

第五章 虚拟存储器



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



第五章 虚拟存储器

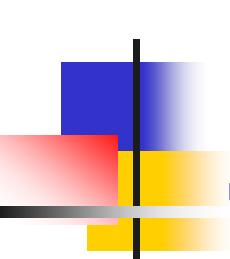
5. 1 虚拟存储器概述

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



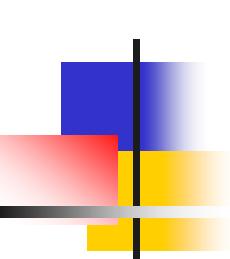
5.1 虚拟存储器概述

前面介绍的各种存储器管理方式，要求将一个作业全部装入内存后方能运行。于是出现了两种情况：

- 有的作业很大，所要求的内存空间超过内存总容量，作业不能被全部装入内存，致使该作业无法运行。
- 有大量作业要求运行，但内存容量不足以容纳所有作业，只能将少数作业装入内存使其运行，其他大量作业留在外存上等待。

解决方法：

- 从物理上增加内存容量
- 从逻辑上扩充内存容量

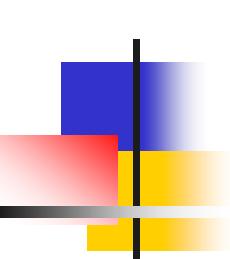


5.1 虚拟存储器概述

5.1.1 常规存储器管理方式的特征和局部性原理

5.1.2 虚拟存储器的定义和特征

5.1.3 虚拟存储器的实现方法



5.1 虚拟存储器概述

5.1.1 常规存储器管理方式的特征和局部性原理

5.1.2 虚拟存储器的定义和特征

5.1.3 虚拟存储器的实现方法

5.1.1 常规存储器管理方式

1. 常规存储器管理方式的特征

- **一次性:** 作业在运行前一次性地全部装入内存
- **驻留性:** 作业装入内存后，便一直驻留在内存中，
直至作业运行结束。

问题: 一次性及驻留性在程序运行时是否是必须的？

5.1.1 常规存储器管理方式

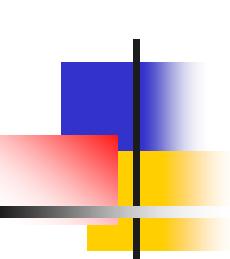
2. 程序运行的局部性原理

1968年，Denning.P提出：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。

- 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。
- 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，但研究表明，过程调用的深度在大多数情况下不超过5。
- 程序中存在许多循环结构，由少数指令构成，但会多次执行。
- 程序中对许多数据结构的处理，往往都局限于很小的范围内。

局部性的表现：

- 时间局部性
- 空间局部性



5.1 虚拟存储器概述

5.1.1 常规存储器管理方式的特征和局部性原理

5.1.2 虚拟存储器的定义和特征

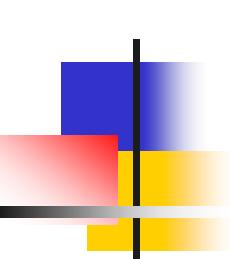
5.1.3 虚拟存储器的实现方法

5.1.2 虚拟存储器的定义和特征

虚拟存储器的定义

基于局部性原理，应用程序在运行前，没有必要全部装入内存，仅将当前要运行的部分页面或段先装入内存即可运行，其余部分暂留在外存上。程序运行时，如果要访问的页(段)已调入内存，便可继续执行；如果尚未调入内存(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，使程序继续执行。如果内存已满，无法再装入新的页(段)，还必须利用页(段)的置换功能，将内存中暂时不用的页(段)调至外存，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。

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，而每位的成本却接近于外存。

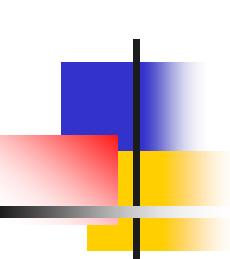


5.1.2 虚拟存储器的定义和特征

- **多次性**: 一个作业被分成多次调入内存运行
- **对换性**: 允许在作业的运行过程中进行换进、换出。
- **虚拟性**: 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。

虚拟性以多次性和对换性为基础；

多次性和对换性又必须建立在离散分配的基础上。



5.1 虚拟存储器概述

5.1.1 常规存储器管理方式的特征和局部性原理

5.1.2 虚拟存储器的定义和特征

5.1.3 虚拟存储器的实现方法

5.1.3 虚拟存储器的实现方法

在虚拟存储器中，允许将一个作业分多次调入内存。如果采用连续分配方式，不仅造成内存资源的浪费，而且无法从逻辑上扩大内存容量。因此，虚拟存储器的实现都是建立在离散分配的存储管理方式的基础上。

