4.6 虚拟存储器的基本概念

前面所介绍的各种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全部装入内存后方能运行。于是，出现了下面这样两种情况：

- 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。
- 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。

4.6.1 虚拟存储器的引入

1．常规存储器管理方式的特征

(1) 一次性。在前面所介绍的几种存储管理方式中，都要求将作业全部装入内存后方能运行，即作业在运行前需一次性地全部装入内存，而正是这一特征导致了上述两种情况的发生。此外，还有许多作业在每次运行时，并非其全部程序和数据都要用到。如果一次性地装入其全部程序，也是一种对内存空间的浪费。

4.6.1 虚拟存储器的引入

(2) 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。尽管运行中的进程会因I/O而长期等待，或有的程序模块在运行过一次后就不再需要(运行)了，但它们都仍将继续占用宝贵的内存资源。

由此可以看出，上述的一次性及驻留性，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。

4.6.1 虚拟存储器的引入

2．局部性原理

早在1968年，Denning.P就曾指出：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。他提出了下述几个论点：

(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。

4.6.1 虚拟存储器的引入

(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。

4.6.1 虚拟存储器的引入

局限性还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。

4.6.1 虚拟存储器的引入

3．虚拟存储器的定义

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。

请求调页:程序在运行时，如果它所要访问的页(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。

页面置换: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。

4.6.2 虚拟存储器的实现方法

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。

4.6.2 虚拟存储器的实现方法

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。

4.6.2 虚拟存储器的实现方法

2) 实现请求分页的软件

这里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。

4.6.2 虚拟存储器的实现方法

2．请求分段系统

这是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。

4.6.2 虚拟存储器的实现方法

为了实现请求分段，系统同样需要必要的硬件支持。一般需要下列支持：

(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。

4.6.3 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。

多次性是虚拟存储器最重要的特征，任何其它的存储管理方式都不具有这一特征。因此，我们也可以认为虚拟存储器是具有多次性特征的存储器系统。

4.6.3 虚拟存储器的特征

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，亦即，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。

换进和换出能有效地提高内存利用率。可见，虚拟存储器具有对换性特征。

4.6.3 虚拟存储器的特征

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。这是虚拟存储器所表现出来的最重要的特征，也是实现虚拟存储器的最重要目标。

值得说明的是，虚拟性是以多次性和对换性为基础的，或者说，仅当系统允许将作业分多次调入内存，并能将内存中暂时不运行的程序和数据换至盘上时，才有可能实现虚拟存储器；而多次性和对换性又必须建立在离散分配的基础上。

4.7 请求分页存储管理方式

请求分页要解决的问题：

- 数据结构：进程页表中要表现该页
  - **是否在内存？**
  - **是否允许读写？**
  - **内存已没有空闲页面，是否将该页淘汰？**
  - **一旦淘汰是否写回外存？**
  - **该页的外存地址是什么？**

4.7 请求分页存储管理方式

解决上述问题，需在页表中增设以下项目：

页面/块号	中断/状态位	权限位	引用位	修改位	外存地址

- 硬件要有：发现缺页、发现越界、发现访问非法、做地址转换等功能，这些功能是在每条指令的译码过程中实现。

- OS要有：缺页处理、非法访问处理、页面淘汰处理等例程。

4.7.1 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。其基本作用仍然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。由于只将应用程序的一部分调入内存，还有一部分仍在盘上，故须在页表中再增加若干项，供程序(数据)在换进、换出时参考。在请求分页系统中的每个页表项如下所示：

页号	物理块号	状态位P	访问字段A	修改位M	外存地址

4.7.1 请求分页中的硬件支持

其中各字段说明：

(1) 状态位P：用于指示该页是否已调入内存，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，供选择换出页面时参考。

4.7.1 请求分页中的硬件支持

(3) 修改位M：表示该页在调入内存后是否被修改过。由于内存中的每一页都在外存上保留一份副本，因此，若未被修改，在置换该页时就不需再将该页写回到外存上，以减少系统的开销和启动磁盘的次数；若已被修改，则必须将该页重写到外存上，以保证外存中所保留的始终是最新副本。简言之，M位供置换页面时参考。

(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。

4.7.1 请求分页中的硬件支持

页表示例

页号	块号	状态位	访问位	修改位	外存地址
0	10	1	1	0	120
1	14	1	0	0	123
2	15	1	2	0	134
3	—	0	—	—	135

4.7.1 请求分页中的硬件支持

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。但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。

4.7.1 请求分页中的硬件支持

(2) 一条指令在执行期间，可能产生多次缺页中断。在图4-24中示出了一个例子。如在执行一条指令COPY A TO B时，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。

4.7.1 请求分页中的硬件支持

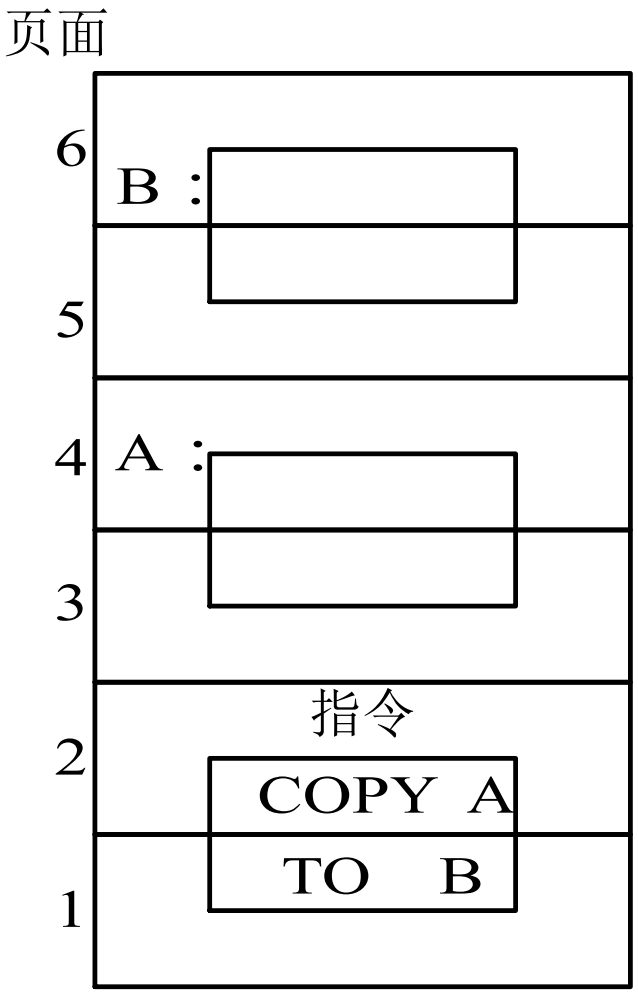


图4-24 涉及6次缺页中断的指令

4.7.1 请求分页中的硬件支持

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。图4-25示出了请求分页系统中的地址变换过程。

在进行地址变换时，首先去检索快表，试图从中找出所要访问的页。若找到，便修改页表项中的访问位。对于写指令，还须将修改位置成“1”，然后利用页表项中给出的物理块号和页内地址形成物理地址。地址变换过程到此结束。

