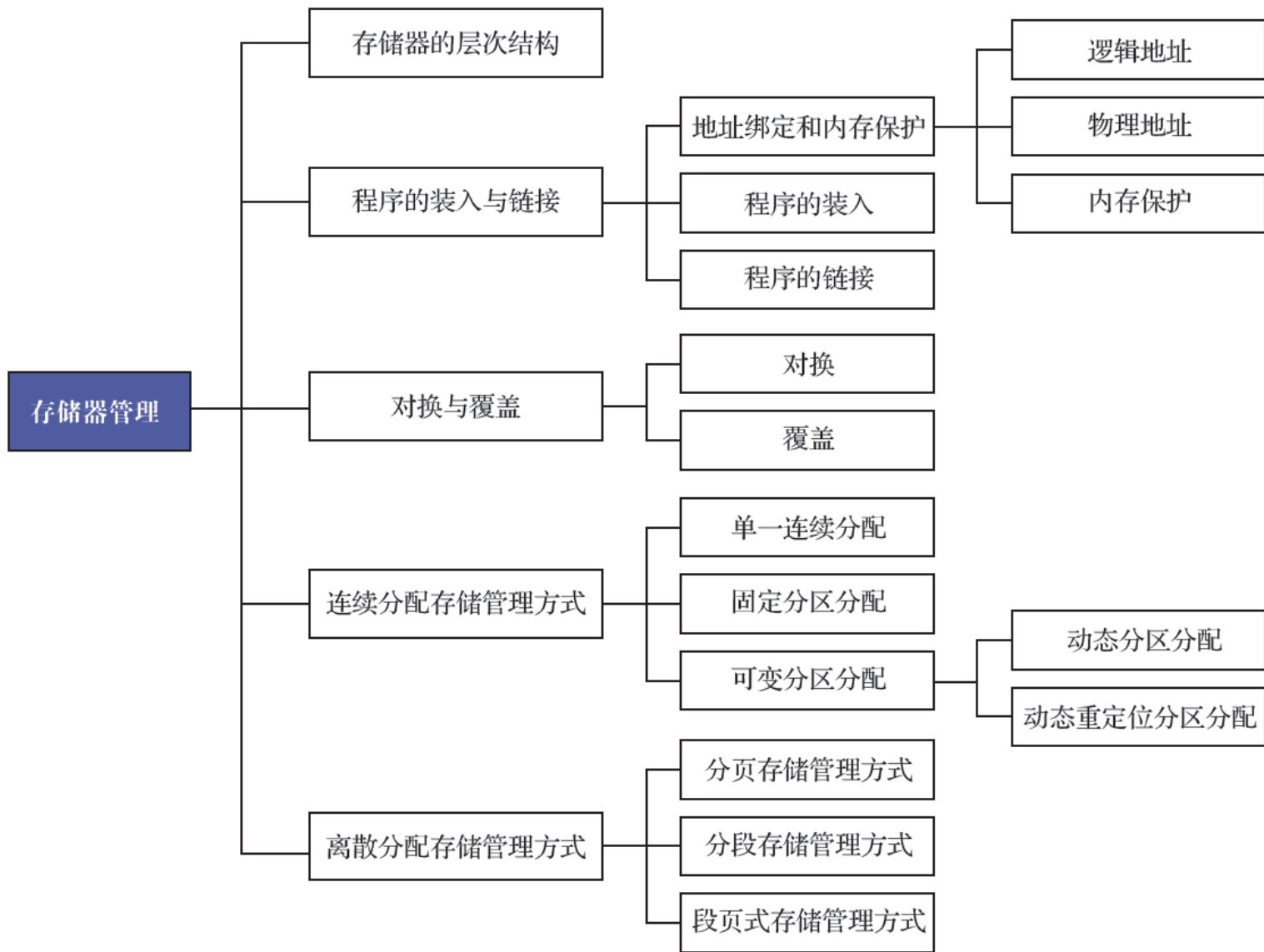
主讲教师：陆丽萍





第5章知识导图

第1章	操作系统引论
第2章	进程的描述与控制
第3章	处理机调度与死锁
第4章	进程同步
第5章	存储器管理
第6章	虚拟存储器
第7章	输入/输出系统
第8章	文件管理





内容导航:

-  **5.1 存储器的层次结构**
-  5.2 程序的装入和链接
-  5.3 对换与覆盖
-  5.4 连续分配存储管理方式
-  5.5 分页存储管理方式
-  5.6 分段存储管理方式
-  5.7 基于IA-32/x86-64架构
的内存管理策略

第5章 存储器管理



存储器的层次结构



存储层次

- CPU寄存器
- 主存：高速缓存、主存储器、磁盘缓存
- 辅存：固定磁盘、可移动介质



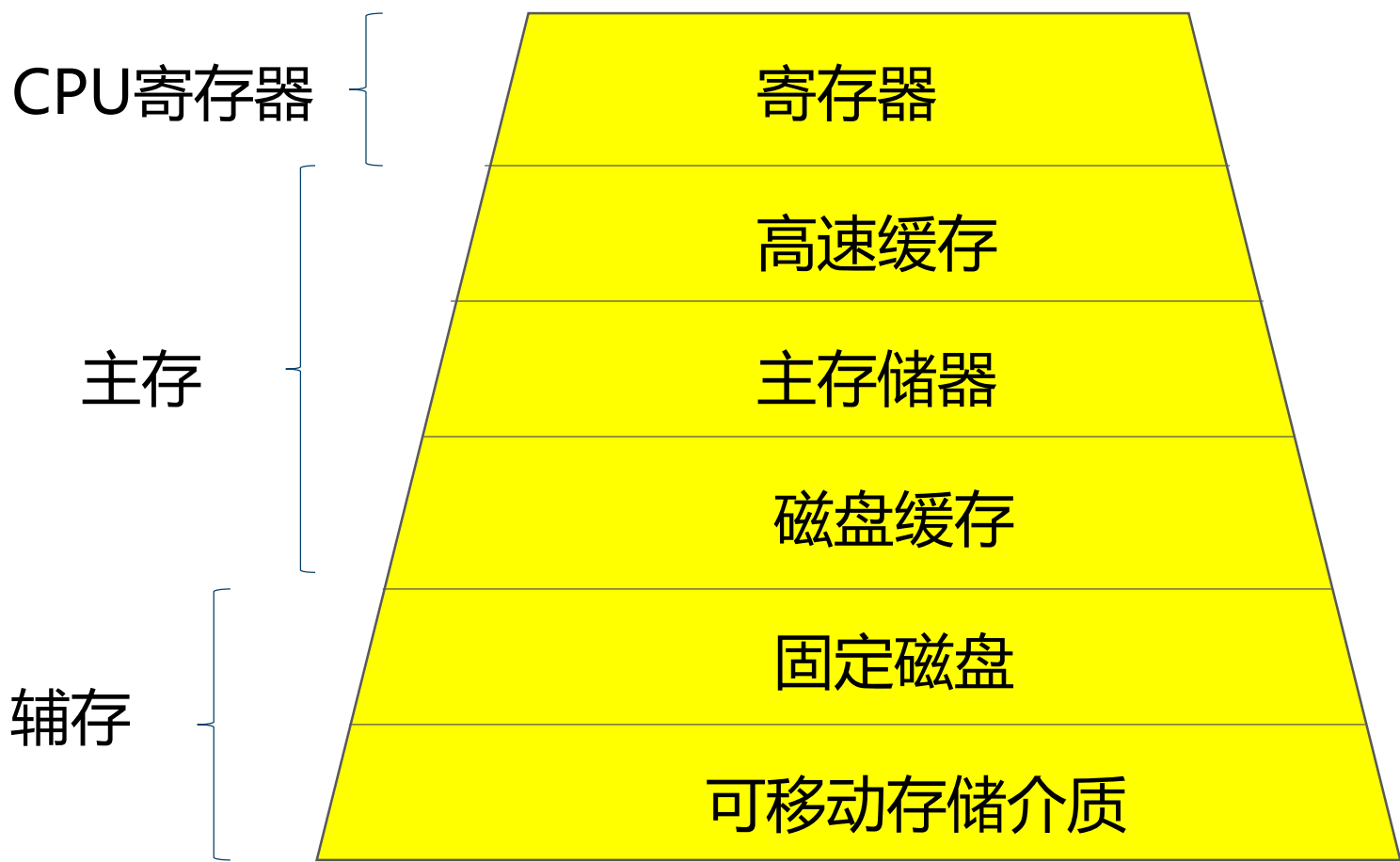
层次越高，访问速度越快，价格也越高，存储容量也最小



寄存器和主存掉电后存储的信息不再存在



辅存的信息长期保存



01

可执行存储器：寄存器和主存储器。

02

主存储器：内存或主存。

03

寄存器：访问速度最快，与CPU协调工作，价格贵。

04

高速缓存：介于寄存器和存储器之间。

- 备份主存主常用数据，减少对主存储器的访问次数；
- 缓和内存与处理机之间的矛盾。

05

磁盘缓存

- 暂时存放频繁使用的一部分磁盘数据和信息；
- 缓和主存和I/O设备在速度上的不匹配；
- 利用主存的部分空间，主存可看成辅存的高速缓存。



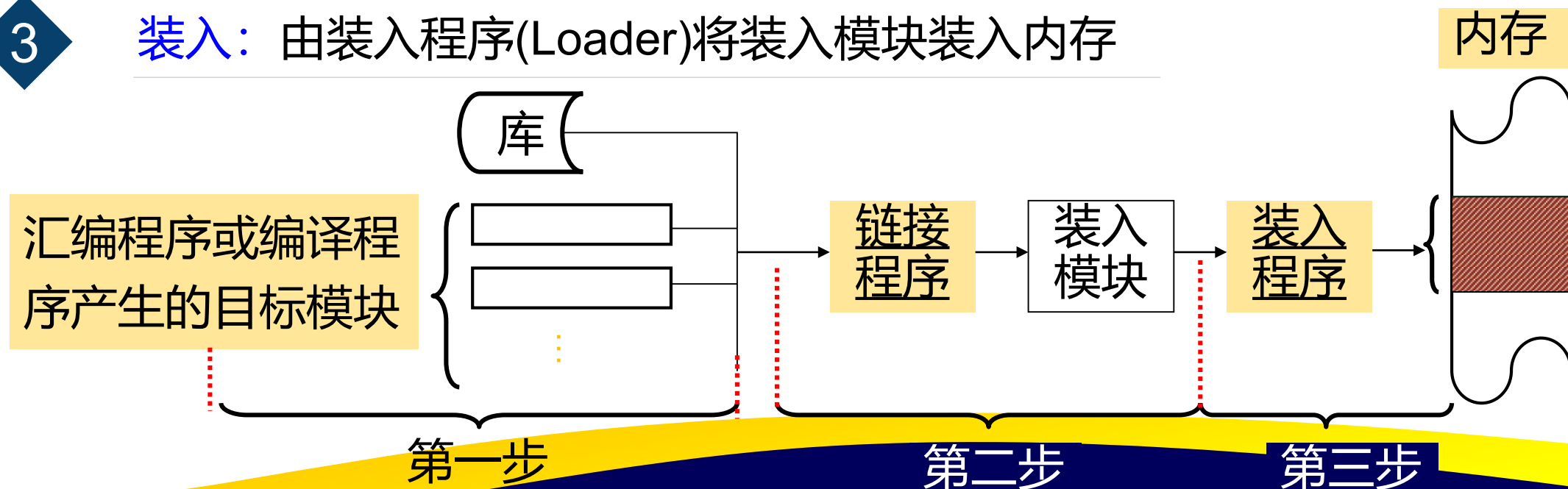
内容导航:

-  5.1 存储器的层次结构
-  **5.2 程序的装入和链接**
-  5.3 对换与覆盖
-  5.4 连续分配存储管理方式
-  5.5 分页存储管理方式
-  5.6 分段存储管理方式
-  5.7 基于IA-32/x86-64架构的内存管理策略

第5章 存储器管理

程序的运行步骤

- 1 **编译**: 由编译程序(Compiler)对源程序进行编译, 形成若干个目标模块
- 2 **链接**: 由链接程序(Linker)将目标模块和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
- 3 **装入**: 由装入程序(Loader)将装入模块装入内存





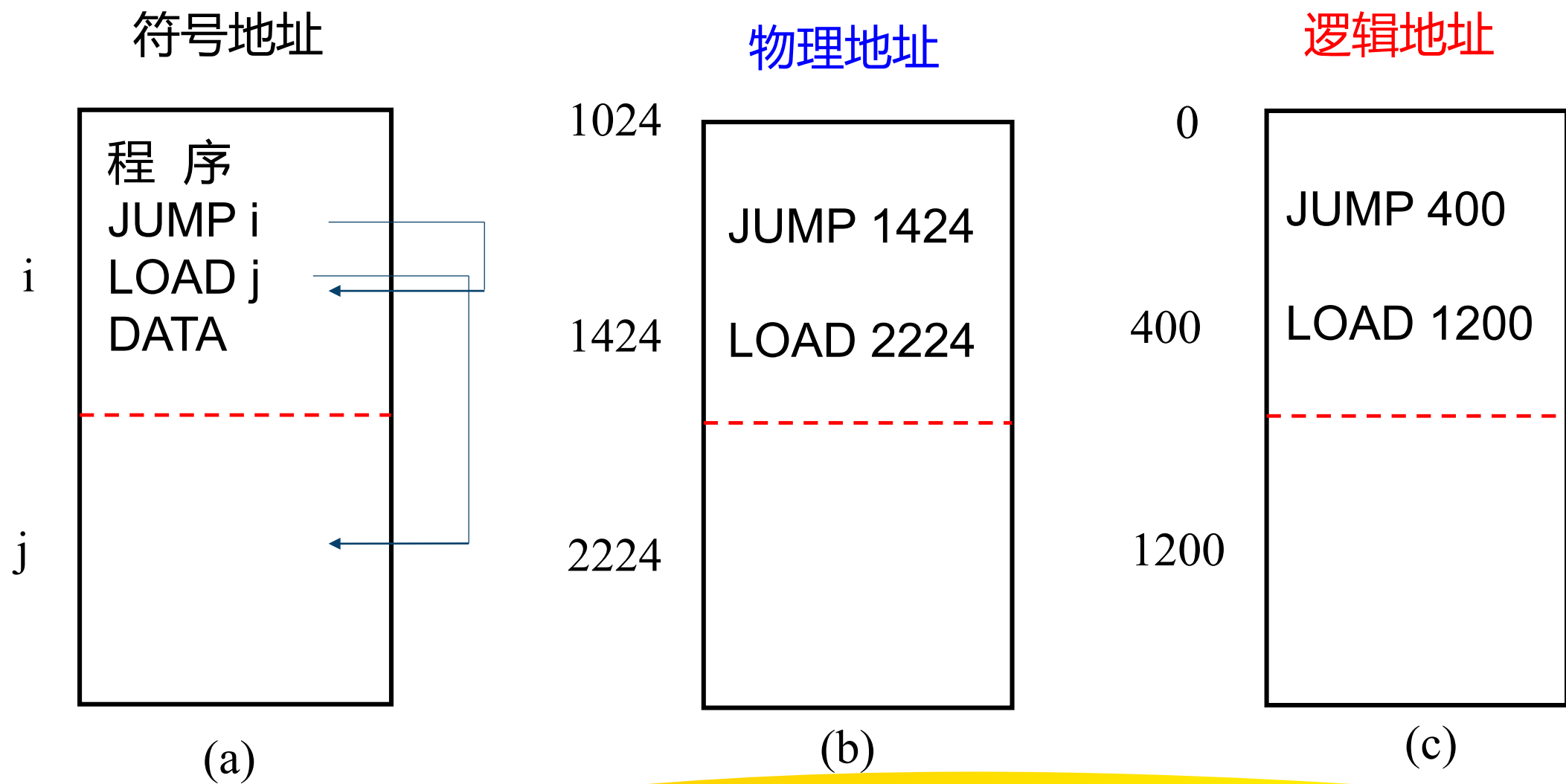
物理地址（绝对地址）

- 物理内存的地址，内存以字节为单位编址
- 物理地址空间：所有物理地址的集合

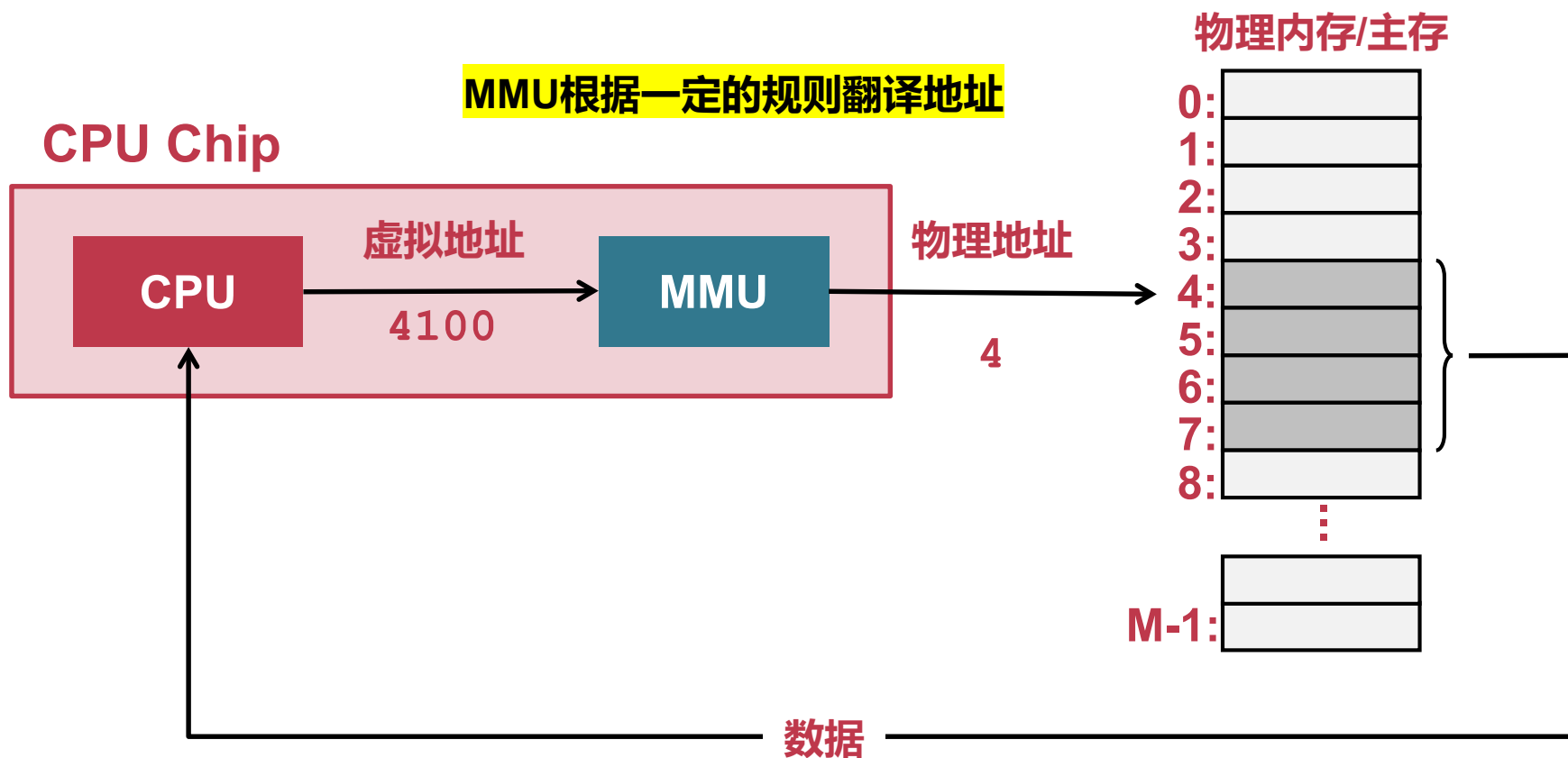


逻辑地址（虚拟地址、相对地址）

- 由CPU产生的地址，即程序编译后使用的相对于0字节的地址
- 逻辑地址空间：由程序所生成的所有逻辑地址的集合



地址翻译过程

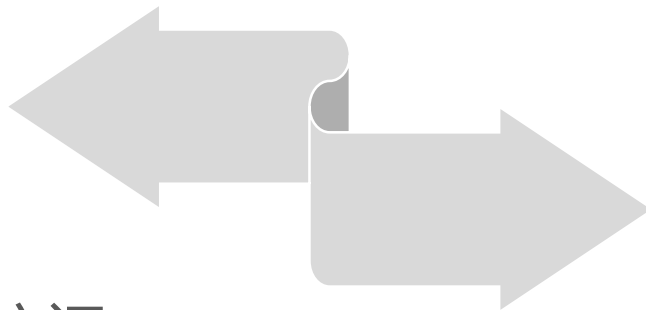


翻译规则取决于虚拟内存采用的组织机制，包括：分段机制和分页机制



内存保护的目

- 保护OS不被用户访问
- 保护用户进程不会相互影响



内存保护的实现：硬件

- 基地址寄存器：保存最小的合法物理内存地址（基地址）
- 界限寄存器：保存合法的地址范围大小（界限地址）
- 内存空间保护的实现
 - 判断 “ $\text{基地址} \leq \text{物理地址} < (\text{基地址} + \text{界限地址})$ ” 是否成立。



