

# 第五章 虚 拟 存 储 器

5.1 虚拟存储器概述

5.2 请求分页存储管理方式

5.3 页面置换算法

5.4 “抖动”与工作集

5.5 请求分段存储管理方式

## 5.1 虚拟存储器概述

第四章所介绍的各种**全部**存储器管理方式有一个共同的特点，即它们都要求将一个作业装入内存后方能运行。于是，出现了下面这样两种情况：

(1) 有的作业很大，其所要求的内存空间超过了内存总容量，**作业不能全部被装入内存**，致使该作业无法运行；

(2) 有大量作业要求运行，但由于内存容量不足以容纳所有这些作业，**只能将少数作业装入内存让它们先运行**，而将其它大量的作业留在外存上等待。

## 5.1.1 常规存储管理方式的特征和局部性原理

### 1. 常规存储器管理方式的特征

(1) 一次性：作业必须一次性地全部装入内存后方能开始运行。

(2) 驻留性：作业被装入内存后，整个作业都一直驻留在内存中，其中任何部分都不会被换出，直至作业运行结束。

一次性和驻留性，使许多在程序运行中不用或暂不用的程序(数据)占据了大量的内存空间，使得一些需要运行的作业无法装入运行。现在要研究的问题是：**一次性及驻留性在程序运行时是否是必需的。**



## 2. 局部性原理

1968年，P.Denning指出：程序在执行时将呈现出**局部性规律**，即**在一较短的时间内，程序的执行仅局限于某个部分，相应地，它所访问的存储空间也局限于某个区域。**

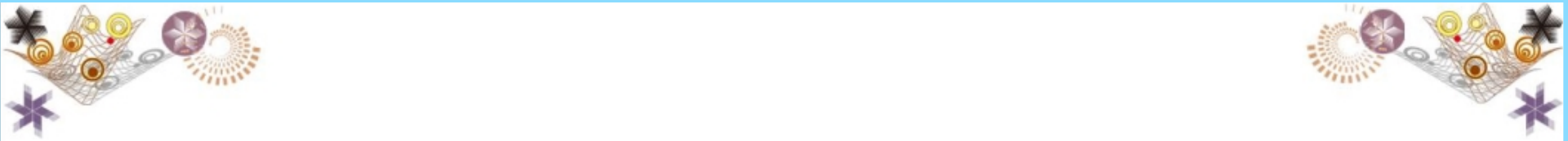
## P. Denning的四个论点

(1) 程序执行时，除了少部分的转移和过程调用指令外，在**大多数情况下仍是顺序执行的**。该论点也在后来的许多学者对高级程序设计语言(如FORTRAN语言、PASCAL语言)及C语言规律的研究中被证实。

(2) 过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域，但经研究看出，**过程调用的深度在大多数情况下都不超过5**。这就是说，程序将会在一段时间内都局限在**这些过程的范围内运行**。

(3) 程序中存在许多**循环结构**，这些虽然只由**少数指令**构成，但是它们将**多次执行**。

(4) 程序中还包括许多对数据结构的处理，如对**数组**进行操作，它们往往都局限于很小的范围内。



局限性又表现在下述两个方面：

(1) 时间局限性：如果程序中的某条指令一旦执行，则不久以后该指令可能再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局限性的典型原因是由于在程序中存在着大量的循环操作。

(2) 空间局限性：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。


### 3. 虚拟存储器的基本工作情况

基于局部性原理，应用程序在运行之前，没有必要全部装入内存，**仅须将那些当前要运行的少数页面或段先装入内存便可运行，其余部分暂留在盘上。**

程序在运行时，如果它所访问的页(段)已调入内存，便可继续执行下去；但如果程序所要访问的页(段)尚未调入内存(称为**缺页**或**缺段**)，此时程序应利用OS所提供的**请求调页(段)功能，将它们调入内存，以使进程能继续执行下去。**

如果此时内存已满，无法再装入新的页(段)，则还须再利用**页(段)的置换功能，将内存中暂时不用的页(段)调至盘上**，腾出足够的内存空间后，再将要访问的页(段)调入内存，使程序继续执行下去。





## 5.1.2 虚拟存储器的定义和特征

### 1. 虚拟存储器的定义

虚拟存储器是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。

其逻辑容量由内存容量和外存容量之和所决定，其运行速度接近于内存速度，而每位的成本又接近于外存。

虚拟存储器是一种性能非常优越的存储器管理技术，广泛地应用于大、中、小型机器和微型机中。



## 2. 虚拟存储器的特征

(1) 多次性：多次性是指一个作业被分成多次调入内存运行。亦即在作业运行时没有必要将其全部装入，只需将当前要运行的那部分程序和数据装入内存即可；以后每当要运行到尚未调入的那部分程序时，再将它调入。

(2) 对换性：对换性是指允许在作业的运行过程中进行换进、换出，在进程运行期间，允许将那些暂不使用的程序和数据，从内存调至外存的对换区(换出)，待以后需要时再将它们从外存调至内存(换进)；甚至还允许将暂时不运行的进程调至外存，待它们重又具备运行条件时再调入内存。

(3) 虚拟性：虚拟性是指能够从逻辑上扩充内存容量，使用户所看到的内存容量远大于实际内存容量。

### 5.1.3 虚拟存储器的实现方法

#### 1. 分页请求系统

这是在分页系统的基础上，增加了请求调页功能和页面置换功能所形成的页式虚拟存储系统。

它允许只装入少数页面的程序(及数据)，便启动运行。以后，再通过调页功能及页面置换功能，陆续地把即将要运行的页面调入内存，同时把暂不运行的页面换出到外存上。置换时以页面为单位。为了能够实现请求调页和置换功能，系统必须提供必要的硬件支持和相应的软件。



## 1) 硬件支持

(1) 请求分页的**页表机制**：它是在纯分页的页表机制上增加若干项而形成的，作为请求分页的数据结构。

(2) **缺页中断机构**：即每当用户程序要访问的页面尚未调入内存时，便产生一缺页中断，以请求OS将所缺的页调入内存；

(3) **地址变换机构**：它同样是在纯分页地址变换机构的基础上发展形成的。



## 2) 实现请求分页的软件


包括有用于实现请求调页的软件和实现页面置换的软件。它们在硬件的支持下，将程序正在运行时所需的页面(尚未在内存中的)调入内存，再将内存中暂时不用的页面从内存置换到磁盘上。



## 2. 请求分段系统

在分段系统的基础上，增加了请求调段及分段置换功能后所形成的段式虚拟存储系统。

它允许只装入少数段(而非所有的段)的用户程序和数据，即可启动运行。以后再通过调段功能和段的置换功能将暂不运行的段调出，同时调入即将运行的段。置换是以段为单位进行的。



## 1) 硬件支持

(1) 请求分段的**段表机制**：这是在纯分段的段表机制基础上增加若干项而形成的。

(2) **缺段中断机构**：每当用户程序所要访问的段尚未调入内存时，产生一个缺段中断，请求OS将所缺的段调入内存。

(3) **地址变换机构**。

## 2) 软件支持

用于实现**请求调段**的软件和实现**段置换**的软件。



## 5.2 请求分页存储管理方式

### 5.2.1 请求分页中的硬件支持

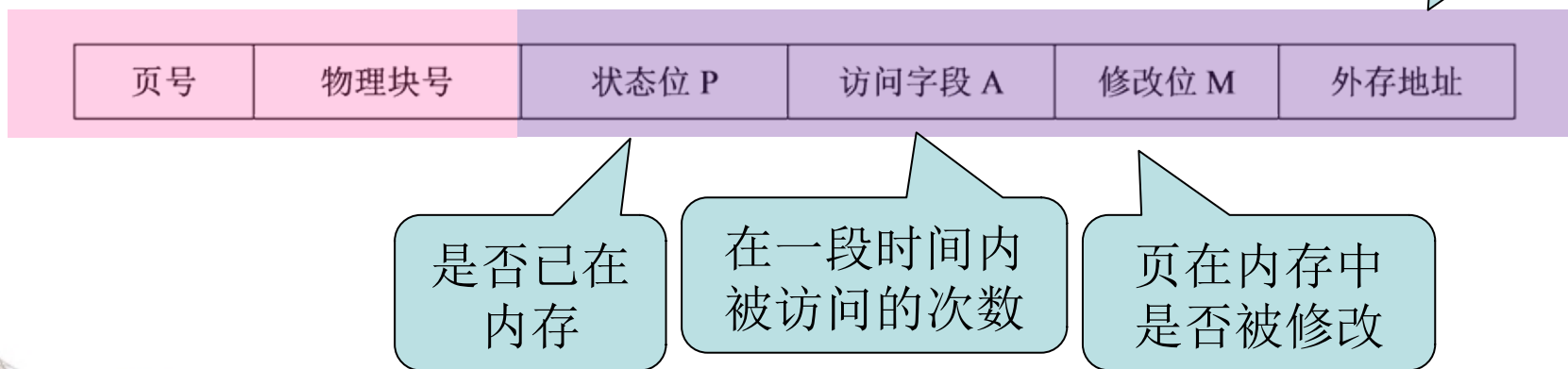
为了实现请求分页，系统必须提供一定的硬件支持。计算机系统除了要求一定容量的内存和外存外，还需要有请求页表机制、缺页中断机构以及地址变换机构。



## 1. 请求页表机制

在请求分页系统中需要的主要数据结构是请求页表，其基本作用仍然是将用户地址空间中的逻辑地址映射为内存空间中的物理地址。

为了满足页面换进换出的需要，在请求页表中又增加了四个字段。这样，在请求分页系统中的每个页表应含以下诸项：





增加的4个字段说明如下：

(1) **状态位P**：用于指示该页是否已经被调入内存，供程序访问时参考

(2) **访问字段A**：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，提供给置换算法在选择换出页面时参考

(3) **修改位M**：标识该页在调入内存后是否被修改过，供置换页面（换出到外存）时参考

(4) **外存地址**：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考

## 2. 缺页中断机构

在请求分页系统中，每当所要访问的页面不在内存时，便产生一缺页中断，请求OS将所缺之页调入内存。**缺页中断**作为中断，它们同样需要经历诸如保护CPU环境、分析中断原因、转入缺页中断处理程序进行处理、恢复CPU环境等几个步骤。

- (1) **在指令执行期间**产生和处理中断信号。
- (2) 一条指令在执行期间可能产生**多次缺页中断**。



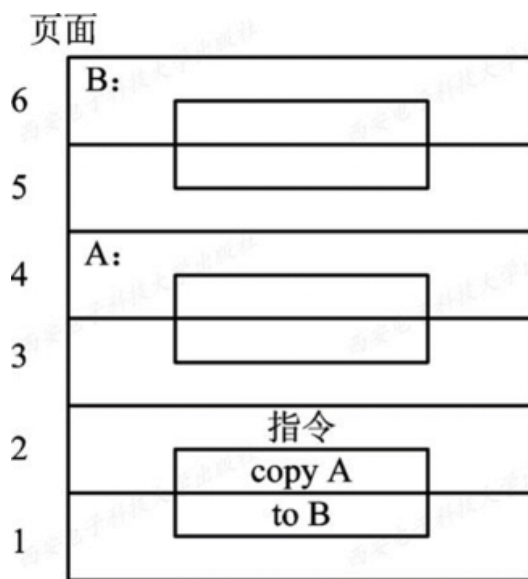
缺页中断又是一种特殊的中断，它与一般的中断相比，有着明显的区别，主要表现在下面两个方面：