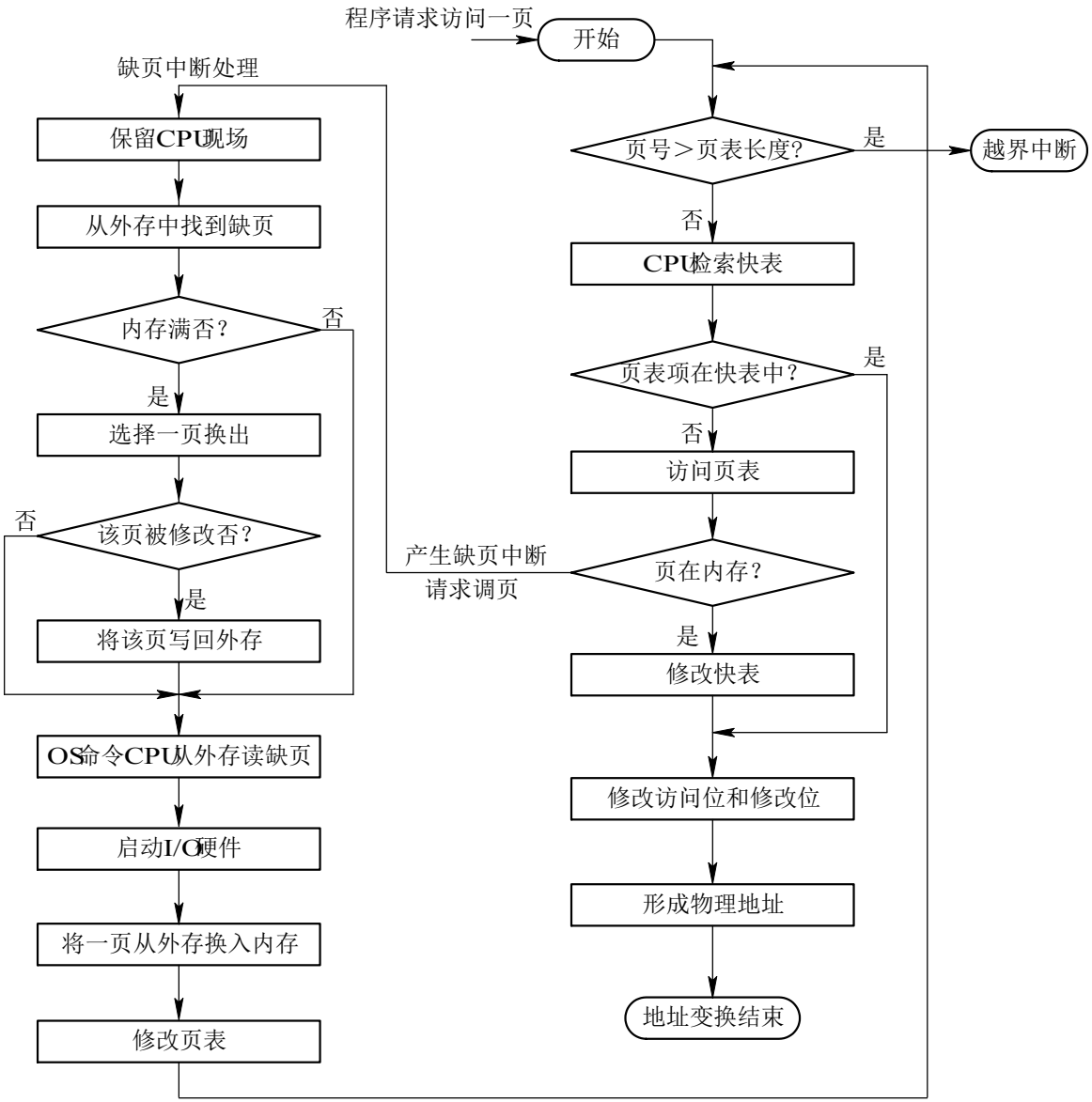
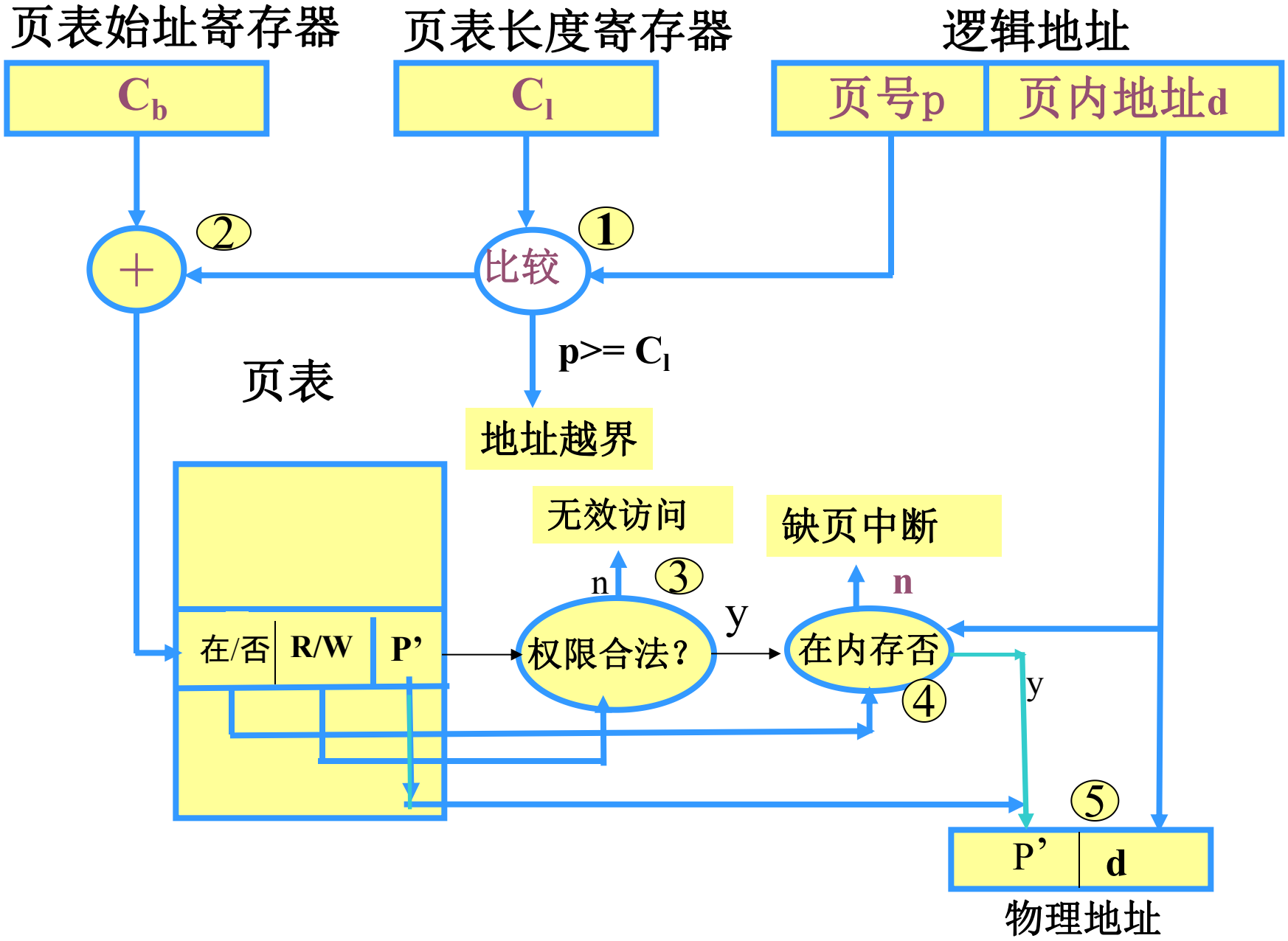