绝对装入方式

- 编译时产生的地址使用绝对地址
- 程序或数据被修改时，需要重新编译程序



可重定位装入方式

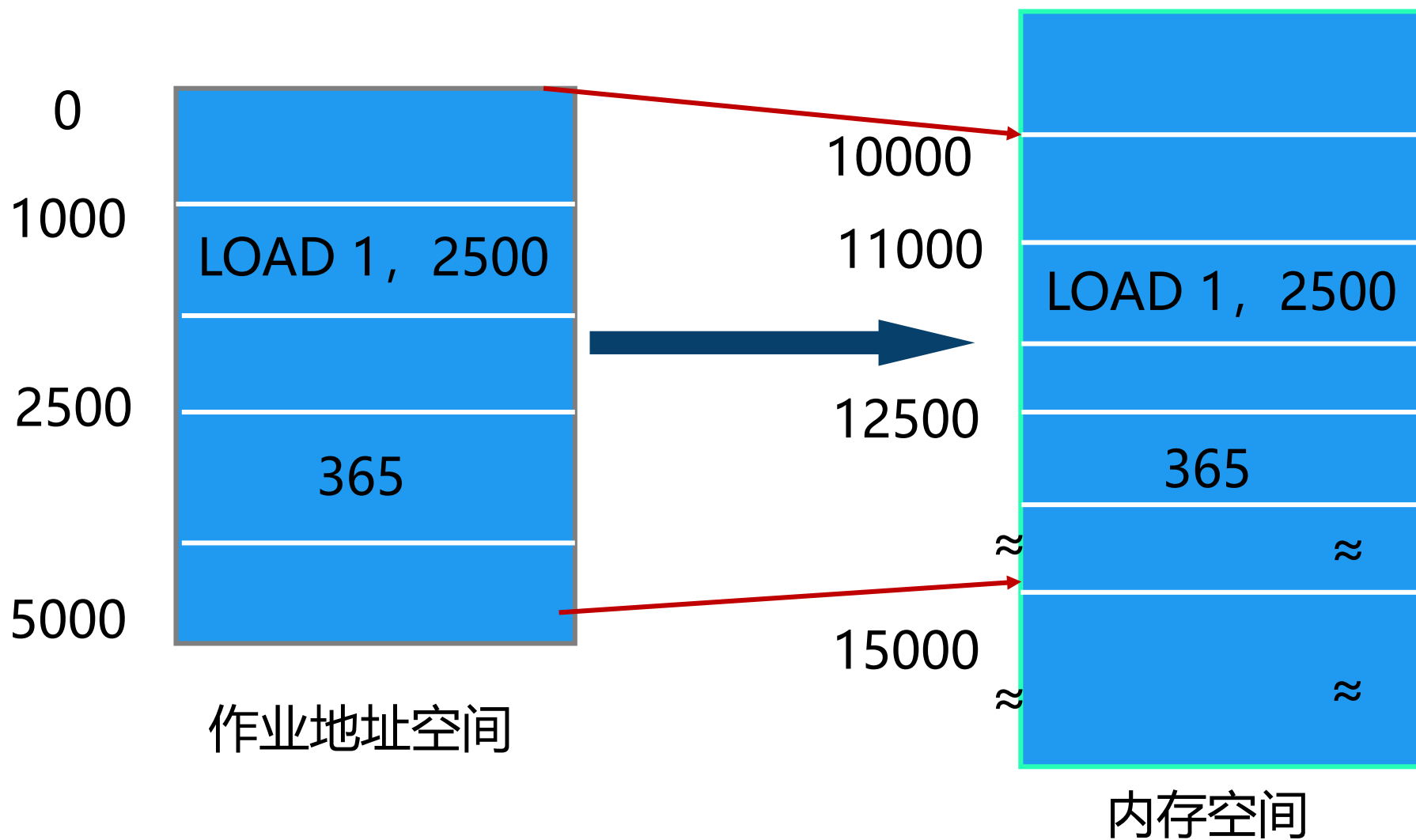
- 编译后的目标模块使用相对地址
- 在装入时，完成**重定位**（静态重定位）
- 需硬件支持

逻辑地址转换为物理地址的过程，称为重定位，也称为地址变换



动态运行时装入方式

- 编译后的目标模块使用相对地址
- 在运行时，程序在内存中的位置不固定，需要完成重定位（动态重定位）
- 地址变换推迟到程序真正要执行时才进行





静态链接

- 在程序运行前，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
- 对相对地址进行修改；变换外部调用符号



装入时动态链接

- 在装入内存时，采用边装入边链接的链接方式
- 便于修改和更新
- 便于实现对目标模块的共享

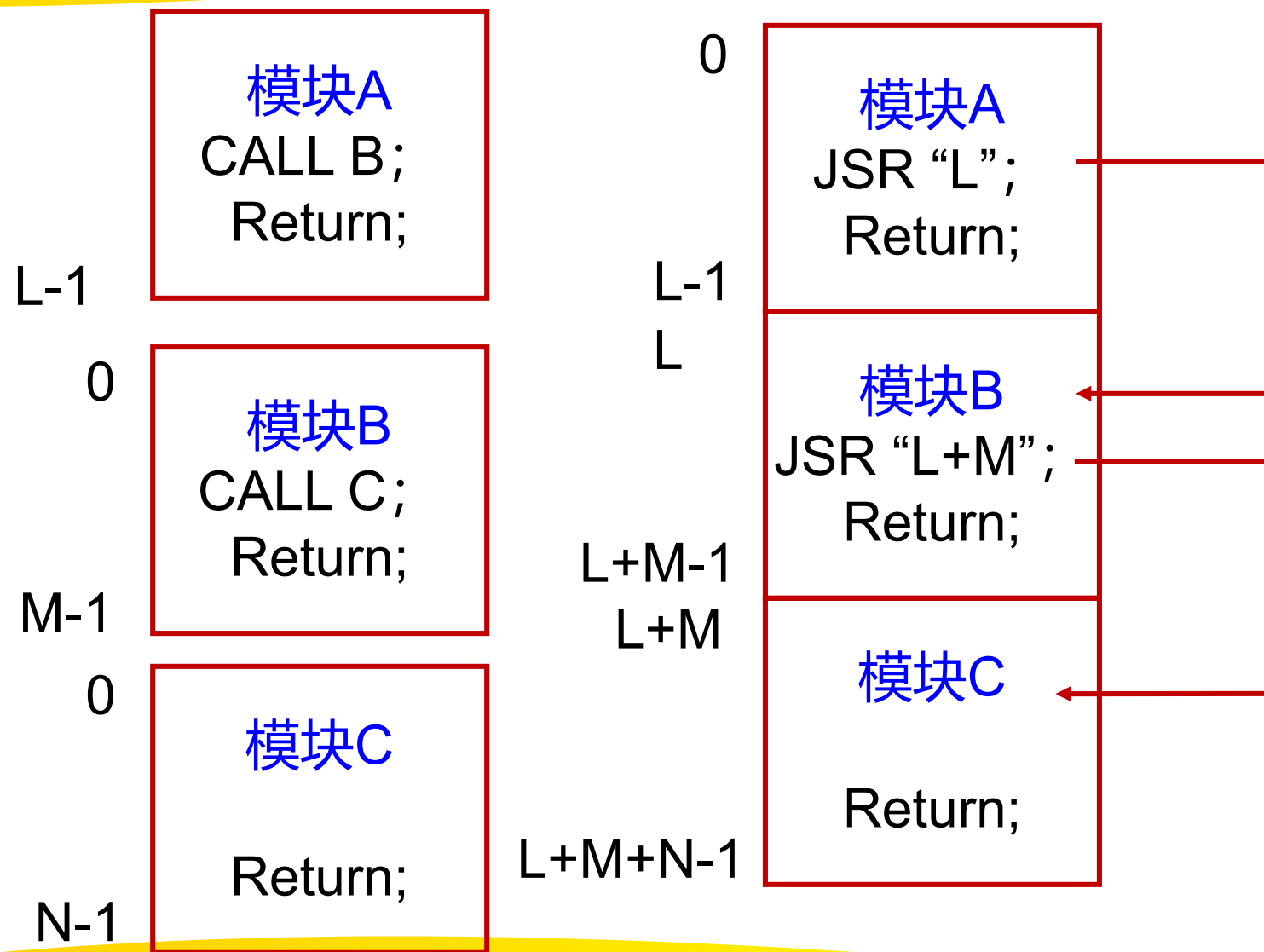


运行时动态链接

- 将某些目标模块的链接推迟到执行时才执行。即在执行过程中，若发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将它装入内存，并把它链接到调用者模块上
- 加快装入过程，节省大量的内存空间




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。





内容导航:

-  5.1 存储器的层次结构
-  5.2 程序的装入和链接
-  **5.3 对换与覆盖**
-  5.4 连续分配存储管理方式
-  5.5 分页存储管理方式
-  5.6 分段存储管理方式
-  5.7 基于IA-32/x86-64架构
的内存管理策略

第5章 存储器管理

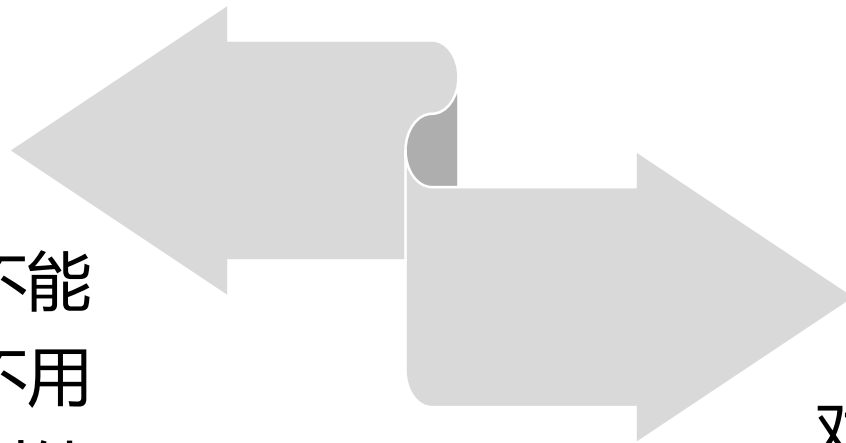
在多道程序环境下：

- **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迫使CPU停止下来等待的情况；
- **另一方面**，却又有着许多作业在外存上等待，因无内存而不能进入内存运行的情况。

浪费资源，降低系统吞吐量。



对换：把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需的程序或数据，调入内存。



对换是提高内存利用率的有效措施，广泛应用于OS中



整体对换：对换以整个进程为单位，也称为**进程对换**

- 被广泛应用于多道程序系统，并作为处理机中级调度



页面(分段)对换：对换是以“页”或“段”为单位进行的，又统称为“部分对换”

- 目的是为了支持虚拟存储系统



为了实现**进程对换**，系统必须能实现三方面的功能

- 对换空间的管理
- 进程的换出
- 进程的换入



对换区管理的主要目标

- 提高进程换入和换出的速度
- 提高文件存储空间的利用率次之
- 应采用连续分配方式，很少考虑碎片问题



盘块管理中的数据结构

- 用于记录外存对换区中的空闲盘块的使用情况
- 与动态分区分配方式相似
- 空闲分区表/空闲分区链：包含对换区首址及大小



对换区的分配与回收

- 与动态分区方式的内存分配与回收方法相似

1. 进程的换出

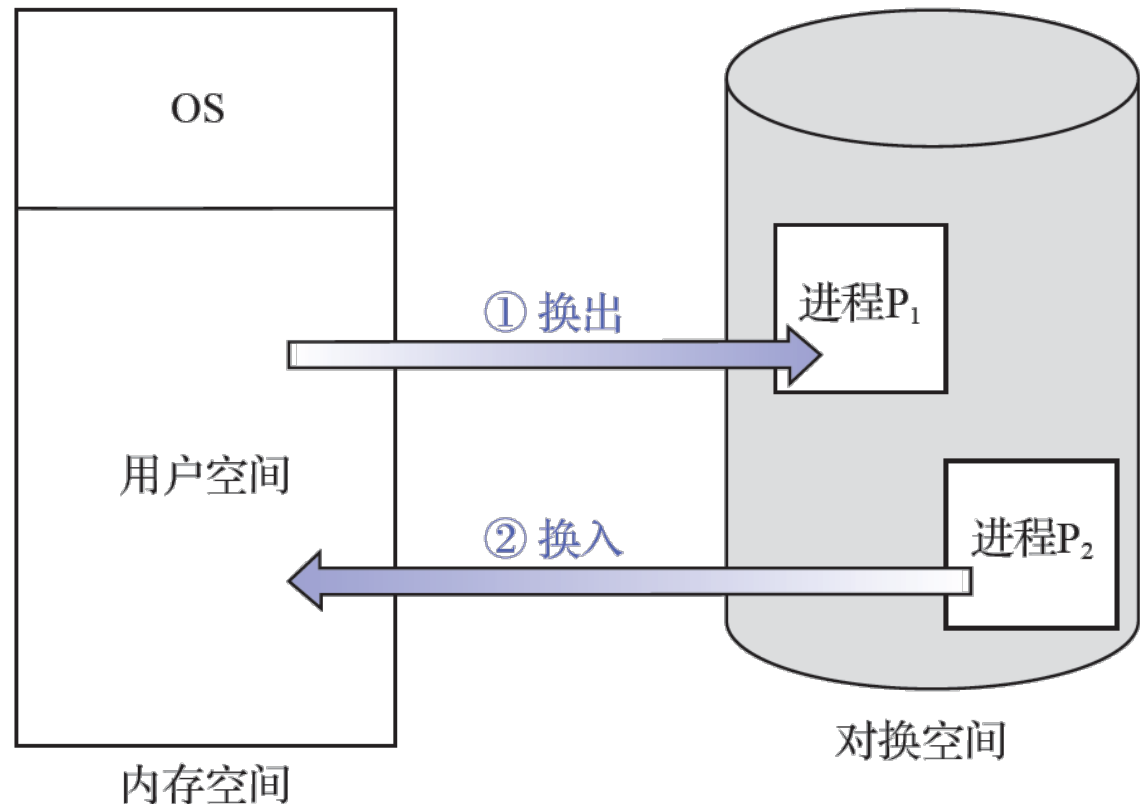
对换进程在实现进程换出时，是将内存中的某些进程调出至对换区，以便腾出内存空间。换出过程可分为以下两步：

(1) 选择被换出的进程。

- ✓ 阻塞状态或睡眠状态的进程
- ✓ 低优先级的进程

(2) 进程换出过程。

- ✓ 只能换出非共享的程序和数据段
- ✓ 先申请交换区→启动磁盘→直到无阻塞进程



2. 进程的换入

对换进程将定时执行换入操作，它首先查看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2 s)的进程作为换入进程，为它申请内存。如果申请成功，可直接将进程从外存调入内存；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。

1

解决问题→程序大小超过物理内存总和

2

程序执行时：

- 只在内存中保留那些在任何时间都需要的指令和数据；
- 程序的不同部分在内存中相互替换。

3

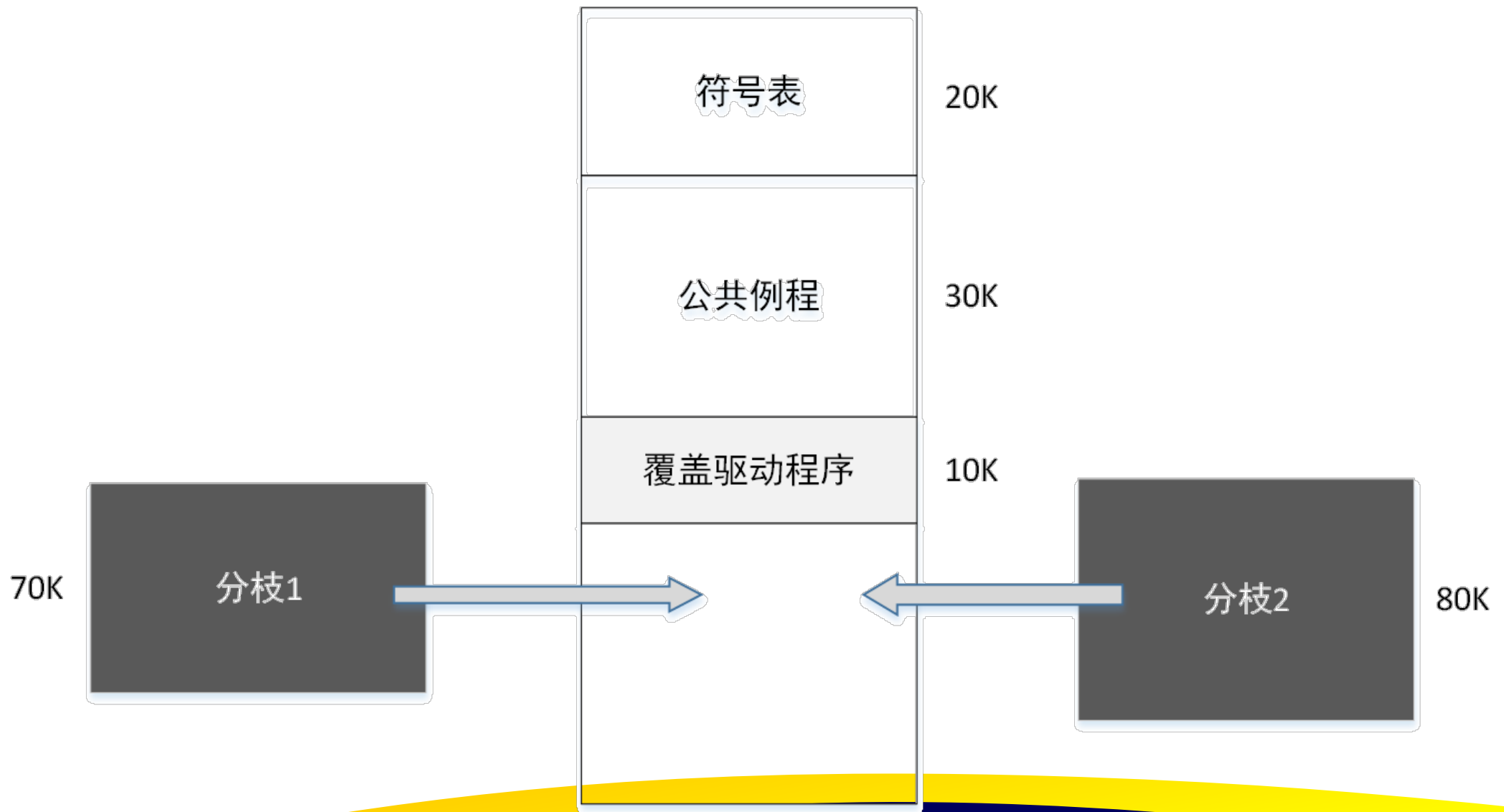
由程序员声明覆盖结构，不需要操作系统的特别支持

4

覆盖结构的程序设计很复杂








5

应用于早期的操作系统





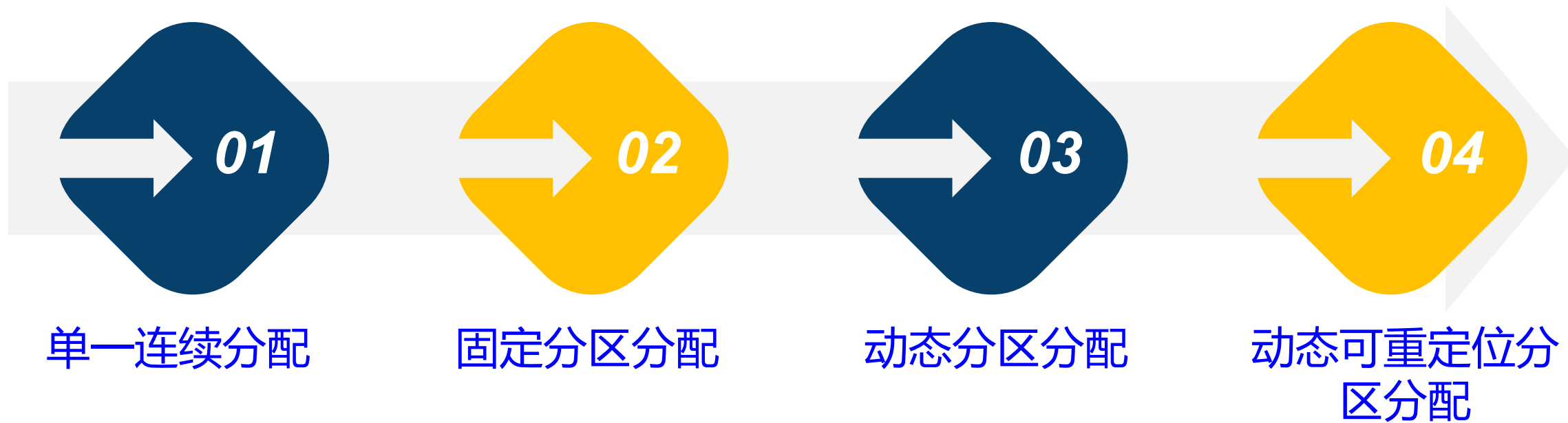
内容导航:

-  5.1 存储器的层次结构
-  5.2 程序的装入和链接
-  5.3 对换与覆盖
-  **5.4 连续分配存储管理方式**
-  5.5 分页存储管理方式
-  5.6 分段存储管理方式
-  5.7 基于IA-32/x86-64架构
的内存管理策略