(1) 在指令执行期间产生和处理中断信号。通常，CPU 都是在一条指令执行完后，才检查是否有中断请求到达。若有，便去响应，否则，继续执行下一条指令。然而，缺页中断是在指令执行期间，发现所要访问的指令或数据不在内存时所产生和处理的。



## (2) 一条指令在执行期间，可能产生多次缺页中断。

例如：执行一条指令COPY A TO B



可能要产生6次缺页中断，其中指令本身跨了两个页面，A和B又分别各是一个数据块，也都跨了两个页面。

基于这些特征，系统中的硬件机构应能保存多次中断时的状态，并保证最后能返回到中断前产生缺页中断的指令处继续执行。



### 3. 地址变换机构

请求分页系统中的地址变换机构是在分页系统地址变换机构的基础上，为实现虚拟存储器，再增加了某些功能所形成的，如产生和处理缺页中断，以及从内存中换出一页的功能等等。



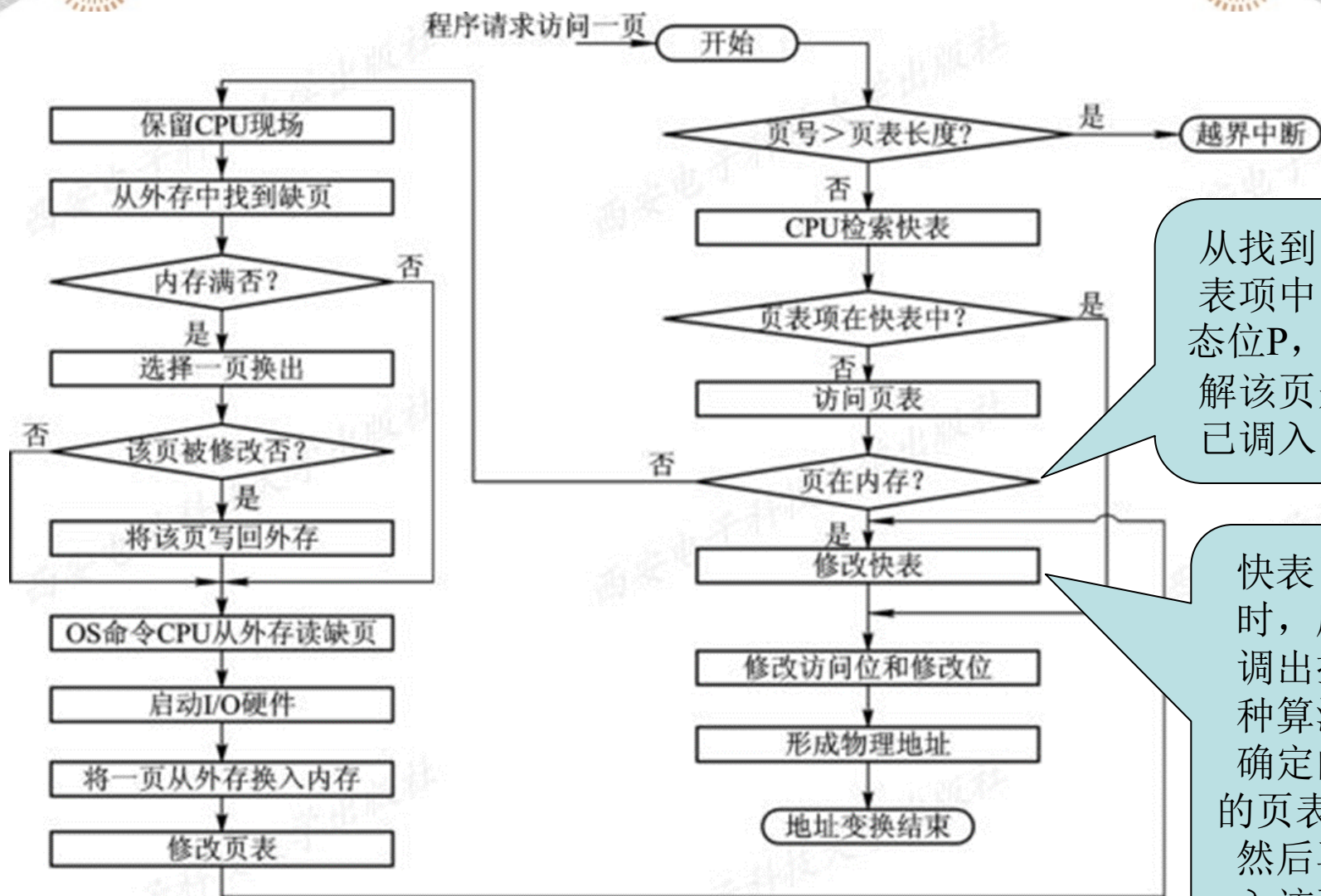


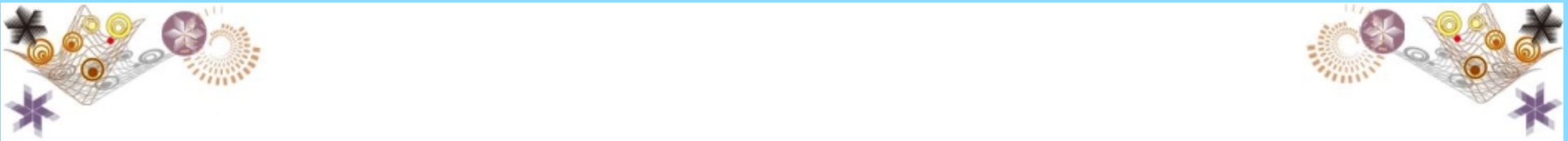
图5-2 请求分页中的地址变换过程



## 5.2.2 请求分页中的内存分配

### 1. 最小物理块数的确定

**最小物理块数**是指**能保证进程正常运行所需的最小物理块数**。当系统为进程分配的物理块数少于此值时，进程将无法运行。进程应获得的最少物理块数与计算机的硬件结构有关，取决于指令的格式、功能和寻址方式。



## 2. 内存分配策略

在请求分页系统中，可采取两种内存分配策略，即**固定和可变分配策略**。

在进行置换时，也可采取两种策略，即**全局置换和局部置换**。于是可组合出以下三种适用的策略。

- 1) 固定分配局部置换(Fixed Allocation, Local Replacement)
- 2) 可变分配全局置换(Variable Allocation, Global Replacement)
- 3) 可变分配局部置换(Variable Allocation, Local Replacement)



## (1) 固定分配局部置换(Fixed Allocation, Local Replacement)

这是指基于进程的类型(交互型或批处理型等), 或根据程序员、程序管理员的建议, 为每个进程分配一定数目的物理块, 在整个运行期间都不再改变。采用该策略时, 如果进程在运行中发现缺页, 则只能从该进程在内存的 $n$ 个页面中选出一个页换出, 然后再调入一页, 以保证分配给该进程的内存空间不变。

实现这种策略的困难在于: 应为每个进程分配多少个物理块难以确定; 太少, 会频繁地出现缺页中断, 降低了系统的吞吐量; 若太多会降低CPU和其它资源的利用率。



## (2) 可变分配全局置换(Variable Allocation, Global Replacement)

这可能是最易于实现的一种物理块分配和置换策略。先为系统中的每个进程分配一定数目的物理块，而OS自身也保持一个空闲物理块队列。当某进程发现缺页时，由OS从空闲物理块队列中取出一个物理块分配给该进程。

仅当空闲物理块队列中的物理块用完时，OS才能从内存中选择一页调出，该页可能是系统中任一进程的页，这样，自然又会使那个进程的物理块减少，进而使其缺页率增加，即一个进程的缺页会影响其他进程。



### (3) 可变分配局部置换(Variable Allocation, Local Replacement)

这同样是基于进程的类型或根据程序员的要求，为每个进程分配一定数目的物理块，但当某进程发现缺页时，只允许从该进程在内存的页面中选出一页换出，这样就不会影响其它进程的运行。



如果进程在运行中频繁地发生缺页中断，则系统须再为该进程分配若干附加的物理块，直至该进程的缺页率减少到适当程度为止；反之，若一个进程在运行过程中的缺页率特别低，则此时可适当减少分配给该进程的物理块数，但不应引起其缺页率的明显增加。



### 3. 物理块分配算法

在采用固定分配策略时，如何将系统中可供分配的所有物理块分配给各个进程，可采用下述几种算法：

(1) 平均分配算法，即将系统中所有可供分配的物理块  
**平均分配**给各个进程。



(2) 按比例分配算法，即根据进程的**大小按比例分配物理块**。

如果系统中共有 $n$ 个进程，每个进程的页面数为 $S_i$ ，则系统中各进程页面数的总和为：

$$S = \sum_{i=1}^n S_i$$

又假定系统中可用的物理块总数为 $m$ ，则每个进程所能分到的物理块数为 $b_i$ 可由下式计算：

$$b_i = \frac{S_i}{S} \times m$$

这里， $b_i$ 应该取整，它必须大于最小物理块数。






### (3) 考虑优先权的分配算法。

在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成，应为它分配较多的内存空间。

通常采取的方法是把内存中可供分配的所有物理块**分成两部分**：一部分**按比例地分配**给各进程；另一部分则根据各进程的**优先权进行分配**，为高优先进程适当地增加其相应份额。

在有的系统中，如重要的实时控制系统，则可能是完全按优先权为各进程分配其物理块的。



### 5.2.3 页面调入策略

为使进程能够正常运行，必须事先将要执行的那部分程序和数据所在的页面调入内存。现在的问题是：

- (1) 系统应在何时调入所需页面；
- (2) 系统应从何处调入这些页面；
- (3) 是如何进行调入的。

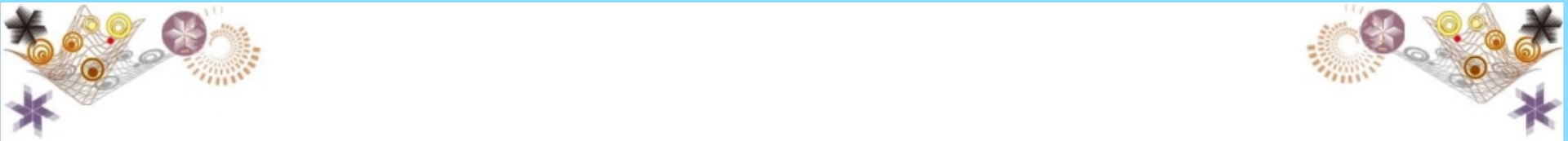


## 1. 何时调入页面

### (1) 预调页策略（访问前调入）

以预测为基础的预调页策略，将那些预计在不久之后便会被访问的页面预先调入内存。

如果预测较准确，那么，这种策略显然是很有吸引力的。但遗憾的是，目前预调页的成功率仅约50%。故这种策略主要用于进程的首次调入时，由程序员指出应该先调入哪些页。



## (2) 请求调页策略（访问时调入）

当进程在运行中需要访问某部分程序和数据时，若发现其所在的页面不在内存，便立即提出请求，由OS将其所需页面调入内存。

由请求调页策略所确定调入的页，是一定会被访问的，再加之请求调页策略比较易于实现，故在目前的虚拟存储器中大多采用此策略。但这种策略每次仅调入一页，故须花费较大的系统开销，增加了磁盘I/O的启动频率。