图 4-25　请求分页中的地址变换过程

4.7.1 请求分页中的硬件支持

如果在快表中未找到该页的页表项时，应到内存中去查找页表，再从找到的页表项中的状态位P，来了解该页是否已调入内存。

若该页已调入内存，这时应将此页的页表项写入快表，当快表已满时，应先调出按某种算法所确定的页的页表项，然后再写入该页的页表项；若该页尚未调入内存，这时应产生缺页中断，请求OS从外存把该页调入内存。



请求页式地址映射及存储保护机制

4.7.2 内存分配策略和分配算法

1．最小物理块数的确定

这里所说的最小物理块数，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。

对于某些简单的机器，若是单地址指令且采用直接寻址方式，则所需的最少物理块数为2。其中，一块是用于存放指令的页面，另一块则是用于存放数据的页面。

4.7.2 内存分配策略和分配算法

如果该机器允许间接寻址时，则至少要求有三个物理块。对于某些功能较强的机器，其指令长度可能是两个或多于两个字节，因而其指令本身有可能跨两个页面，且源地址和目标地址所涉及的区域也都可能跨两个页面。

正如前面所介绍的在缺页中断机构中要发生6次中断的情况一样，对于这种机器，至少要为每个进程分配6个物理块，以装入6个页面。

4.7.2 内存分配策略和分配算法

2．物理块的分配策略

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。若太少，会频繁地出现缺页中断，降低了系统的吞吐量；若太多，又必然使内存中驻留的进程数目减少，进而可能造成CPU空闲或其它资源空闲的情况，而且在实现进程对换时，会花费更多的时间。

4.7.2 内存分配策略和分配算法

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，已用于若干个OS中。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物理块队列。当某进程发现缺页时，由系统从空闲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(缺)页装入其中。

这样，凡产生缺页(中断)的进程，都将获得新的物理块。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，自然又会使该进程的物理块减少，进而使其缺页率增加。

4.7.2 内存分配策略和分配算法

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。

4.7.2 内存分配策略和分配算法

3．物理块分配算法

1) 平均分配算法

这是将系统中所有可供分配的物理块平均分配给各个进程。例如，当系统中有100个物理块，有5个进程在运行时，每个进程可分得20个物理块。这种方式貌似公平，但实际上是不公平的，因为它未考虑到各进程本身的大小。如有一个进程其大小为200页，只分配给它20个块，这样，它必然会有很高的缺页率；而另一个进程只有10页，却有10个物理块闲置未用。

4.7.2 内存分配策略和分配算法

2) 按比例分配算法

这是根据进程的大小按比例分配物理块的算法。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，将有：

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

b应该取整，它必须大于最小物理块数。

4.7.2 内存分配策略和分配算法

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，适当地增加其相应份额后，分配给各进程。

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来为各进程分配其物理块的。

4.7.3 调页策略

1．调入页面的时机

1) 预调页策略

如果进程的许多页是存放在外存的一个连续区域中，则一次调入若干个相邻的页，会比一次调入一页更高效些。但如果调入的一批页面中的大多数都未被访问，则又是低效的。

可采用一种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。

4.7.3 调页策略

2) 请求调页策略

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。

4.7.3 调页策略

2．确定从何处调入页面

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，可分成如下三种情况：

(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。

4.7.3 调页策略

(2) 系统缺少足够的对换区空间，这时凡是不会被修改的文件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。但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。

4.7.3 调页策略

3．页面调入过程

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。整个页面的调入过程对用户是透明的。

4.8 页面置换算法

4.8.1 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内不再被访问的，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。现举例说明如下。

Example

3 个物理块

逻辑页引用次序为 4，3，2，1，4，3，5，4，3，2，1，5

OPT	4	3	2	1	4	3	5	4	3	2	1	5
页1	4	3	2	1	1	1	5	5	5	2	1	1
页2		4	3	3	3	3	3	3	3	5	5	5
页3			4	4	4	4	4	4	4	4	4	4
		x	x	x	x	3	3	x	3	3	x	x	3