第5章 存储器管理

连续分配方式： 为一个用户程序分配一个连续的内存空间。

分类：





内存

- 系统区：供OS使用、低址部分
- 用户区：供用户使用



分配方式：单道程序环境下，仅装有一道用户程序，即整个内存的用户空间由该程序独占。

- 内存分配管理十分简单，内存利用率低
- 用于单用户、单任务OS
- CP/M、MS-DOS、RT11



未采取存储器保护措施

- 节省硬件
- 方案可行



最早的、也是最简单的一种可运行多道程序的存储管理方式。

预先把可分配的主存储器空间分割成若干个连续区域，称为一个分区。

每个分区的大小可以相同也可以不同。但分区大小固定不变，每个分区装一个且只能装一个作业。

内存分配：如果有一个空闲区，则分配给进程。



分区大小一样

- 缺乏灵活性
 - 程序太小：浪费内存
 - 程序太大：装不下
- 有些场合适用，如利用一台计算机同时控制多个相同对象

分区大小不等

- 多个小分区
- 适量中分区
- 少量大分区

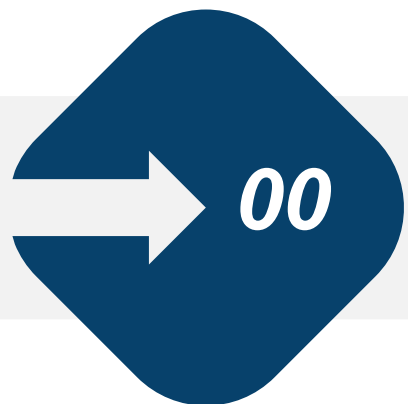


分区号	大小(K)	起始地址(K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	未分配

固定分区使用表

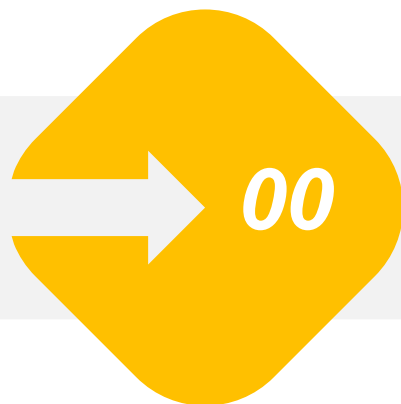
	操作系统
20K	作业A
32K	
	作业B
64K	
	作业C
128K	
256K	

动态分区分配:



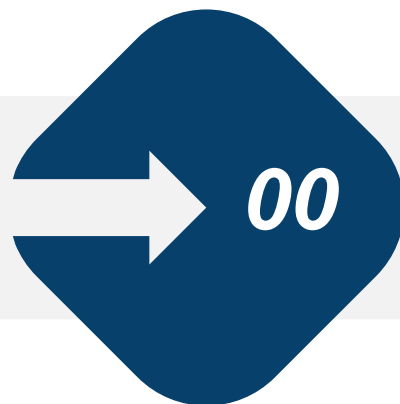
- 又称为**可变分区分配**，根据进程的实际需要，动态地为之分配内存空间。

数据结构



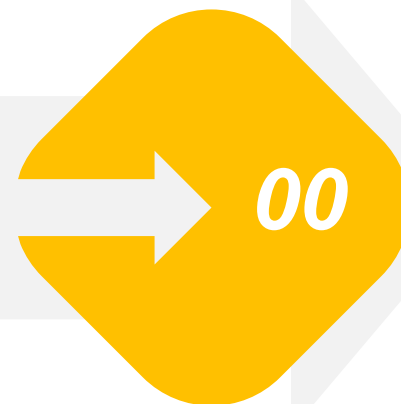
- 空闲分区表
- 空闲分区链

分配算法



- 顺序式分配算法
- 索引式分配算法

分配操作



- 分配内存
- 回收内存

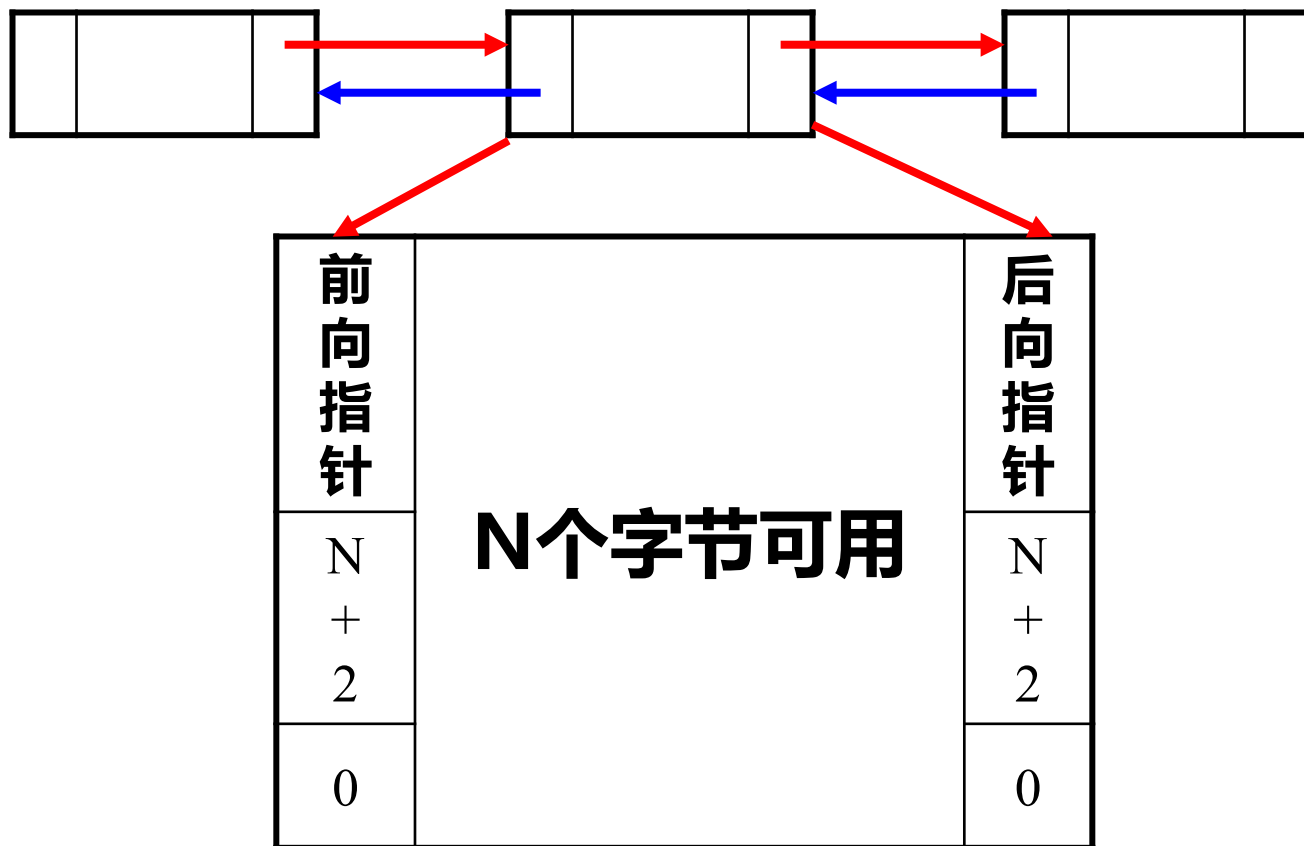
空闲分区表

- 一个空闲分区占一个表目
- 分区序号、分区始址、分区大小、状态等

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

空闲分区链

- 为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些用于控制分区分配的信息，以及用于链接各分区所用的前向指针；在分区的尾部设置后向指针，通过前、后向指针，可将所有的空闲分区链接成一个双向链。





基于顺序搜索的动态分区分配算法

- 依次搜索空闲分区链上的空闲分区，寻找一个其大小能够满足要求的分区
- 首次适应算法、循环首次适应算法、最佳适应算法、最坏适应算法



基于索引搜索的动态分区分配算法

- 提高搜索空闲分区的速度，在大、中型系统中采用
- 快速适应算法、伙伴系统和哈希算法



首次适应算法 (first fit, FF)

- 空闲分区链以地址递增的次序链接
- 从链首开始顺序查找，直到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。
- 缺点：低址部分留下许多小碎片



循环首次适应算法(next fit, NF)

- 从上次找到的空闲分区的下一个空闲分区开始查找，直到找到一个能满足要求的空闲分区
- 空闲分区分布更均匀，减少了查找的开销
- 缺乏大的空闲分区



最佳适应算法(best fit, BF)

- 搜索整个序列，找到适合条件的最小的分区进行分配
- 空闲分区按其容量从小到大的顺序链接
- 用最小空间满足要求；但留下许多难以利用的小碎片



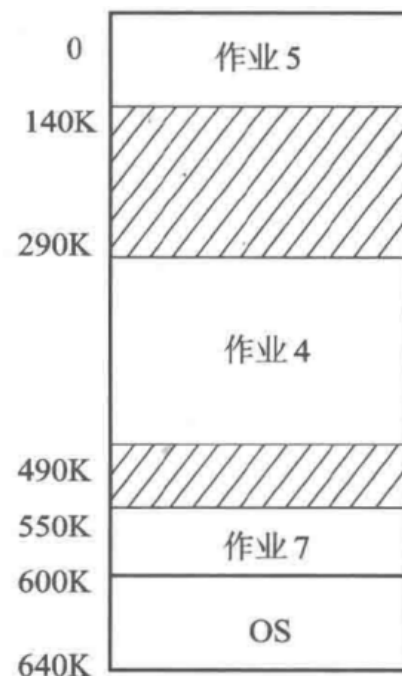
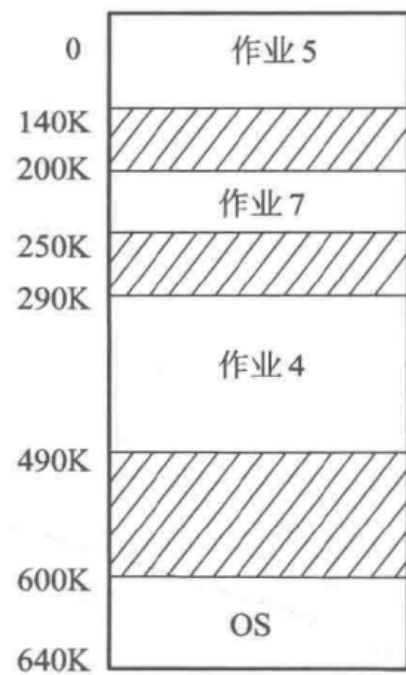
最坏适配算法(worst fit, WF)

- 搜索整个序列，寻找最大的分区进行分配
- 空闲分区按其容量从大到小的顺序链接
- 分割后空闲块仍为较大空块；缺乏大的空闲分区

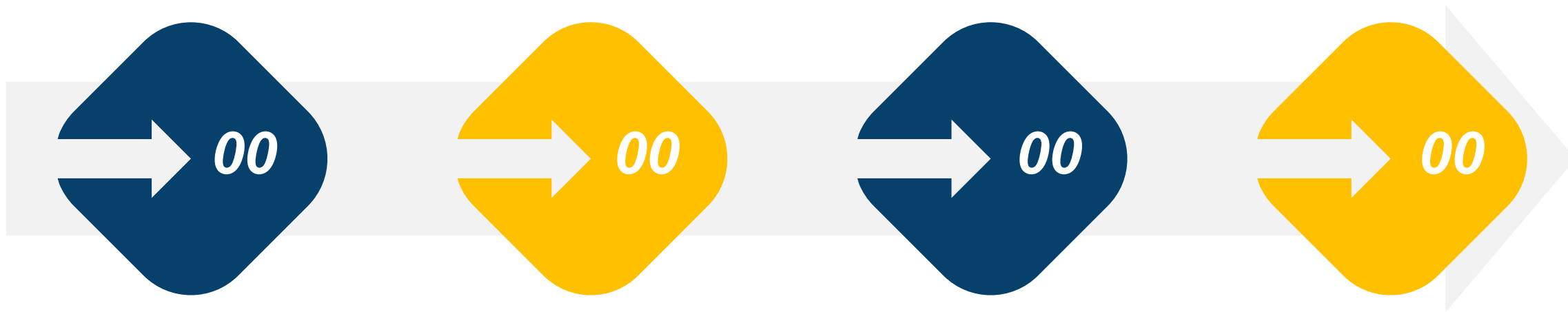
某系统采用动态分区分配方式管理内存，内存空间为640KB，高端40KB用来存放OS。在分配内存时，系统优先使用空闲区低端的空间。对下列的请求序列：作业1申请130KB、作业2申请60KB、作业3申请100KB、作业2释放60KB、作业4申请200KB、作业3释放100KB、作业1释放130KB、作业5申请140KB、作业6申请60KB、作业7申请50KB、作业6释放60KB，请分别画图表示出使用首次适应算法和最佳适应算法进行内存分配和回收后，内存的实际使用情况。

作业1申请130KB、作业2申请60KB、作业3申请100KB、作业2释放60KB、作业4申请200KB、作业3释放100KB、作业1释放130KB、作业5申请140KB、作业6申请60KB、作业7申请50KB、作业6释放60KB，

动 作	首次适应算法		最佳适应算法	
	已分配分区 (作业, 始址, 大小)	空闲分区 (始址, 大小)	已分配分区 (作业, 始址, 大小)	空闲分区 (始址, 大小)
作业 1 申请 130K	1, 0, 130	130, 470	1, 0, 130	130, 470
作业 2 申请 60K	1, 0, 130 2, 130, 60	190, 410	1, 0, 130 2, 130, 60	190, 410
作业 3 申请 100K	1, 0, 130 2, 130, 60 3, 190, 100	290, 310	1, 0, 130 2, 130, 60 3, 190, 100	290, 310
作业 2 释放 60K	1, 0, 130 3, 190, 100	130, 60 290, 310	1, 0, 130 3, 190, 100	130, 60 290, 310
作业 4 申请 200K	1, 0, 130 3, 190, 100 4, 290, 200	130, 60 490, 110	1, 0, 130 3, 190, 100 4, 290, 200	130, 60 490, 110
作业 3 释放 100K	1, 0, 130 4, 290, 200	130, 160 490, 110	1, 0, 130 4, 290, 200	490, 110 130, 160
作业 1 释放 130K	4, 290, 200	0, 290 490, 110	4, 290, 200	490, 110 0, 290
作业 5 申请 140K	4, 290, 200 5, 0, 140	140, 150 490, 110	4, 290, 200 5, 0, 140	490, 110 140, 150
作业 6 申请 60K	4, 290, 200 5, 0, 140 6, 140, 60	200, 90 490, 110	4, 290, 200 5, 0, 140 6, 490, 60	550, 50 140, 150
作业 7 申请 50K	4, 290, 200 5, 0, 140 6, 140, 60 7, 200, 50	250, 40 490, 110	4, 290, 200 5, 0, 140 6, 490, 60 7, 550, 50	140, 150
作业 6 释放 60K	4, 290, 200 5, 0, 140 7, 200, 50	140, 60 250, 40 490, 110	4, 290, 200 5, 0, 140 7, 550, 50	490, 60 140, 150



快速适应算法



将空闲分区按其容量大小进行分类，具有相同容量的所有空闲分区设有一个空闲分区链表

系统设有一张管理索引表，每一项对应一个空闲分区类型

分配时，根据进程长度，从索引表中寻找到能容纳它的最小空闲分区链表；从链表中取下第一块进行分配

特点

- **优点：**不分割分区，不产生碎片，查找效率高
- **缺点：**分区归还主存时算法复杂，系统开销较大，存在浪费

伙伴系统

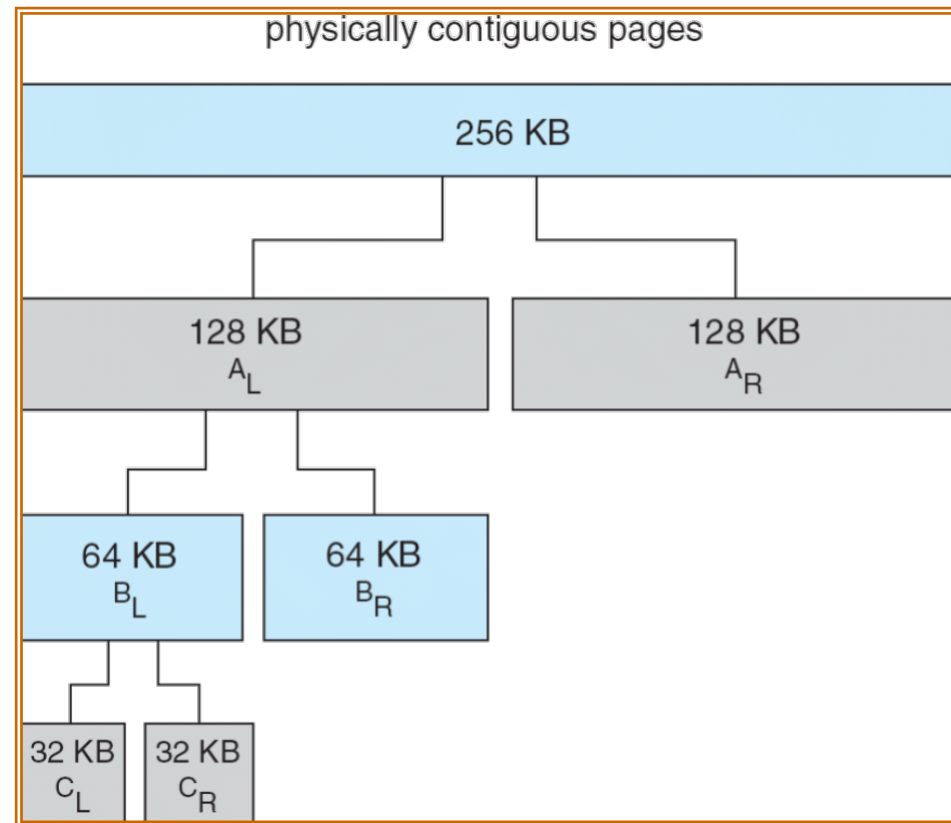


分区大小均为2的k次幂



内存按2的幂的大小来分配，即4KB、8KB等

- 满足要求是以2的幂为单位的
- 如果请求不为2的幂，则需要调整到下一个更大的2的幂：先计算一个i值，使 $2^{i-1} < n \leq 2^i$ ，在2ⁱ的空闲区表中找
- 当分配需求小于现在可用内存时，当前段就分为两个更小的2的幂段，这两个分区称为“一对伙伴”，其中一个分区用于分配，另一个加入2ⁱ的空闲分区链表中
 - 继续上述操作直到合适的段大小

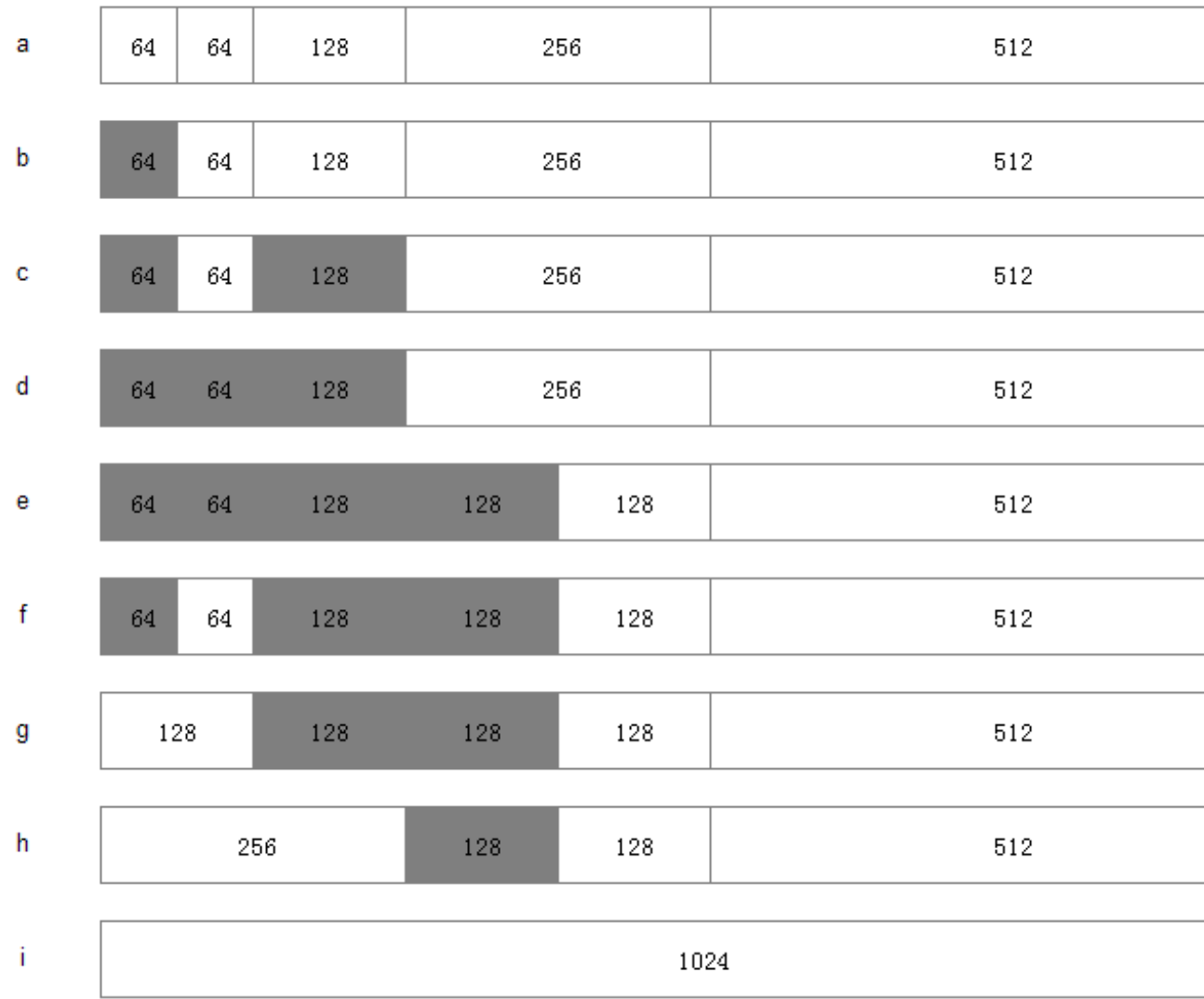




伙伴系统的应用案例-1

假设系统中有 1MB 大小的内存需要动态管理，按照伙伴算法的要求：需要将这 1M大小的内存进行划分。这里，我们将这1M的内存分为 64K、64K、128K、256K、和512K 共五个部分

