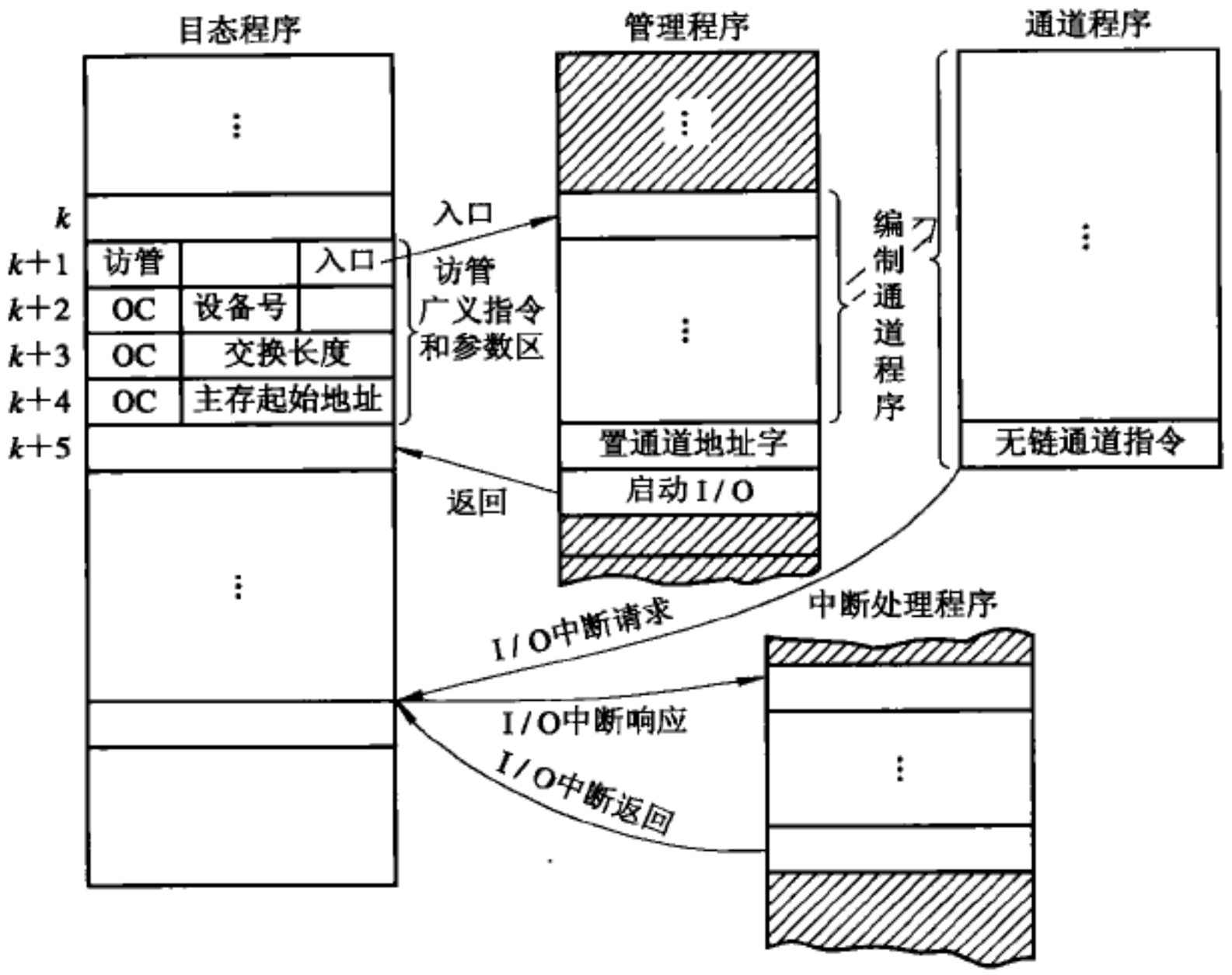


图 3－15　通道处理机输入/输出的主要过程

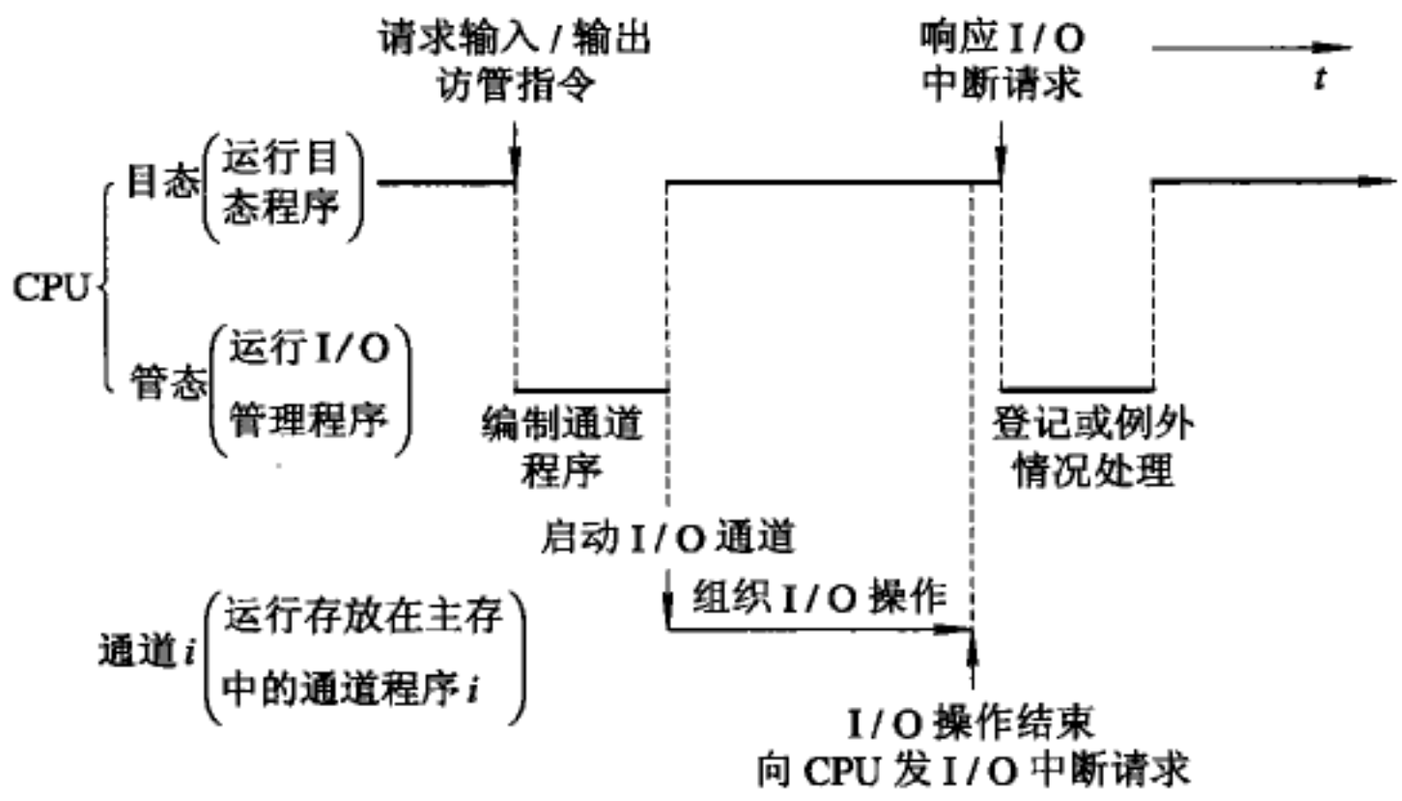


图 3－16　通道处理机输入/输出主要过程的时间关系示意图

管理程序根据广义指令中所提供的参数，如设备号、交换信息的主存起始地址、要交换的信息长度等编制通道程序。通道程序能完成 CPU 一条输入/输出指令所要求执行的许多操作。编制好的通道程序存在主存对应该通道的通道缓冲区中，通道程序的入口地址被置入主存中通道地址字单元，并由管理程序指明操作方式。之后，管理程序就执行“启动 I/O”指令，进入通道开始选择设备期。

“启动 I/O”指令是主要的输入/输出指令，属管态指令，其操作流程如图 3－17 所示。先选择指定的通道、子通道，如它被连通且空闲时，就从主存中取出通道地址字，按通道地址字给出的通道程序首地址，从主存通道缓冲区中取出第一条通道指令。经校验，其格式无误后，再选择相应设备控制器和设备。如该设备是被连着的，就向设备发启动命令。如果设备启动成功，即用全“0”字节回答通道，则结束通道开始选择设备期。

图 3 - 17 “启动 I/O”指令流程

通道被启动后，CPU 退出管态，继续运行目态程序。而通道进入通道数据传送期，执行通道程序，组织 I/O 操作，开始通道与设备间的数据传送。当通道程序执行完无链通道指令后，传送完成，转入通道数据传送结束期，向 CPU 发出 I/O 中断请求。当然。如果出现故障、错误等异常时也向 CPU 发出 I/O 中断请求。CPU 响应此中断请求后，第二次转管态，调出相应管理程序对中断请求进行处理。如属正常结束，就进行登记计费；如属故障、错误，则进行处理。之后，再返回目态，继续目态程序的运行。这样，每完成一次输入/输出