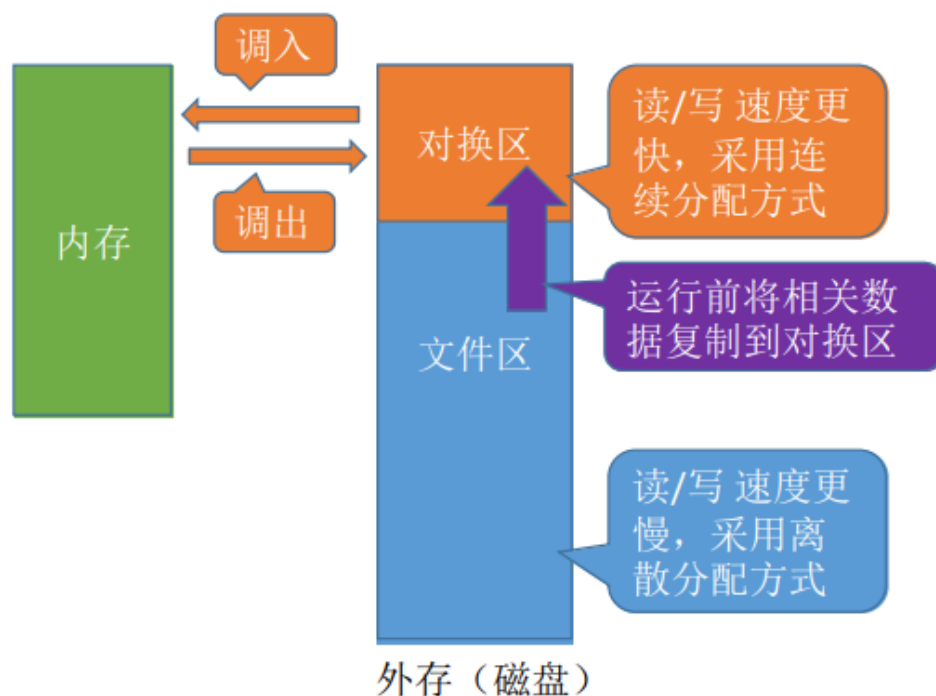
## 2. 从何处调入页面

在请求分页系统中的外存分为两部分：用于存放文件的文件区和用于存放对换页面的对换区。

通常，由于对换区是采用连续分配方式，而文件区是采用离散分配方式，故对换区的磁盘I/O速度比文件区的高。这样，每当发生缺页请求时，系统应从何处将缺页调入内存

(1) 系统拥有足够的对换区空间，这时可以全部从对换区调入所需页面，以提高调页速度。

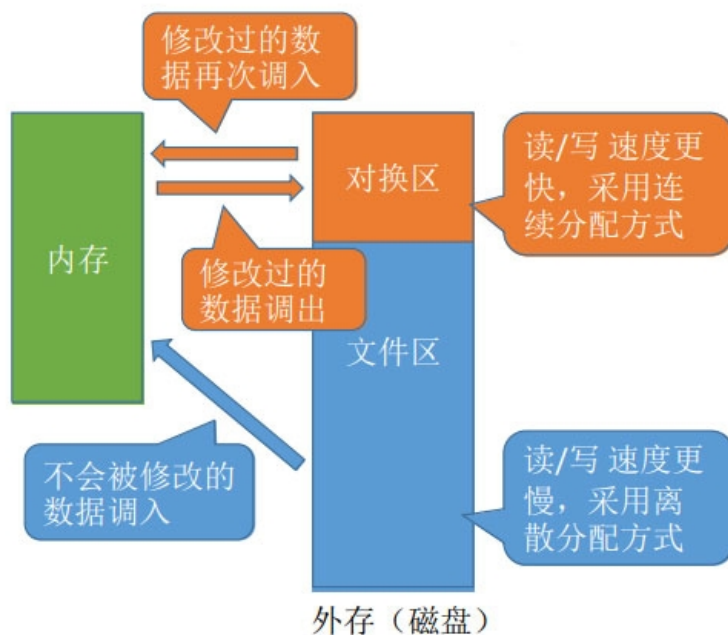
为此，在进程运行前，便须将与该进程有关的文件从文件区拷贝到对换区。



(2) 系统缺少足够的对换区空间，

①不会被修改的页面都直接从文件区调入；而当换出这些页面时，由于它们未被修改而不必再将它们换出，以后再调入时，仍从文件区直接调入。

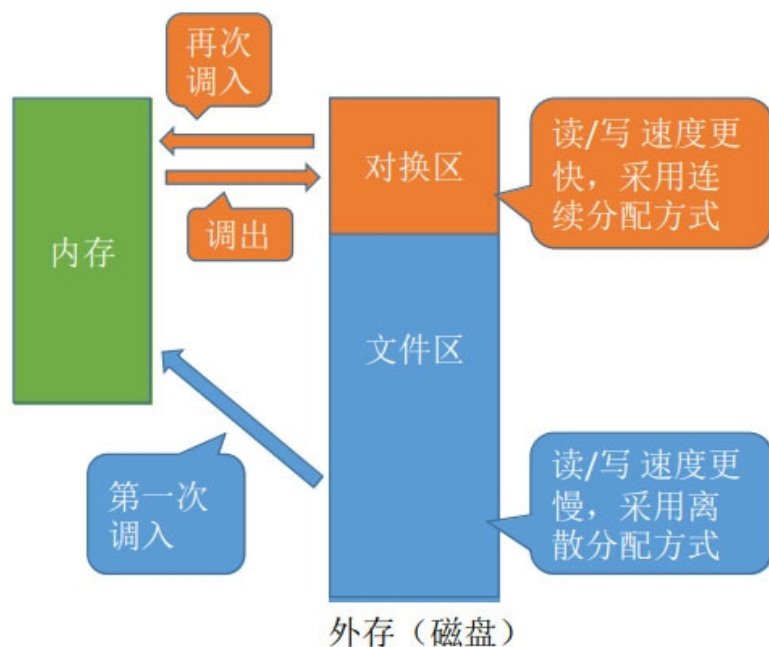
②对于那些可能被修改的部分，在将它们换出时，便须调到对换区，以后需要时，再从对换区调入。





### (3) UNIX方式。

由于与进程有关的文件都放在文件区，故凡是未运行过的页面，都应从文件区调入。而对于曾经运行过但又被换出的页面，由于是被**放在对换区**，因此在下次调入时，应从对换区调入。由于UNIX系统允许页面共享，因此，某进程所请求的页面有可能已被其它进程调入内存，此时也就无须再从对换区调入。



### 3. 页面调入过程

每当程序所要访问的页面未在内存时(存在位为“0”),便向CPU发出一缺页中断,中断处理程序**首先保留CPU环境**,分析中断原因后**转入缺页中断处理程序**。

该程序通过**查找页表**,得到该页在外存的物理块后,如果此时**内存能容纳新页**,则启动磁盘I/O将所缺之页**调入内存**,然后**修改页表**。如果**内存已满**,则须先按照某种置换算法从内存中选出一页准备**换出**;如果该页未被修改过,可不必将该页写回磁盘;但如果此页已被修改,则必须将它写回磁盘,然后再把所缺的页调入内存,并修改页表中的相应表项,置其存在位为“1”,并将此页表项写入快表中。

在缺页调入内存后,利用修改后的页表,去形成所要访问数据的物理地址,再去访问内存数据。

程序请求访问一页

开始

页号 > 页表长度?

是

越界中断

否

CPU检索快表

页表项在快表中?

是

访问页表

页在内存?

否

修改快表

修改访问位和修改位

形成物理地址

地址变换结束

保留CPU现场

从外存中找到缺页

内存满否?

否

是

选择一页换出

该页被修改否?

否

是

将该页写回外存

OS命令CPU从外存读缺页

启动I/O硬件

将一页从外存换入内存



修改页表

页面调入过程

#### 4. 缺页率

假设一个进程的逻辑空间为 $n$ 页，系统为其分配的内存物理块数为 $m(m \leq n)$ 。如果在进程的运行过程中，访问页面成功(即所访问页面在内存中)的次数为 $S$ ，访问页面失败(即所访问页面不在内存中，需要从外存调入)的次数为 $F$ ，则该进程总的页面访问次数为 $A = S + F$ ，那么该进程在其运行过程中的缺页率即为

$$f = \frac{F}{A}$$



事实上，在缺页中断处理时，当由于空间不足，需要置换部分页面到外存时，选择被置换页面还需要考虑到置换的代价，如页面是否被修改过。没有修改过的页面可以直接放弃，而修改过的页面则必须进行保存，所以处理这两种情况时的时间也是不同的。

假设被置换的页面被修改的概率是 $\beta$ ，其缺页中断处理时间为 $t_a$ ，被置换页面没有被修改的缺页中断时间为 $t_b$ ，那么，缺页中断处理时间的计算公式为

$$t = \beta \times t_a + (1 - \beta) \times t_b$$

## 5.3 页面置换算法

在进程运行过程中，若其所要访问的页面不在内存，而需把它们调入内存，但内存已无空闲空间时，为了保证该进程能正常运行，系统必须从内存中调出一页程序或数据送到磁盘的对换区中。但**应将哪个页面调出，须根据一定的算法来确定**。通常，把选择换出页面的算法称为页面置换算法 (Page-Replacement Algorithms)。置换算法的好坏将直接影响到系统的性能。



### 5.3.1 最佳置换算法和先进先出置换算法

#### 1. 最佳 (Optimal) 置换算法

最佳置换算法是由Belady于1966年提出的一种理论上的算法。其所选择的被淘汰页面，将是以后永不使用的，或许是在最长**(未来)时间内不再被访问的页面。**

采用最佳置换算法，通常可保证获得最低的缺页率。

但由于人们目前还无法预知一个进程在内存的若干个页面中，哪一个页面是未来最长时间不再被访问的，因而该算法是无法实现的，但可以利用该算法去评价其它算法。



假定系统为某进程分配了三个物理块，并考虑有以下的页面号引用串：

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1																				
内存块2																				
内存块3																				
是否缺页																				

图5-3 利用最佳页面置换算法时的置换图



访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	→																
内存块2		0	0																	
内存块3			1																	
是否缺页	√	√	√																	

图5-3 利用最佳页面置换算法时的置换图





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2																
内存块2		0	0	0																
内存块3			1	1																
是否缺页	√	√	√	√																