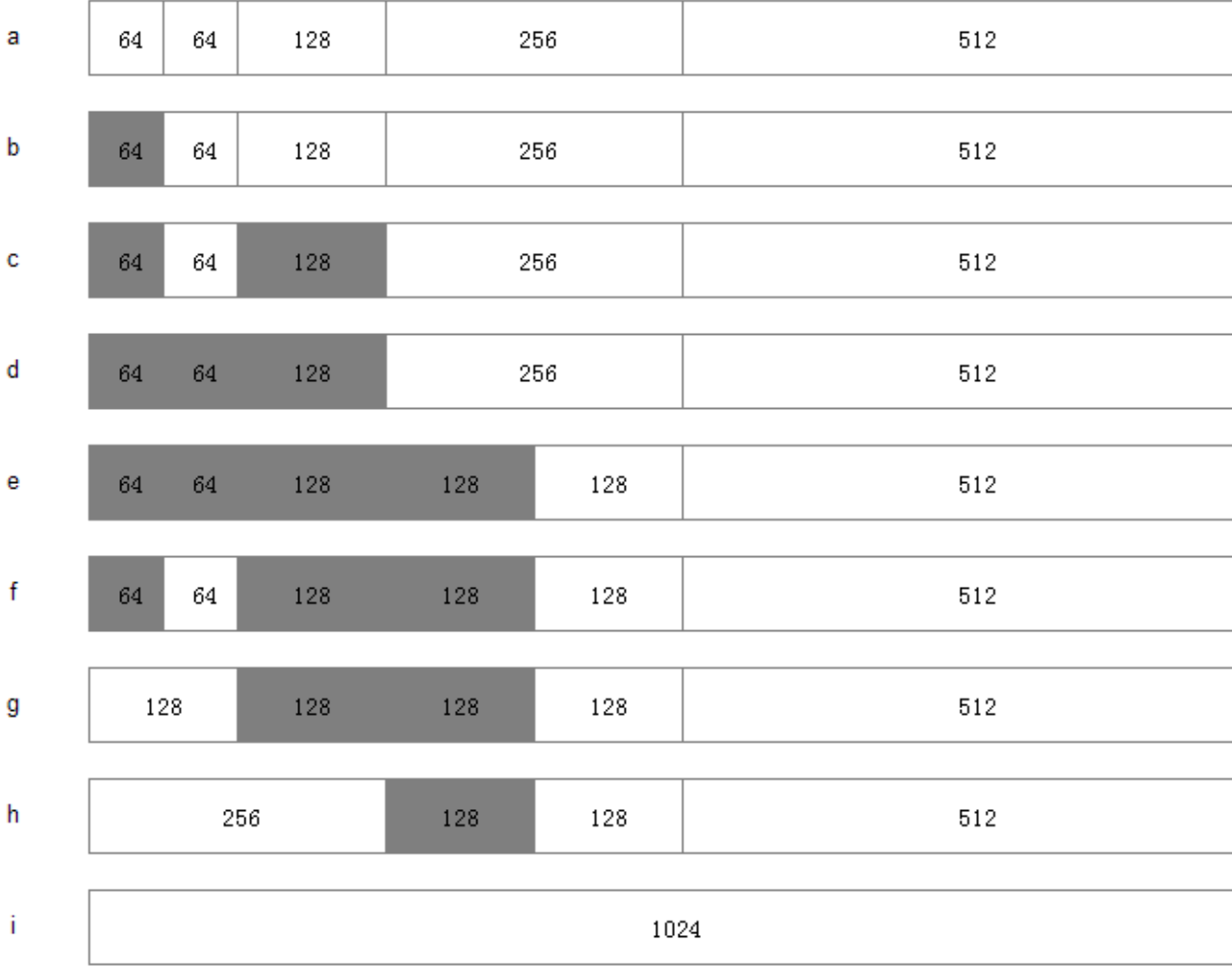
- 1.此时，如果有一个程序A想要申请一块 45K大小的内存，则系统会将第一块64K的内存块分配给该程序（产生内部碎片为代价），如图b所示；
- 2.然后程序B向系统申请一块68K大小的内存，系统会将128K内存分配给该程序，如图c所示；
- 3.接下来，程序C要申请一块大小为35K的内存。系统将空闲的64K内存分配给该程序，如图d所示；





4.之后程序D需要一块大小为90K的内存。当程序提出申请时，系统本该分配给程序D一块128K大小的内存，但此时内存中已经没有空闲的128K内存块了，于是根据伙伴算法的原理，系统会将256K大小的内存块平分，将其中一块分配给程序D，另一块作为空闲内存块保留，等待以后使用，如图e所示；

5.紧接着，程序C释放了它申请的64K内存。在内存释放的同时，系统还负责检查与之相邻并且同样大小的内存是否也空闲，由于此时程序A并没有释放它的内存，所以系统只会将程序C的64K内存回收，如图f所示；

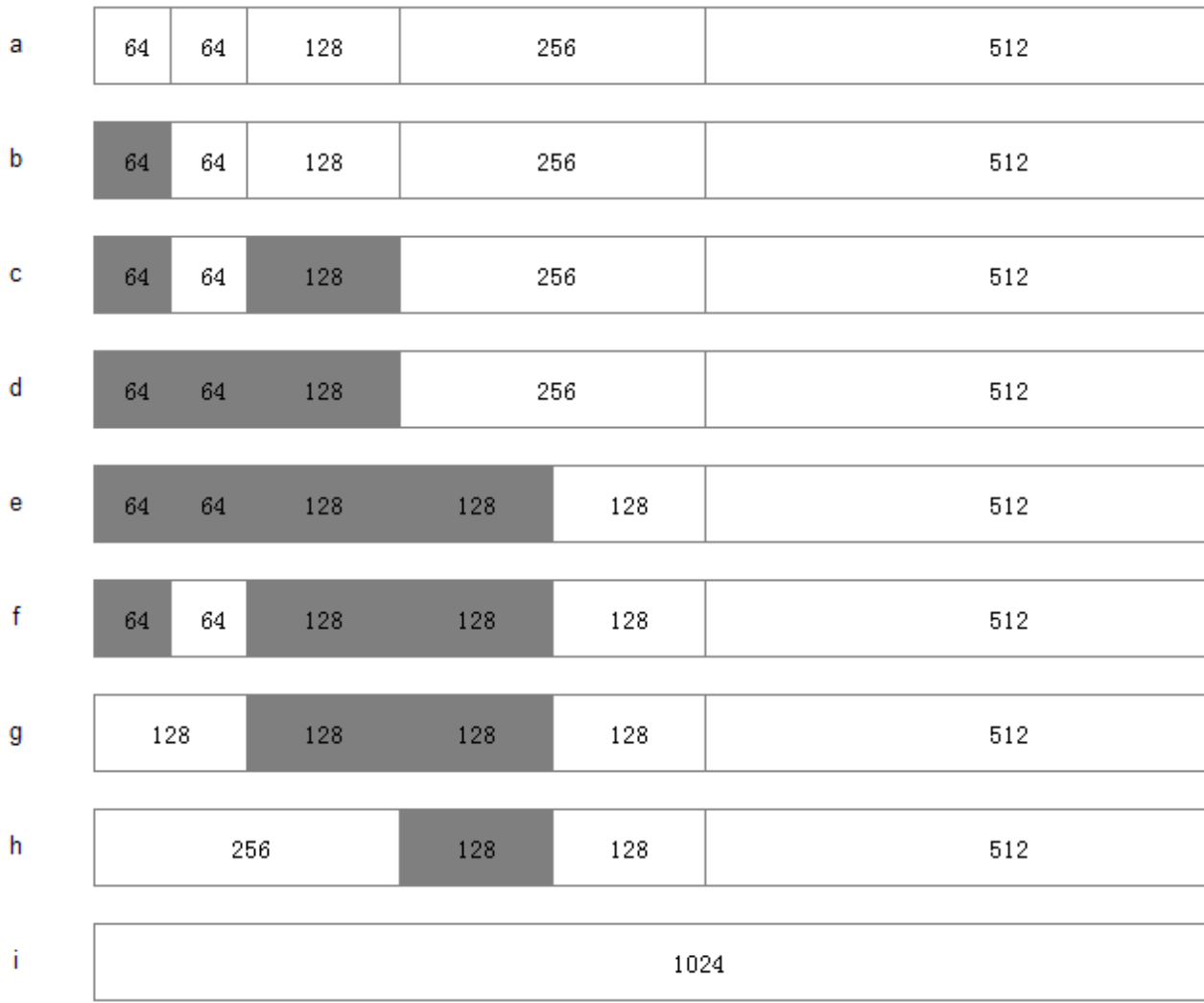




6.然后程序A也释放掉由它申请的64K内存，系统随机发现与之相邻且大小相同的一段内存块恰好也处于空闲状态。于是，将两者合并成128K内存，如图g所示；

7.之后程序B释放掉它的128k，系统也将这块内存与相邻的128K内存合并成256K的空闲内存，如图h所示；

8.最后程序D也释放掉它的内存，经过三次合并后，系统得到了一块1024K的完整内存，如图i所示。



哈希算法

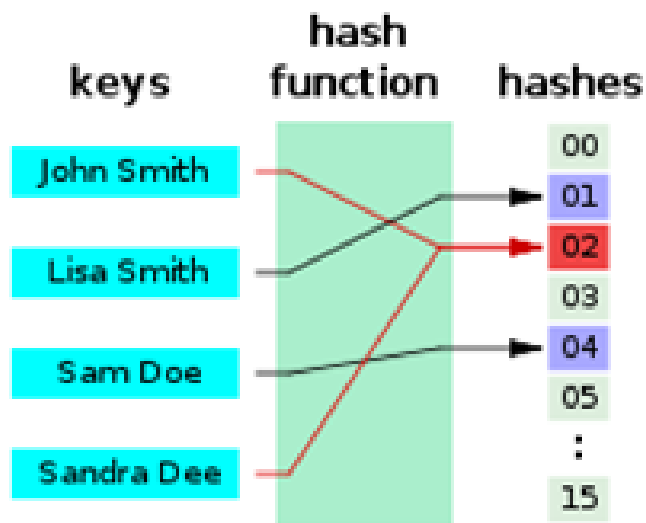
建立哈希函数，构造一张以空闲分区大小为关键字的哈希表，该表的每一个表项对应于一个空闲分区链表的头指针。

进行分配时，根据空闲区大小，通过计算哈希函数，得到在哈希表中的位置，找到对应的空闲分区链表。

优点：查找快速！

哈希函数 (Hash Function)，也称为散列函数，给定一个输入 x ，它会算出相应的输出 $H(x)$ 。哈希函数的主要特征是：

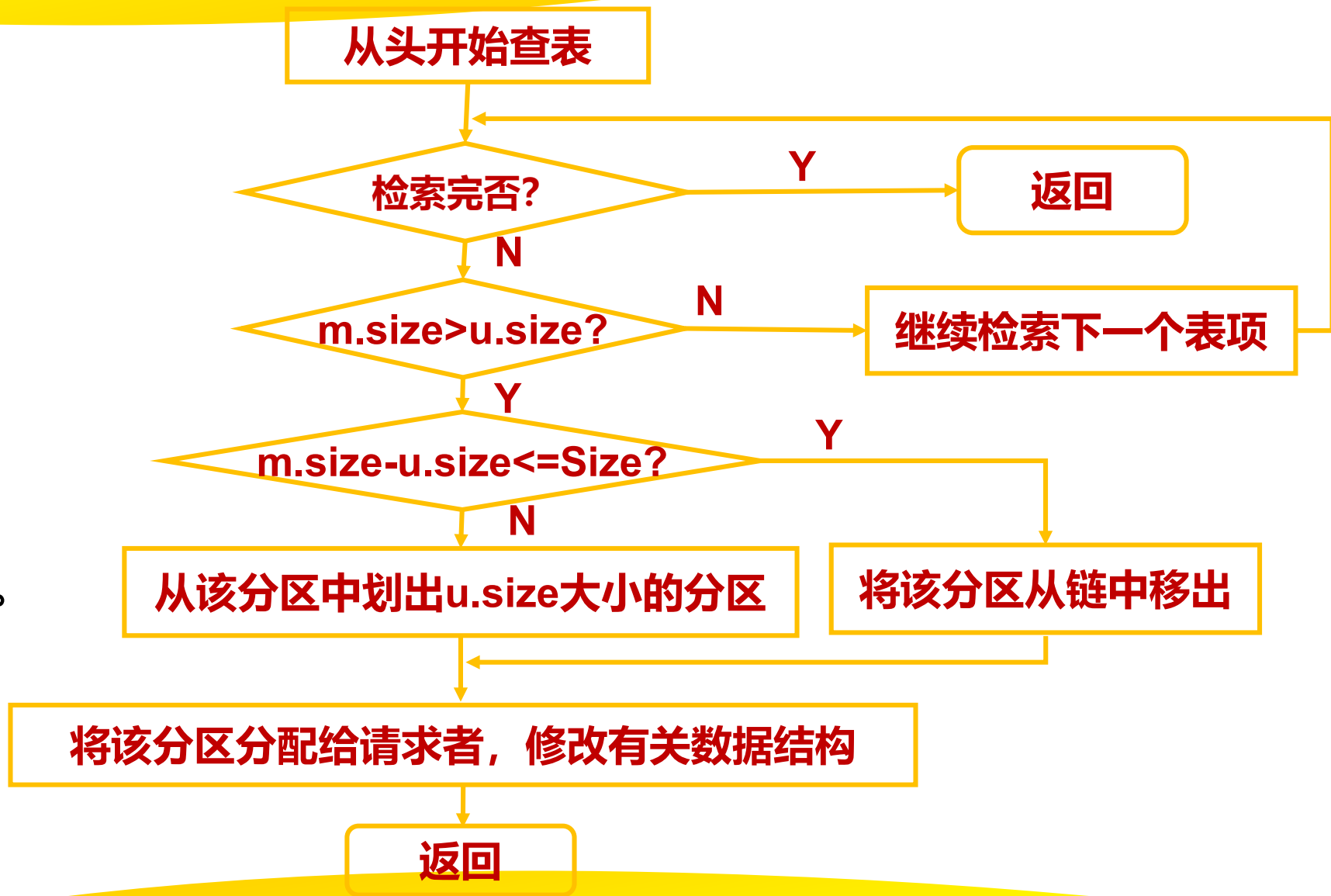
- 输入 x 可以是任意长度的字符串
- 输出结果即 $H(x)$ 的长度是固定的
- 计算 $H(x)$ 的过程是高效的 (对于长度为 n 的字符串 x ，计算出 $H(x)$ 的时间复杂度应为 $O(n)$)



1) 分配内存

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 $m.size$ 。
- ✓ $Size$ 是事先规定的不再分割的剩余分区的大小

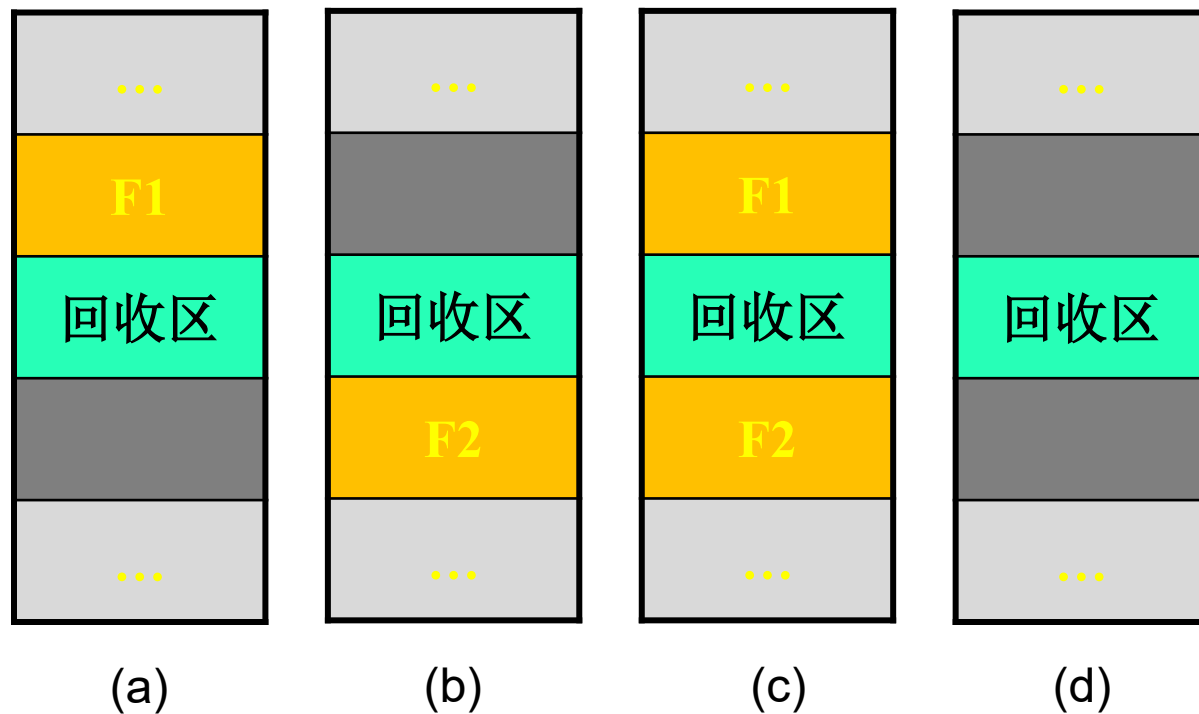


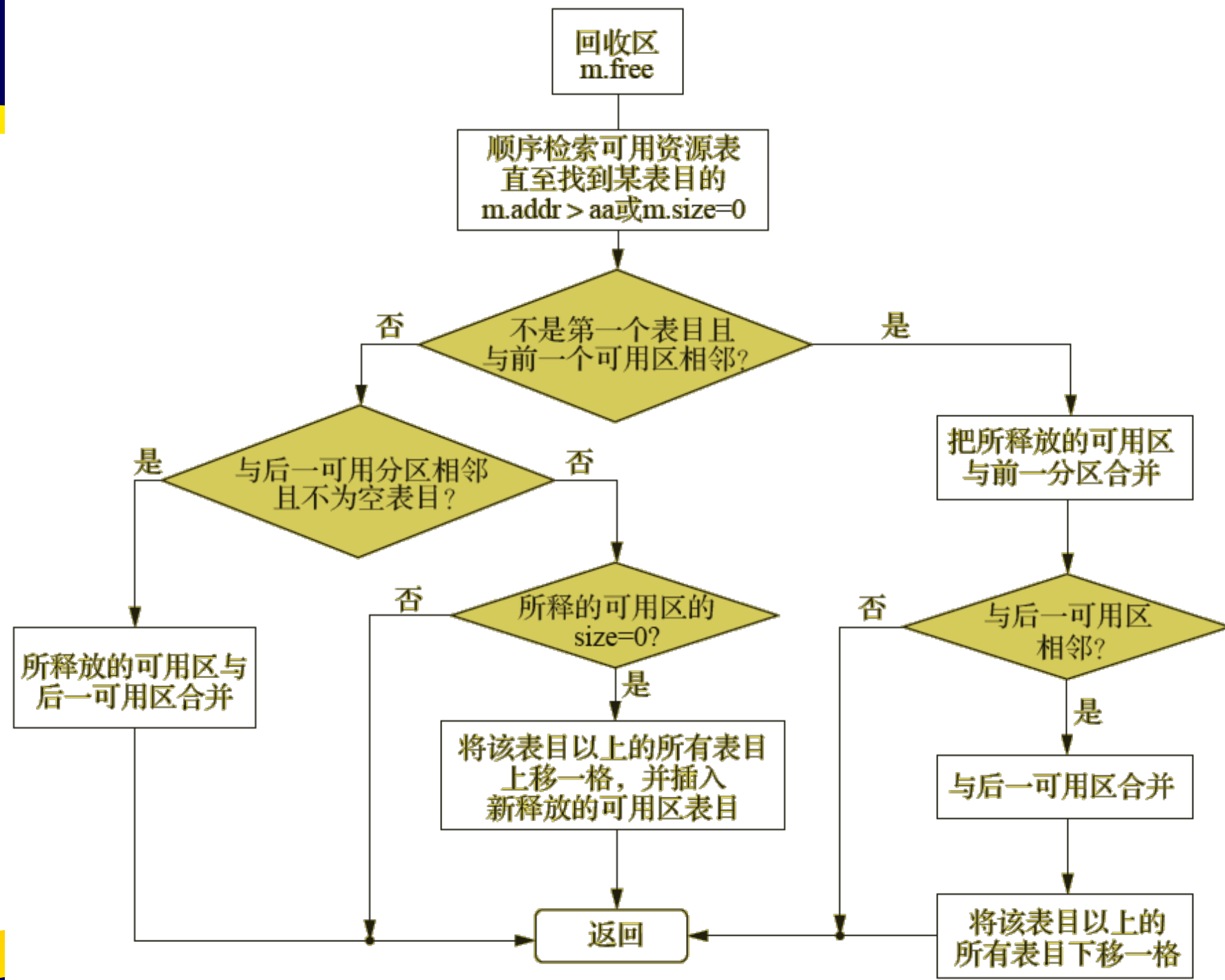
(1) 回收区与插入点的前一个空闲分区 F_1 相邻接, 见(a)。此时应将回收区与插入点的前一分区合并, 不必为回收分区分配新表项, 而只需修改其前一分区 F_1 的大小。