1. 分页请求系统

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。它允许只装入部分页面的程序(及数据)，便启动运行。以后再通过调页功能及页面置换功能，陆续将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。

为实现请求调页和置换功能，系统必须提供必要的支持：

- **硬件支持：**①请求分页的页表机制 ②缺页中断机构
 ③地址变换机构
- **软件支持：**①实现请求调页的软件 ②实现页面置换的软件

5.1.3 虚拟存储器的实现方法

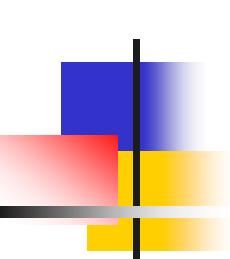
1. 分页请求系统

2. 请求分段系统

在分段系统的基础上，增加了请求调段功能和分段置换功能所形成的段式虚拟存储系统。它允许只装入若干段的程序(及数据)，便启动运行。以后再通过调段功能及段的置换功能，把暂不运行的段调出，同时调入即将要运行的段。置换时以段为单位。

为实现请求调段和置换功能，系统必须提供必要的支持：

- 硬件支持：①请求分段的段表机制 ②缺段中断机构
 ③地址变换机构
- 软件支持：①实现请求调段的软件 ②实现段的置换的软件



第五章 虚拟存储器

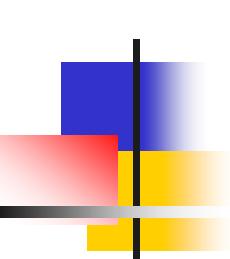
5.1 虚拟存储器概述

5.2 请求分页存储管理方式

5.3 页面置换算法

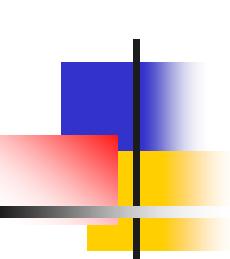
5.4 “抖动”与工作集

5.5 请求分段存储管理方式



5.2 请求分页存储管理方式

请求分页系统是建立在基本分页基础上的，增加了请求调页功能和页面置换功能。换入和换出的基本单位都是长度固定的页面，因而在实现上比请求分段系统简单。

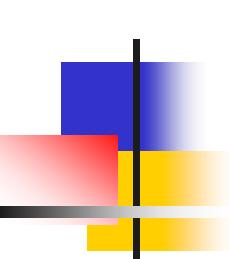


5.2 请求分页存储管理方式

5.2.1 请求分页中的硬件支持

5.2.2 请求分页中的内存分配

5.2.3 页面调入策略



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

5.2.2 请求分页中的内存分配

5.2.3 页面调入策略

5.2.1 请求分页中的硬件支持

1. 页表机制

在请求分页系统中所需要的主要数据结构是页表。基本作用仍是将用户地址空间中的逻辑地址变换为内存空间中的物理地址。由于只将程序的一部分装入内存，还有一部分在外存中，因此须在页表中增加若干项，供程序或数据在换进、换出时参考。

请求分页系统中的页表项：

页号	物理块号	状态位P	访问字段A	修改位M	外存地址
----	------	------	-------	------	------

- (1) **状态位P**：指示该页是否已调入内存。
- (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。
- (3) **修改位M**：表示该页在调入内存后是否被修改过。
- (4) **外存地址**：用于指出该页在外存上的地址。

5.2.1 请求分页

1. 页表机制

2. 缺页中断机构

在请求分页系统中，每当要产生一缺页中断，请求OS将所缺为中断，同样需要经历诸如保护入缺页中断处理程序进行处理、缺页中断是一种特殊的中断，与

- 在指令执行期间产生和处理中断
- 一条指令在执行期间，可能产生

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，再为实现虚拟存储器而增加了某些功能而形成的，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。

页面

6

B:

5

A:

3

指令

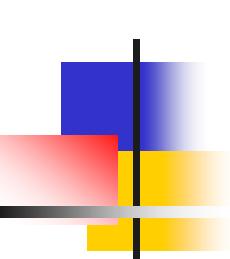
copy A

TO B

2

1

未理



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

5.2.2 请求分页中的内存分配

5.2.3 页面调入策略

5.2.2 请求分页中的内存分配

为进程分配内存时，涉及三个问题：

- 最小物理块数的确定
- 物理块的分配策略
- 物理块的分配算法

1. 最小物理块数的确定

最小物理块数，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。

进程应获得的最小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。

对于某些简单机器，若是单地址指令且采用直接寻址方式，最小物理块数应为2；如果该机器允许间接寻址，则至少要求3个物理块。

对于某些功能较强的机器，指令长度可能是两个或两个以上字节，至少要为进程分配6个物理块。

5.2.2 请求分页中的内存分配

1. 最小物理块数的确定

2. 物理块的分配策略

在请求分页系统中，可以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，也可以采取两种策略，即全局置换和局部置换。于是可以组合出三种适合的策略。

■ 固定分配局部置换

基于进程的类型，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，在整个运行期间不再改变。

采用该策略时，如果进程在运行中发现缺页，只能从该进程在内存中的n个页面中选出一页换出，然后再调入一页。

困难：应为每个进程分配多少个物理块难以确定。

■ 可变分配全局置换

在采用这种策略时，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的物理块队列。如果某进程发生缺页时，由系统从空闲的物理块队列中，取出一个物理块分配给该进程，并将欲调入的页装入其中。

5.2.2 请求分页中的内存分配

1. 最小物理块数的确定

2. 物理块的分配策略

- 固定分配局部置换
- 可变分配全局置换
- 可变分配局部置换

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，如果某进程发生缺页时，只允许从该进程在内存的页面中选出一页换出，不会影响其他进程执行。如果进程在运行中频繁发生缺页中断，则系统再为进程分配若干物理快；如果进程在运行中缺页率特别低，则适当减少分配给该进程的物理块。

5.2.2 请求分页中的内存分配

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物理块分配给各个进程，可采用以下几种算法：

(1) 平均分配算法：将系统中所有可供分配的物理块，平均分配给各个进程。

缺点：未考虑各进程本身的大小。

(2) 按比例分配算法：根据进程的大小按比例分配物理块。

设系统中共有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 ，将有：

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

b_i 应该取整，必须大于最小物理块数。

■ 5.2.2 请求分页中的内存分配

1. 最小物理块数的确定

2. 物理块的分配策略

3. 物理块分配算法

(1) 平均分配算法

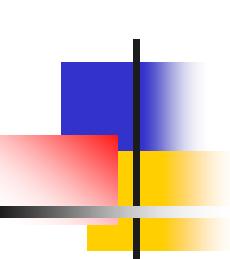
(2) 按比例分配算法

(3) 考虑优先权的分配算法

在实际应用中，为了照顾重要的、急迫的作业尽快完成，应为它分配较多的内存空间。

方法：

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，适当地增加其相应份额后，分配给各进程。



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

5.2.2 请求分页中的内存分配

5.2.3 页面调入策略

5.2.3 页面调入策略

1. 何时调入页面

1) 预调页策略

采用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。

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

当程序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将其所需要的页面调入内存。

优点：由请求调页策略所确定调入的页，一定会被访问；请求调页策略比较容易实现。

缺点：每次仅调入一页，需花费较大的系统开销，增加了磁盘I/O的启动频率。

5.2.3 页面调入策略

1. 何时调入页面

2. 从何处调入页面

在请求分页系统中的外存分为文件区和对换区。

每当发生缺页时，系统应从何处将缺页调入内存，可分为：

- **系统拥有足够的对换区空间：** 可以全部从对换区调入所需页面，以提高调页速度。
- **系统缺少足够的对换区空间：** 凡不会被修改的文件，直接从文件区调入；换出时不用换，再调入时仍从文件区调入。可能被修改的部分，换出时需调到对换区，换入时从对换区调入。
- **UNIX方式：** 与进程有关的文件放在文件区，故未运行的页面应从文件区调入。曾经运行但又被换出的页面被放在对换区，下次调入应从对换区调入。进程请求的共享页面可能已被其他进程调入，无需再从对换区调入。

5.2.3 页面调入策略

1. 何时调入页面

2. 从何处调入页面

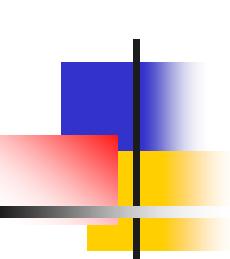
3. 页面调入过程

(1) 每当程序所要访问的页面未在内存时，便向CPU发出一缺页中断，中断处理程序首先保护CPU环境，分析中断原因后，转入缺页中断处理程序。

(2) 该程序通过查找页表，得到该页在外存上的物理块后，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。