图5-3 利用最佳页面置换算法时的置换图





↓

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2														
内存块2		0	0	0		0	→													
内存块3			1	1		3														
是否缺页	√	√	√	√		√														

图5-3 利用最佳页面置换算法时的置换图





↓

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		2												
内存块2		0	0	0		0		4	→											
内存块3			1	1		3		3												
是否缺页	√	√	√	√		√		√												

图5-3 利用最佳页面置换算法时的置换图






↓

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		2			2									
内存块2		0	0	0		0		4			0									
内存块3			1	1		3		3			3									
是否缺页	√	√	√	√		√		√			√									

图5-3 利用最佳页面置换算法时的置换图





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		2			2			2						
内存块2		0	0	0		0		4			0			0						
内存块3			1	1		3		3			3			1						
是否缺页	√	√	√	√		√		√			√			√						

图5-3 利用最佳页面置换算法时的置换图





共产生6次置换

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		2			2			2				7		
内存块2		0	0	0		0		4			0			0				0		
内存块3			1	1		3		3			3			1				1		
是否缺页	√	√	√	√		√		√			√			√				√		

图5-3 利用最佳页面置换算法时的置换图

## 2. 先进先出 (FIFO) 页面置换算法

该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。实现简单，只需把一个进程已调入内存的页面，按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。

但该算法与进程实际运行的规律不相适应，因为在进程中，有些页面经常被访问，比如，含有全局变量、常用函数、例程等的页面，FIFO算法并不能保证这些页面不被淘汰。



访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7 →																	
内存块2		0	0																	
内存块3			1																	
是否缺页	√	√	√																	

图5-4 利用FIFO置换算法时的置换图





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2																
内存块2		0	0	0																
内存块3			1	1																
是否缺页	√	√	√	√																

图5-4 利用FIFO置换算法时的置换图





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2														
内存块2		0	0	0		3														
内存块3			1	1		1	→													
是否缺页	√	√	√	√		√														

图5-4 利用FIFO置换算法时的置换图



同理，可以得到后续置换

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2	2													
内存块2		0	0	0		3	3													
内存块3			1	1		2	0													
是否缺页	√	√	√	√		√	√													

图5-4 利用FIFO置换算法时的置换图

共产生12次置换

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
内存块2		0	0	0		3	3	3	2	2	2			1	1			1	0	0
内存块3			1	1		2	0	0	0	3	3			3	2			2	2	1
是否缺页	√	√	√	√		√	√	√	√	√	√			√	√			√	√	√

图5-4 利用FIFO置换算法时的置换图



### 5.3.2 最近最久未使用和最少使用置换算法

#### 1. LRU (Least Recently Used) 置换算法的描述

最近最久未使用(LRU)的页面置换算法，是**根据页面调入内存后的使用情况进行决策的。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似**，因此，LRU置换算法是选择**最近最久未使用的页面**予以淘汰。

该算法赋予每个页面一个**访问字段**，用来记录一个页面自上次被访问以来所经历的时间 $t$ ，当须淘汰一个页面时，选择现有页面中其 $t$ 值最大的，即最近最久未使用的页面予以淘汰。



访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7 →																	
内存块2		0	0																	
内存块3			1																	
是否缺页	√	√	√																	

图5-5 LRU页面置换算法





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2																
内存块2		0	0	0																
内存块3			1	1																
是否缺页	√	√	√	√																

图5-5 LRU页面置换算法





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2														
内存块2		0	0	0		0														
内存块3			1	1		3														
是否缺页	√	√	√	√		√														

图5-5 LRU页面置换算法





访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		4												
内存块2		0	0	0		0		0												
内存块3			1	1		3		3												
是否缺页	√	√	√	√		√		√												

图5-5 LRU页面置换算法



同理，可以得到后续置换

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		4	4											
内存块2		0	0	0		0		0	0											
内存块3			1	1		3		3	2											
是否缺页	√	√	√	√		√		√	√											

图5-5 LRU页面置换算法

共产生9次置换

访问页面	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
内存块1	7	7	7	2		2		4	4	4	0			1		1		1		
内存块2		0	0	0		0		0	0	3	3			3		0		0		
内存块3			1	1		3		3	2	2	2			2		2		7		
是否缺页	√	√	√	√		√		√	√	√	√			√		√		√		

图5-5 LRU页面置换算法



## 2. LRU置换算法的硬件支持

LRU置换算法须有两类硬件之一的支持：**寄存器或栈**。

### 1) 寄存器

为了记录某进程在内存中各页的使用情况，须为每个在内存中的页面配置一个移位寄存器，可表示为

$$R = R_{n-1}R_{n-2}R_{n-3} \dots R_2R_1R_0$$




当进程访问某物理块时，要将相应寄存器的 $R_{n-1}$ 位置成1。此时，定时信号将每隔一定时间(例如100 ms)将寄存器右移一位。如果我们把 $n$ 位寄存器的数看作是一个整数，那么，具有最小数值的寄存器所对应的页面，就是最近最久未使用的页面。



800ms内访问次数

实 页 \ R	R <sub>7</sub>	R <sub>6</sub>	R <sub>5</sub>	R <sub>4</sub>	R <sub>3</sub>	R <sub>2</sub>	R <sub>1</sub>	R <sub>0</sub>	
1	0	1	0	1	0	0	1	0	3
2	1	0	1	0	1	1	0	0	4
3	0	0	0	0	0	1	0	0	1
4	0	1	1	0	1	0	1	1	5
5	1	1	0	1	0	1	1	0	5
6	0	0	1	0	1	0	1	1	4
7	0	0	0	0	0	1	1	1	3
8	0	1	1	0	1	1	0	1	5



图5-6 某进程具有8个页面时的LRU访问情况



## 2) 栈

可利用一个特殊的栈保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶始终是最新被访问页面的编号，而栈底则是最近最久未使用页面的页面号。假定现有一进程，它分有五个物理块，所访问的页面的页面号序列为：

4, 7, 0, 7, 1, 0, 1, 2, 1, 2, 6

4	7	0	7	1	0	1	2	1	2	6
							2	1	2	6
				1	0	1	1	2	1	2
		0	7	7	1	0	0	0	0	1
	7	7	0	0	7	7	7	7	7	0
4	4	4	4	4	4	4	4	4	4	7

图5-7 用栈保存当前使用页面时栈的变化情况



### 3. 最少使用 (Least Frequently Used, LFU) 置换算法

在采用LFU算法时，应为在内存中的每个页面设置一个移位寄存器，用来记录该页面被访问的频率。该置换算法选择在最近时期使用最少的页面作为淘汰页。

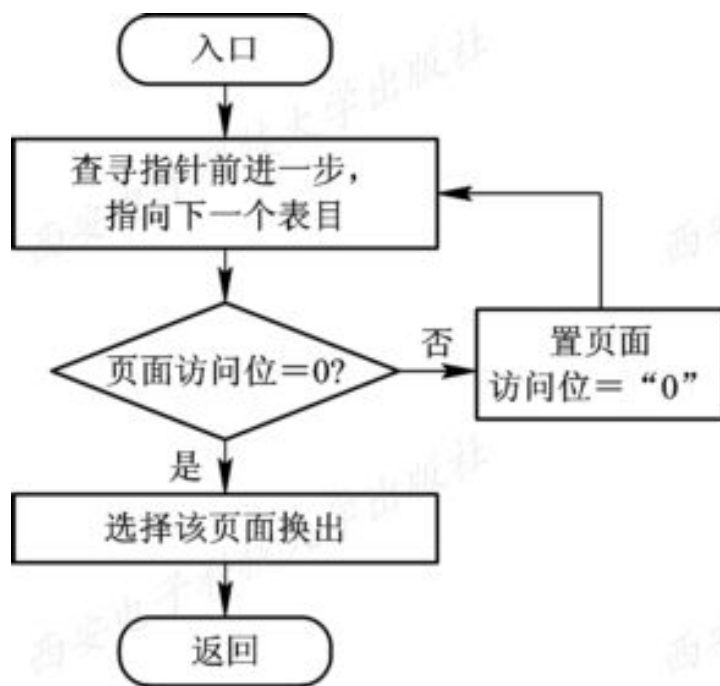
LFU置换算法的页面访问图，与LRU置换算法的访问图完全相同。

### 5.3.3 Clock置换算法

#### 1. 简单的Clock置换算法

当利用简单Clock算法时，只需为每页设置一位访问位，再将内存中的所有页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位被置1。

置换算法在选择一页淘汰时，只需检查页的访问位。如果是0，就选择该页换出；若为1，则重新将它置0，暂不换出，而给该页第二次驻留内存的机会，再按照FIFO算法检查下一个页面。当检查到队列中的最后一个页面时，若其访问位仍为1，则再返回到队首去检查第一个页面。



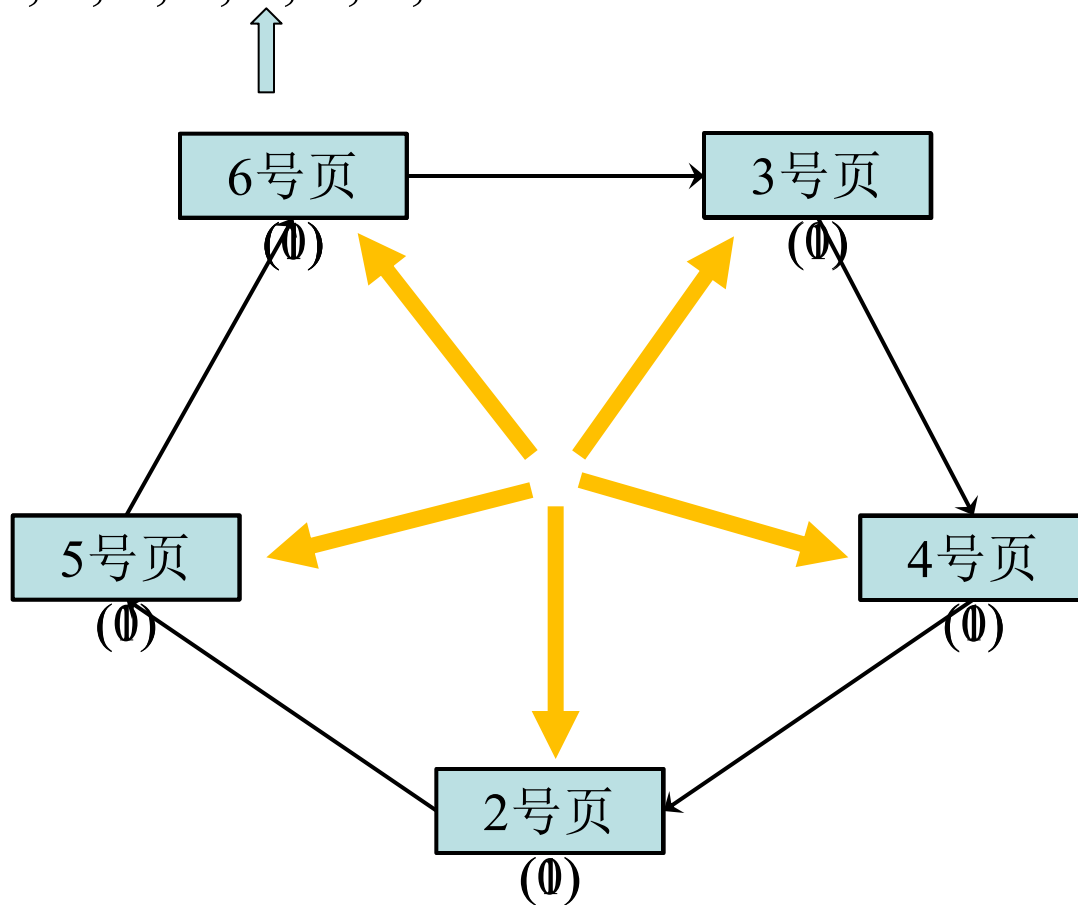
块号	页号	访问位	指针
0			
1			
2	4	0	
3			
4	2	1	
5			
6	5	0	
7	1	1	