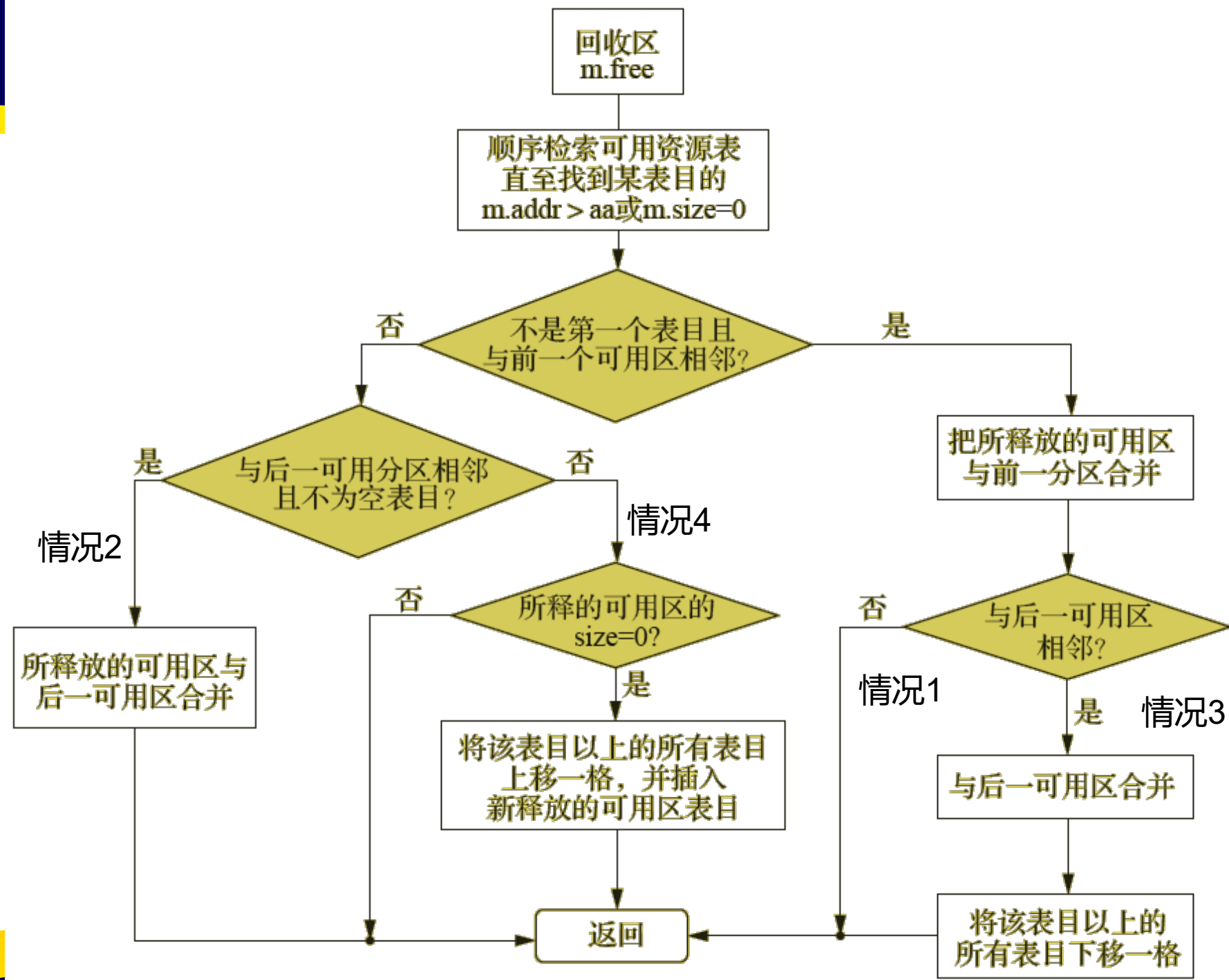
(2) 回收分区与插入点的后一空闲分区 F_2 相邻接, 见(b)。此时也可将两分区合并, 形成新的空闲分区, 但用回收区的首址作为新空闲区的首址, 大小为两者之和。

(3) 回收区同时与插入点的前、后两个分区邻接, 见图(c)。此时将三个分区合并, 使用 F_1 的表项和 F_1 的首址, 取消 F_2 的表项, 大小为三者之和。

(4) 回收区既不与 F_1 邻接, 又不与 F_2 邻接, 见图(d)。这时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。









连续分配方式存在的问题

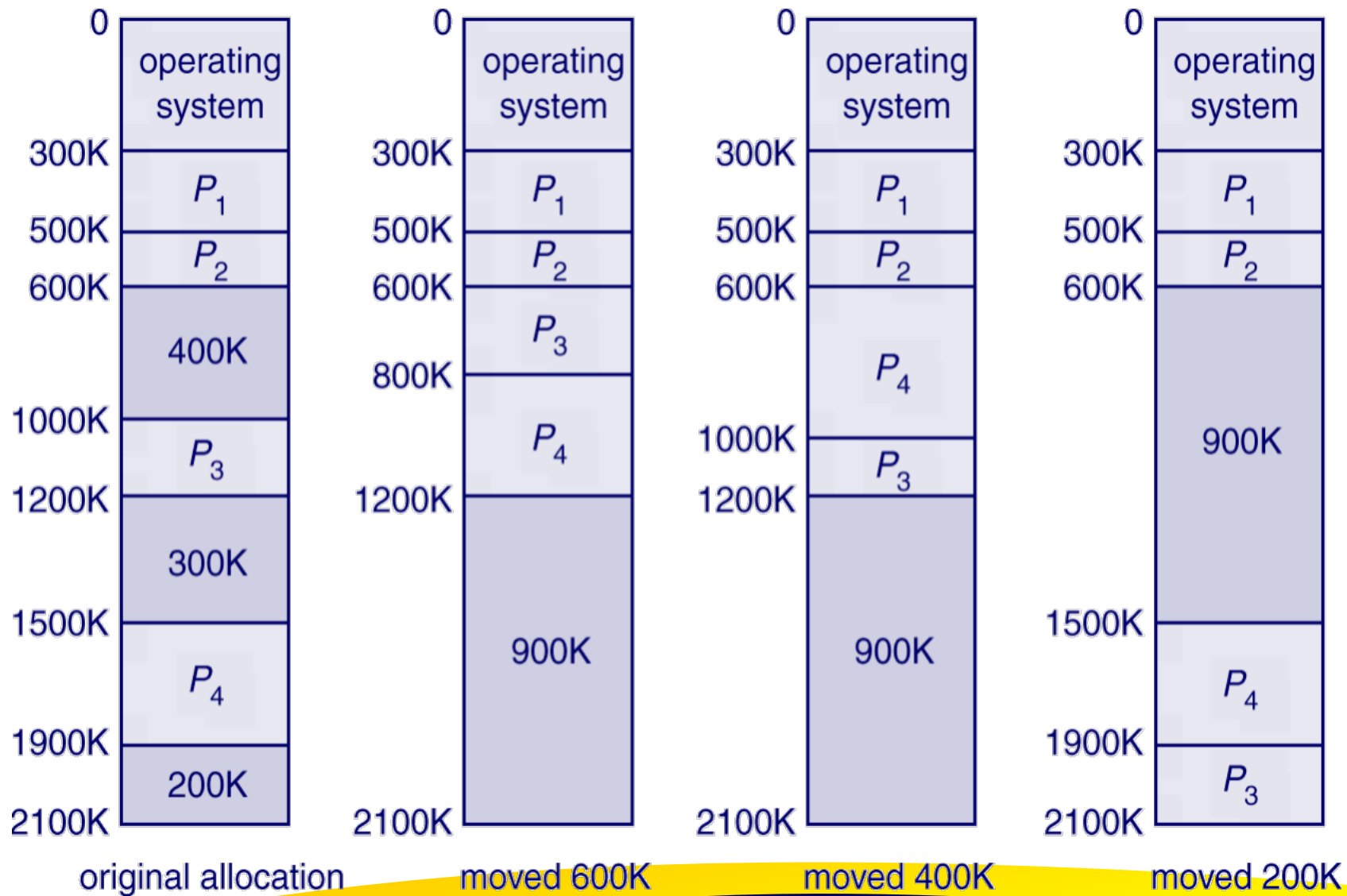
- **碎片**：不能被利用的小分区
- 解决方案：紧凑，要求代码和数据可以在内存中移动
- **紧凑**：通过移动内存中的作业位置，以把原来多个分散的小分区拼接成一个大分区的方法，也叫“拼接”

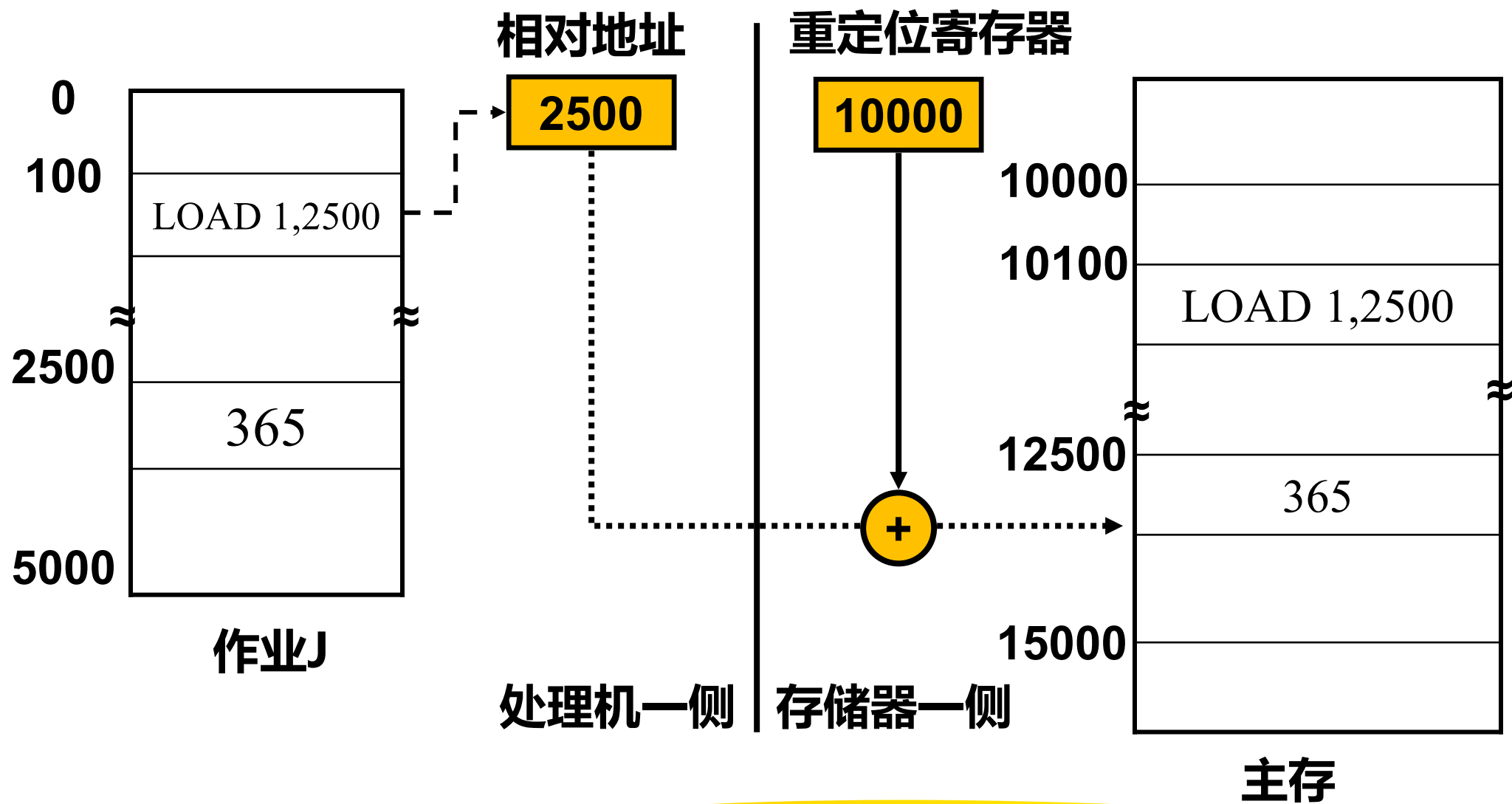


动态重定位：在指令运行时，实现地址转换(相对地址转换为绝对地址)



分配算法：类似于动态分区分配算法，增加了紧凑的功能





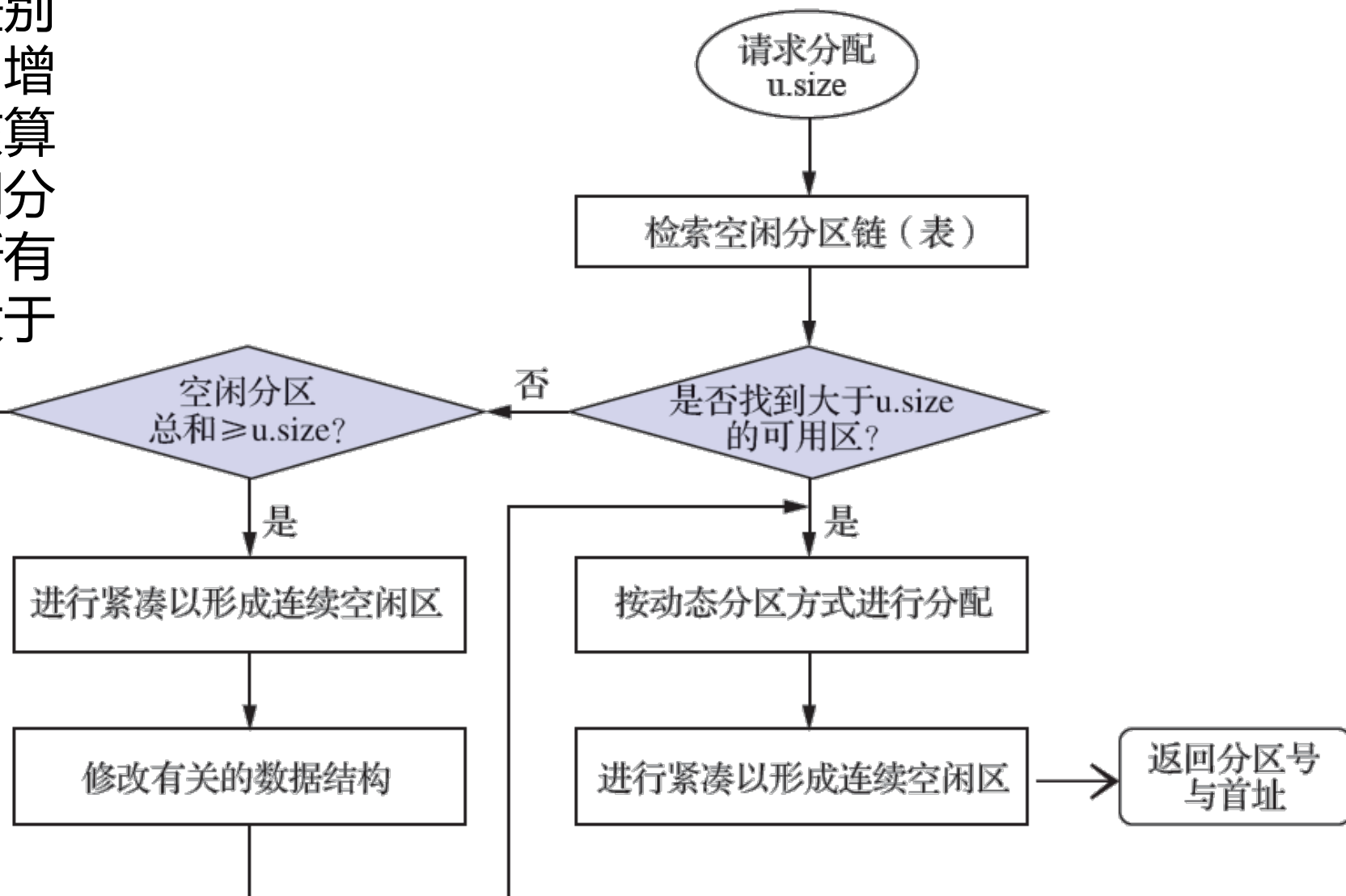


动态分区分配算法流程

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。通常，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，


无法分配
返回

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。





内容导航:

-  5.1 存储器的层次结构
-  5.2 程序的装入和链接
-  5.3 对换与覆盖
-  5.4 连续分配存储管理方式
-  **5.5 分页存储管理方式**
-  5.6 分段存储管理方式
-  5.7 基于IA-32/x86-64架构
的内存管理策略

第5章 存储器管理

- OS 进程的逻辑地址空间可能是不连续的，如果有可用的物理内存，它将分给进程。
- OS 把物理内存分成大小固定的块，称为**物理块**(frame)。（大小为2的幂，通常为1KB~8KB）
- OS 把逻辑内存也分成固定大小的块，称为**页**(page)。
- OS 保留所有空闲块的记录。
- OS 运行一个有N页大小的程序，需要找到N个空的页框来装入程序
- OS 存在页**内碎片**：进程最后一页经常装不满，而形成不可利用的碎片

31

12 11

0

页号P

位移量W (页内地址)

分页地址中的结构 (32位)

页号P

- ◆ 12-31位：20位
- ◆ 地址空间最多允许有1M (2^{20}) 页

位移量W (页内地址)

- ◆ 0-11: 12位
- ◆ 每页大小为4KB (2^{12})



对某特定机器，地址结构是一定的。



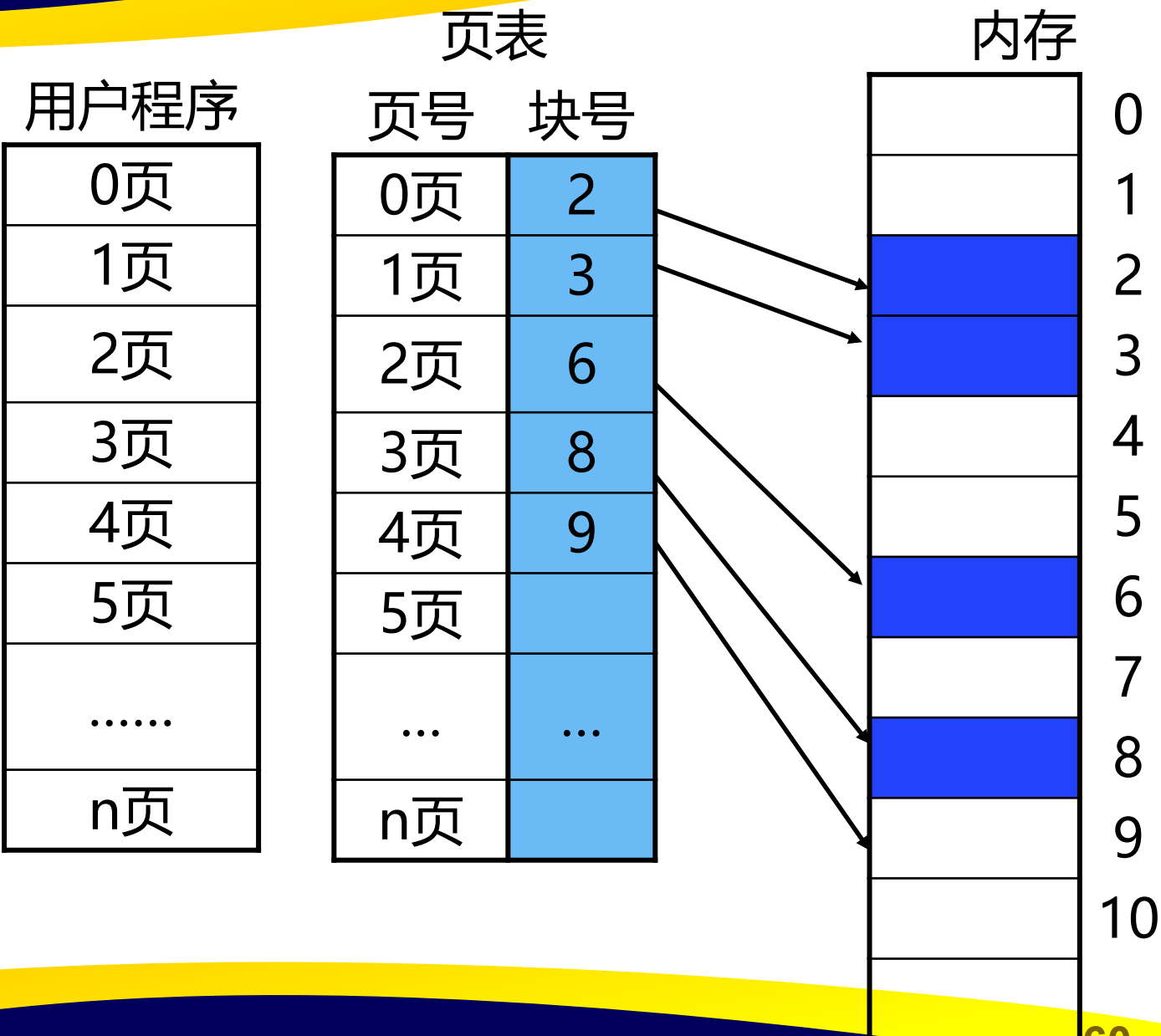
若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得