(3) 如果内存已满，则需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，可不必写回磁盘；但如果此页已被修改，则必须将它写回磁盘，然后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。

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



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

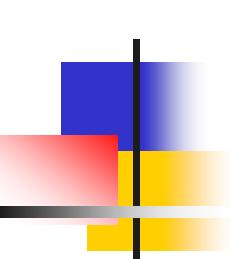
5.3 页面置换算法

在进程运行过程中，若其访问的页面不在内存而需将其调入，但内存已无空闲空间时，需从内存中调出一页程序或数据，送入磁盘的对换区。但应将哪个页面调出，需根据一定的算法来确定。把选择换出页面的算法称为页面置换算法，其好坏直接影响系统的性能。

一个好的置换算法应具有较低的页面更换频率。从理论上讲，应将那些以后不会再访问的页面换出，或者把那些在较长时间内不会再访问的页面换出。

设作业执行中访问页面的总次数为A，其中有F次访问的页面尚未装入主存，故产生了F次缺页中断。现定义**缺页中断率**为：

$$f = F/A$$



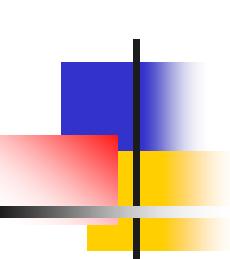
5.3 页面置换算法

5.3.1 最佳置换算法和先进先出置换算法

5.3.2 最近最久未使用 (LRU) 算法

5.3.3 Clock 置换算法

5.3.4 页面缓冲算法



5.3 页面置换算法

5.3.1 最佳置换算法和先进先出置换算法

5.3.2 最近最久未使用（LRU）算法

5.3.3 Clock置换算法

5.3.4 页面缓冲算法

5.3.1 最佳置换算法和先进先出置换算法

最佳置换算法是一种理想化的算法，具有最好的性能，但难于实现。先进先出置换算法最直观，但可能性能最差，故应用极少。

1. 最佳置换算法

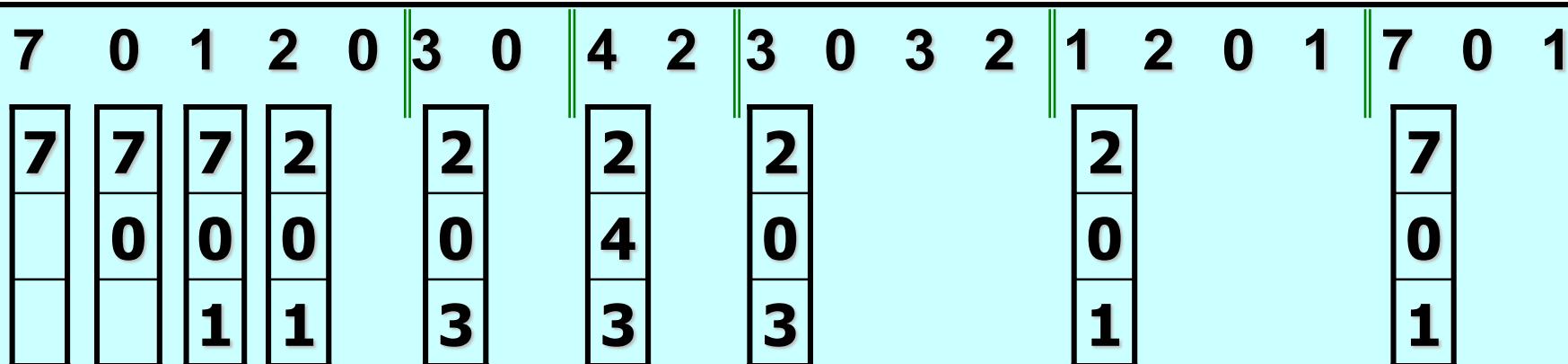
其所选择的被淘汰页面，将是以后永不再用的，或许是在最长(未来)时间内不再被访问的页面。

优点：保证获得最低的缺页率

缺点：无法预知一个进程在内存的若干个页面，哪个在未来最长
时间内不再被访问。

算法无法实现，但可评价其他算法。

设作业分配3个物理块，开始3页不算缺页（后面算法同）。则：



页框(物理块)