替换  
指针

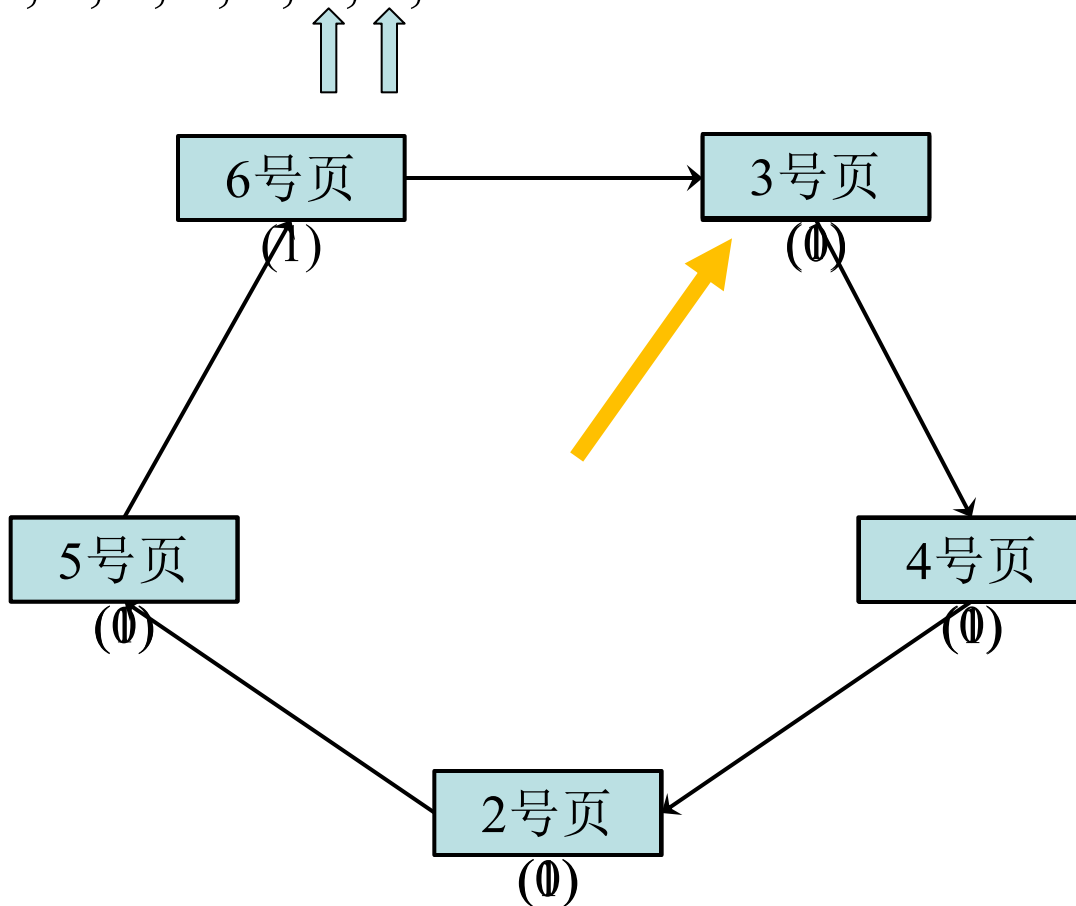
图5-8 简单Clock置换算法的流程和示例



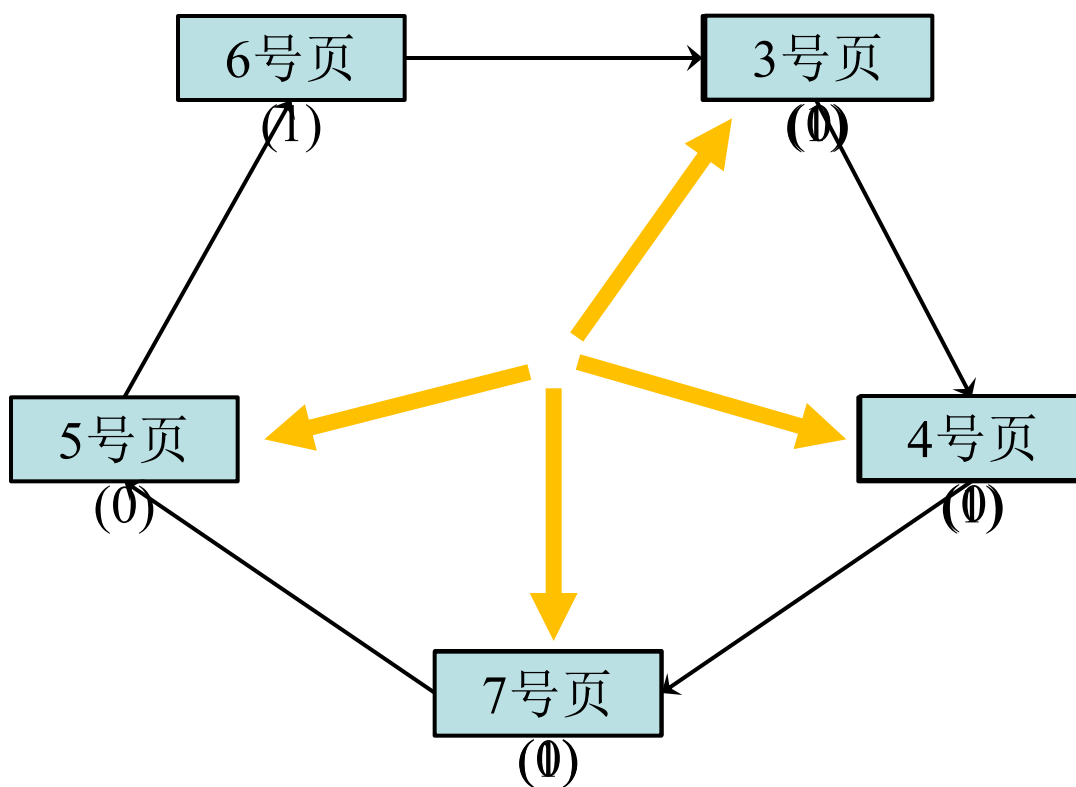
例：假设某进程分配了五个内存块，并考虑到如下的页面引用串：1, 3, 4, 2, 5, 6, 3, 4, 7



例：假设某进程分配了五个内存块，并考虑到如下的页面引用串：1, 3, 4, 2, 5, 6, 3, 4, 7



例：假设某进程分配了五个内存块，并考虑到如下的页面引用串：1, 3, 4, 2, 5, 6, 3, 4, 7





Clock置换该算法只有一位访问位，只能用它表示该页是否已经使用过，而置换时是将未使用过的页面换出去，故又把该算法称为**最近未用算法NRU**(Not Recently Used)。





## 2. 改进型Clock置换算法

在将一个页面换出时，如果该页已被修改过，便须将该页重新写回到磁盘上；但如果该页未被修改过，则不必将它拷回磁盘。换言之，对于修改过的页面，在换出时所付出的开销比未修改过的页面大，或者说，置换代价大。在改进型Clock算法中，除须考虑页面的使用情况外，还须再增加一个因素——**置换代价**。

选择页面换出时，既要是**未使用过的页面**，又要是**未被修改过的页面**。把同时满足这两个条件的页面作为首选淘汰的页面。



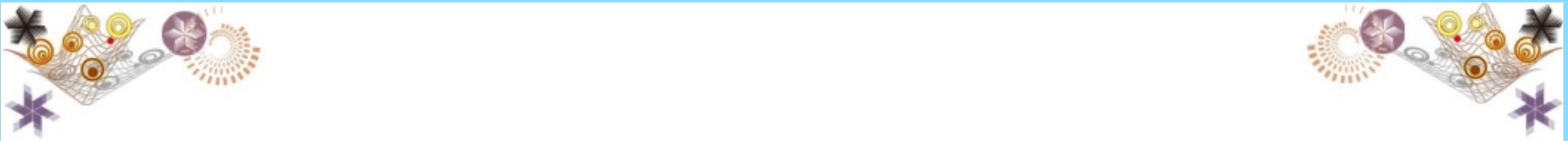
由访问位A和修改位M可以组合成下面四种类型的页面：

1类( $A=0$ ,  $M=0$ ): 表示该页最近既未被访问, 又未被修改, 是最佳淘汰页。

2类( $A=0$ ,  $M=1$ ): 表示该页最近未被访问, 但已被修改, 并不是很好的淘汰页。

3类( $A=1$ ,  $M=0$ ): 表示该页最近已被访问, 但未被修改, 该页有可能再被访问。



4类( $A=1$ ,  $M=1$ ): 表示该页最近已被访问且被修改, 该页可能再被访问。



采用与简单Clock算法相类似的算法，其差别在于该算法须同时检查访问位与修改位，以确定该页是四类页面中的哪一种。其执行过程可分成以下三步：

(1) 从指针所指示的当前位置开始，扫描循环队列，**寻找 $A=0$ 且 $M=0$** 的第一类页面，将所遇到的第一个页面作为所选中的淘汰页。在第一次扫描期间**不改变访问位 $A$** 。

(2) 如果第一步失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，**寻找 $A=0$ 且 $M=1$** 的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将**所有扫描过的页面的访问位都置0**。



(3) 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复0。然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页。