共缺页中断7次

4.8.1 最佳置换算法和先进先出置换算法

2．先进先出(FIFO)页面置换算法

这是最早出现的置换算法。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。该算法实现简单，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，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。

Belady现象

- 3 个物理块，逻辑页引用次序为 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 12次引用9次缺页

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺 页	x	x	x	x	x	x	x	√	√	x	X	√

Belady现象

- 4 个物理块
- 12次引用10次缺页

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺 页	x	x	x	x	√	√	x	x	x	x	x	x

4.8.2 最近最久未使用(LRU)置换算法

1．LRU(Least Recently Used)置换算法的描述

FIFO置换算法性能之所以较差，是因为它所依据的条件是各个页面调入内存的时间，而页面调入的先后并不能反映页面的使用情况。

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。

LRU	4	3	2	1	4	3	5	4	3	2	1	5
页1	4	3	2	1	4	3	5	4	3	2	1	5
页2		4	3	2	1	4	3	5	4	3	2	1
页3			4	3	2	1	4	3	5	4	3	2
	x	x	x	x	x	x	x	3	3	x	x	x

共缺页中断10次

4.8.2 最近最久未使用(LRU)置换算法

2．LRU置换算法的硬件支持

LRU置换算法虽然是一种比较好的算法，但要求系统有较多的支持硬件。为了了解一个进程在内存中的各个页面各有久未被进程访问，以及如何快速地知道哪一页是最近最久未使用的页面，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

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

4.8.2 最近最久未使用(LRU)置换算法

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位寄存器的数看做是一个整数，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。

4.8.2 最近最久未使用(LRU)置换算法

实页 \ R	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	0	0	0	0	0	1	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	1	0	1	0	1	1
7	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	0	1

图4-29　某进程具有8个页面时的LRU访问情况

4.8.2 最近最久未使用(LRU)置换算法

2) 栈

可利用一个特殊的栈来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。假定现有一进程所访问的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如图4-30所示。在访问页面6时发生了缺页，此时页面4是最近最久未被访问的页，应将它置换出去。

4.8.2 最近最久未使用(LRU)置换算法

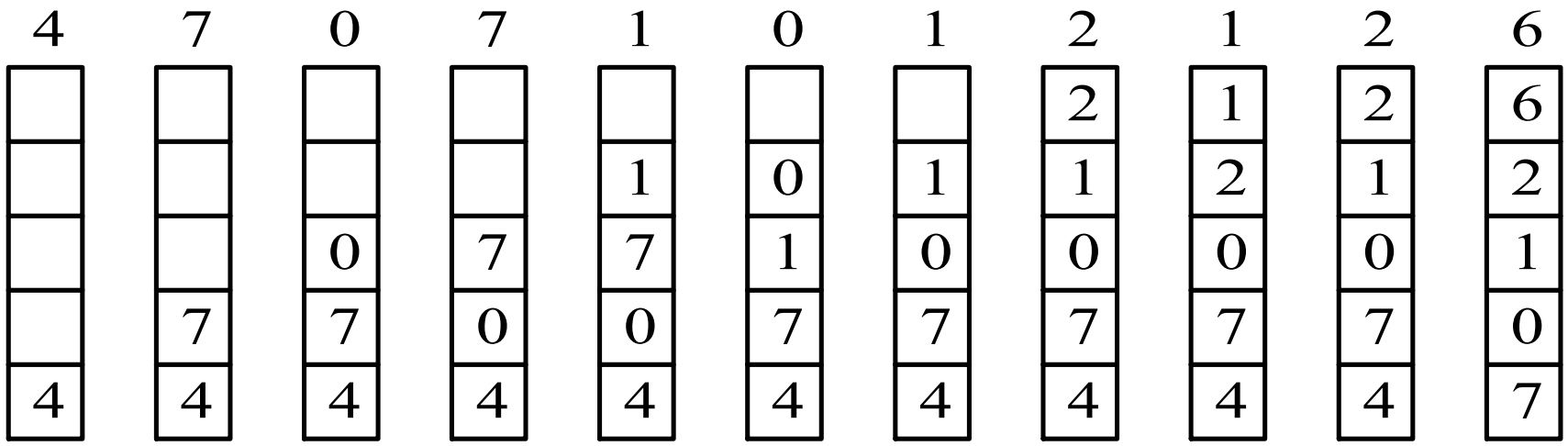


图4-30 用栈保存当前使用页面时栈的变化情况

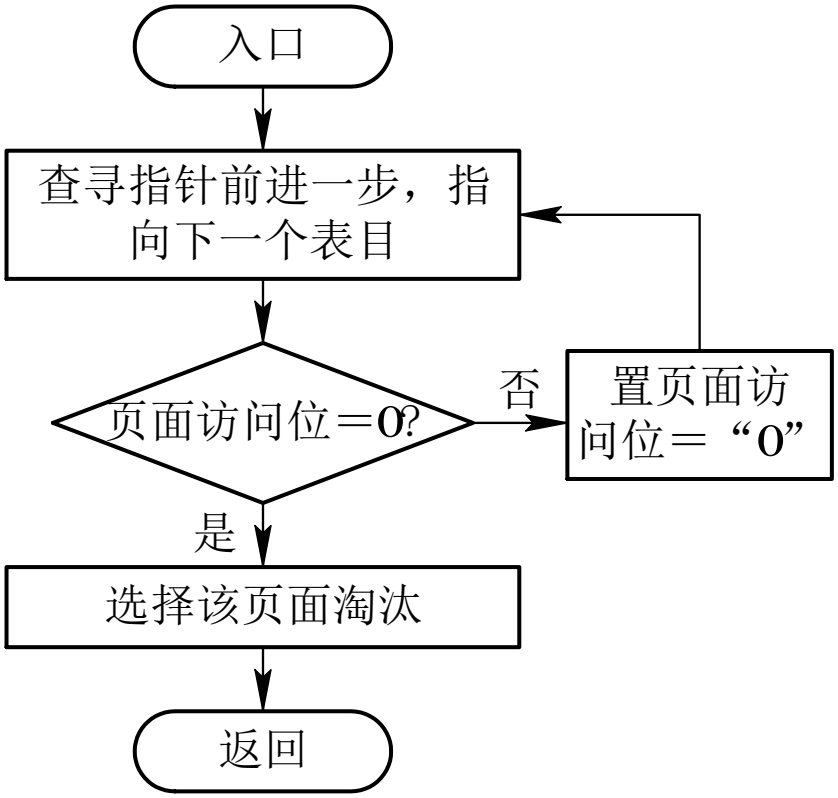
4.8.3 Clock置换算法

1．简单的Clock置换算法

当采用简单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，即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。