只需两次进管，大大减少了对目态程序的干扰，显著提高了 CPU 运算和外设操作的重叠度。系统中多个通道各自的通道程序可以同时运行，使多种、多台设备可以并行工作。

通道在通道数据传送期里，当所连接的多台设备同时要求交换信息，或者是通道的数据宽度与要传送的信息宽度不等时，还要多次选择当前要传送信息的是哪台设备。即每传送一个数据宽度就要重新选择设备。

根据通道数据传送期中信息传送方式的不同，分为字节多路、数组多路和选择 3 类通道。

字节多路通道适用于连接大量的像光电机那样的字符类低速设备。它们传送一个字符(字节)的时间很短，但字符(字节)间的等待时间很长。因此，通道数据宽度为单字节，以字节交叉方式轮流为多台低速设备服务，以提高效率。字节多路通道又可有多个子通道，各子通道能独立执行通道指令，并行地操作，以字节宽度分时进出通道。接在每个子通道上的多台设备也能分时使用子通道。

数组多路通道适合于连接多台像磁盘那样的高速设备。这些设备的传送速率很高，但传送开始前的寻址辅助操作时间很长。为了充分利用并尽可能重叠各台高速设备的辅助操作时间，不让通道空闲等待，采用成组交叉方式工作。其数据宽度为定长块，传送完 K 个字节数据后就重新选择下个设备。它可有多个子通道，同时执行多个通道程序。所有子通道能分时共享输入/输出通道，但它是以成组交叉方式传送的，既具有多路并行操作的能力，又具有很高的数据传送速率。

选择通道适合于连接优先级高的磁盘等高速设备，让它独占通道，只能执行一道通道程序。数据传送以不定长块方式进行，相当于数据宽度为可变长块，一次对 N 个字节全部传送完。所以，在数据传送期内只选择一次设备。

IBM 370 的通道系统如图 3 - 18 所示，它是 CPU/主存—通道—设备控制器—外设 4 级结构。其三类通道与各种速度的外设相连，形成数据流量平衡的 I/O 系统。

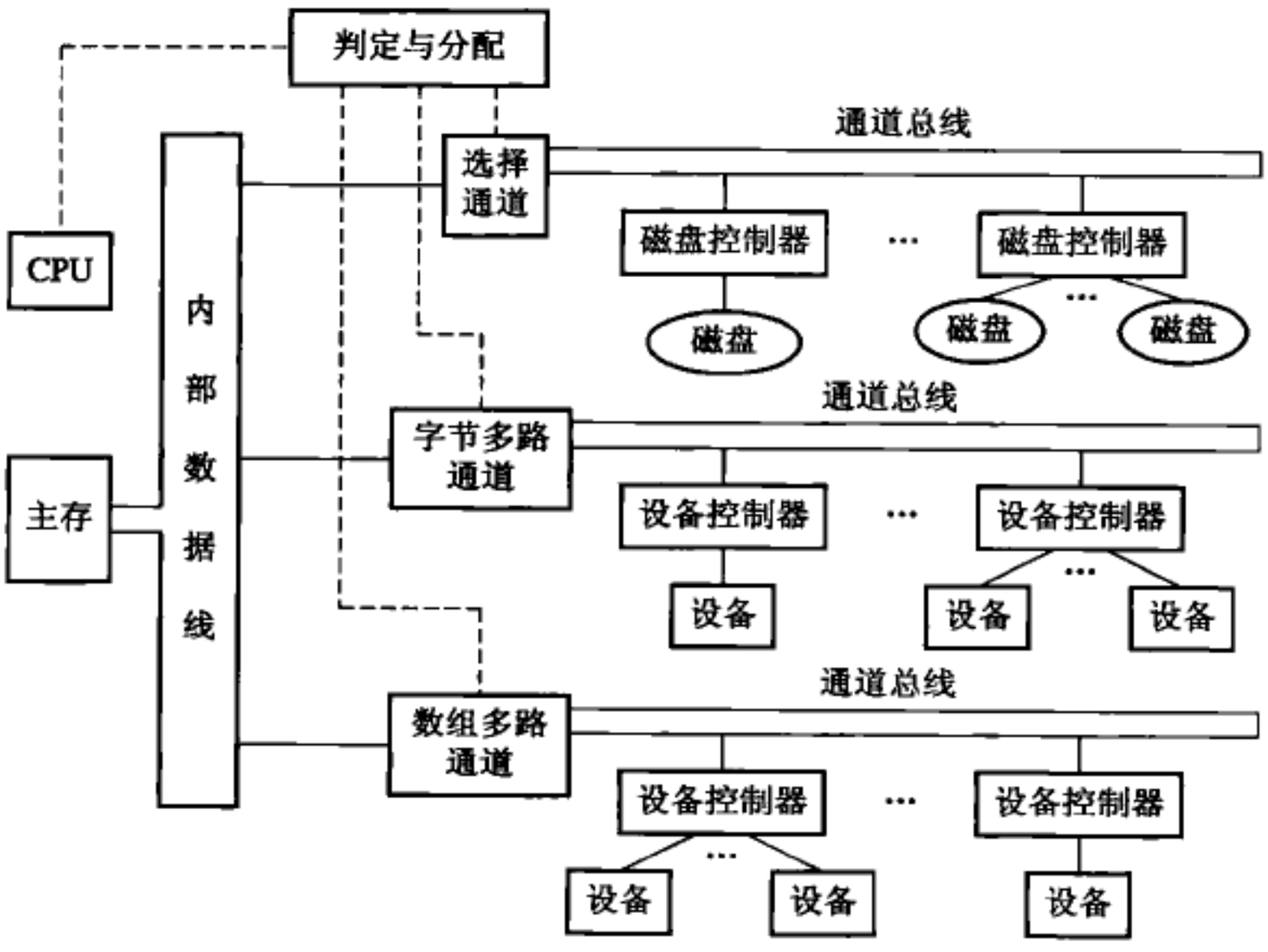


图 3 - 18　IBM 370 的 I/O 结构

2. 通道流量的设计

通道流量是通道在数据传送期内，单位时间内传送的字节数。它能达到的最大流量称为通道极限流量。通道的极限流量与其工作方式、数据传送期内选择一次设备的时间 T_S 和传送一个字节的时间 T_D 的长短有关。

字节多路通道每选择一台设备只传送一个字节，其通道极限流量为

$$f_{max\cdot byte} = \frac{1}{T_S + T_D}$$

数组多路通道每选择一台设备就能传送完 K 个字节。如果要传送 N 个字节，就得分 $\lceil N/K \rceil$ 次传送才行，每次传送都要选一次设备，通道极限流量为

$$f_{max\cdot block} = \frac{K}{T_S + KT_D} = \frac{1}{\frac{T_S}{K} + T_D}$$

选择通道每选择一台设备就把 N 个字节全部传送完，通道极限流量为

$$f_{max\cdot select} = \frac{N}{T_S + NT_D} = \frac{1}{\frac{T_S}{N} + T_D}$$

显然，若通道的 T_S、T_D 一定，且 $N>K$ 时，字节多路方式的极限流量最小，数据多路方式的居中，选择方式的最大。

由通道工作原理可知，当挂上设备后，设备要求通道的实际最大流量与三种通道工作方式有关。以字节多路方式工作的应是该通道所接各设备的字节传送速率之和，即

$$f_{byte\cdot j} = \sum_{i=1}^{p_j} f_{i\cdot j}$$

数组多路和选择方式的应是所接各设备的字节传送速率中之最大者，即

$$f_{block\cdot j} = \max_{i=1}^{p_j} f_{i\cdot j}$$

$$f_{select\cdot j} = \max_{i=1}^{p_j} f_{i\cdot j}$$

式中，j 为通道的编号，$f_{i\cdot j}$ 为第 j 号通道上所挂的第 i 台设备的字节传送速率，p_j 为第 j 号通道中所接设备的台数。

为了保证第 j 号通道上所挂设备在满负荷的最坏情况下都不丢失信息，必须使设备要求通道的实际最大流量不超过通道的极限流量，因此，上述三类通道应分别满足

$$f_{byte\cdot j} \leqslant f_{max\cdot byte\cdot j}$$

$$f_{block\cdot j} \leqslant f_{max\cdot block\cdot j}$$

$$f_{select\cdot j} \leqslant f_{max\cdot select\cdot j}$$

如果 I/O 系统有 m 个通道，其中 1 至 m_1 为字节多路，m_1+1 至 m_2 为数组多路，m_2+1 至 m 为选择，则 I/O 系统的极限流量为

$$f_{max} = \sum_{j=1}^{m_1} f_{max\cdot byte\cdot j} + \sum_{j=m_1+1}^{m_2} f_{max\cdot block\cdot j} + \sum_{j=m_2+1}^{m} f_{max\cdot select\cdot j}$$

必然会满足

$$f_{\max} \geqslant \sum_{j=1}^{m_1} \sum_{i=1}^{p_j} f_{i,j} + \sum_{j=m_1+1}^{m_2} \max_{i=1}^{p_j} f_{i,j} + \sum_{j=m_2+1}^{m} \max_{i=1}^{p_j} f_{i,j}$$