该算法与简单Clock算法比较，可减少磁盘的I/O操作次数。但为了找到一个可置换的页，可能须经过几轮扫描。换言之，实现该算法本身的开销将有所增加。





### 5.3.4 页面缓冲算法 (Page Buffering Algorithm, PBA)

#### 1. 影响页面换进换出效率的若干因素

- (1) 页面置换算法。
- (2) 写回磁盘的频率。
- (3) 读入内存的频率。

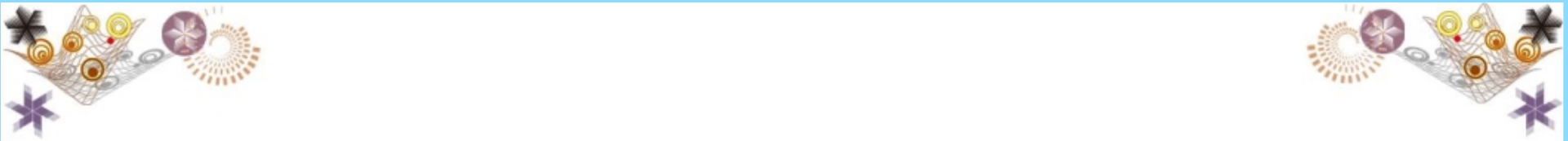


## 2. 页面缓冲算法PBA

PBA算法的主要特点是：

① 显著地降低了页面换进、换出的频率，使磁盘I/O的操作次数大为减少，因而减少了页面换进、换出的开销；

② **正是由于**换入换出的开销大幅度减小，才能使其采用一种较简单的置换策略，如先进先出(FIFO)算法，它不需要特殊硬件的支持，实现起来非常简单。



VAX/VMS操作系统中，内存分配策略采用了可变分配和局部置换方式，系统为每一个进程分配一定数目的物理块，系统自己保留一部分空闲物理块。为了能显著地降低页面换进、换出的频率，在内存中设置了两个链表：

1) 空闲页面链表：一个空闲物理块链表，其中的每个物理块都是空闲的。当需要读入一个页面时，可利用空闲物理块链表中的第一个物理块来装入该页。当有一个未被修改的页要换出时，实际上并不将它换出内存，而是把该未被修改的页所在的物理块挂在自由页链表的末尾。

2) 修改页面链表：由已修改的页面所形成的链表。设置该链表的目的是为了减少已修改页面换出的次数。当进程需要将一个已修改的页面换出时，系统并不立即把它换出到外存上，而是将它所在的物理块挂在修改页面链表的末尾。

### 5.3.5 访问内存的有效时间

与基本分页存储管理方式不同，在请求分页管理方式中，内存有效访问时间不仅要考虑访问页表和访问实际物理地址数据的时间，还必须还要考虑到缺页中断的处理时间。



在具有快表机制的请求分页管理方式中，存在三种方式的内存访问操作，其有效访问时间的计算公式也有所不同。



① **被访问页在内存中**，其对应的页表项在快表中。

设内存的有效访问时间为EAT，查找快表时间为 $\lambda$ ，访问实际物理地址所用的时间为 $t$ ，则：

$$EAT = \lambda + t$$





② 被访问页在内存中，其对应的页表项不在快表中。

需要两次访问内存，一次读取页表，一次读取数据，另外还需要更新快表。

内存的有效访问时间分为：查找快表的时间，查找页表的时间，修改快表的时间和访问实际物理地址的时间。

$$EAT = \lambda + t + \lambda + t = 2 \times (\lambda + t)$$





③ 被访问页不在内存中。

需要进行缺页中断处理。

内存的有效访问时间分为：查找快表的时间，查找页表的时间，处理缺页中断的时间，更新快表的时间和访问实际物理地址的时间。假设中断处理时间为 $\varepsilon$ ，则

$$EAT = \lambda + t + \varepsilon + \lambda + t = \varepsilon + 2 \times (\lambda + t)$$





加入命中率  $a$  和缺页率  $f$  后，内存的有效访问时间的计算公式为：

$$EAT = \lambda + a \times t + (1 - a) \times [t + f \times (\epsilon + \lambda + t) + (1 - f) \times (\lambda + t)]$$

如果不考虑命中率，仅考虑缺页率，设缺页中断处理时间为  $\phi$ ，可得：

$$EAT = t + f \times (\phi + t) + (1 - f) \times t$$





## 5.4 “抖动”与工作集

由于请求分页式虚拟存储器系统的性能优越，在正常运行情况下，它能有效地减少内存碎片，提高处理机的利用率和吞吐量，故是目前最常用的一种系统。

但如果在系统中运行的进程太多，进程在运行中会**频繁地发生缺页情况**，这又会对系统的性能产生很大的影响，故还须对请求分页系统的性能做简单的分析。



## 5.4.1 多道程序度与“抖动”

### 1. 多道程序度与处理机的利用率

由于虚拟存储器系统能从逻辑上扩大内存，这时，只需装入一个进程的部分程序和数据便可开始运行，故人们希望在系统中能运行更多的进程，即**增加多道程序度**，以提高处理机的利用率。

但处理机的实际利用率却如图5-9中的实线所示。

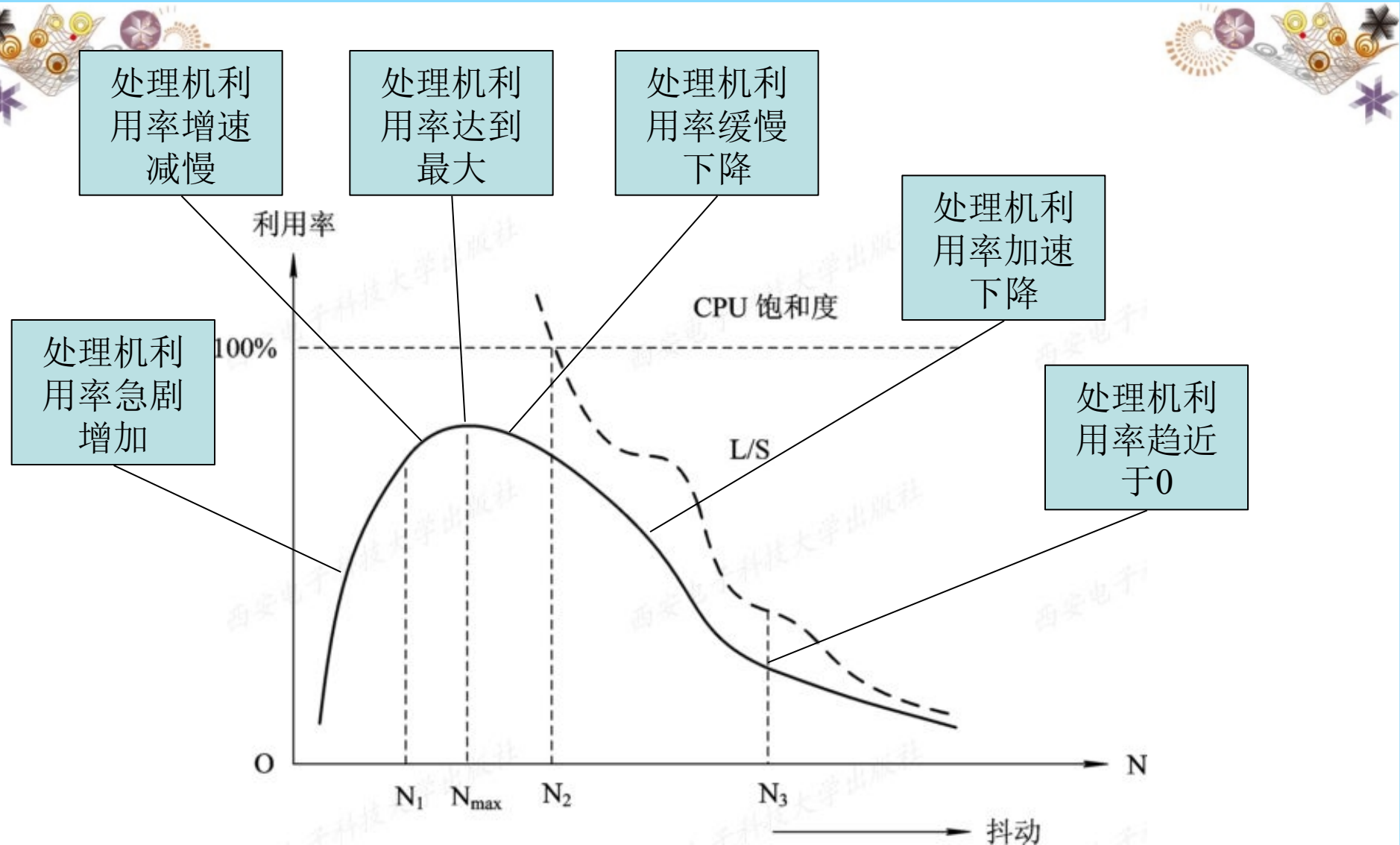



图5-9 处理机的利用率



## 2. 产生“抖动”的原因

发生“抖动”的根本原因是，同时在系统中运行的进程太多，由此分配给每一个进程的物理块太少，不能满足进程正常运行的基本要求，致使每个进程在运行时，频繁地出现缺页，必须请求系统将所缺之页调入内存。

这会使得在系统中排队等待页面调进/调出的进程数目增加。显然，对磁盘的有效访问时间也随之急剧增加，造成每个进程的大部分时间都用于页面的换进/换出，而几乎不能再去做任何有效的工作，从而导致发生处理机的利用率急剧下降并趋于0的情况。我们称此时的进程是处于“抖动”状态。

## 5.4.2 工作集

### 1. 工作集的基本概念

进程发生缺页率的时间间隔与进程所获得的物理块数有关。图5-10示出了缺页率与物理块数之间的关系。

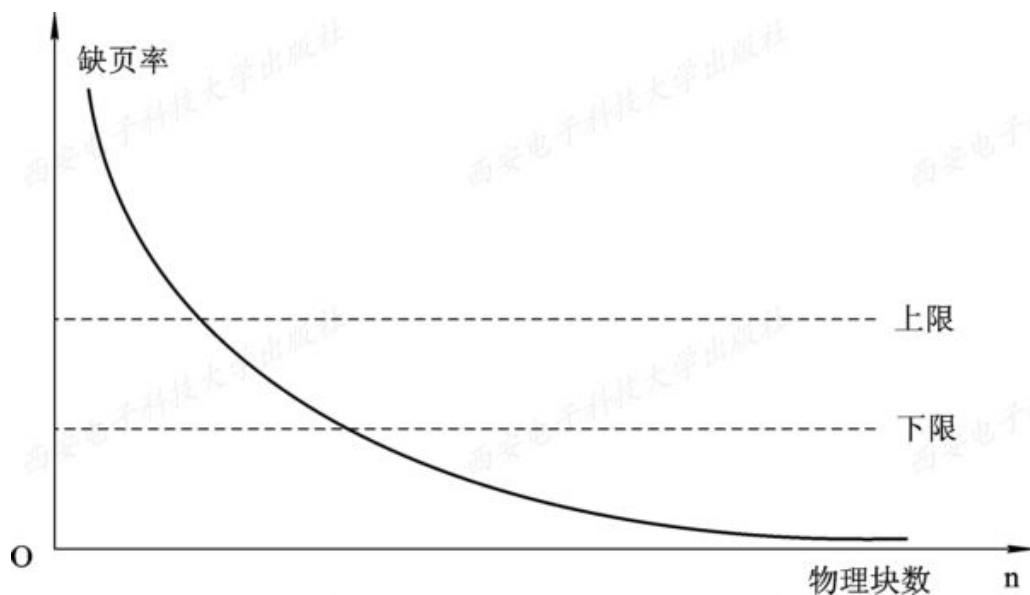


图5-10 缺页率与物理块数之间的关系



## 2. 工作集的定义

所谓工作集，是指**在某段时间间隔 $\Delta$ 里，进程实际所要访问页面的集合。**

Denning指出，虽然程序只需要少量的几页在内存便可运行，但为了较少地产生缺页，**应将程序的全部工作集装入内存中。**



然而我们**无法事先预知**程序在不同时刻将访问哪些页面，故仍只有像置换算法那样，用**程序的过去某段时间内的行为**作为程序在将来某段时间内行为的近似。

把进程在时间 $t$ 的工作集记为 $w(t, \Delta)$ ,  $\Delta$ 称为工作集的窗口尺寸。工作集也可以定义为, 进程在时间间隔 $(t - \Delta, t)$ 中引用的页面的集合。