最佳置换算法（缺页6次）

5.3.1 最佳置换算法和先进先出置换算法

1. 最佳置换算法

2. 先进先出置换算法

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。

算法实现简单，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(替换指针)，使它总是指向最老的页面。

算法与进程的实际运行规律不相适应，因为进程中的某些页面经常被访问，但先进先出置换算法不能保证这些页面不被淘汰。

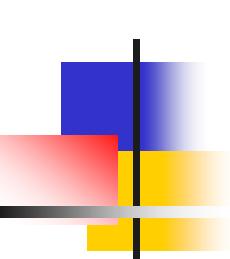
7	0	1	2	0
7	7	7	2	
0	0	0	0	
1	1			

3	0	4	2	3	0	3
2	2	4	4	4	0	
3	3	3	2	2	2	
1	0	0	0	3	3	

1	2	0	1	7	0	1
0	0		1	7	7	
1	1		2	0	0	
3	2		2	2	1	

页框(物理块)

先进先出置换算法（缺页12次）



5.3 页面置换算法

5.3.1 最佳置换算法和先进先出置换算法

5.3.2 最近最久未使用 (LRU) 算法

5.3.3 Clock 置换算法

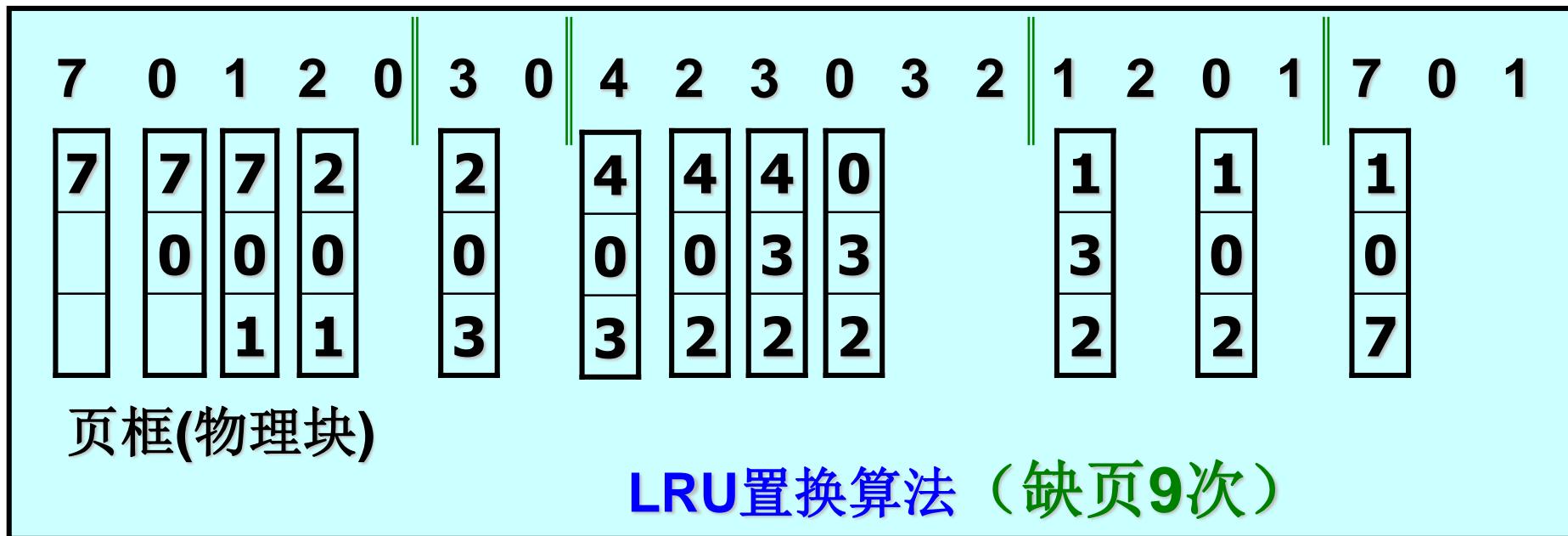
5.3.4 页面缓冲算法

5.3.2 最近最久未使用(LRU)置换算法

1. LRU置换算法的描述

算法根据页面调入内存后的使用情况进行决策。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，当需淘汰一个页面时，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。



5.3.2 最近最久未使用(LRU)置换算法

最佳置换算法是从“[向后看](#)”的观点出发的，即它是根据以后各页的使用情况；而LRU算法则是“[向前看](#)”的，即根据各页以前的使用情况来判断，而页面过去和未来的走向之间并无必然的联系。

1. LRU置换算法的描述

2. LRU置换算法的硬件支持（实现）

LRU置换算法虽然较好，但需较多的硬件支持，为了了解一个进程在内存中的各个页面各有多少时间未被进程访问，以及如何快速地知道哪一页是最近最久未使用的页面，需有以下两类硬件之一的支持：