可以用不等式左右两边的差值衡量 I/O 系统流量的利用率。差值越小，其利用率越高，设计越合理。

$f_{\max}$也是 I/O 系统对主存频宽 B_m 的要求。除 I/O 系统外，CPU 也要使用主存。从保持计算机系统各部分频带平衡出发，由 $f_{\max}$根据一定比例可大致估算出主存应达到的频带宽度 B_m。I/O 系统占主存频宽 B_m 中的比例与机器的用途有很大关系。

【例 3-4】 如果通道在数据传送期中，选择设备需 9.8 μs，传送一个字节数据需 0.2 μs。某低速设备每隔 500 μs 发出一个字节数据请求，那么至多可接几台这种低速设备？对于如下 A～F 6 种高速设备，一次通信传送的字节数不少于 1024 个字节，则哪些设备可挂，哪些不能挂？其中，A～F 设备每发一个字节数据传送请求的时间间隔分别如表 3-4 所示。

表 3-4 A～F 设备发请求间隔时间

设备	A	B	C	D	E	F
发请求间隔时间/μs	0.2	0.25	0.5	0.19	0.4	0.21

通道在数据传送期中，低速设备每隔 500 μs 发出一个字节数据传送请求，不难得出，挂低速设备的通道应该是按字节多路通道方式工作的。那么，由于字节多路通道的通道极限流量是

$$f_{\max \cdot byte} = \frac{1}{T_S + T_D}$$

所以，在各设备均被启动后，满负荷的最坏情况下，要想在宏观上不丢失设备的信息，通道极限流量就应大于或等于设备对通道要求的流量 f_{byte}，即应满足

$$f_{\max \cdot byte} \geqslant f_{byte}$$

而在字节多路通道上，设备对通道要求的流量应是所挂全部设备的速率之和。如果字节多路通道上所挂设备台数为 m，设备的速率 f_i 实际上就是设备发出字节传送请求的间隔时间的倒数。m 台相同设备，其速率之和为 mf_i，这样，为不丢失信息，就应满足

$$\frac{1}{T_S + T_D} \geqslant mf_i$$

于是可求得在字节多路通道上能挂的设备台数 m 应满足

$$m \leqslant \frac{1}{(T_S + T_D) \cdot f_i}$$

这样，$m \leqslant \frac{1}{(T_S + T_D) f_i} = \frac{500\ \mu s}{(9.8 + 0.2)\mu s} = 50$ 台。所以，至多可挂 50 台设备。

对于第 2 个问题，A～F 属高速设备，一次通信传送的字节数 n 不少于 1024 个字节，意味着此通道是选择通道。如果通道上挂有 m 台设备，则选择通道的极限流量为

$$f_{\max \cdot select} = \frac{n}{T_S + nT_D} = \frac{1}{\frac{T_S}{n} + T_D} = \frac{1\ B}{\frac{9.8\ \mu s}{n\ B} + 0.2\ \mu s/B}$$

所以，限制通道上所挂的设备速率

$$f_i \leqslant \frac{1}{\frac{9.8}{n}+0.2}\mathrm{B}\cdot\mu\mathrm{s}^{-1}, \quad n \geqslant 1024$$

才行，根据所给出的各台设备每发一个字节数据传送请求的间隔时间，可得各台设备的速率如表 3 - 5 所示。

表 3 - 5　A～F 各台设备的速率

设备	A	B	C	D	E	F
设备速率 f_i/B·μs^{-1}	$\frac{1}{0.2}$	$\frac{1}{0.25}$	$\frac{1}{0.5}$	$\frac{1}{0.19}$	$\frac{1}{0.4}$	$\frac{1}{0.21}$

这样，能满足上述 f_i 不等式要求，只能挂 B、C、E、F 4 台设备，A 和 D 因为超过了 $f_{\max\cdot\text{byte}}$，所以不能挂。

【例 3 - 5】 假设有一个字节多路通道，它有 3 个子通道："0"号、"1"号高速印字机各占一个子通道；"0"号打印机、"1"号打印机和"0"号光电输入机合用一个子通道。假定数据传送期内高速印字机每隔 25 μs 发送一个字节请求，低速打印机每隔150 μs 发送一个字节请求，光电输入机每隔 800 μs 发送一个字节请求，则这 5 台设备要求通道的流量为

$$f_{\text{byte}\cdot j}=\sum_{i=1}^{5} f_{i\cdot j}=\frac{1}{25}+\frac{1}{25}+\left(\frac{1}{150}+\frac{1}{150}+\frac{1}{800}\right)\approx 0.095\ \text{MB/s}$$

根据流量设计的基本要求，该通道的极限流量可设计成0.1 MB/s，即所设计的通道工作周期 $T_S+T_D=10\ \mu s$，这样各设备的请求就都能及时得到响应和处理，不会丢失信息。

通常，高速设备请求的响应优先级也高。让各设备请求得到响应的优先次序定为："0"号高速印字机→"1"号高速印字机→"0"号低速打印机→"1"号低速打印机→"0"号光电输入机。如果各设备要求传送字节数据的请求时刻如图 3 - 19 中的"↑"所示，则由图可见，每台设备都是在发出下一个申请之前或最多是同时就处理完了上次的申请，不会丢码，但各设备处理完每个字节请求的间隔时间并不相等。

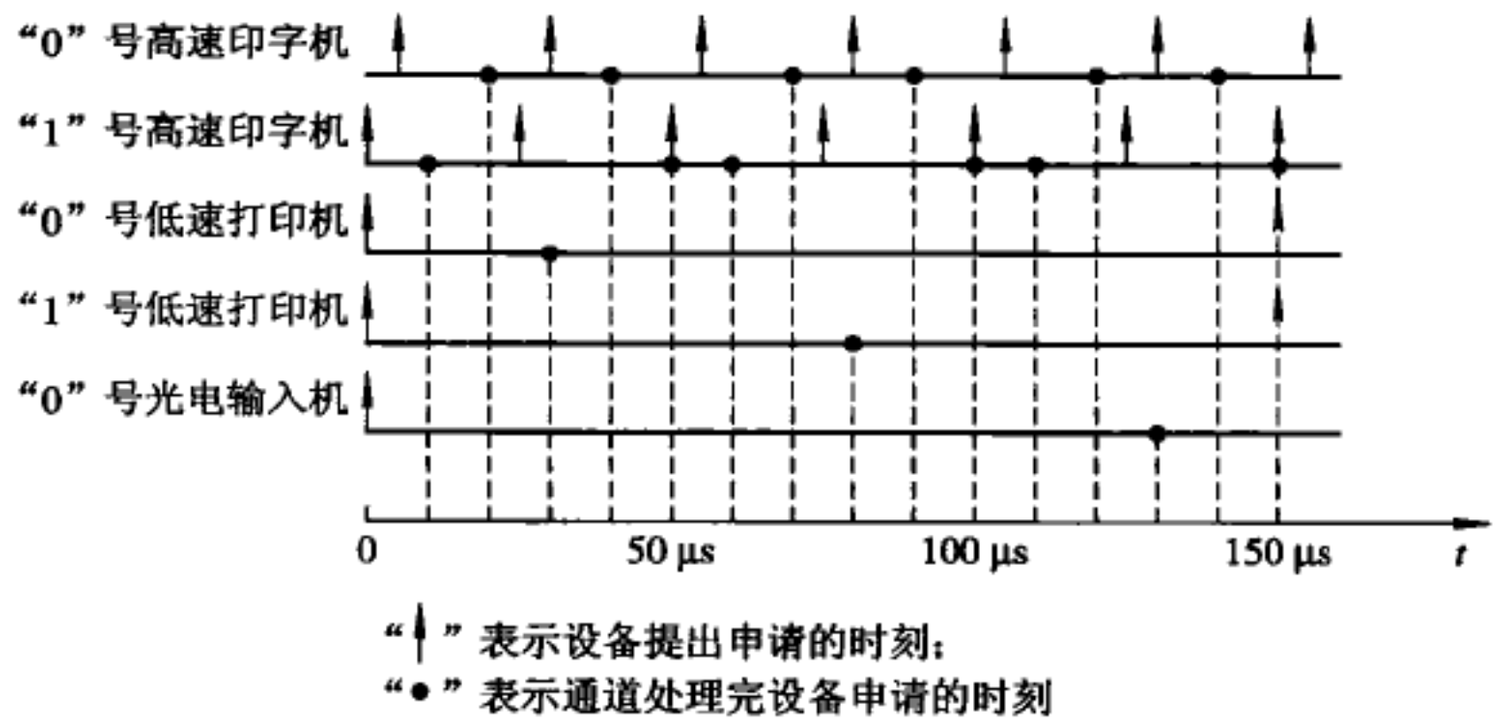


图 3 - 19　字节多路通道响应和处理各设备请求的时间

注意：上述流量设计的基本条件只保证了宏观上不丢失设备信息，并不能保证微观上每一个局部时刻都不丢失信息。特别是当设备要求通道的实际最大流量非常接近于通道极限流量时，由于高速设备频繁发出请求并总是优先得到响应和处理，速率较低的设备就可