引用页序列	窗口大小		
	3	4	5
24	24	24	24
15	15 24	15 24	15 24
18	18 15 24	18 15 24	18 15 24
23	23 18 15	23 18 15 24	23 18 15 24
24	24 23 18	—	—
17	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17 24	—	—
24	—	—	—
18	—	—	—
17	—	—	—
17	—	—	—
15	15 17 18	15 17 18 24	—
24	24 15 17	—	—
17	—	—	—
24	—	—	—
18	18 24 17	—	—

图5-11 窗口为3、4、5时进程的工作集





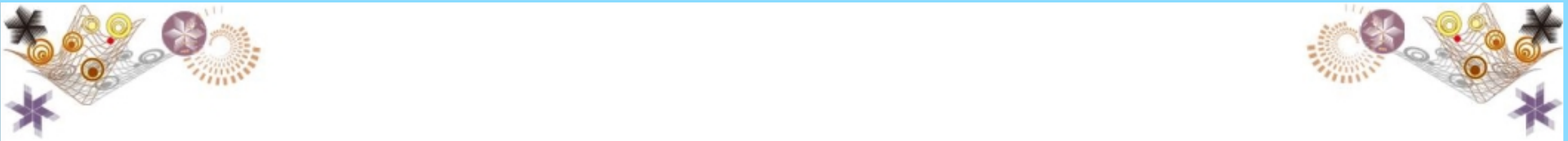
工作集 $w(t, \Delta)$ 是一个二元函数，即在不同时间 $t$ 的工作集大小不同，所含页面数也不同：工作集与窗口尺寸 $\Delta$ 有关，是窗口尺寸 $\Delta$ 的非降函数，即

$$w(t, \Delta) \subseteq w(t, \Delta+1)$$

工作集的大小可能小于窗口尺寸。实际应用中，操作系统可以统计进程的工作集大小，根据工作集大小给进程分配若干内存块。

例如：窗口尺寸为5，经过一段时间的监测发现某进程的工作集最大为3，那么给这个进程分配3个以上的内存块即可满足的运行需要。





### 5.4.3 “抖动”的预防方法

#### 1. 采取局部置换策略

在页面分配和置换策略中，如果采取的是可变分配方式，则为了预防发生“抖动”，可采取局部置换策略。

根据这种策略，当某进程发生缺页时，只能在分配给**自己的内存空间**内进行置换，不允许从其它进程去获得新的物理块。

该方法虽然简单易行，但是效果不是很好。因为在某进程发生“抖动”后，它还会长期处在磁盘I/O的等待队列中，使得队列的长度增加，这回延长其他进程缺页中断的处理时间。

## 2. 把工作集算法融入到处理机调度中

当调度程序发现处理机利用率低下时，它将试图从外存调入一个新作业进入内存，来改善处理机的利用率。

如果在调度中融入了工作集算法，在调度程序从外存调入作业之前，必须先检查每个进程在内存的驻留页面是否足够多。

如果都已足够多，此时便可以从外存调入新的作业，不会因新作业的调入而导致缺页率的增加；反之，如果有些进程的内存页面不足，则应首先为那些缺页率居高的作业增加新的物理块，此时将不再调入新的作业。



### 3. 利用“ $L=S$ ”准则调节缺页率

Denning于1980年提出了“ $L=S$ ”的准则来调节多道程序度，其中 $L$ 是缺页之间的平均时间， $S$ 是平均缺页服务时间，即用于置换一个页面所需的时间。

如果是 $L$ 远比 $S$ 大，说明很少发生缺页，磁盘的能力尚未得到充分的利用；反之，如果是 $L$ 比 $S$ 小，则说明频繁发生缺页，缺页的速度已超过磁盘的处理能力。

只有当 $L$ 与 $S$ 接近时，磁盘和处理机都可达到它们的最大利用率。理论和实践都已证明，利用“ $L=S$ ”准则，对于调节缺页率是十分有效的。

#### 4. 选择暂停的进程

当多道程序度偏高时，已影响到处理机的利用率，为了防止发生“抖动”，系统必须减少多道程序的数目。

此时应基于某种原则选择暂停某些当前活动的进程，将它们调出到磁盘上，以便把腾出的内存空间分配给缺页率发生偏高的进程。



## 5.5 请求分段存储管理方式

### 5.5.1 请求分段中的硬件支持

为了实现请求分段式存储管理，应在系统中配置多种硬件机构，以支持快速地完成请求分段功能。与请求分页系统相似，在请求分段系统中所需的硬件支持有段表机制、缺段中断机构，以及地址变换机构。

## 1. 请求段表机制

在请求分段式管理所需的主要数据结构是请求段表。在该表中除了具有请求分页机制中有的访问字段A、修改位M、存在位P和外存始址四个字段外，还增加了存取方式字段和增补位。这些字段供程序在调进、调出时参考。

段名	段长	段基址	存取方式	访问字段 A	修改位 M	存在位 P	增补位	外存始址
----	----	-----	------	--------	-------	-------	-----	------



(1) 存取方式：用于标识本分段的存取属性是只执行、只读，还是允许读/写。

(2) 访问字段A：其含义与请求分页的相应字段相同，用于记录该段被访问的频繁程度。

(3) 修改位M：用于表示该页在进入内存后是否已被修改过，供置换页面时参考。

(4) 存在位P：指示本段是否已调入内存，供程序访问时参考。

(5) 增补位：这是请求分段式管理中所特有的字段，**用于表示本段在运行过程中是否做过动态增长。**

(6) 外存始址：指示本段在外存中的起始地址，即起始盘块号。





## 2. 缺段中断机构

在请求分段系统中采用的是请求调段策略。每当发现运行进程所要访问的段尚未调入内存时，便由缺段中断机构产生一**缺段中断**信号，进入OS后，由缺段中断处理程序将所需的段调入内存。

与缺页中断机构类似，缺段中断机构同样需要**在一条指令的执行期间产生和处理中断**，以及在一条指令执行期间，可能产生**多次缺段中断**。但由于分段是信息的逻辑单位，因而不可能出现一条指令被分割在两个分段中，和一组信息被分割在两个分段中的情况。



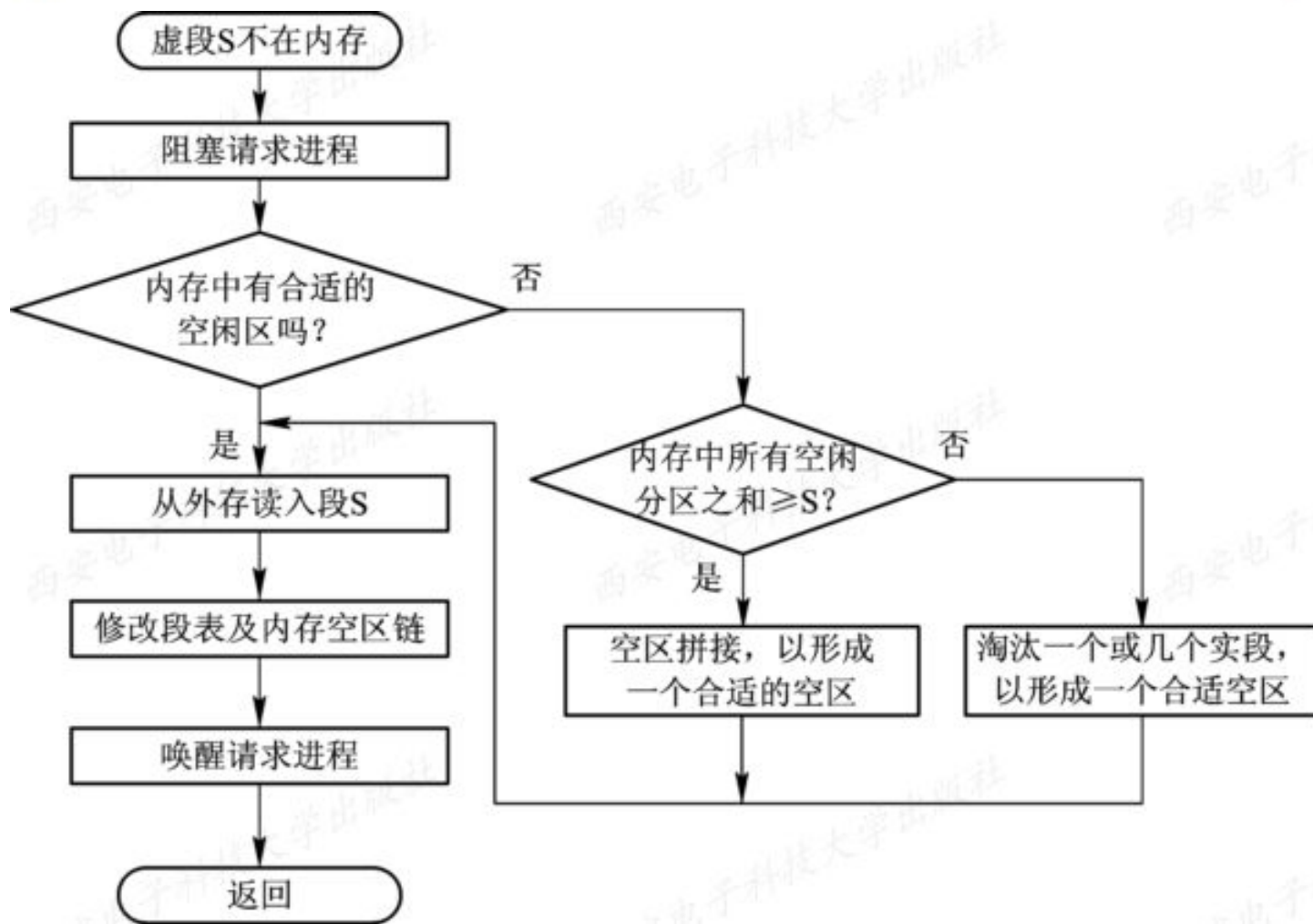


图5-12 请求分段系统中的中断处理过程

### 3. 地址变换机构

在地址变换时，若发现所要访问的段不在内存，必须**先**将所缺的段调入内存，并**修改段表**，然后才能再利用段表进行地址变换。为此，在地址变换机构中又增加了某些功能，如缺段中断的请求及处理等。