图4-31示出了该算法的流程和示例。由于该算法是循环地检查各页面的使用情况，故称为Clock算法。但因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)。

4.8.3 Clock置换算法



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4-31　简单Clock置换算法的流程和示例

4.8.3 Clock置换算法

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。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，即置换代价，这样，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：

4.8.3 Clock置换算法

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。

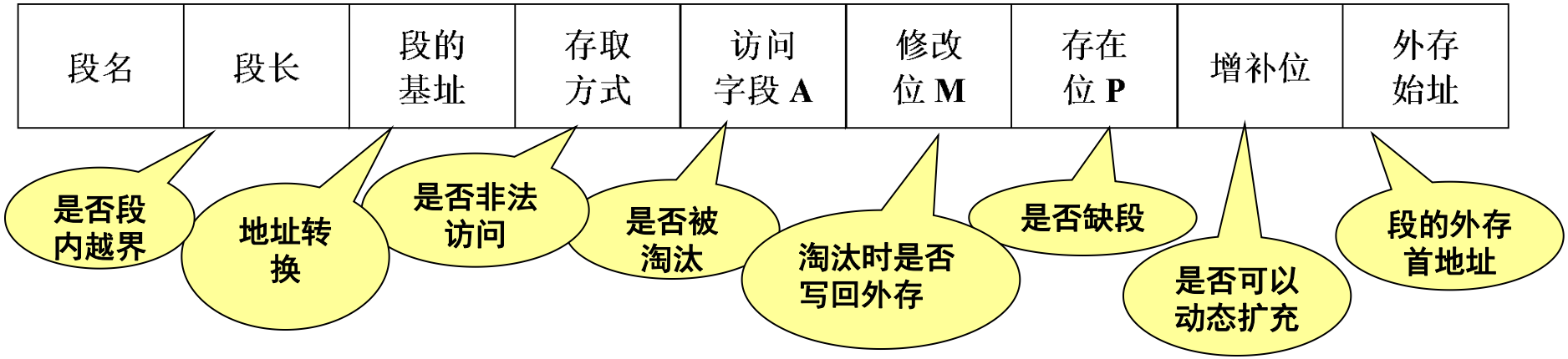
4类(A=1，M=1)：表示该页最近已被访问且被修改，该页可能再被访问。

4.9 请求分段存储管理方式

4.9.1 请求分段中的硬件支持

1．段表机制

在请求分段式管理中所需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，故须在段表中增加若干项，以供程序在调进、调出时参考。下面给出请求分段的段表项。