能长期得不到通道而丢失信息。为此，可在设备或设备控制器中设置一定容量的缓冲器以缓冲一时来不及处理的信息，或是通过动态提高低速设备的响应优先级来保证从微观上也不丢失信息。但很明显，上述流量设计的基本条件如果得不到满足，则无论设置多大容量的缓冲器或无论怎样改变通道响应设备请求的优先次序也还是要丢失信息的。

3.4.3 外围处理机

通道处理机实际上不能看成是独立的处理机，因为其指令(通道指令)的功能简单，只具有面向外设控制和数据传送的功能，又没有大容量的存储器。就是在输入/输出的过程中，也还需要 CPU 承担以下工作：输入/输出的前处理和后处理，设备或通道出现错误、异常后的处理，对所传送数据信息的代码和格式转换，数据块整体的正确性校验及像文件管理、设备管理等操作系统的工作。另外，为使 CPU 能高速运行所采用的流水等组成技术常会因为遇到输入/输出中断而发挥不了作用，速度严重下降。通道处理机那种每调用一次输入/输出设备就得经“访管”中断转入输入/输出管理程序的做法，不仅妨碍了 CPU 资源的合理利用，也利用不上 CPU 本来具有的高速性能。为此发展了外围处理机(PPU)，希望让 CPU 进一步摆脱对输入/输出操作的控制，以便更好地集中精力专注于自己的事情。

这种外围处理机很接近于一般的处理机，有时干脆就采用一般的通用机。由于它具有较丰富的指令，功能也较强，因此还有利于简化设备控制器，甚至于还可进一步承担起诊断、维修、显示系统工作情况及改善人机界面的功能。

外围处理机基本上是独立于主处理机异步工作的。它可以与主处理机共享主存，也可以不共享主存。像 CDC－CYBER、ASC、B6700 等系统都采用共享主存的连接方式。这种方式的外围处理机存储器(局存)容量较小，外围处理机要执行的例行程序一般放在主存中，为各台 PPU 共享，只有当需要用到时才通过加载或更换覆盖等形式把它调入相应 PPU 的主存储器中。在这种共享主存的连接方式中，有的系统，如 B6700，各 PPU 具有独立的运算部件与主存相连。而另外的系统，像 CDC－CYBER 和 ASC 则是让各 PPU 合用同一运算部件和指令处理部件，并通过公用部件与主存通信，这可以降低外围处理机子系统的造价，但控制较复杂。STAR－100 则属于不共享主存的连接方式，各 PPU 具有更强的独立性，但却需要有很大容量的内存。

采用外围处理机方式，可以自由选择通道和设备进行通信。主存、PPU、通道和设备控制器相互独立，可以视需要用程序动态地控制它们之间的连接，具有比通道处理机方式强得多的灵活性。由于 PPU 是独立的处理机，具有一定的运算功能，可以承担一般的外围运算处理和操作控制任务，还可以让各台外设不必通过主存就可以直接交换信息，这些都进一步提高了整个计算机系统的工作效率。

I/O 处理机功能的进一步扩展已超出单纯进行输入/输出设备管理和数据传送的范围，出现了各种前端机(如网络、远程终端控制前端机)以及后台机(如数据库机器等)。

外围处理机方式就其硬件利用率和成本来讲，不如通道处理机方式好，但随着微处理器和微处理机的迅速发展，不仅功能不断提高和加强，而且成本也在迅速下降。在设计I/O系统时，这种硬件的利用率和成本已不再是着重强调的问题，而是应当考虑怎样才能进一步减少 CPU 对 I/O 系统的介入，充分提高整个系统的功能和性能。为此，进一步增强

输入/输出设备与设备控制器的“智能化”，发展智能外设，让管理、控制操作尽可能在端点完成，使调用外部设备的过程变成是在 I/O 系统中各微处理器之间及各缓冲存储器之间的信息传送过程，这些都将会继续提高 I/O 系统的数据吞吐率并减轻 CPU 输入/输出控制管理负担。

3.5 本章小结

3.5.1 知识点和能力层次要求

(1) 领会并行主存系统的组织形式、极限频宽和实际频宽的关系。领会通过使用并行主存的组成技术提高主存实际频宽的可能性、局限性和发展存储体系的必要性。

(2) 领会为什么要将中断源分成不同的类和级，一般可以将中断源分成哪几类和哪几级。领会设置中断级屏蔽位的作用和中断嵌套的基本原则。熟练掌握按所要求的中断处理(完)的次序来设置各中断处理程序中中断级屏蔽位的状态，并正确画出发生多种中断请求时，CPU 执行程序时的状态转移过程示意图。这部分内容要达到综合应用层次。

(3) 识记专用和非专用总线的定义、优缺点及适用的场合。领会非专用总线的 3 种总线控制方式的总线分配过程、优缺点及所需增加的辅助控制总线线数。领会总线采用同步和异步通信方式的通信过程、优缺点及适用场合。领会数据宽度的定义和它与数据通路宽度定义的区别以及 5 种数据宽度的适用场合。识记在满足性能和流量设计要求的前提下，可采取减少总线线数的办法。

(4) 领会在高性能多用户的计算机系统中，I/O 系统应当是面向操作系统来设计的概念。识记 I/O 系统的三种方式中 I/O 处理机的两种处理方式。

(5) 领会通道方式 I/O 处理机进行输入/输出工作的全过程及通道处理机的工作原理。掌握字节多路、数组多路和选择三类通道各自采用的数据宽度是什么，它们各自适用于什么场合。掌握通道处理机和 I/O 系统的流量计算和分析，字节多路通道流量的计算、通道工作周期的设计。能够画出通道处理机响应和处理完各外设请求的时空示意图。以上这些内容要求达到综合应用层次。

(6) 领会外围处理机的工作原理以及它与通道处理机的不同。

3.5.2 重点和难点

1. 重点

中断为什么要分类和分级；中断处理次序的安排和实现；非专用总线的总线控制方式；数据宽度及其分类；通道的工作过程、流量的分析和设计。

2. 难点

如何按中断处理优先次序的要求，设置各中断处理程序中中断级屏蔽位的状态，正确画出中断处理过程示意图；通道的流量设计；画字节多路通道响应和处理完各外部设备请求的时空图。

习 题 3

3－1 程序存放在模32单字交叉存储器中，设访存申请队的转移概率λ为25%，求每个存储周期能访问到的平均字数。当模数为16呢？由此可得到什么结论？

3－2 设主存每个分体的存取周期为2 μs，宽度为4个字节。采用模m多分体交叉存取，但实际频宽只能达到最大频宽的60%。现要求主存实际频宽为4 MB/s，问主存模数m应取多少方能使两者速度基本适配(m取2的幂)？

3－3 对中断进行分类的根据是什么？这样分类的目的是什么？IBM 370机把中断分为哪几类？

3－4 为什么要将中断类分成优先级？如何分级？IBM 370的中断响应优先次序是什么？

3－5 设中断级屏蔽位“1”对应于开放，“0”对应于屏蔽，各级中断处理程序的中断级屏蔽位设置如表3－6所示。

表3－6 习题3－5中的中断级屏蔽位设置

中断处理程序级别	中断级屏蔽位			
	第1级	第2级	第3级	第4级
第1级	0	0	0	0
第2级	1	0	1	1
第3级	1	0	0	0
第4级	1	0	1	0

(1) 当中断响应优先次序为1→2→3→4时，其中断处理次序是什么？

(2) 设所有的中断处理都各需3个单位时间，中断响应和中断返回时间相对中断处理时间少得多。当机器正在运行用户程序时，同时发生第2、3级中断请求，过两个单位时间后，又同时发生第1、4级中断请求，试画出程序运行过程示意图。

3－6 若机器共有5级中断，中断响应优先次序为1→2→3→4→5，现要求其实际的中断处理次序为1→4→5→2→3，回答下面问题：

(1) 设计各级中断处理程序的中断级屏蔽位(令“1”对应于屏蔽，“0”对应于开放)；

(2) 若在运行用户程序时，同时出现第4、2级中断请求，而在处理第2级中断未完成时，又同时出现第1、3、5级中断请求，请画出此程序运行过程示意图。

3－7 总线控制方式有哪三种？各需要增加几根用于总线控制的控制线？总线控制优先级可否由程序改变？

3－8 简要列举出集中式串行链接、定时查询和独立请求三种总线控制方式的优、缺点。同时分析硬件产生故障时通信的可靠性。

3－9 列举在定时查询方式下进行总线分配，用程序控制优先次序的四种方法以及对应可实现什么样的总线使用优先次序。

3－10　简述字节多路、数组多路和选择通道的数据传送方式。

3－11　某字节多路通道连接6台外设，其数据传送速率分别如表3－7中所列。

表3－7　设备的数据传送速率

设 备 号	传送速率/(KB/s)
1	50
2	15
3	100
4	25
5	40
6	20

(1) 计算所有设备都工作时的通道实际最大流量。

(2) 设计的通道工作周期使通道极限流量恰好与通道实际最大流量相等，以满足流量设计的基本要求，同时让速率越高的设备被响应的优先级越高。当6台设备同时发出请求时，画出此通道在数据传送期内响应和处理各外设请求的时间示意图。由此，能发现什么问题和得出什么结论？