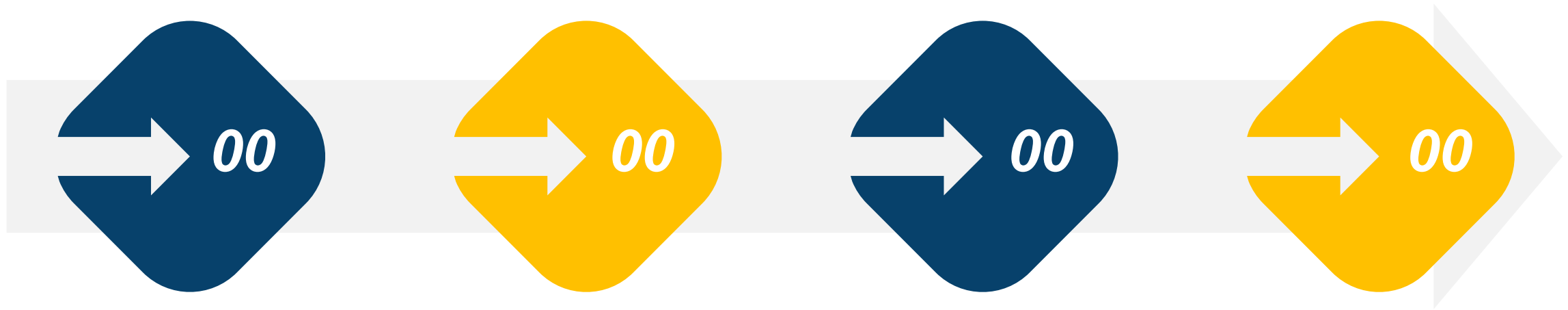
$$P = \text{INT} \left[\frac{A}{L} \right]$$

$$d = [A] \text{ MOD } L$$

- 其中，INT：整除函数，MOD：取余函数
- 例如：系统页面大小为1KB，设A=5168B，则P=5, d=48

- ❑ 系统为每个进程建立了一张页表。
- ❑ 逻辑地址空间内的所有页，依次在页表中有一表项，记录相应页在内存中对应的物理块号。
- ❑ 页表的作用：实现从页号到块号的地址映射。



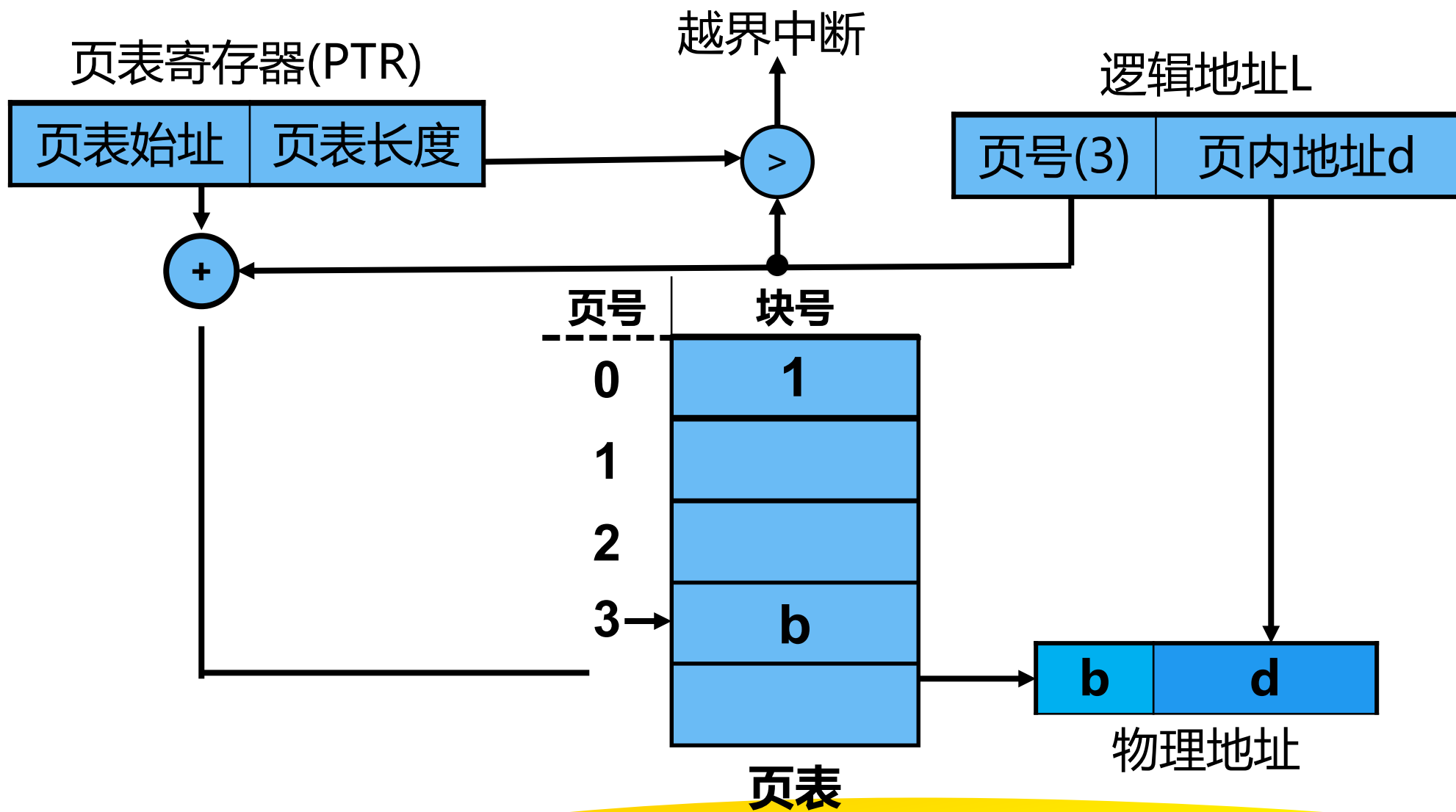


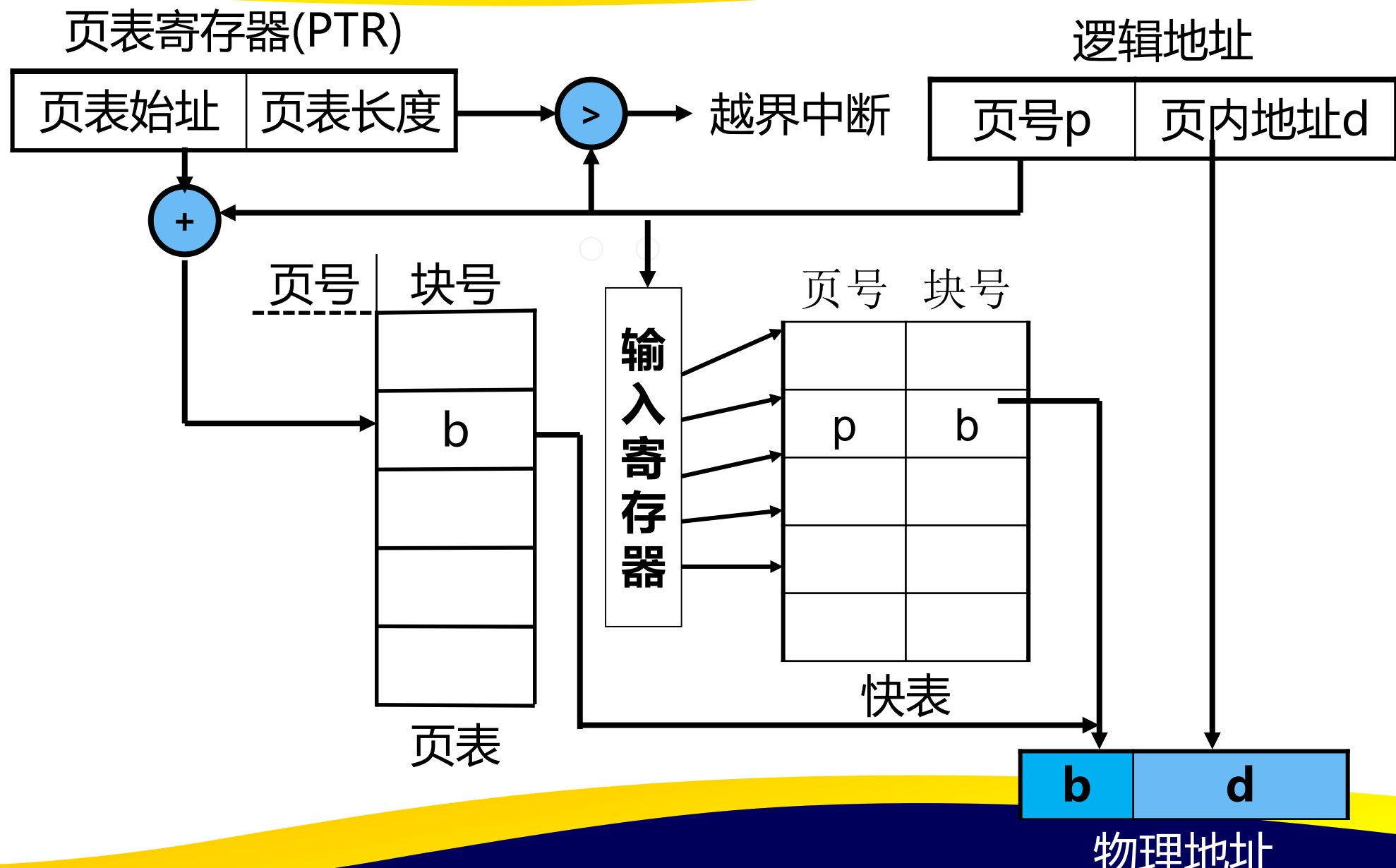
页表被保存在
主存中

页表寄存器(PTR)
指向页表的起始
地址和长度

在这个机制中，每
一次的数据/指令存
取需要**两次内存访
问**，一次是访问页
表，一次是访问数
据/指令

解决两次访问的问
题，是采用小但专
用且快速的硬件缓
冲，这种缓冲称为
转换表缓冲器(TLB)
或联想寄存器。







引入快表后的有效访问时间EAT

- 查找快表需要的时间为 λ
- 假设访问内存一次需要的时间为 t
 $EAT = t + t = 2t$ (基本分页存储系统中)
- 命中率—在联想寄存器中找到页号的百分比，比率与联想寄存器的大小有关，假设为 a
- 引入快表的有效访问时间 (EAT)
 $EAT = \lambda * a + (t + \lambda)(1 - a) + t$

快表大小：16-512个页表项，局部性：90%

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



现代的大多数计算机系统，都支持非常大的逻辑地址空间。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。



解决办法：

- 对页表所需要的内存空间，采用离散分配方式
- 部分页表调入内存



页表结构

- 两级页表
- 多级页表
- 反置页表

例子 (32位逻辑地址空间)



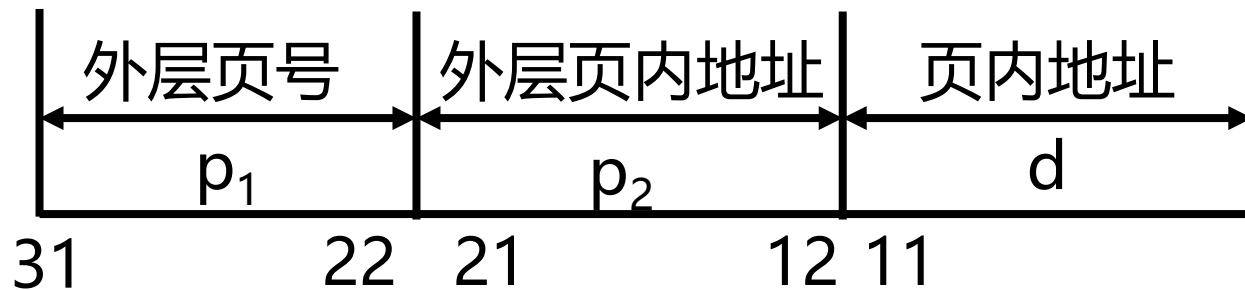
页面大小为4KB时 (12位), 若采用一级页表结构, 应具有20位的页号, 即页表项应有1M个;



在采用两级页表结构时, 再对页表进行分页, 使每页中包含 2^{10} 个页表项, 最多允许有 2^{10} 个页表分页。

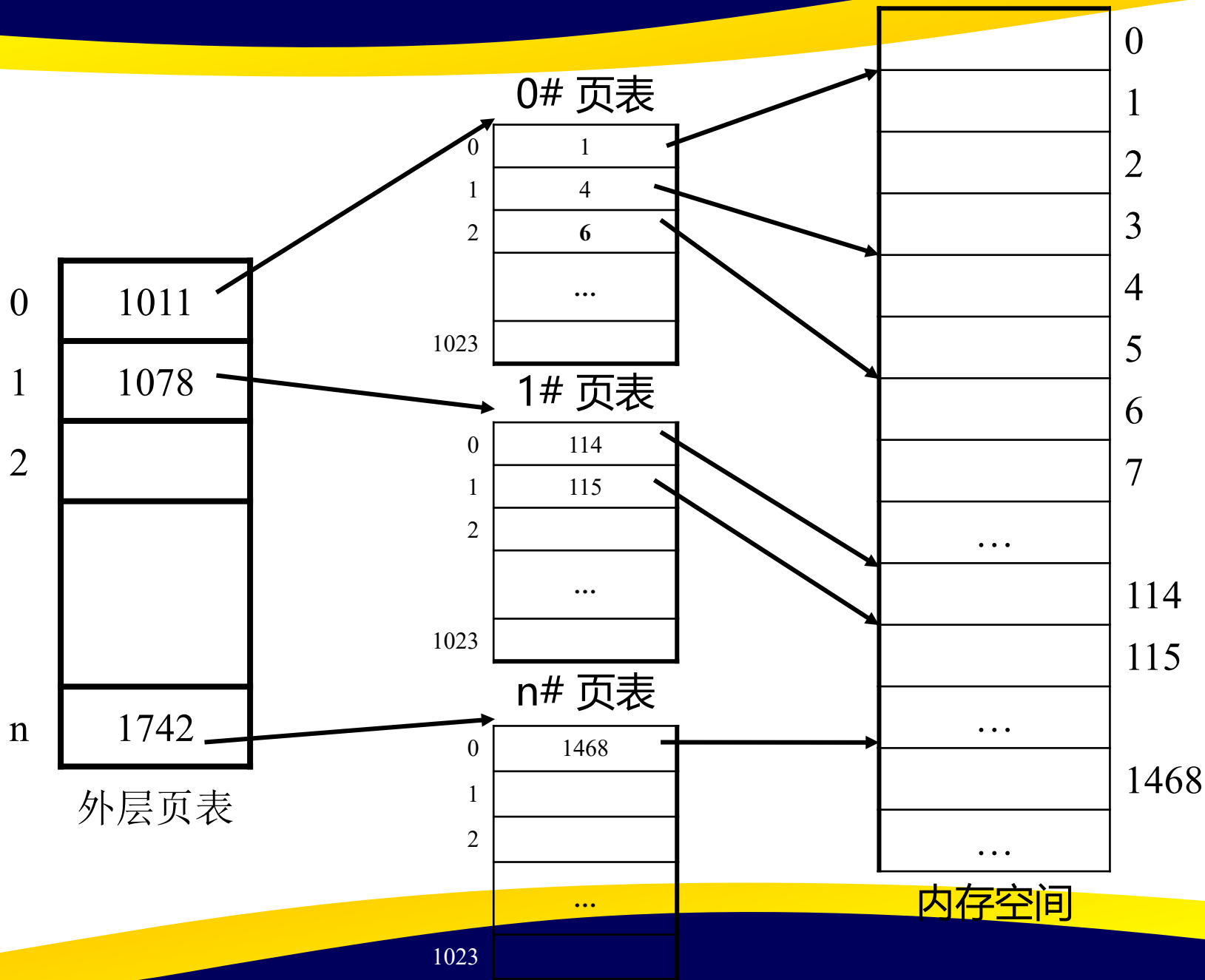


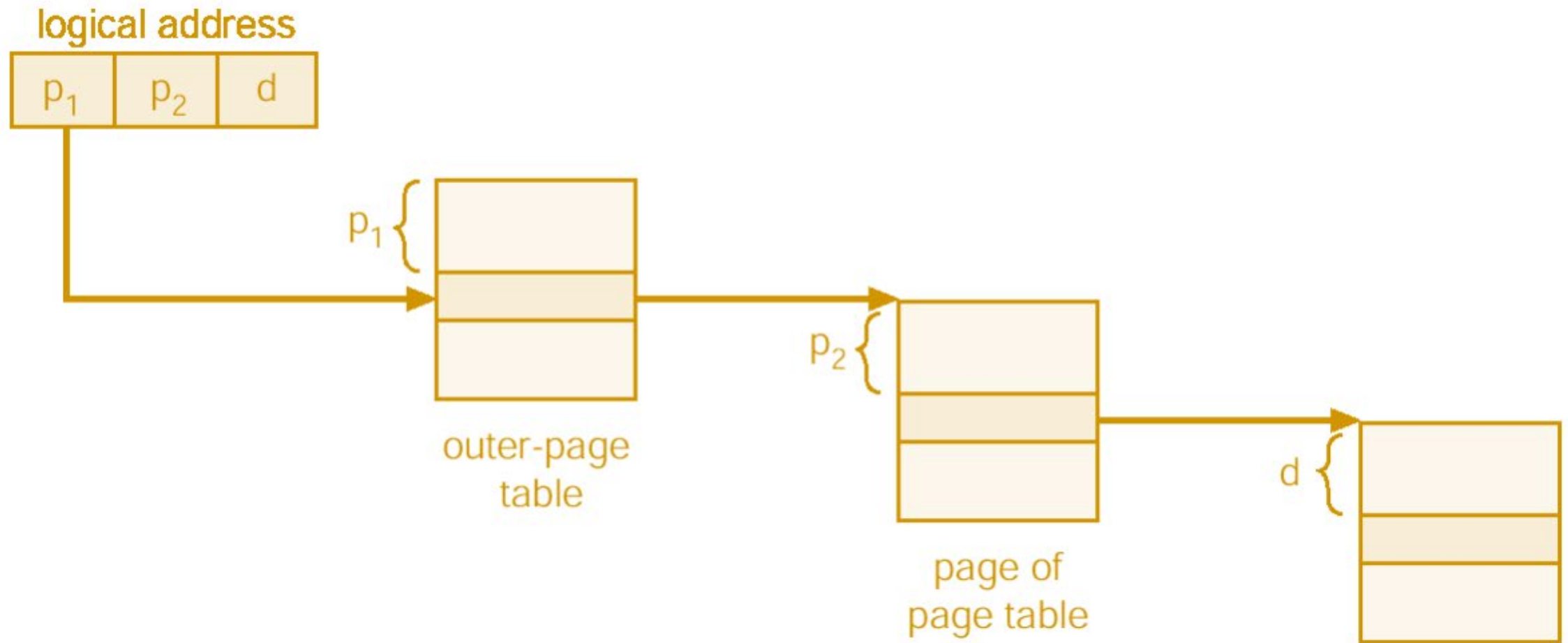
地址变换机构上,
增设一个外层页表
寄存器





两级页表结构





对于64位的机器，采用两级页表是否仍可使用？



如果页面大小为4KB，那么还剩下52位，假定物理块大小为4K来划分页表，则余下的42位用于外层页号。此时在外层页表中可能有4096G个页表项，要占用16384GB的连续内存空间

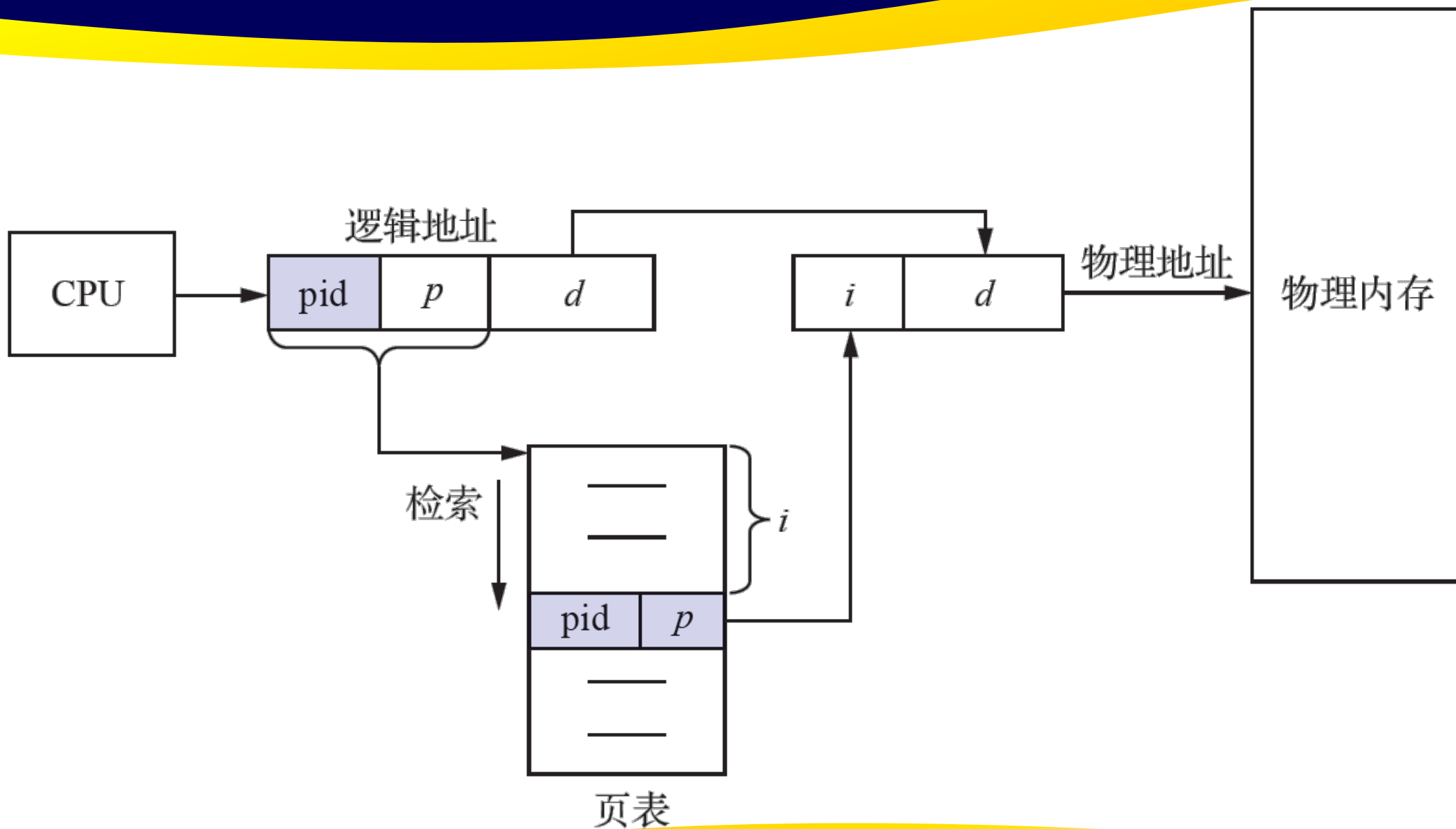


使用多级页表

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

- OS 为减少页表占用的内存空间，引入反置页表，一个系统一张页表
- OS 对每个内存物理块设置一个条目。
- OS 每个条目保存在真正内存位置的页的虚拟地址，以及包括拥有这个页的进程的信息。
- OS 减少了需要储存每个页表的内存，但是当访问一个页时，增加了寻找页表需要的时间。
- OS 可使用哈希表来将查找限制在一个或少数几个页表条目。
- OS AS/400、IBM RISC System 和IBM RT采用