4.9.1 请求分段中的硬件支持

在段表项中，除了段名(号)、段长、段在内存中的起始地址外，还增加了以下表项。

(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。

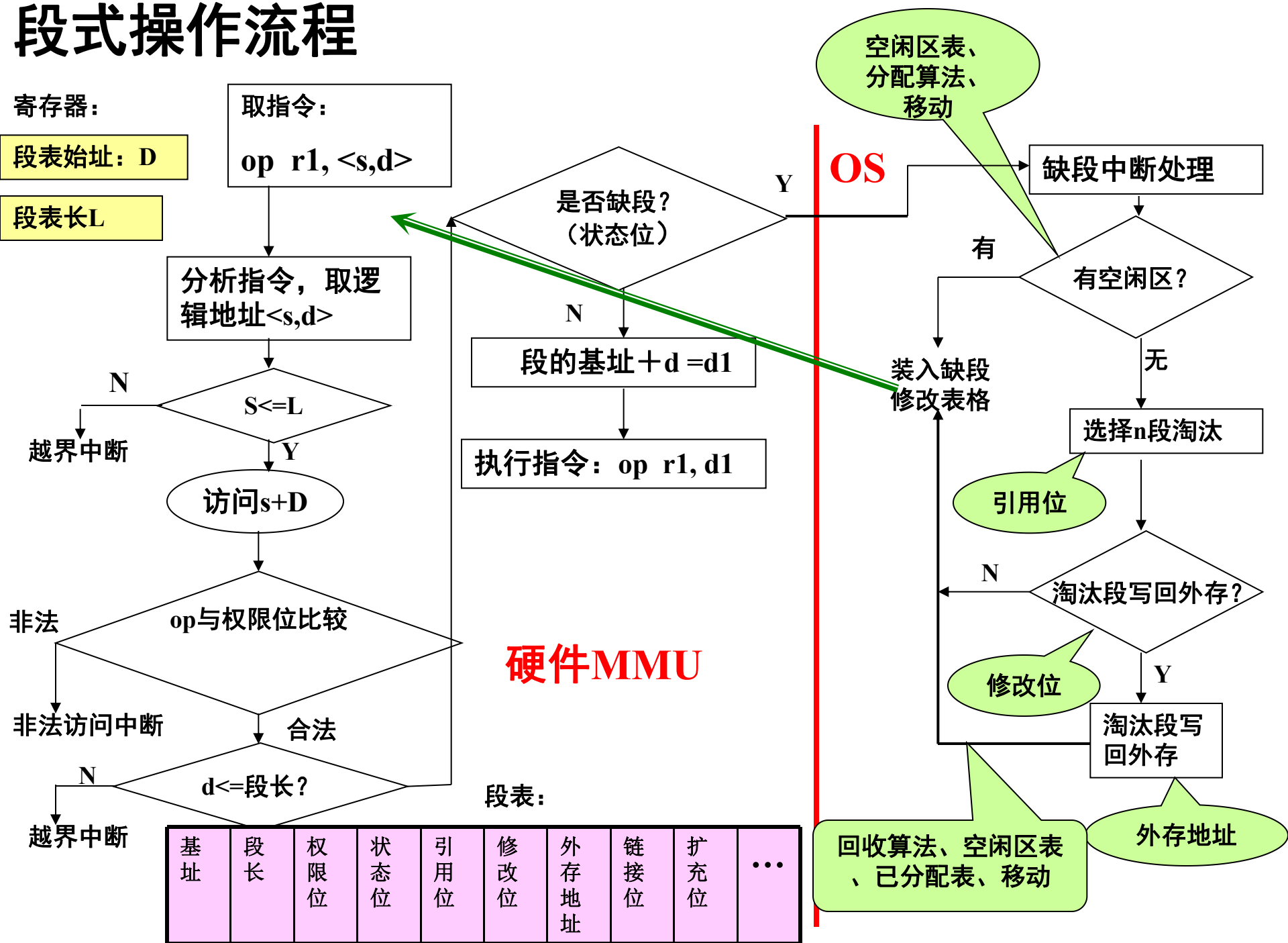
4.9.1 请求分段中的硬件支持

(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。

段式操作流程


寄存器：
段表始址：D
段表长L
取指令：
op r1, <s,d>
分析指令，取逻辑地址<s,d>
S<=L
N
越界中断
Y
访问s+D
op与权限位比较
非法
非法访问中断
合法
d<=段长？
N
越界中断
是否缺段？（状态位）
Y
N
段的基址＋d =d1
执行指令：op r1, d1
硬件MMU
段表：
基址
段长
权限位
状态位
引用位
修改位
外存地址
链接位
扩充位
…
OS
空闲区表、分配算法、移动
缺段中断处理
有空闲区？
有
无
装入缺段修改表格
选择n段淘汰
引用位
淘汰段写回外存？
N
Y
修改位
淘汰段写回外存
回收算法、空闲区表、已分配表、移动
外存地址

4.9.1 请求分段中的硬件支持

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。

缺段中断机构与缺页中断机构类似，它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。缺段中断的处理过程如图4-32所示。由于段不是定长的，这使对缺段中断的处理要比对缺页中断的处理复杂。

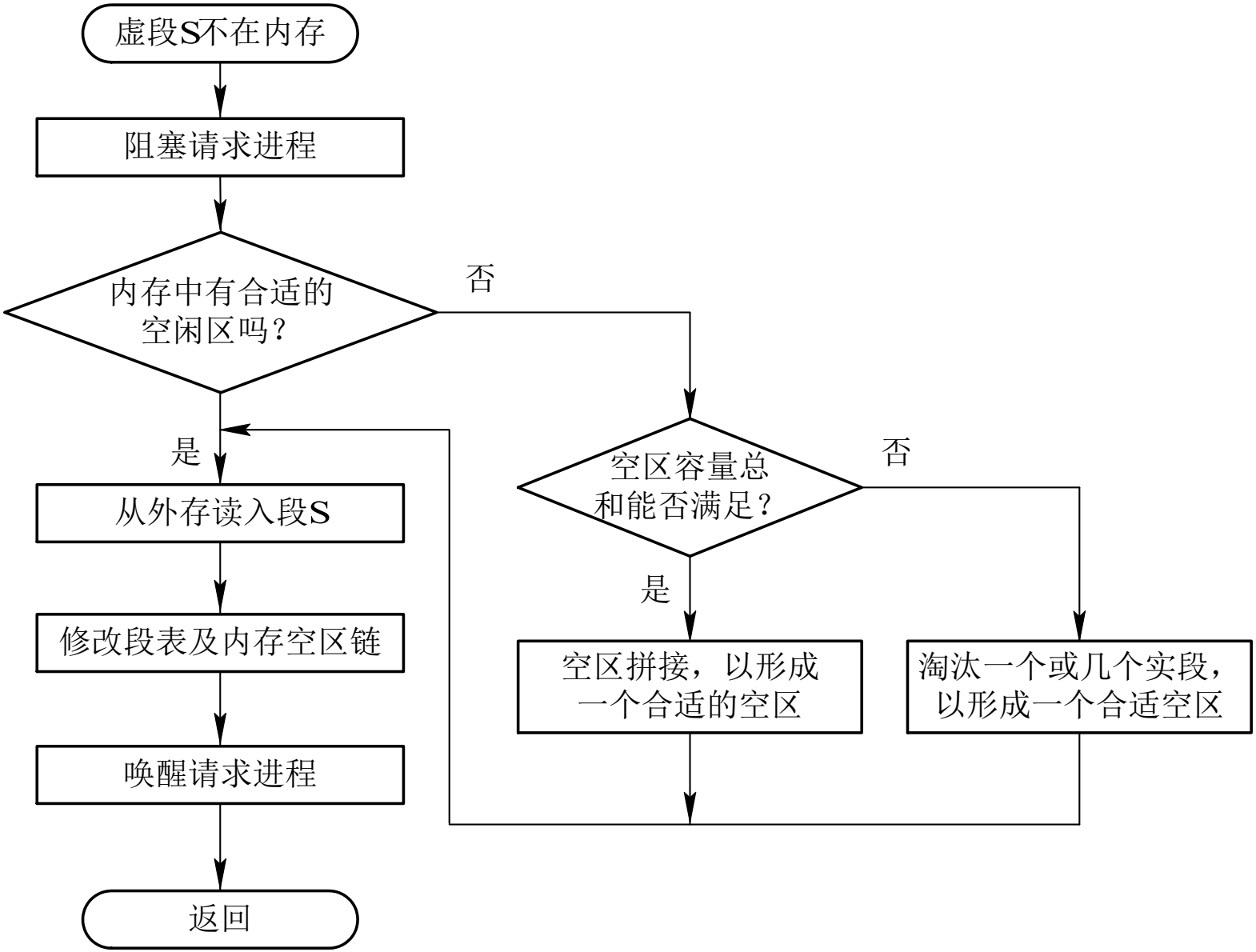


图4-32　请求分段系统中的中断处理过程

4.9.1 请求分段中的硬件支持

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，所以在地址变换时，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4-33示出了请求分段系统的地址变换过程。