(3) 在问题(2)的基础上，在哪台设备内设置多少个字节的缓冲器就可以避免设备信息丢失？那么，这是否说明书中关于流量设计的基本要求是没有必要的？为什么？

3－12　有8台外设，各设备要求传送信息的工作速率分别如表3－8所示。

表3－8　设备要求传送信息的工作速率

设　　备	工作速率/(KB/s)
A	500
B	240
C	100
D	75
E	50
F	40
G	14
H	10

现设计的通道在数据传送期，每选择一次设备需2 μs，每传送一个字节数据也需要2 μs。

(1) 若用作字节多路通道，通道工作的最高流量是多少？

(2) 作字节多路通道用时，希望同时不少于4台设备挂在此通道上，最好多挂一些，且高速设备尽量多挂一些，请问应选哪些设备挂在此通道上？为什么？

(3) 若用作数组多路通道，通道工作的最高流量是多少？设定长块大小取成512 B。

(4) 作数组多路通道用时，应选哪些设备挂在此通道上？为什么？

3－13　通道型 I/O 系统由一个字节多路通道 A(其中包括两个子通道 A_1 和 A_2)、两个数组多路通道 B_1 和 B_2 及一个选择通道 C 构成，各通道所接设备和设备的数据传送速率如表 3－9 所示。

表 3－9　通道所接设备和设备的数据传送速率

通道号		所接设备的数据传送速率/(KB/s)							
字节多路通道	子通道 A_1	50	35	20	20	50	35	20	20
	子通道 A_2	50	35	20	20	50	35	20	20
数组多路通道 B_1		500		400		350		250	
数组多路通道 B_2		500		400		350		250	
选择通道 C		500		400		350		250	

(1) 分别求出各通道应具有多大设计流量才不丢失信息；

(2) 设 I/O 系统流量占主存流量的 1/2 时才算流量平衡，则主存流量应达到多少？

第4章　存储体系

本章着重讲述存储体系的基本概念，介绍虚拟存储器和 Cache 存储器的原理、虚实地址的映像和变换、替换算法及其实现、影响性能的因素，分析有关软、硬件功能分配中的某些问题，最后讲述存储器的存储保护。

4.1　基本概念

4.1.1　存储体系及其分支

前面已经讲过，为了同时满足存储系统的大容量、高速度和低价格，需要将多种不同工艺的存储器组织在一起。但是，如果不能从逻辑上构成一个完整的整体，充其量只能是一个存储系统，不是我们这里所要介绍的存储体系。

存储体系(即存储层次)是在构成存储系统的几种不同的存储器($M_1 \sim M_n$)之间，配上辅助软、硬件或辅助硬件，使之从应用程序员来看，在逻辑上是一个整体。存储层次的等效访问速度是接近于 M_1 的，容量是 M_n 的，每位价格是接近于 M_n 的。基本的二级存储体系是虚拟存储器和 Cache 存储器，这是存储体系的两个不同的分支。

虚拟存储器是因为主存容量满足不了要求而提出来的。在主存和辅存之间，增设辅助的软、硬件设备，让它们构成一个整体，所以也称之为主存—辅存存储层次，如图 4-1 所示。从 CPU 看，速度是接近于主存的，容量是辅存的，每位价格是接近于辅存的。从速度上看，主存的访问时间约为磁盘的访问时间的 10^{-5}，即快 10 万倍。从价格上看，主存的每位价格约为磁盘的每位价格的 10^3，即贵 1000 倍。如果存储层次能以接近辅存的每位价格去构成等于辅存容量的快速主存，就会大大提高存储系统的性能价格比。应用程序员可用机器指令的地址对整个程序统一编址，称该地址为虚地址(程序地址)，而把实际主存地址称为实地址(实存地址)。当虚存空间(程序空间)远远大于实地址空间(实存空间)时，只需将程序空间分割成较小的段或页，由系统程序按需要调入物理主存，并用辅助映像表建立其虚、实地址空间的对应关系即可。在用虚地址访问主存时，由系统硬件查看这个虚地址所对应单元的内容是否已装入主存。如果该单元已在主存内，就将虚地址变换成主存实地址去访问主存；如果不在主存内，就经辅助软、硬件将包含所要访问的单元在内的那个段(或页)的程序块由辅存调入主存，建立好映像关系，再进行访问。这样，不论是虚、实地址的变换还是

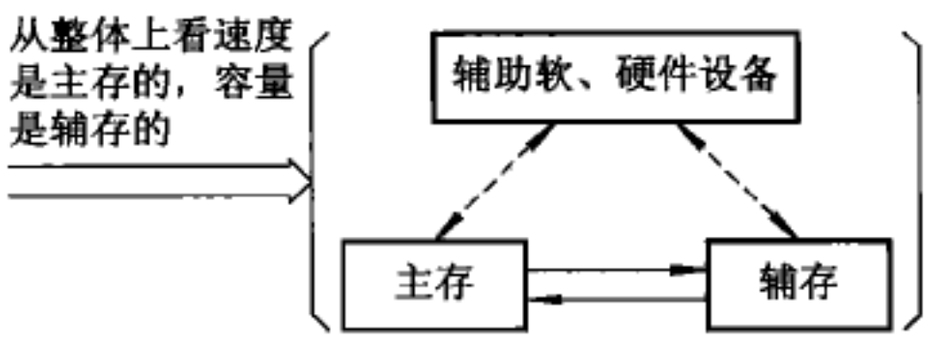


图 4-1　主存—辅存存储层次

程序由辅存调入主存都不必由应用程序员安排，即这些操作和辅助软、硬件对应用程序员来讲都是透明的。事实上，只要是存储层次，这些都必须对应用程序员是透明的。

因主存速度满足不了要求而引出了Cache存储器。在CPU和主存之间增设高速、小容量、每位价格较高的Cache，用辅助硬件将其和主存构成整体，如图4－2所示，称之为Cache存储器(或称为Cache—主存存储层次)。从CPU看，其速度接近于Cache的，容量是主存的，每位价格接近于主存的。由于CPU与主存的速度只差一个数量级，信息在Cache与主存之间的传送就只能全部用辅助硬件实现，因此，Cache存储器不仅对应用程序员是透明的，而且对系统程序员也是透明的。

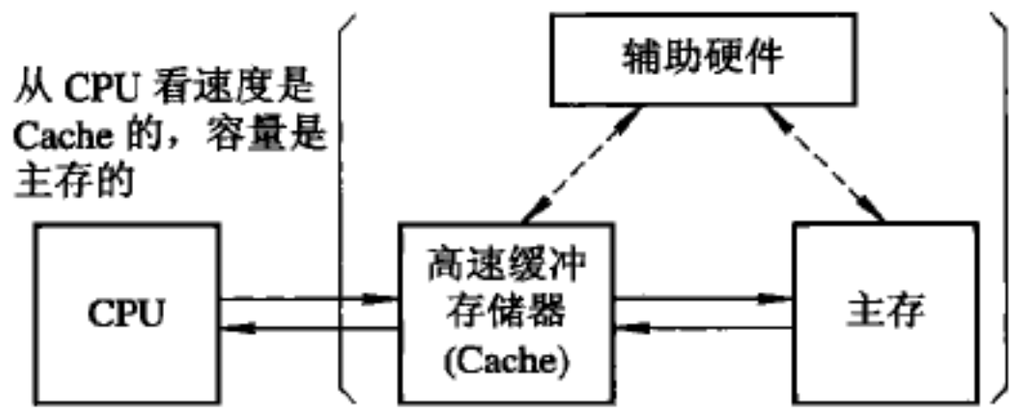


图 4－2　Cache—主存存储层次

由二级存储层次可组合成如图4－3所示的多级存储层次。从CPU看，它是一个整体，有接近于最高层 M_1 的速度，最低层 M_n 的容量，并有接近于最低层 M_n 的每位价格。

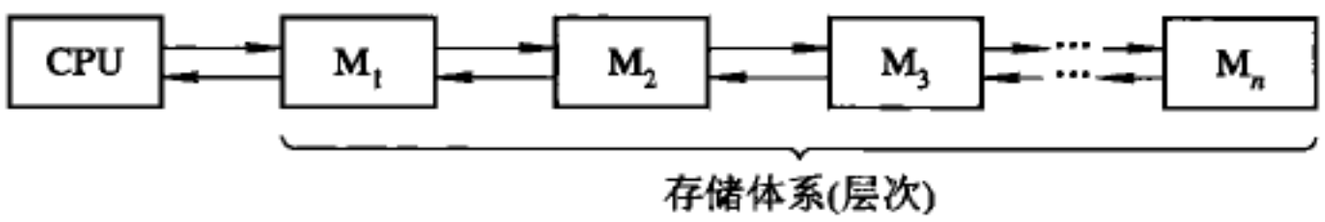


图 4－3　多级存储层次

4.1.2　存储体系的构成依据

为了使存储体系能有效地工作，当CPU要用到某个地址的内容时，总希望它已在速度最快的 M_1 中，这就要求未来被访问信息的地址能预知，这对存储体系的构成是非常关键的。这种预知的可能性基于计算机程序具有局部性，它包括时间上的局部性和空间上的局部性。前者指的是在最近的未来要用到的信息很可能是现在正在使用的信息，这是因为程序存在循环。后者指的是在最近的未来要用到的信息很可能与现在正在使用的信息在程序空间上是邻近的，这是因为指令通常是顺序存放、顺序执行，数据通常是以向量、阵列、树形、表格等形式簇聚地存放的。所以，程序执行时所用到的指令和数据是相对簇聚成自然的块或页(存储器中较小的连续单元区)。这样，层次的 M_1 级不必存入整个程序，只需将近期用过的块或页(根据时间局部性)存入即可。在从 M_2 级取所要访问的字送 M_1 时，一并把该字所在的块或页整个取来(根据空间的局部性)，就能使要用的信息已在 M_1 的概率显著增大。这是存储层次构成的主要依据。