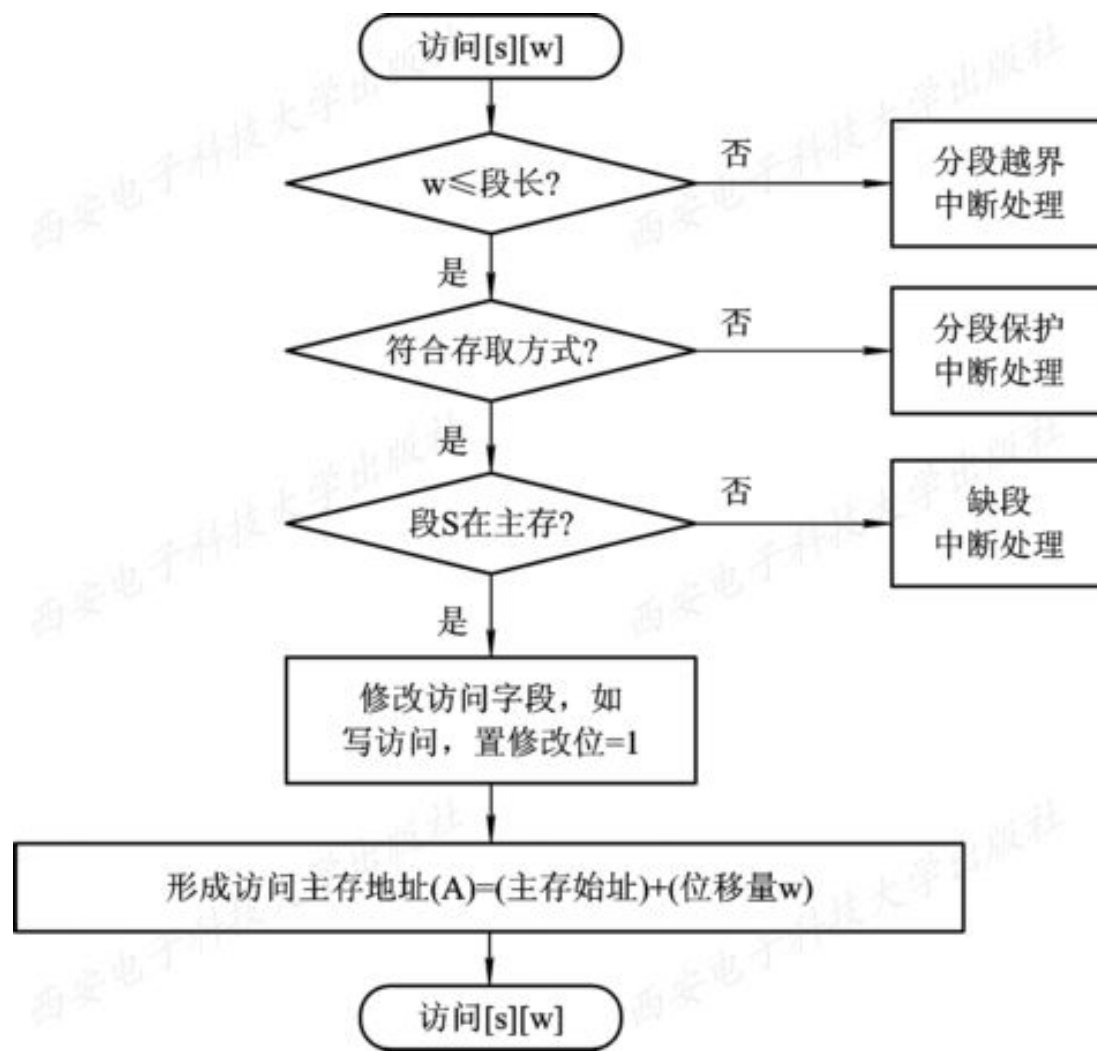
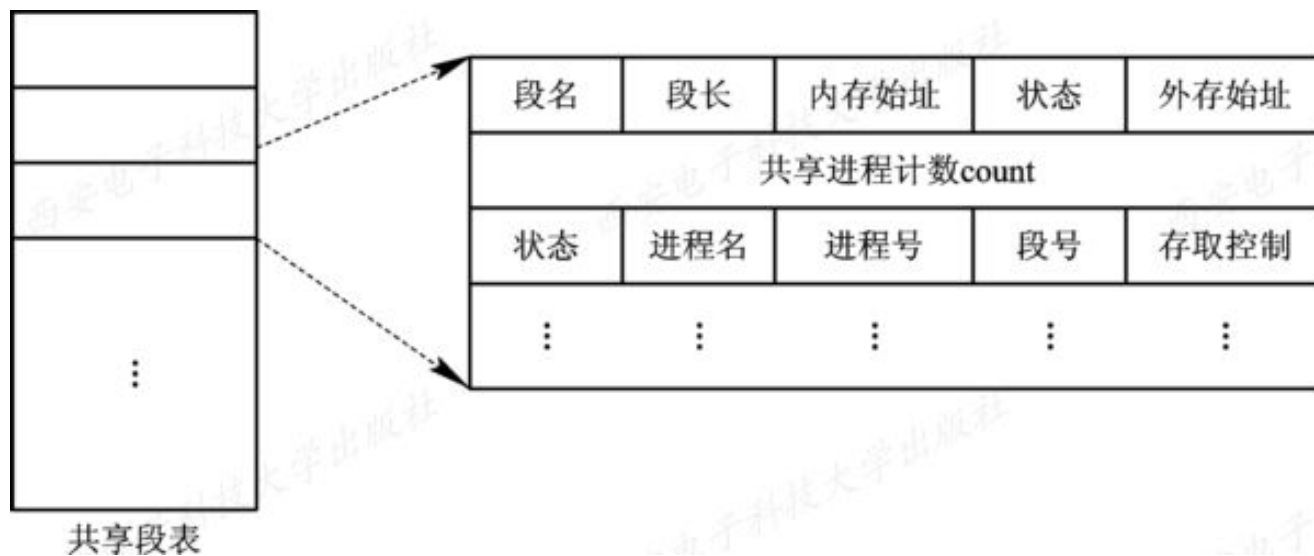


图5-13 请求分段系统的地址变换过程

## 5.5.2 分段的共享与保护

### 1. 共享段表

为了实现分段共享，可在系统中配置一张共享段表，所有各共享段都在共享段表中占有一表项。表项中记录了共享段的段号、段长、内存始址、存在位等信息，并记录了共享此分段的每个进程的情况。





### (1) 共享进程计数count。

共享段是为多个进程所需要的，当某进程不再需要而释放它时，系统并不回收该段所占内存区，仅当所有共享该段的进程全都不再需要它时，才由系统回收该段所占内存区。

为了记录有多少个进程需要共享该分段，特设置了一个整型变量count。



## (2) 存取控制字段。

对于一个共享段，应给不同的进程以不同的存取权限。  
例如，对于文件主，通常允许他读和写；而对其它进程，则可能只允许读，甚至只允许执行。

## (3) 段号。

对于一个共享段，不同的进程可以各用不同的段号去共享该段。

## 2. 共享段的分配与回收

### 1) 共享段的分配

在为共享段分配内存时，对第一个请求使用该共享段的进程，由系统为该共享段分配一物理区，再把共享段调入该区，同时将该区的始址填入请求进程的段表的相应项中，还须在共享段表中增加一表项，填写有关数据，把count置为1。

当又有其它进程需要调用该共享段时，由于该共享段已被调入内存，故此时无须再为该段分配内存，而只需在调用进程的段表中增加一表项，填写该共享段的物理地址；在共享段的段表中，填上调用进程的进程名、存取控制等，再执行 $\text{count} := \text{count} + 1$ 操作，以表明有两个进程共享该段。



## 2) 共享段的回收

当共享此段的某进程不再需要该段时，应将该段释放，包括撤消在该进程段表中共享段所对应的表项，以及执行  $\text{count} := \text{count} - 1$  操作。

若结果为0，则须由系统回收该共享段的物理内存，以及取消在共享段表中该段所对应的表项，表明此时已没有进程使用该段；否则(减1结果不为0)，只是取消调用者进程在共享段表中的有关记录。



### 3. 分段保护

#### 1) 越界检查

在段表寄存器中放有段表长度信息；同样，在段表中也为每个段设置有段长字段。

在进行存储访问时，首先将逻辑地址空间的段号与段表长度进行比较，如果段号等于或大于段表长度，将发出地址越界中断信号；其次，还要检查段内地址是否等于或大于段长，若大于段长，将产生地址越界中断信号，从而保证了每个进程只能在自己的地址空间内运行。



## 2) 存取控制检查

在段表的每个表项中，都设置了一个“**存取控制**”字段，用于规定对该段的访问方式。通常的访问方式有：

(1) **只读**，即只允许进程对该段中的程序或数据进行读访问。

(2) **只执行**，即只允许进程调用该段去执行，但不准读该段的内容，也不允许对该段执行写操作。

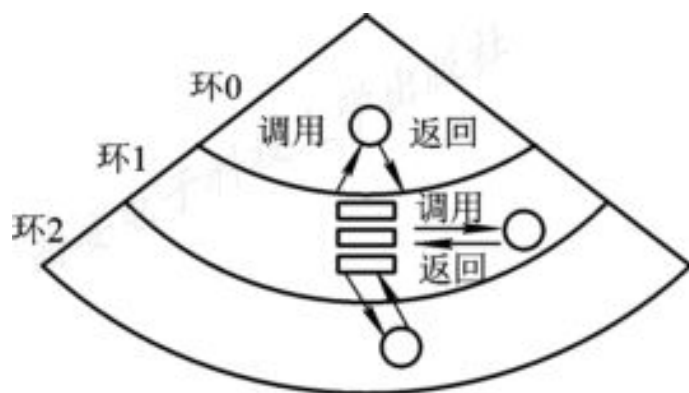
(3) **读/写**，即允许进程对该段进行读/写访问。



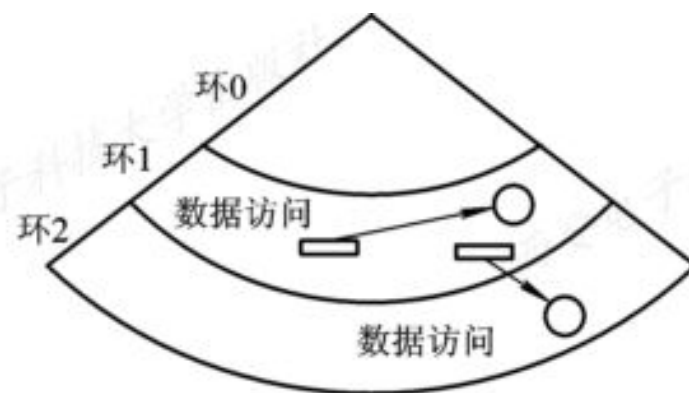
### 3) 环保护机构

这是一种功能较完善的保护机制。在该机制中规定：低编号的环具有高优先权。OS核心处于0环内；某些重要的实用程序和操作系统服务占居中间环；而一般的应用程序则被安排在外环上。在环系统中，程序的访问和调用应遵循以下规则：

- (1) 一个程序可以访问驻留在相同环或较低特权环中的数据。
- (2) 一个程序可以调用驻留在相同环或较高特权环中的服务。



(a) 程序间的控制传输



(b) 数据访问

图5-15 环保护机构