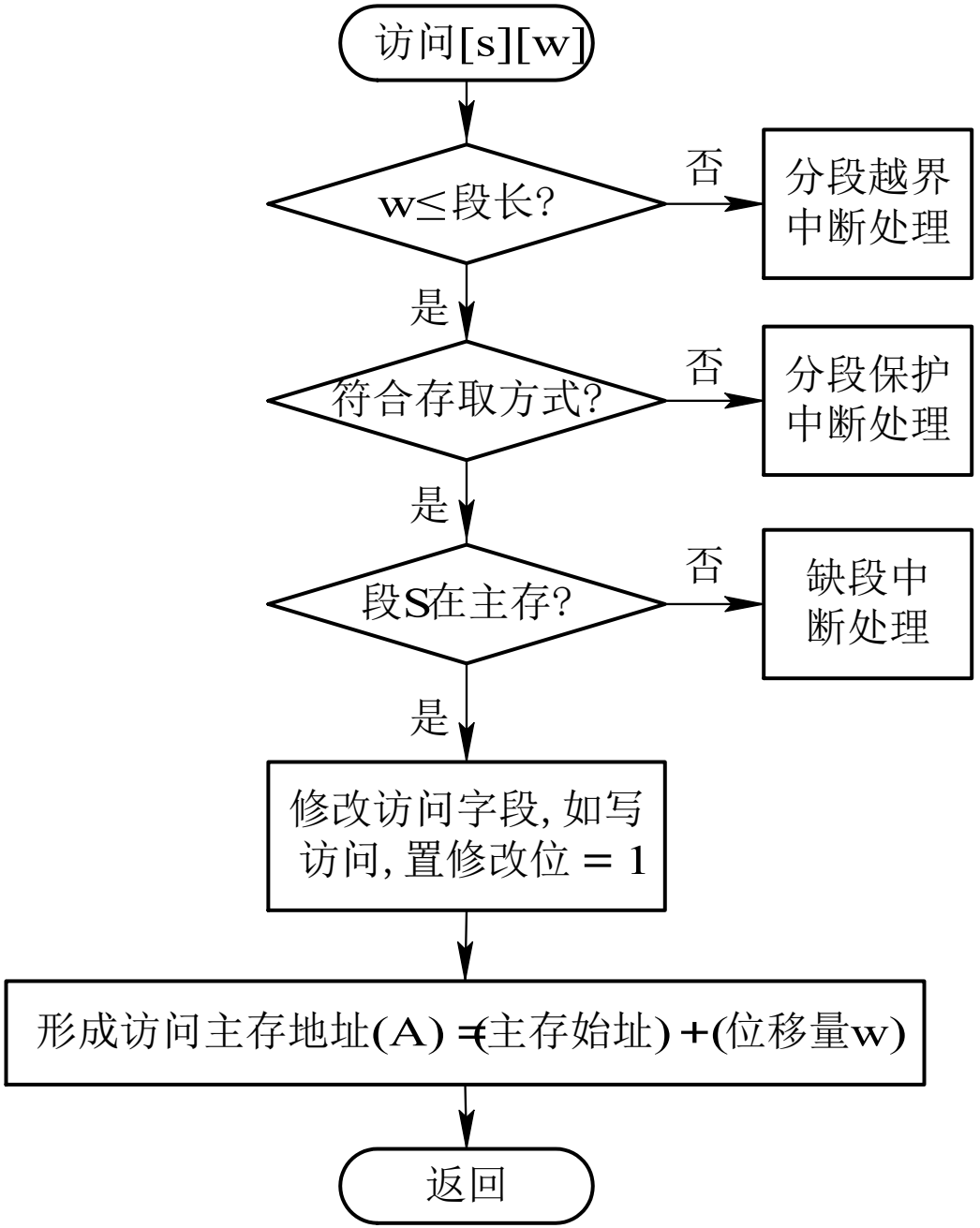


图4-33　请求分段系统的地址变换过程

4.9.2 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。共享段表如图4-34 所示。其中各项说明如下。

4.9.2 分段的共享与保护

(1) 共享进程计数count。非共享段仅为一个进程所需要。当进程不再需要该段时，可立即释放该段，并由系统回收该段所占用的空间。而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。