预知的准确性是存储层次设计好坏的主要标志，很大程度取决于所用算法和地址映像变换的方式。一旦被访问信息不在 M_1 中，原先申请访存的程序就暂停执行或被挂起，直到所需信息被调到 M_1 为止。这指的是虚拟存储器。若 M_1 为Cache，则不将程序挂起，只是暂停执行，等待信息调入 M_1。同时为缩短CPU的空等时间，还让CPU不只与Cache，也和主存有直接通路。就是说，虚拟存储器只适用于多道程序(多用户)环境，而Cache存储器既可以用于单用户环境也可以用于多用户环境。

4.1.3 存储体系的性能参数

为简单起见，以图 4－4 所示的二级存储体系(M_1，M_2)为例来分析。设 c_i 为 M_i 的每位价格，S_{M_i} 为 M_i 的以位计算的存储容量，T_{A_i} 为 CPU 访问到 M_i 中的信息所需的时间。为评价存储层次性能，引入存储层次的每位价格 c、命中率 H 和等效访问时间 T_A。

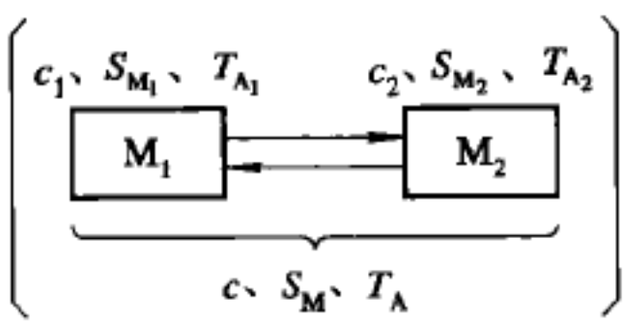


图 4－4　二级存储体系的评价

存储层次的每位价格为

$$c = \frac{c_1 \cdot S_{M_1} + c_2 \cdot S_{M_2}}{S_{M_1} + S_{M_2}}$$

总希望存储层次的每位价格能接近于 c_2，为此应使 $S_{M_1} \ll S_{M_2}$。同时，上式中并未把采用存储体系所增加的辅助软、硬件价格计算在内，所以要使 c 接近于 c_2，还应限制所增加的这部分辅助软、硬件价格只能是总价格中一个很小的部分，否则将显著降低存储体系的性能价格比。

命中率 H 定义为 CPU 产生的逻辑地址能在 M_1 中访问到(命中到)的概率。命中率可用实验或模拟方法求得，即执行或模拟一组有代表性的程序，若逻辑地址流的信息能在 M_1 中访问到的次数为 R_1，当时在 M_2 中还未调到 M_1 的次数为 R_2，则命中率 $H = R_1/(R_1+R_2)$。显然命中率 H 与程序的地址流、所采用的地址预判算法及 M_1 的容量都有很大关系。我们总希望 H 越大越好，即 H 越接近于 1 越好。相应地，不命中率或失效率是指由 CPU 产生的逻辑地址在 M_1 中访问不到的概率。对二级存储层次，失效率为 $1-H$。

存储层次的等效访问时间 $T_A = HT_{A_1} + (1-H)T_{A_2}$。希望 T_A 越接近于 T_{A_1}，即存储层次的访问效率 $e = T_{A_1}/T_A$ 越接近于 1 越好。

设 CPU 对存储层次相邻二级的访问时间比 $r = T_{A_2}/T_{A_1}$，则

$$e = \frac{T_{A_1}}{T_A} = \frac{T_{A_1}}{HT_{A_1} + (1-H)T_{A_2}} = \frac{1}{H + (1-H)r}$$

据此，可得 $e = f(r, H)$的关系如图 4－5 所示。

由图 4－5 可知，要使访问效率 e 趋于 1，在 r 值越大时，就要求命中率 H 越高。例如，$r=100$ 时，为使 $e>0.9$，必须使 $H>0.998$；而当 $r=2$ 时，只需 $H>0.889$ 即可。例如主－辅层次的主、辅存速度比达 10^5，要求 H 极高才能有高的访问效率，但 H 很难极高。为了降低对 H 的要求，可以减小相邻二级存储器的访问速度比，还可减小相邻二级存储器的容量比，也能提高 H，但这与为降低每位平均价格而要求容量比要大相矛盾。因此，在主、辅存之间增加一级电子磁盘，使级间 r 值不会过大，有利于降低对 H 的要求，以获得同样的 e。所以要想使存储层次的访问效率 e 趋于 1，就要在选择具有高命中率的算法、相邻二级的容量比和速度比及增加的辅助软、硬件的代价等因素间综合权衡，进行优化设计。

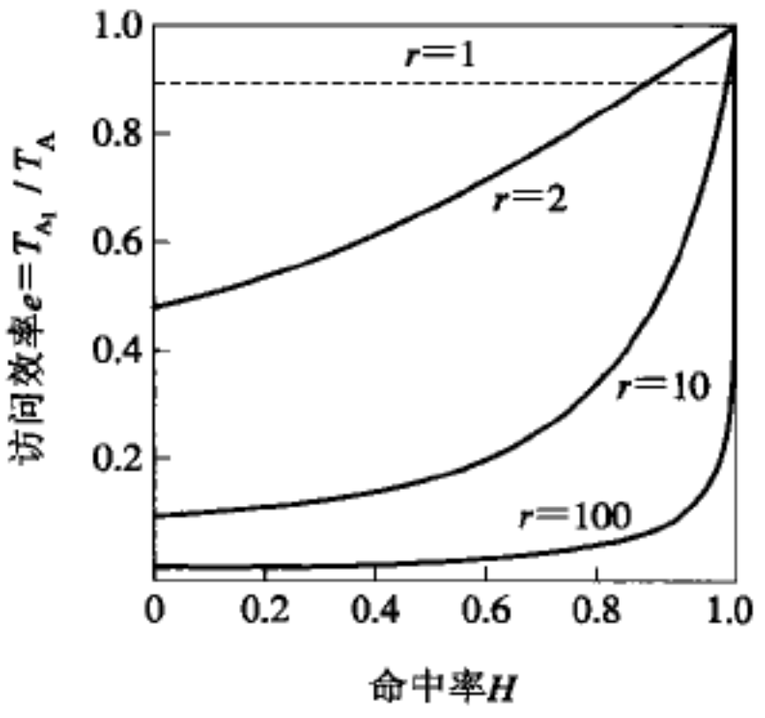


图 4－5　对于不同的 r，命中率 H 与访问效率 e 的关系

4.2 虚拟存储器

4.2.1 虚拟存储器的管理方式

虚拟存储器通过增设地址映像表机构来实现程序在主存中的定位。将程序分割成若干个段或页，用相应的映像表指明该程序的某段或某页是否已装入主存。若已装入，同时指明其在主存中的起始地址；若未装入，就去辅存中调段或调页，将其装入主存后再在映像表中建立好程序空间和实存空间的地址映像关系。这样，程序执行时先查映像表，将程序(虚)地址变换成实(主)存地址后再访主存。

根据存储映像算法的不同，可有多种不同存储管理方式的虚拟存储器，其中主要有段式、页式和段页式三种。

1. 段式管理

程序都有模块性，一个复杂的大程序总可以分解成多个在逻辑上相对独立的模块。这些模块可以是主程序、子程序或过程，也可以是数据块。模块的大小各不相同，有的甚至事先无法确定。每个模块都是一个单独的段，都以该段的起点为 0 相对编址。当某个段由辅存调入主存时，只要系统赋予该段一个基址(即该段存放在主存中的起始地址)，就可以由此基址和单元在段内的相对位移形成单元在主存中的实际地址。将主存按段分配的存储管理方式称为段式管理。

为了进行段式管理，每道程序在系统中都有一个段(映像)表来存放该道程序各段装入主存的状况信息。参看图 4 - 6，段表中的每一项(对应表中的每一行)描述该道程序一个段的基本状况，由若干个字段提供。段名字段用于存放段的名称，段名一般是有其逻辑意义的，也可以转换成用段号指明。由于段号从 0 开始顺序编号，正好与段表中的行号对应，如 2 段必是段表中的第 3 行，这样，段表中就可不设段号(名)字段。装入位字段用来指示该段是否已经调入主存，“1”表示已调入，“0”表示未调入。在程序的执行过程中，各段的装入位随该段是否活跃而动态变化。当装入位为“1”时，地址字段用于表示该段装入主存中的起始(绝对)地址；当装入位为“0”时，则无效(有的机器用它表示该段在辅存中的起始地址)。段长字段指明该段的大小，一般以字数或字节数为单位，取决于所用的编址方式。段长字段是供判断所访问的地址是否越出段界的界限保护检查用的。访问方式字段用来标记该段允许的访问方式，如只读、可写、只能执行等，以提供段的访问方式保护。除此之外，段表中还可以根据需要设置其他的字段。段表本身也是一个段，一般常驻在主存中，也可以存在辅存中，需要时再调入主存。