在一个分页存储管理系统中，页面大小为4KB，系统中的地址占24位，给定页面变换表如下表所示。

- (1) 计算逻辑地址（页号为3，页内地址为100）的物理地址。
- (2) 说明地址变换过程。

页号P	块号B
0	3
1	4
2	9
3	7

在一个分页存储管理系统中，页面大小为4KB，系统中的地址占24位，给定页面变换表如下表所示。

- (1) 计算逻辑地址（页号为3，页内地址为100）的物理地址。
- (2) 说明地址变换过程。

页号P	块号B
0	3
1	4
2	9
3	7


(1) 逻辑地址（页号3，页内地址100）的物理地址为：

$$7 \times 4\text{KB} + 100 = 28\text{KB} + 100 = 28772$$

(2) 在请求分页存储管理中，系统是通过页表进行地址转换。先将逻辑地址分解成页号P和页内地址W两部分，然后查页表，可得页号P对应的物理块号为B，从而变换出对应的物理地址为：物理地址=块号×页面大小+页内地址



内容导航:

-  5.1 存储器的层次结构
-  5.2 程序的装入和链接
-  5.3 对换与覆盖
-  5.4 连续分配存储管理方式
-  5.5 分页存储管理方式
-  **5.6 分段存储管理方式**
-  5.7 基于IA-32/x86-64架构
的内存管理策略

第5章 存储器管理

- **引入目的:** 为了满足用户（程序员）在编程和使用上多方面的要求
- 一个程序是一些段的集合，一个段是一个逻辑单位，如：

main program,
procedure,
function,
local variables, global variables,
common block,
stack,
symbol table, arrays





方便编程

- 用户将自己的作业按照逻辑关系划分为若干个段，每个段都是从0开始编址，并有自己的名字和长度。因此，希望访问的逻辑地址是由段名（段号）和段内偏移量（段内地址）决定的



信息共享

- 在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。分页系统中的“页”只是存放信息的物理单位，并无完整的意义，不便于实现共享；然而段却是信息的逻辑单位。



信息保护

- 信息保护同样是对信息的逻辑单位进行保护



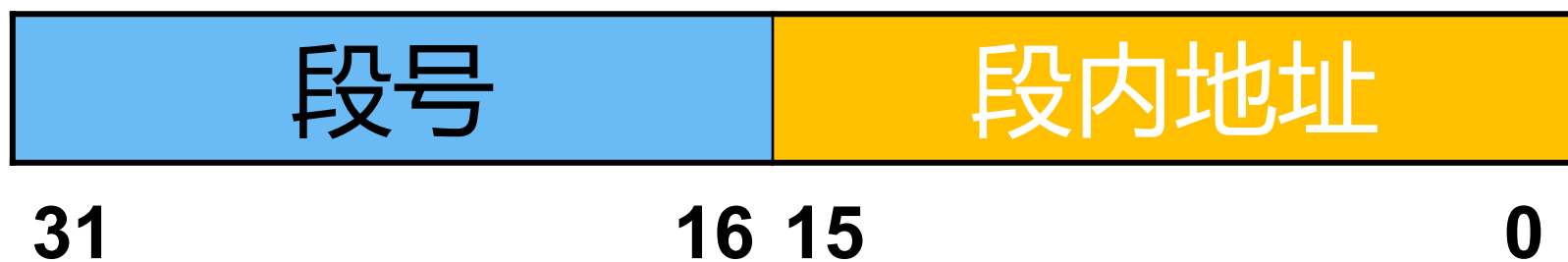
动态链接



动态增长

- 在实际应用中，往往有些段（特别是数据段），在使用过程中会不断地增长

- 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。
- 每个段都有自己的名字，通常用一个段号来代替段名，每个段都从0开始编址，并存储在一段连续的地址空间内
- 段的长度由相应的逻辑信息组的长度决定，因此各段长度不等。

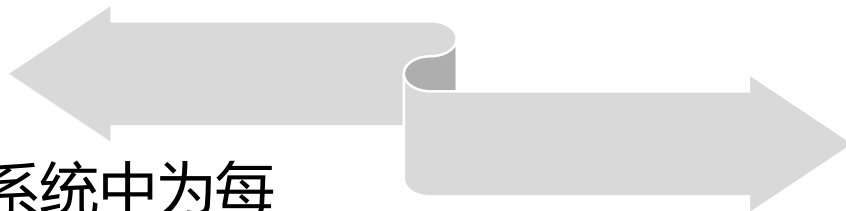


在分段式存储管理系统中，为每个分段分配一个连续的分区。进程中的各个段可以离散地装入内存中不同的分区中。



类似于分页系统，在系统中为每个进程建立一张段映射表（段表），用于实现从逻辑段到物理内存区的映射。

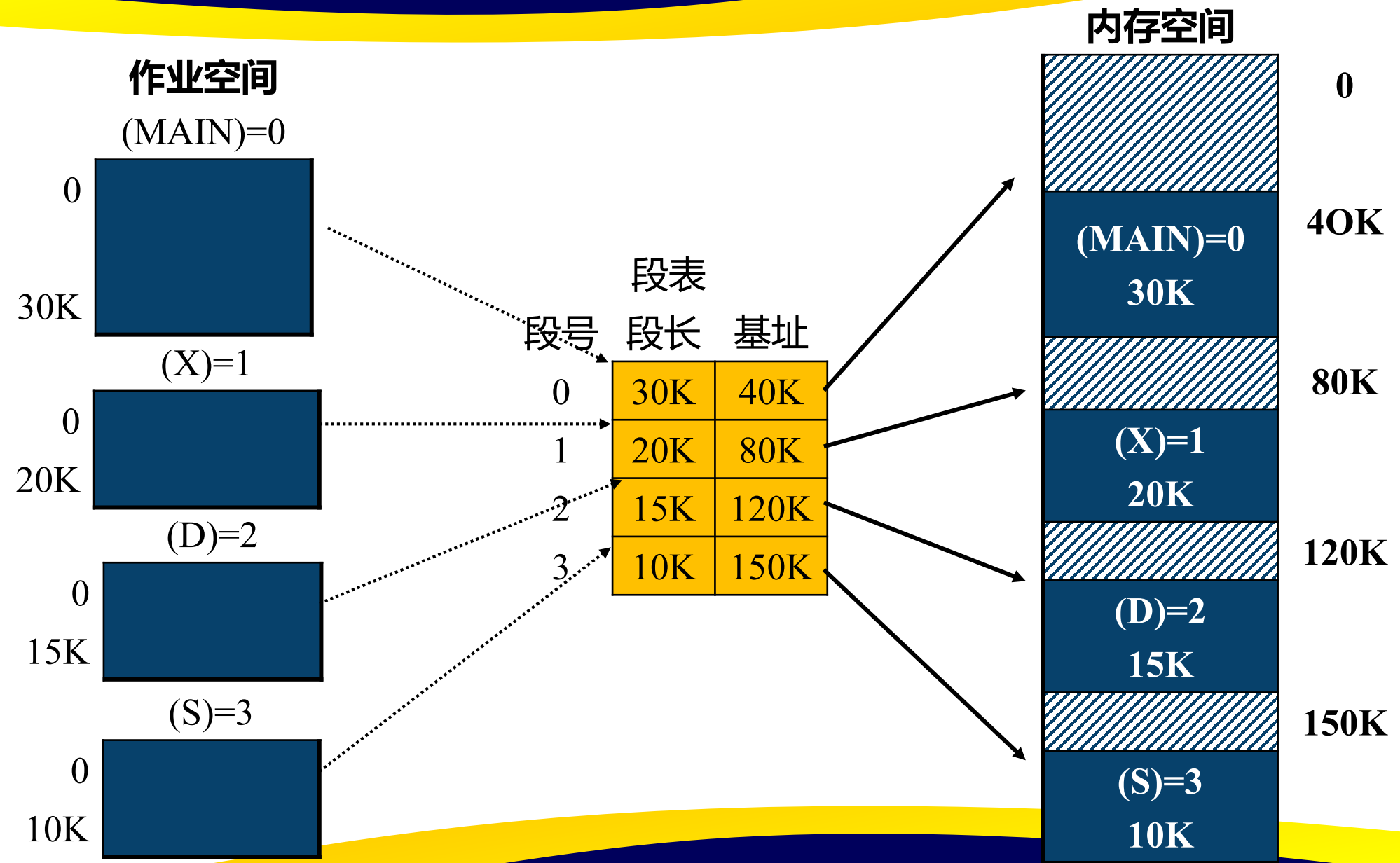
- 每个段在表中占有一个表项，记录了该段在内存中的起始地址（基址）和段的长度。

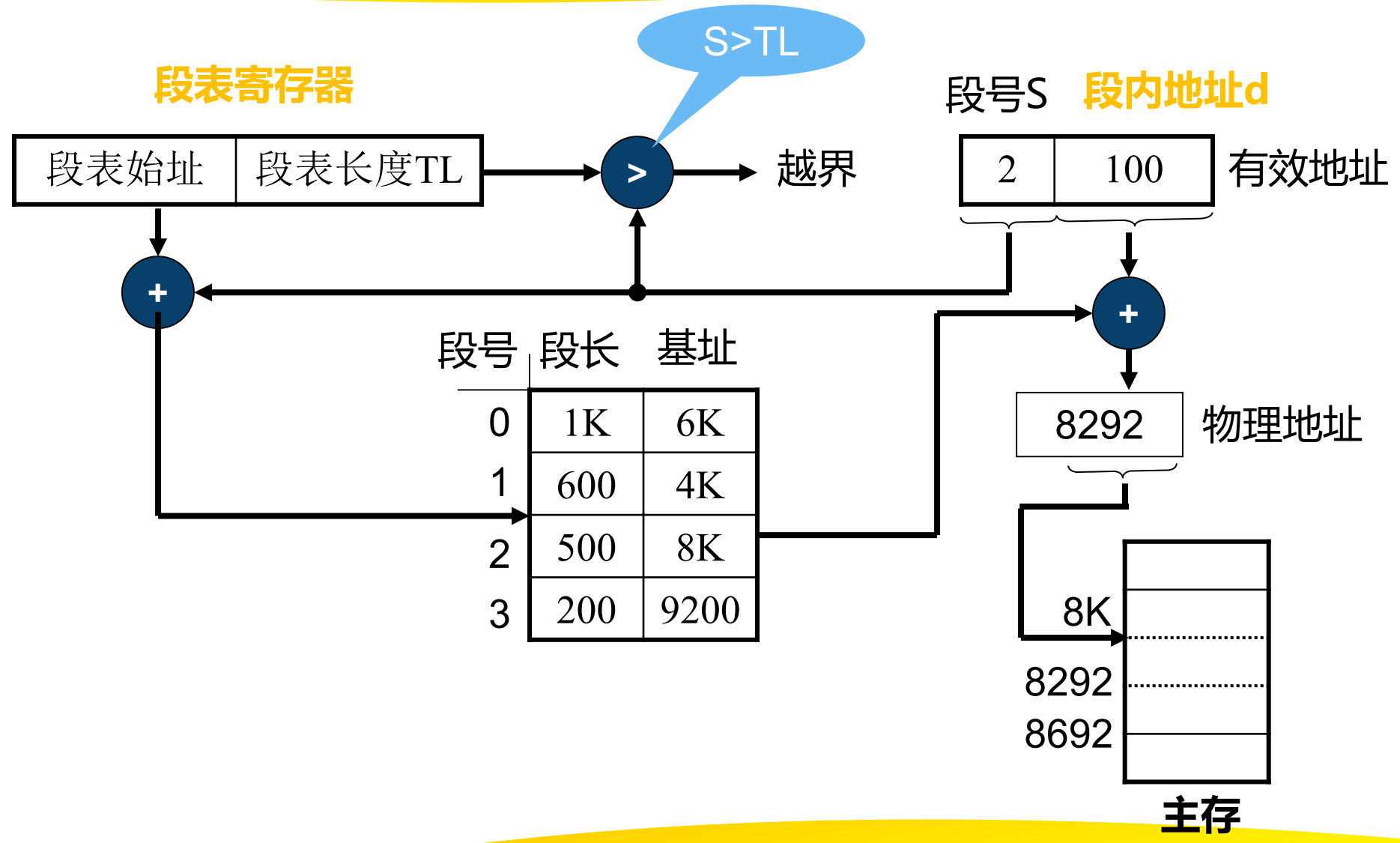


段表保存在内存中，由控制寄存器保存其地址。



利用段表实现地址映射



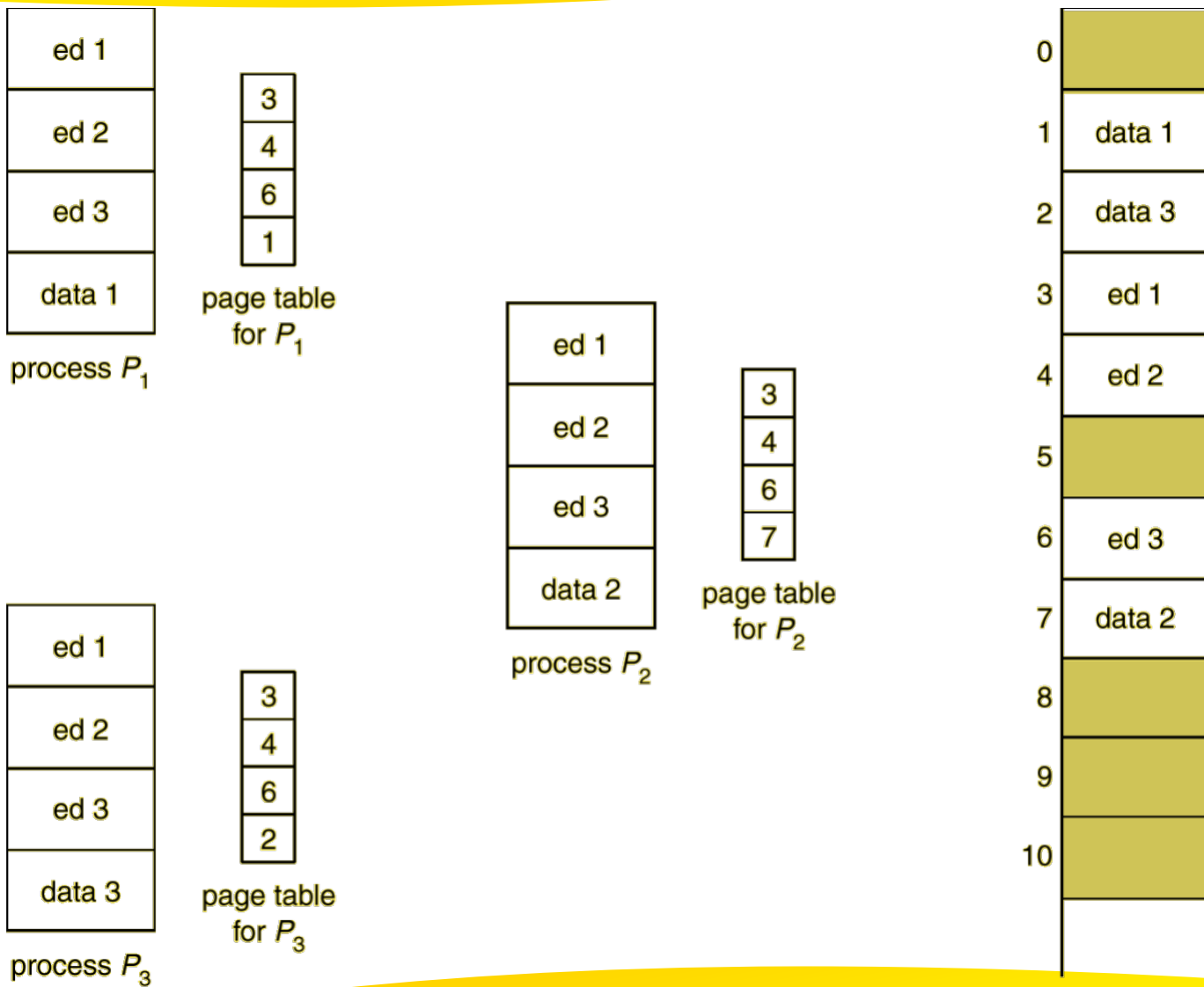


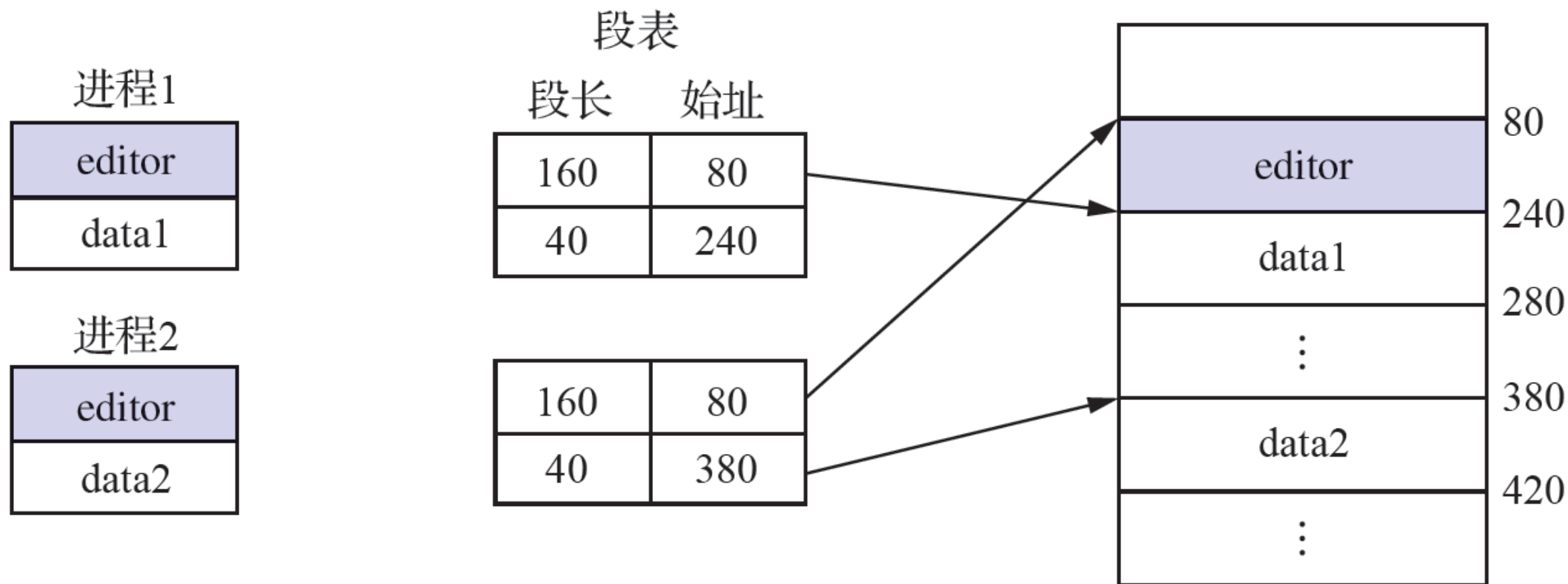


分页和分段的主要区别

	分页	分段
信息单位	页	段（逻辑上有意义）
信息完整性	离散分配方式	意义相对完整
页/段的大小	固定，由系统决定	不固定，由程序员决定

- 分段的一个突出优点，是易于实现段的共享，即允许若干个进程共享一个或多个分段，且对段的保护也十分简单易行。
- 在分页系统中，虽然也能实现程序和数据的共享，但远不如分段系统来得方便。
- **可重入代码**（纯代码）：一种允许多个进程同时访问的代码。