4.9.2 分段的共享与保护

(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。

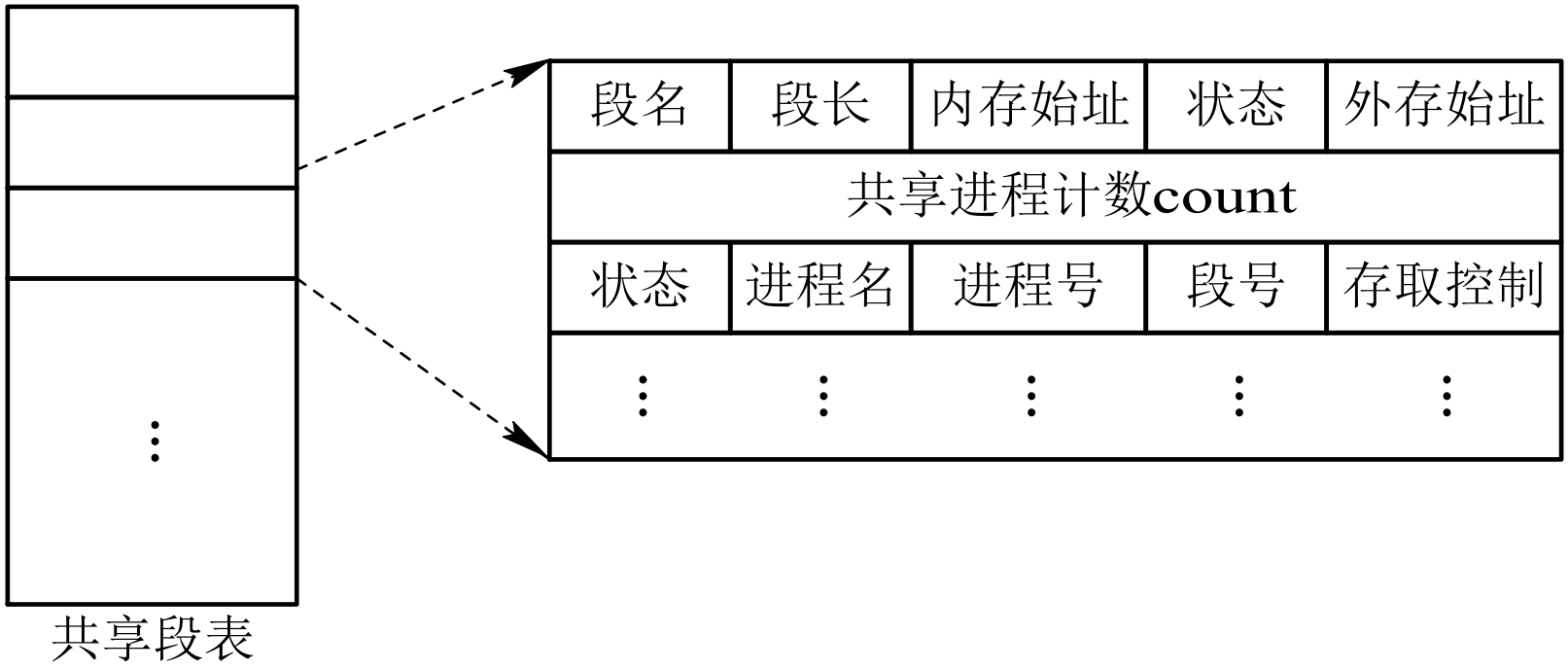


图4-34 共享段表项

4.9.2 分段的共享与保护

2．共享段的分配与回收

1) 共享段的分配

由于共享段是供多个进程所共享的，因此，对共享段的内存分配方法与非共享段的内存分配方法有所不同。

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；之后，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count :=count+1操作，以表明有两个进程共享该段。

4.9.2 分段的共享与保护

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count :=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。

4.9.2 分段的共享与保护

3．分段保护

在分段系统中，由于每个分段在逻辑上是独立的，因而比较容易实现信息保护。目前，常采用以下几种措施来确保信息的安全。

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。

4.9.2 分段的共享与保护

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。

小结

- 概念：重定位、逻辑地址、物理地址、链接、连续、完整、页、页面、段、页表、段表、MMU、局部性原理、缺页率、快表、慢表、越界、非法访问、虚存、覆盖、交换
- 存储管理的功能
- 六种管理方案的全面比较
- 可变式分区的分配、回收算法
- 页式管理的页表格式，地址转换过程、缺页中断处理
- 段式管理的段表格式，地址转换过程、缺段中断处理
- 段页式管理的段表、页表格式、地址转换过程

几种存储管理方案总结

类型	分配回收	保护	共享	扩充	硬件支持	地址空间	碎片	数据结构	OS算法	总体性能
单连续	简单	基址	无	覆盖	基址寄存器	一维	浪费严重、内部碎片大			速度快、资源利用率低、道数最少
固定分区	简单、分区表、算法	基址、限长	无	覆盖、交换	基址、限长寄存器 简单MMU	一维	浪费较严重、内部碎片大	分区表	分配、回收算法、越界处理、非法访问处理、缺页处理	速度快、资源利用率较低、道数较少
可变分区	复杂、	基址、限长	无	覆盖、交换	基址、限长寄存器 简单MMU	一维	外部碎片、移动开销	空闲分区表、已分配分区表	最先适应、最佳适应、最坏适应、回收要合并、越界处理、非法访问处理、	速度快、资源利用率较低、道数较多
页式	较简单	页表长、权限位	可共享、代码段页号相同	虚存	页表始址、长度寄存器、复杂MMU、联想存储器	一维	较小内部碎片（1/2页）	位示图、页表、共享页表	越界处理、非法访问处理、缺页处理	速度慢、资源利用率高、道数多、整体性能好
段式	复杂	段表长、段长、权限位	易共享	虚存	段表始址、长度寄存器、复杂MMU、联想存储器	二维	外部碎片多	空闲区表、已分配区表、段表、共享段表	最先适应、最佳适应、最坏适应、回收要合并、越界处理、非法访问处理、缺段处理	速度慢、资源利用率高、道数多 整体性能较好
段页式	较简单	段表长、页表长、权限位	易共享	虚存	段表始址、长度寄存器、复杂MMU、联想存储器	二维	较小内部碎片n×（1/2页）	位示图、段表、页表、共享页表	越界处理、非法访问处理、缺段处理、缺页处理	执行速度最慢、资源利用率高、道数多、速度慢、资源利用率高、道数多、整体性能最好

作业（1）

设有二维数组 **var A; array [1..100] of array [1..100] of integer;** 其中数组元素**A《1，1》**存放在页面大小为**200**的分页存储管理系统中的地址**200**处，数组按行存储。使用该数组的一个较小的程序存放在第**0**页中（地址**0 - 199**），这样将只会从第0页取指令。 假定现有三个页面，第一个页面存放程序，其余两个页面初始为空。试问：若使用**LRU**替换算法，下面的数组初始化循环将会产生多少次缺页中断？

（1）for j:=1 to 100 do
for i:=1 to 100 do
A[i,j]:=0;

(2)for i:=1 to 100 do
for j:=1 to 100 do
A[i,j]:=0;

作业（2）

考虑下面的页访问串：**1，2，3，4，2，1，5，6，2，1，2，3，7，6，3，2，1，2，3，6** 假定为该进程分配4个页面。试问：若应用下面的页面替换算法，各会出现多少次缺页中断？注意，所给定的页块初始均为空，因此，首次访问一页时就会发生缺页中断。

（1）**LRU**替换算法 （2）**FIFO**替换算法。（3）**Optimal**替换算法。

作业（3）

若在一分页存储管理系统中，某作业的页表如下所示。已知页面大小为1024字节，试将逻辑地址（十进制）1011、2148、3000、4000，5012转化为相应的物理地址。

页号	块号
0 1 2 3	2 3 1 6

作业（4）

若在一分页存储管理系统中，某作业的页表如下所示。已知页面大小为1024字节，在将下面给出的指令与逻辑地址转化为相应的物理地址时,给出每条指令的执行过程描述。 (R, 2311)、(R ,0542)、(W ,3972)、(W, 5665)

设：前后指令无关，采用LRU淘汰算法。中断位1：表示缺页；引用位1：表示最近访问过，修改位1：表示在内存修改过。

页表(page table)

逻辑页号	页面号	中断位	引用位	修改位	存取权限	外存地址
0	3	1	1	0	R	
1	7	0	0	1	RW	
2	5	0	1	1	RW	
3	11	1	0	0	R	
4	19	0	1	0	W	