- 寄存器
- 栈

5.3.2 最近最久未使用(LRU)置换算法

1. LRU置换算法的描述

2. LRU

某进程具有8个页面时的LRU访问情况

1) 算

进程

此时
位寄
应的

实质	R	R ₇	R ₆	R ₅	R ₄	R ₃	R ₂	R ₁	R ₀
1	0	1	0	1	0	0	0	1	0
2	1	0	1	0	1	1	1	0	0
3	0	0	0	0	0	0	1	0	0
4	0	1	1	0	0	1	0	1	1
5	1	1	0	1	1	0	1	1	0
6	0	0	1	0	0	1	0	1	1
7	0	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	1	0	1

某

。

n

对

5.3.2 最近最久未使用(LRU)置换算法

1. LRU置换算法的描述

2. LRU置换算法的硬件支持（实现）

1) 寄存器

2) 栈

利勝

每当进和

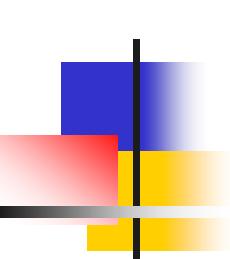
它压入相

底则是量

设一进程访问页面的页面号序列为：

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

随着进程的访问，栈中页面号的变化情况：



5.3 页面置换算法

5.3.1 最佳置换算法和先进先出置换算法

5.3.2 最近最久未使用 (LRU) 算法

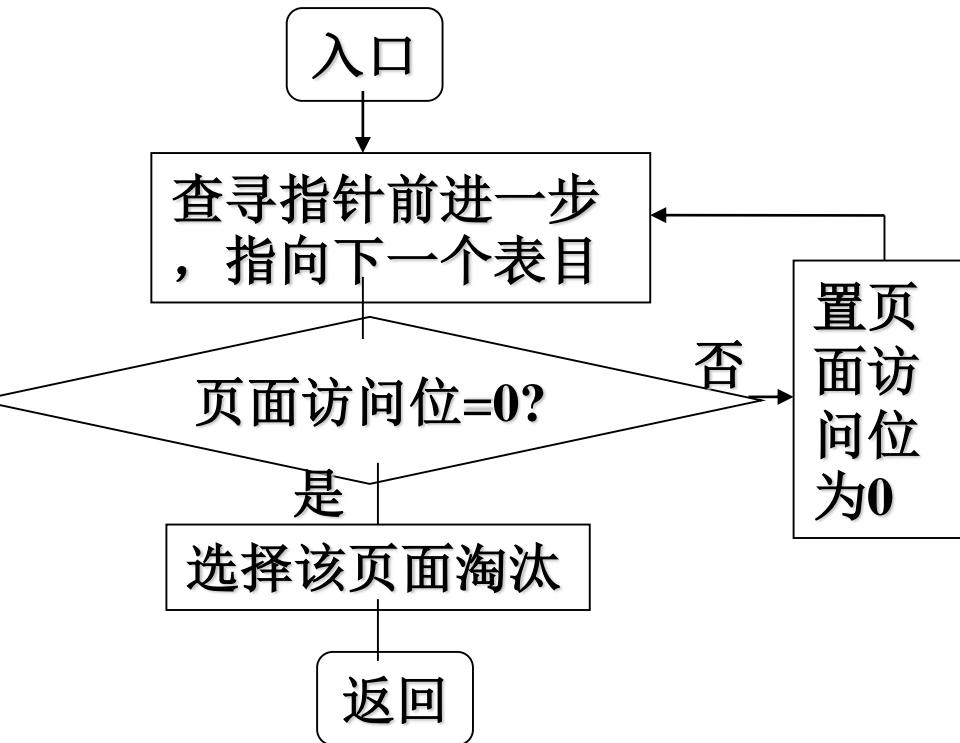
5.3.3 Clock 置换算法

5.3.4 页面缓冲算法

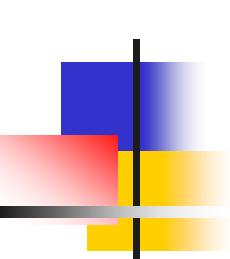
5.3.3 Clock置换算法

又称最近未使用算法(NRU, Not Recently Used), LRU和FIFO的折衷。

原理: 每页有一个使用标志位(use bit), 若该页被访问则置 user bit=1。置换时采用一个指针, 从当前指针位置开始按地址先后检查各页, 寻找use bit=0的页面作为被置换页。指针经过的user bit=1的页都修改user bit=0, 最后指针停留在被置换页的下一个页。