假设系统在主存中最多可同时有 N 道程序，可设 N 个段表基址寄存器。对应于每道程序，由基号(程序号)指明使用哪个段表基址寄存器。段表基址寄存器中的段表基地址字段指向该道程序的段表在主存中的起始地址。段表长度字段指明该道程序所用段表的行数，即程序的段数。由系统赋予某道程序(用户、进程)一个基号，并在调入/调出过程中对有关段表基址寄存器和段表的内容进行记录和修改，所有这些对用户程序员都是透明的。某道活跃的程序在执行过程中产生的指令或操作数地址只要与基号组合成系统的程序地址，即可通过查表

自动转换成主存的物理地址。图 4－6 示意性地表示了这一地址变换的过程。

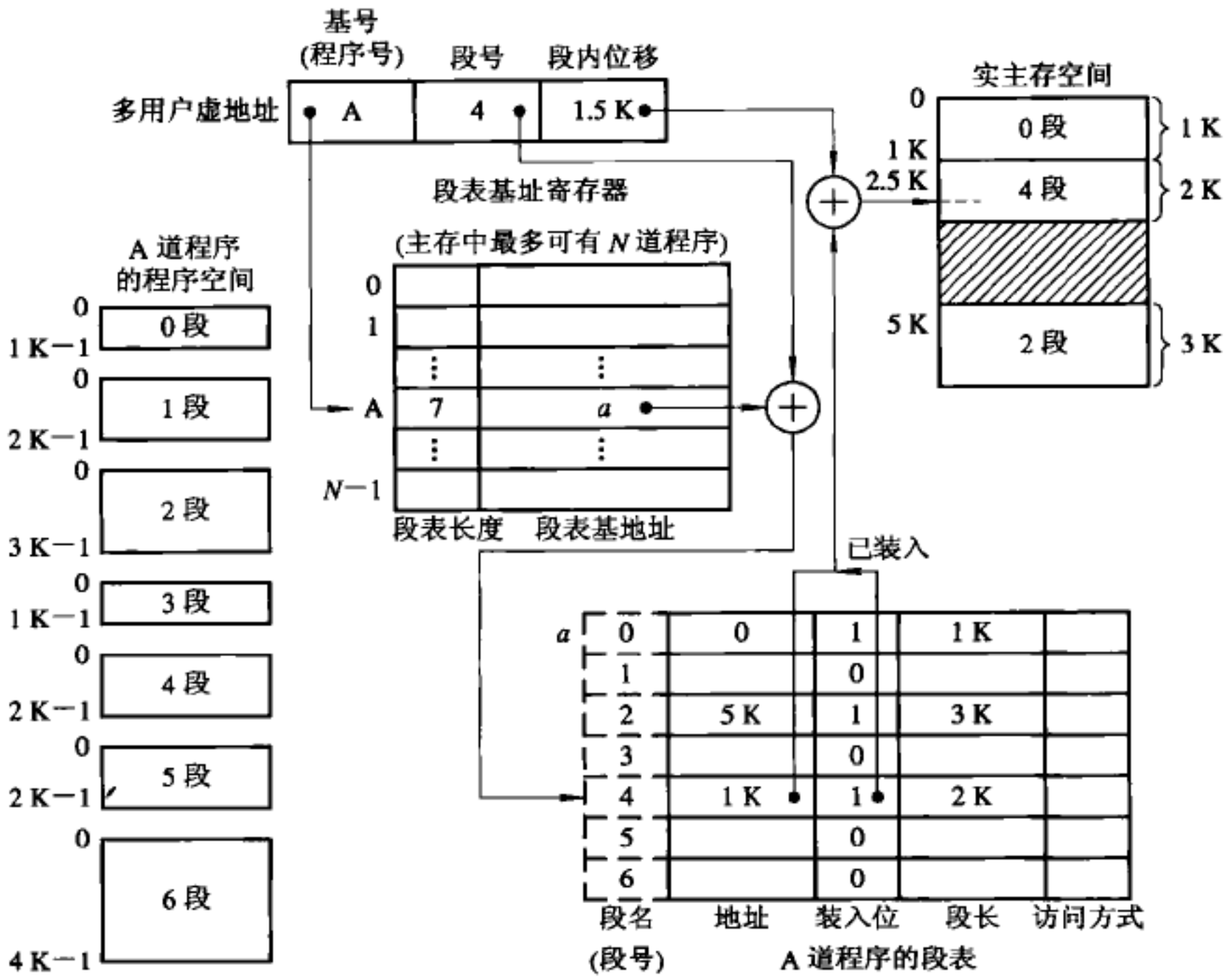


图 4－6　段式管理的定位映像机构及地址的变换过程

分段方法能使大程序分模块编制，从而可使多个程序员并行编程，缩短编程时间，在执行或编译过程中对不断变化的可变长段也便于处理。各个段的修改、增添并不影响其他各段的编制，各用户以段的连接形成的程序空间可以与主存的实际容量无关。

分段还便于几道程序共用已在主存内的程序和数据，如编译程序、各种子程序、各种数据和装入程序等，不必在主存中重复存储，只需把它们按段存储，并在几道程序的段表中设置其公用段的名称及同样的基址值即可。

由于各段是按其逻辑特点组合的，因而容易以段为单位实现存储保护。例如，可以安排成常数段只能读不能写；操作数段只能读或写，不能作为指令执行；子程序段只能执行，不能修改；有的过程段只能执行，不能读也不能写，如此等等。一旦违反规定就中断，这对发现程序设计错误和非法使用是很有用的。

段式管理的虚拟存储器由于各个段的长度完全取决于段自身，因此不会恰好如图 4－6 所示的那样是 1 K 的整数倍，段在主存中的起点也会是随意的，这就给高效地为调入段分配主存区域带来困难。为了进行段式管理，除了系统需要为每道程序分别设置段映像表外，还得由操作系统为整个主存系统建立一个实主存管理表，它包括占用区域表和可用区域表两部分。占用区域表的每一项(行)用来指明主存中哪些区域已被占用，被哪道程序的哪个段占用以及该段在主存的起点和长度。此外，还可以设置标识该段是否进入主存后被改写过的字段，以便决定该段由主存中释放时，是否还要将其写回到辅存中原先的位置来减少辅助操作。可用区域表的每一项(行)则指明每一个未被占用区的基地址和区域大小。当一个段从辅存装入主存时，操作系统就在占用区域表中增加一项，并修改可用区域表。

而当一个段从主存中退出时，就将其在占用区域表的项(行)移入可用区域表中，并进行有关它是否可与其他可用区归并的处理，修改可用区域表。当某道程序全部执行结束或者是被优先级更高的程序所取代时，也应将该道程序的全部段的项从占用区域表移入可用区域表并作相应的处理。

2. 页式管理

段式存储中各段装入主存的起点是随意的，段表中的地址字段很长，必须能表示出主存中任意一个绝对地址，加上各段长度也是随意的，段长字段也很长，这既增加了辅助硬件开销，降低了查表速度，也使主存管理麻烦。段式管理和存储还会带来大的段间零头浪费。例如，主存中已有 A、B、C 三个程序，其大小和位置如图 4－7 所示，现有一长度为 12 KB 的 D 道程序想要调入。段式管理时尽管 D 道程序长度小于主存所有可用区零头总和 16 KB，但没有哪一个零头能装得下它，所以无法装入。于是提出了页式存储。