基本原理：

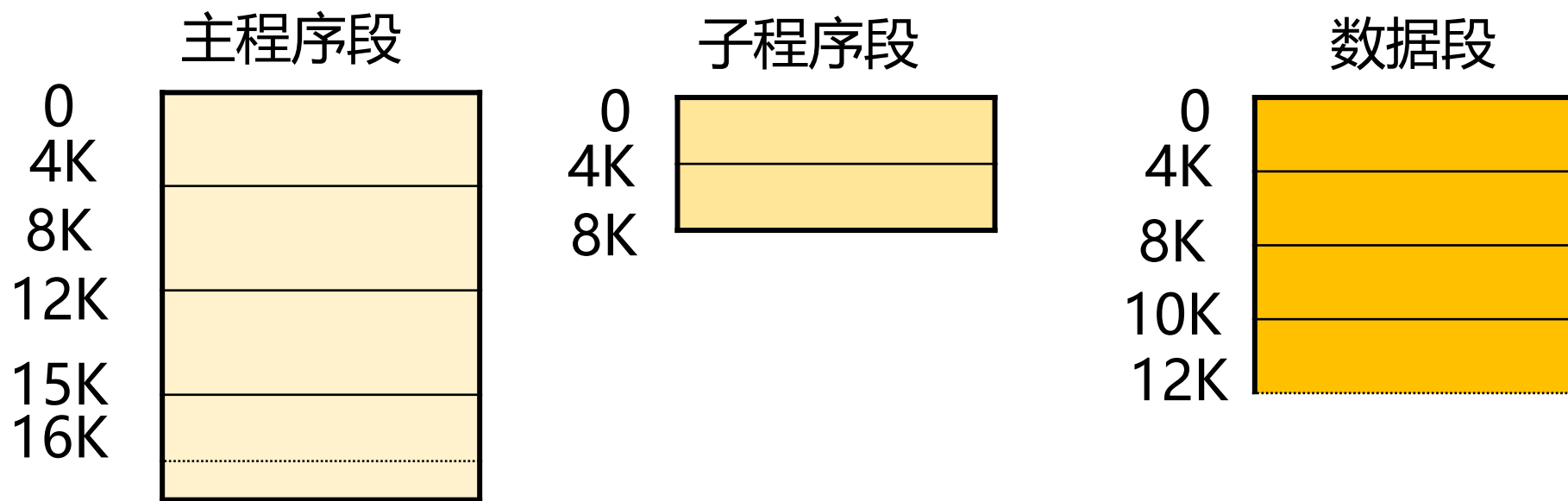
- **分段和分页原理的结合**，即先将用户程序分成若干段，再把每个段分成若干个页，并为每个段赋予一个段名。

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。

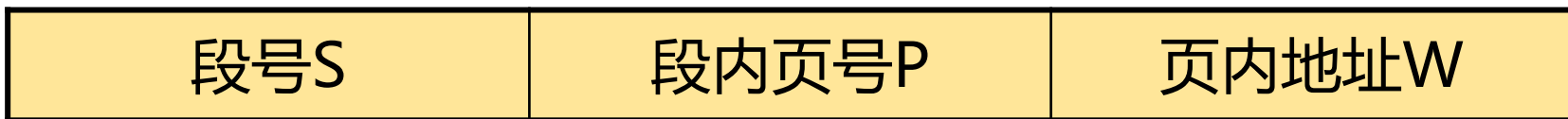


优点：

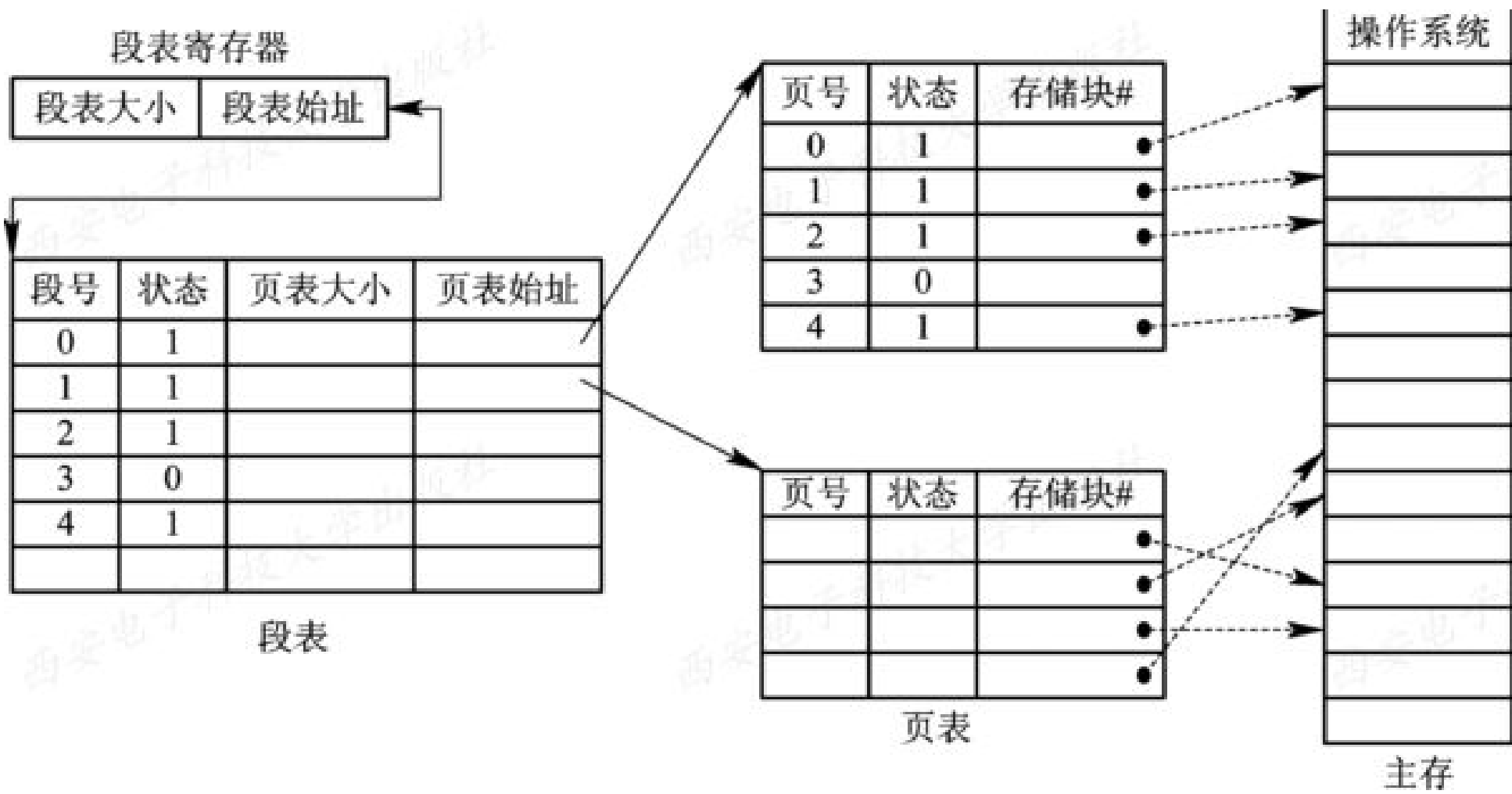
- 既有分段系统的便于实现、可共享、易于保护、可动态链接；
- 又能像分页系统，很好地解决内存的外部碎片问题。



(a)



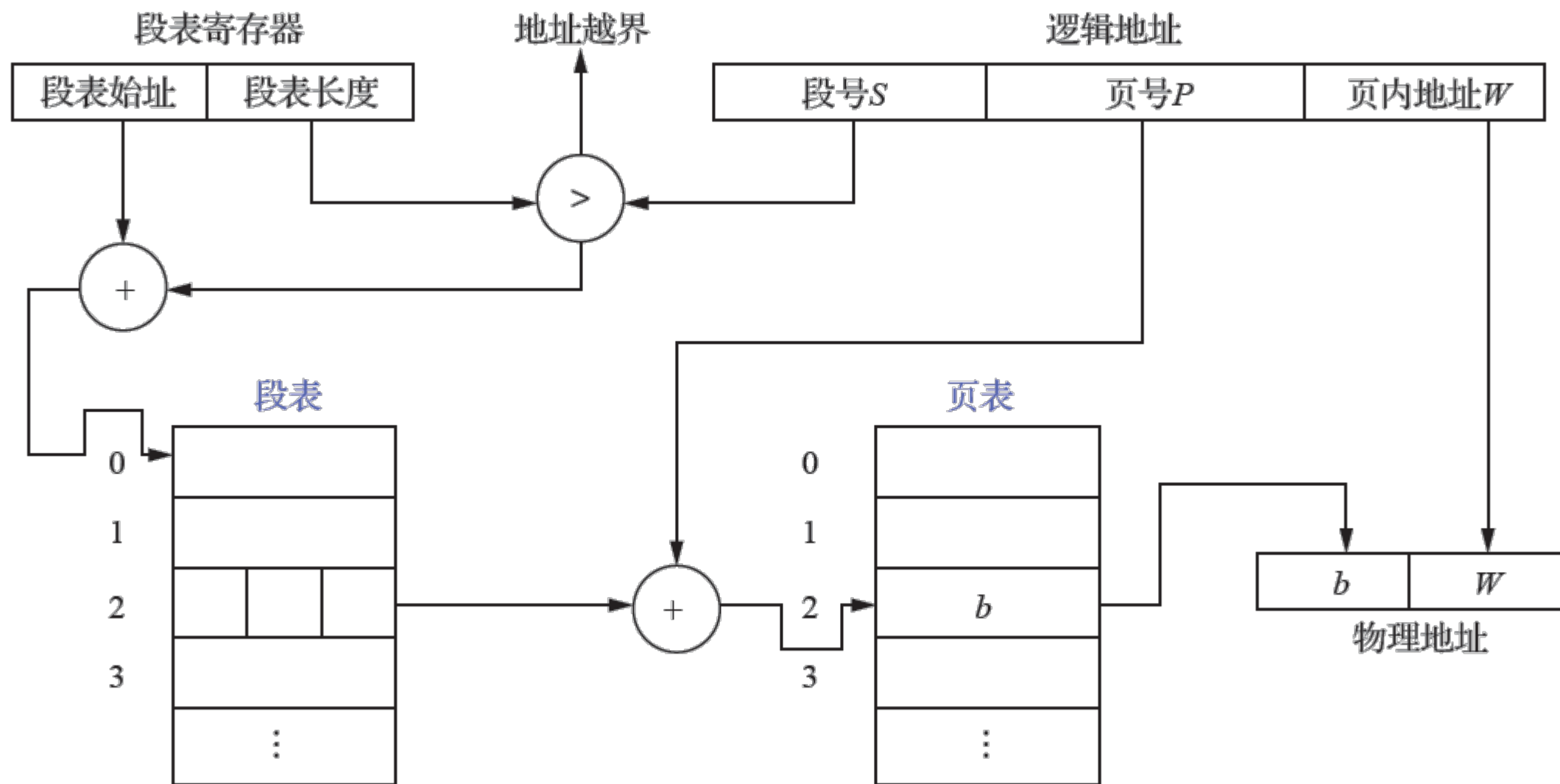
(b)



2. 地址变换过程

在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中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 $S < TL$ ，表示未越界，于是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，并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，从中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。图4-25示出了段页式系统中的地址变换机构。

OS 段页式存储管理方式的地址变换过程





内容导航:

-  5.1 存储器的层次结构
-  5.2 程序的装入和链接
-  5.3 对换与覆盖
-  5.4 连续分配存储管理方式
-  5.5 分页存储管理方式
-  5.6 分段存储管理方式
-  **5.7 基于IA-32/x86-64架构
的内存管理策略**

第5章 存储器管理

分为两部分：分段和分页

工作方案：



01

**CPU生成逻辑地址，
并将其交给分段单
元**

02

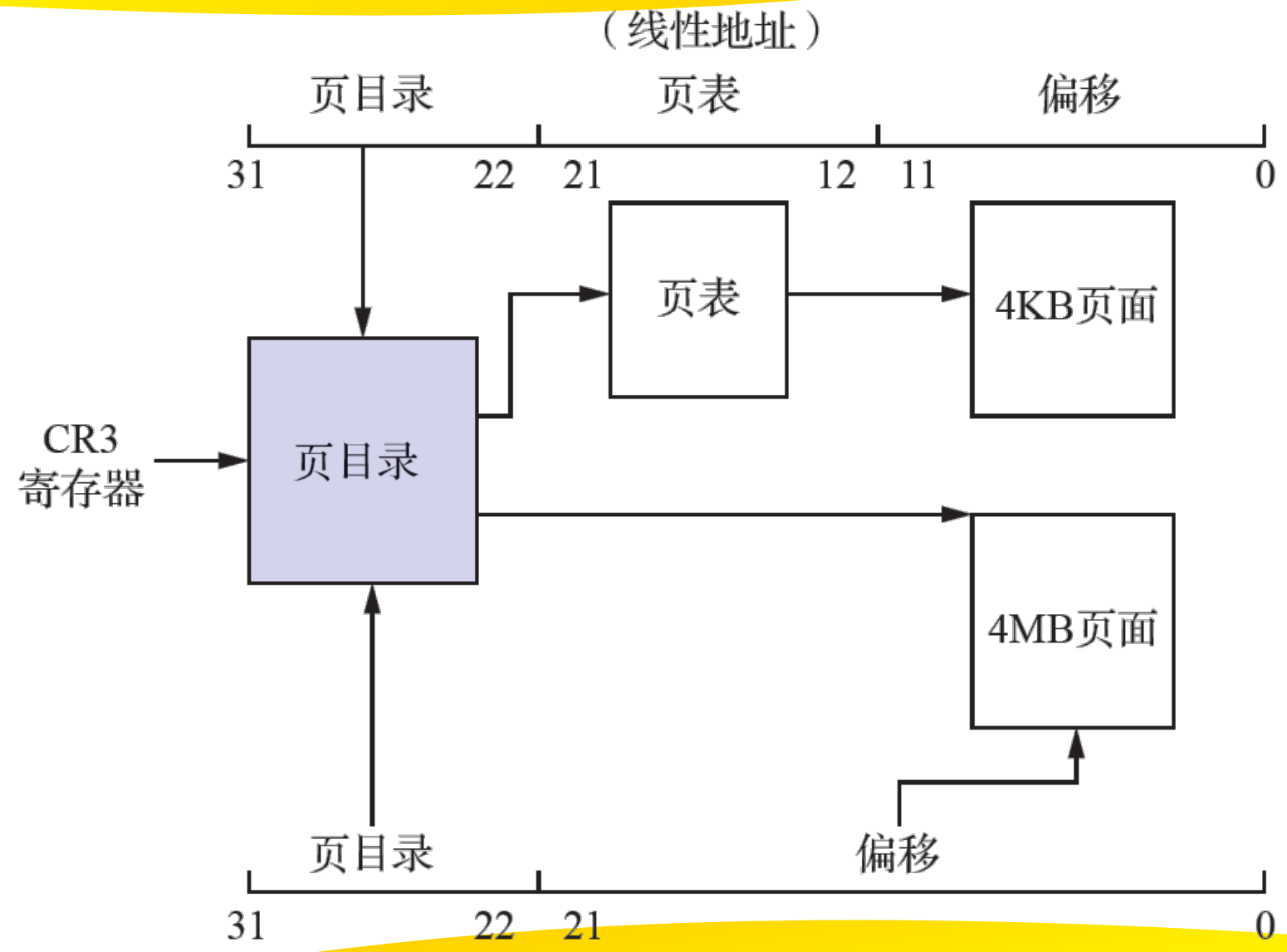
分段单元为每个逻辑地址生成一个线性地址

03

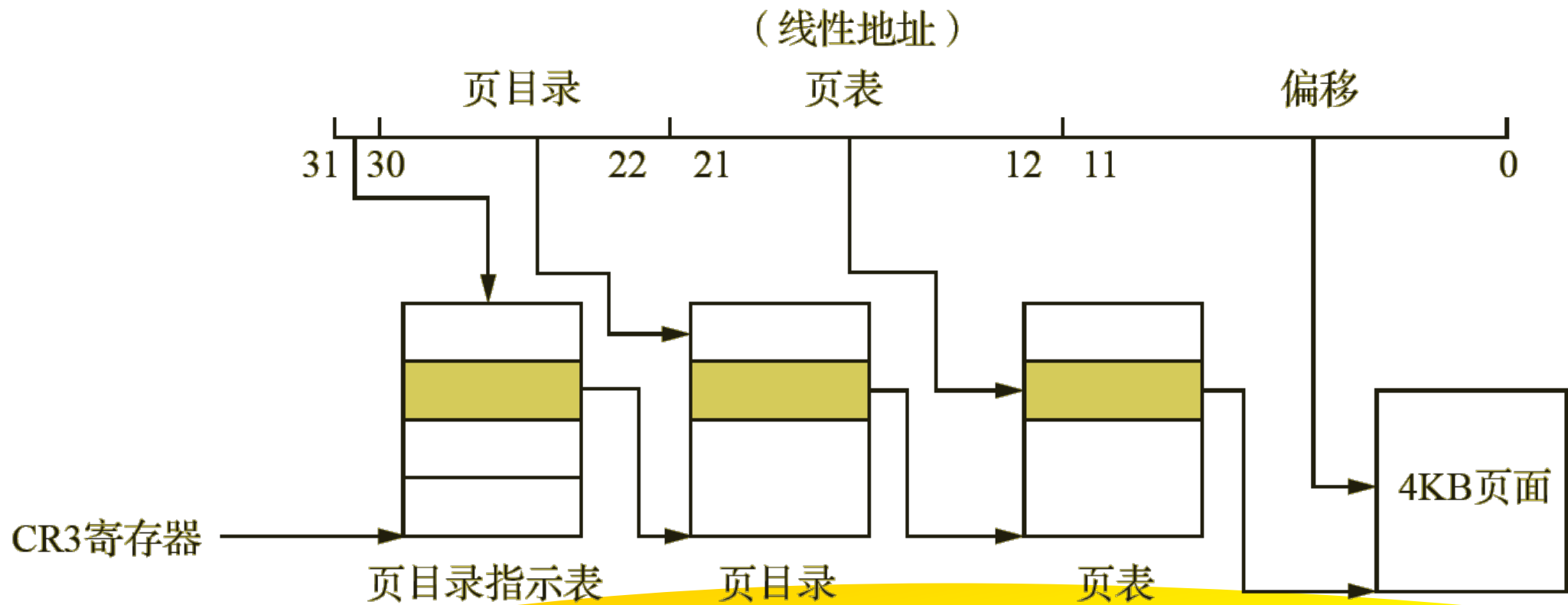
将线性地址交给分页单元，以生成内存的物理地址



IA-32架构的分页（二级分页）



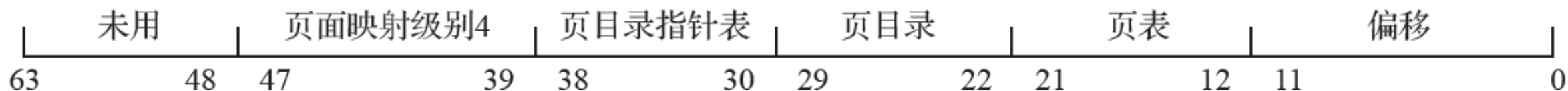
- 针对64位的硬件平台，二级分页不合适
- Linux采用了适合32和64位架构的三级分页模式
- 采用页地址扩展PAE，可访问大于4GB的物理地址空间





X86-64架构的地址架构的分页（四级分页）

- 64位的地址空间意味着寻址的内存达到264B，即大于16EB（1EB = 220TB）。
- 目前采用四级分页模式，支持48位虚地址，它的页面大小可为4KB、2MB或1GB。
- 能够采用PAE，系统虚拟地址的大小为48位，可支持52位的物理地址（4096TB）。





第一次作业

简答题

1

2

3

4

5

6

7

8

9

10

11

计算题

12

13

14

15

16

17

综合应用题

18

19

20

标黄色为本次作业



第二次作业

简答题

1	2	3	4	5	6	7	8
9	10	11					

计算题

12	13	14	15	16	17
----	----	----	----	----	----

综合应用题

18	19	20
----	----	----

标黄色为本次作业



警惕Windows 10系统安全问题

坚持走自主创新发展国产操作系统之路

2014年10月1日，微软公司在美国旧金山召开新品发布会，对外展示了新一代Windows系统，并将其命名为“Windows 10”。2015年3月18日，微软公司中国官网正式推出Windows 10系统中文介绍页面；7月29日，微软公司发布了Windows 10正式版。

在此背景下，中国信息安全测评中心承担了对Windows 10系统的网络安全审查工作，并于审查过程中发现其存在安全问题，即没有通过审查！

实际上，我们并不排斥引进国外操作系统，但是要求引进的国外操作系统必须完全可控。微软公司首先未向我们提供全部可重构的源代码，即有百万行量级的源代码未对我们开放（这等于是“黑盒子”），那么这部分源代码就无法做“白盒测试”，在这种情况下我们就难以对其安全性、可控性做准确评估。



警惕Windows 10系统安全问题

坚持走自主创新发展国产操作系统之路

同时Windows 10系统捆绑了可信计算技术和杀毒软件，这也对我国可信计算厂商和国内外信息安全厂商构成了不正当竞争！因此，我们要高度警惕Windows 10系统的安全问题，力求通过自主创新发展国产操作系统。

此外，考虑到我国的超级计算机、知名互联网企业等均采用了开源Linux系统这一基础软件，且经过实践检验证明其切实可行。

因此，我们应该深刻理解“国产操作系统才是安全可控国产信息技术体系的核心”这一关键点，基于开源Linux系统，依靠各方力量创新发展安全可靠、自主可控的国产操作系统。