块号	页号	访问位	指针	替换指针
0				
1				
2	4	0	4	指向最近被替换的 4 号页所在块的块号 2
3				
4	2	1→0	6	2→6 被调入页的页号为 6
5				
6	5→6	0→1	7	
7	1	1	2	



5.3 页面置换算法

5.3.1 最佳置换算法和先进先出置换算法

5.3.2 最近最久未使用 (LRU) 算法

5.3.3 Clock 置换算法

5.3.4 页面缓冲算法

5.3.4 页面缓冲算法

1. 最少使用置换算法LFU（Least Frequently Used）

选择到当前时间为止被访问次数最少的页面被置换；

- 基本方法

记录每个页面的访问次数，最少访问的页面首先考虑淘汰。

- 实际采取方法

为页面设置移位寄存器。

与LRU的区别：
 $R1=10000000$

$R2=01110100$

LRU-----淘汰R2

LFU-----淘汰R1

5.3.4 页面缓冲算法

1. 最少使用置换算法LFU (Least Frequently Used)

2. 页面缓冲算法(Page Buffering Algorithm)

是对FIFO算法的发展，通过被置换页面的缓冲，有机会找回刚被置换的页面；该算法在页面分配时，采用可变分配和局部置换的方式。

被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。

- 如果页面未被修改，就将其归入到空闲页面链表的末尾
- 否则将其归入到已修改页面链表。

需要调入新的物理页面时，将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。

空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，只需较小开销，而被访问的页面可以返还作为进程的内存页。

当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表，这样能大大减少I/O操作的次数。

习题

在一个请求分页系统中，采用 **LRU**页面置换算法时，假如一个作业的页面访问顺序为**4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5**，当分配给该作业的物理块数**M**分别为**3**和**4**时，试计算访问过程中所发生的缺页次数**A**和**B**，缺页率分别为**A/C**和**B/C**，其中**C**为访问次数。比较所得的结果为**D**。

A, B, C: (1) 7; (2) 8; (3) 9; (4) 10; (5)11;
(6)12; (7)13。

D: (1) 正常现象，即存储块增加，缺页次数减少；
(2) 存在奇异现象，即存储块增加，缺页次数反而增加；
(3) 存储块增加，缺页次数不变。

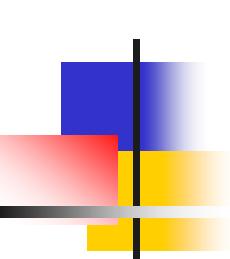
答案：

习题

已知一个有快表的请求分页系统,内存读写周期为**Tr(1us)**内外存之间传送一个页面的时间 **Tt(5ms)**, 快表的命中率为 **p (80%)**, 页面失效率为 **f (10%)**。计算快表命中时存取时间为**A**, 快表不命中、内存中页表查到时存取时间为**B**, 页面失效时存取时间为**C**。求系统和内存有效存取周期**EAT**(平均存取时间)。

A、B、C: (1) **1μs**; (2) **2μs**; (3) **3μs**; (4) **500μs**;
(5) 501μs; (6) 5ms.

答案:



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

5.4 “抖动”与工作集

请求分页式虚拟存储器系统在正常运行情况下，能有效地减少内存碎片，提高处理机的利用率和吞吐量。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。

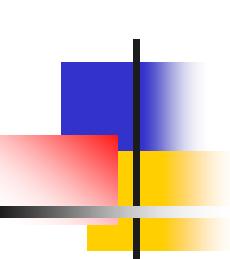
多道程序度与“抖动”

随着进程数目的增加，处理机利用率趋于0时是系统中已发生了“抖动”。

产生“抖动”的原因是同时在系统中运行的进程太多，而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，频繁的出现缺页，必须请求系统将所缺之页调入内存。

工作集

是指在某段时间间隔 \triangle 里，进程实际所要访问页面的集合。



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

5.5 请求分段存储管理方式

在请求分段系统中，程序运行之前，只需先调入若干个分段(不必调入所有的分段)，便可启动运行。当所访问的段不在内存中时，可请求OS将所缺的段调入内存。

5.5.1 请求分段中的硬件支持

1. 段表机制

在请求分段系统中所需要的主要数据结构是段表。由于在应用程序的许多段中，只有一部分段装入内存，其余的一些段仍留在外存上，故需在段表中增加若干项，以供程序在调进、调出时参考。

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

1. 段表机制

请求分段系统中的段表项：

段名	段长	段的 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。
- (2) 访问字段A：用于记录本段被访问的频繁程度。
- (3) 修改位M：表示该段在调入内存后是否被修改过。
- (4) 存在位(状态位)P：指示该段是否已调入内存。
- (5) 增补位：用于表示该段在运行中是否做过动态增长。
- (6) 外存地址：用于指出该段在外存上的起始地址(盘块号)。

5.5.1 请求分段中的硬件支持

1. 段表机制

段名	段长	段的 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

2. 缺段中断机构

在请求分段系统中，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。缺段中断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。但不会出现一条指令被分割在两个分段中或一组信息被分割在两个分段中的情况。

3. 地址变换机构

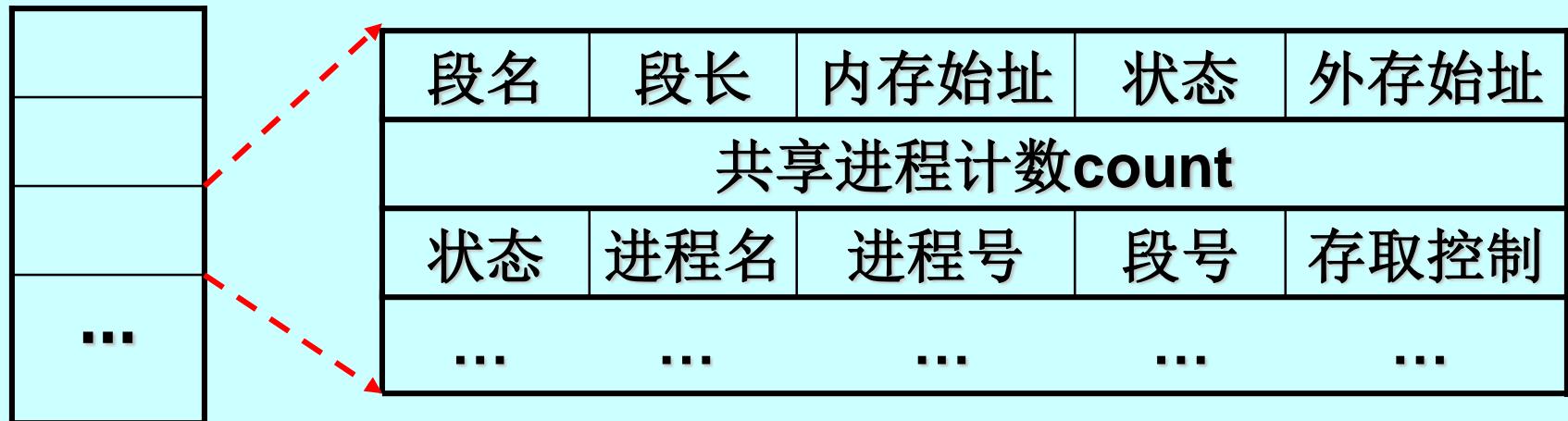
请求分段系统中的地址变换机构，是在分段系统地址变换机构的基础上形成的。因为被访问的段并非全在内存，因而在地址变换时，若发现要访问的段不在内存，必须先将所缺的段调入内存，并修改段表，然后才能利用段表进行地址变换。在地址变换机构中需增加缺段中断的处理及请求等功能。

5.5.2 分段的共享和保护

1. 共享段表

为了实现共享，可在内存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。

(1) 共享计数**count**: 共享段为多个进程所需要，当某进程不再需要它而释放它时，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



共享段表

5.5.2 分段的共享和保护

1. 共享段表
 2. 共享段的分享与回收
- ### 1) 共享段的分配

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；之后，当又有其他进程需要调用该共享段时，无需再为该段分配内存，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**count:=count+1**操作。

- ### 2) 共享段的回收

当共享该段的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**count:=count-1**操作。如果结果为0，则需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，否则只取消调用者进程在共享段表中的有关记录。

5.5.2 分段的共享和保护

1. 共享段表

2. 共享段的分享与回收

3. 分段保护

1) 越界检查

段表寄存器存放了段表长度；段表中存放了每个段的段长。在进行存储访问时，将段号与段表长度比较，段内地址与段长比较。

2) 存取控制检查

段表中的每个表项都设置了“存取控制”字段，用于规定该段的访问方式：只读；只执行；读/写

3) 环保护机构

规定：低编号的环具有高优先权

遵循的原则：

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