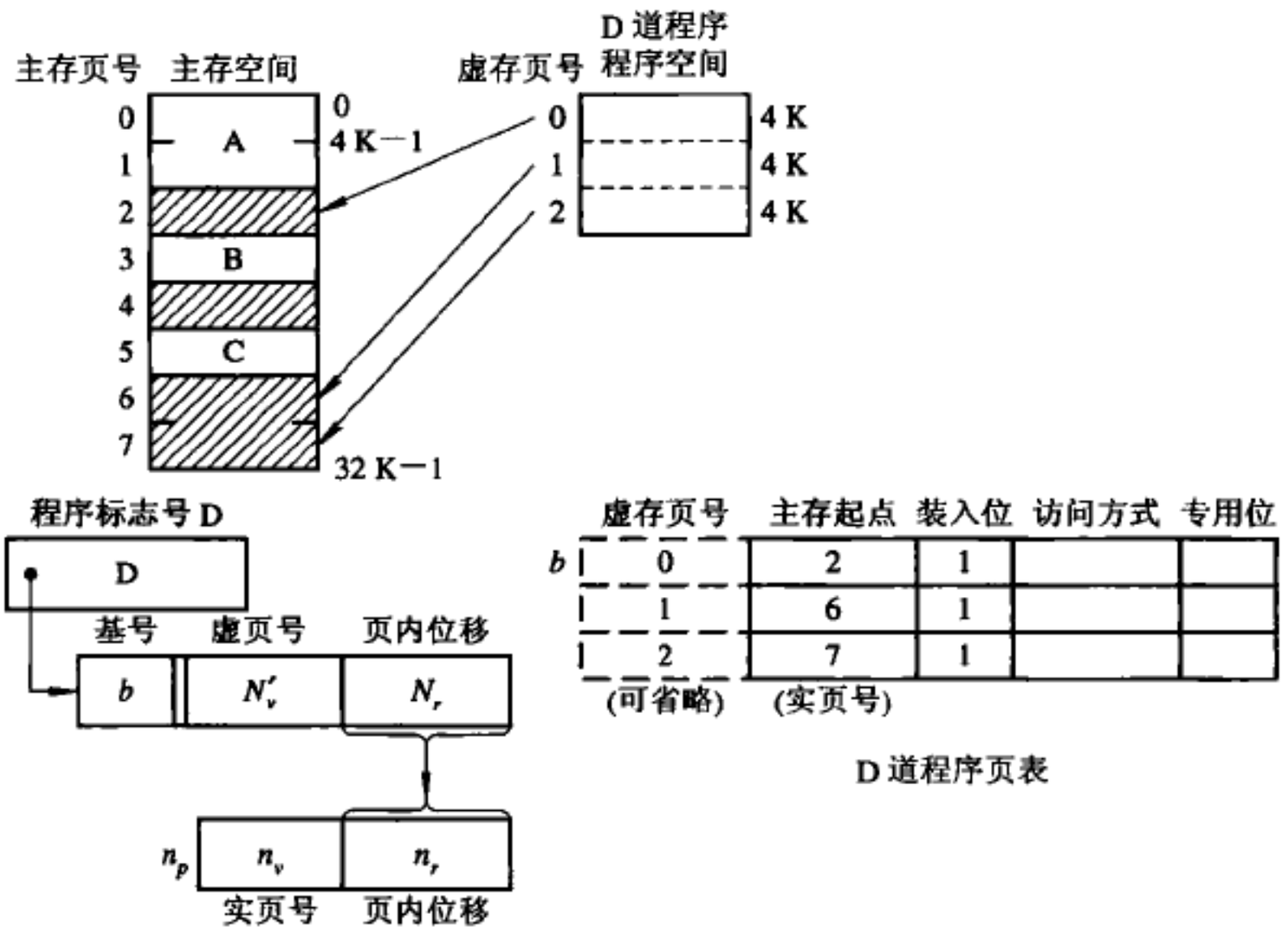


图 4－7　采用页式存储后 D 道程序仍可装入

页式存储是把主存空间和程序空间都机械等分成固定大小的页(页面大小随机器而异，一般在 512 B 到几 KB 之间)，按页顺序编号。这样，任一主存单元的地址 n_p 就由实页号 n_v 和页内位移 n_r 两个字段组成。每个独立的程序也有自己的虚页号顺序。如此例中，若页面大小取 4 KB，则独立编址的 D 程序就有 3 页长，页号为 0～2。如果虚存中的每一页均可装入主存中任意的实页位置，如图 4－7 所示，那么 D 程序中各页就可分别装入主存的第 2、6、7 三个实页位置，只要系统设置相应的页(映像)表，保存好虚页装入实页时的页面对应关系，就可由给定的程序(虚)地址查页表，变换成相应的实(主)存地址访存。

由于页式存储中程序的起点必处于一个页面的起点，用户程序中每一个虚地址就由虚页号字段 N_v' 和页内位移字段 N_r 组成。而虚存和实(主)存的页面大小又一样，所以页表中只需记录虚页号 N_v' 和实(主)存页号 n_v 的对应关系，不用保存页内位移。而虚页号与页表的行号是对应的，如虚页号 2 必对应于页表中第 3 行，所以不用专设虚页号字段。页面大小固定，页长字段也省了。所有这些都简化了映像表硬件，也利于加快查表。当然与段表

类似，页表也必须设置装入位字段以表示该页是否已装入主存。当装入位为“1”时，实页号字段中的内容才是有效的，否则无效。为便于存储保护，页表中也可设置相应的访问方式字段等。可以看出，对于由虚地址查表变换成实地址过程，段式管理需要较长的加法器进行将段起始地址加上段内位移的操作，而页式管理只需将主存实页号与页内位移拼装在一起即可，大大加快了地址变换的速度，也利于提高形成实地址的可靠性。

假设系统内最多可在主存中容纳 N 道程序，对每道程序都将有一个页表。由用户标识号 u 指明该道程序使用哪个页表基址寄存器，从而可以找到该道程序的页表在主存中的起点。就整个多用户虚拟存储器的虚存空间来说，其虚地址应有用户(进程、程序)标识号 u、虚页号 N_v' 和页内位移 N_r 三个字段。如同段式管理一样，在程序装入和运行过程中，页表基址寄存器和页表的内容全部由存储层次来完成设置和修改，对用户完全是透明的。图 4－8 示意出页式管理的定位映像机构及其虚、实地址的变换过程。

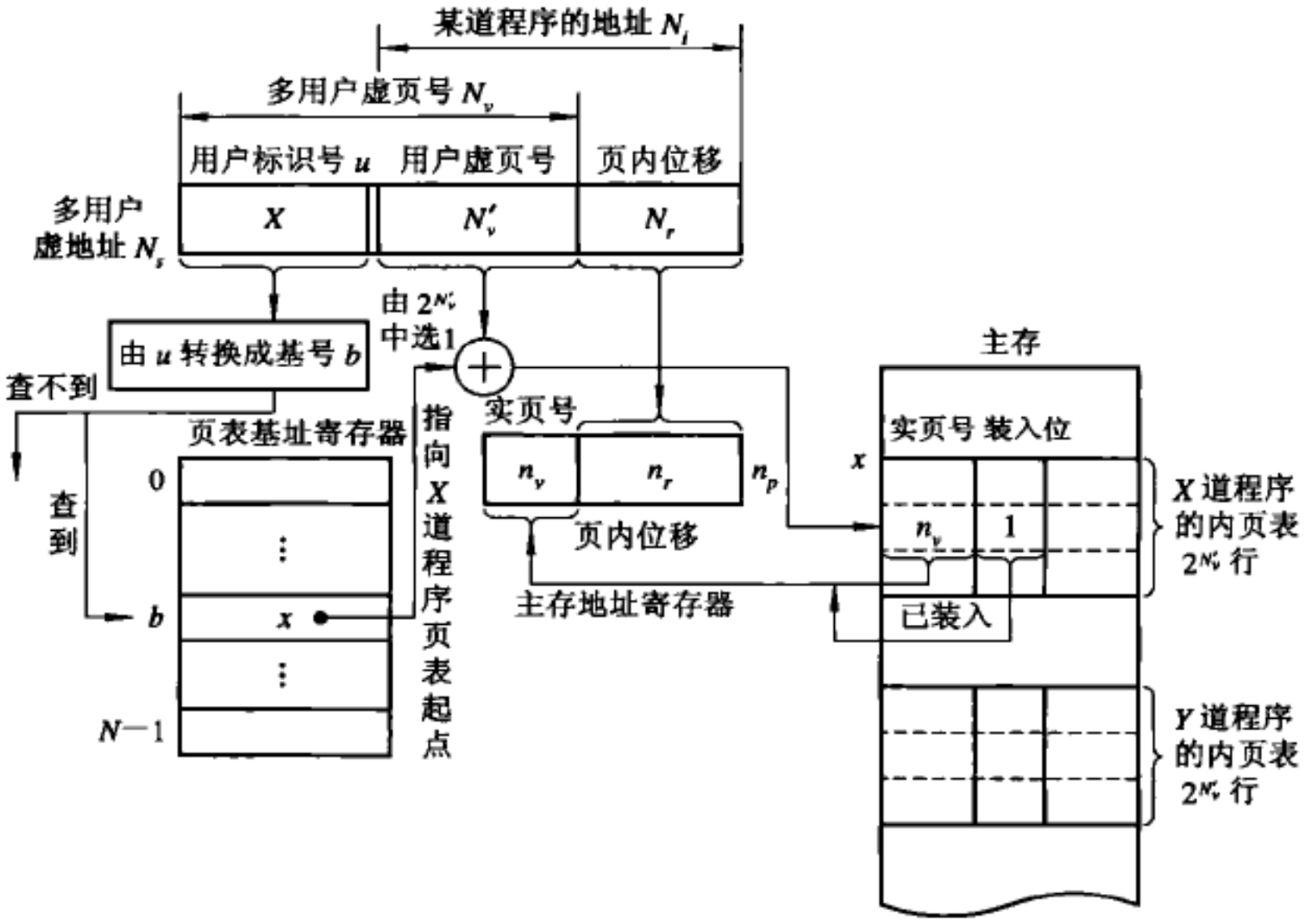


图 4－8　页式管理的定位映像机构及其虚、实地址的变换过程

为了对整个主存空间进行管理，与段式管理类似，页式管理系统中还应设置专门的主存页面管理表，以指明主存中每个页面位置的使用状况及其他信息，关于这一点我们将在下一节介绍。

3. 段页式管理

从以上介绍中可以看出，段式和页式虚拟存储器在许多方面是不同的，因而各有不同的优缺点。页式管理对应用程序员完全透明，所需映像表硬件较少，地址变换的速度快，调入操作简单，这些方面都优于段式管理。但页式管理不能完全消除主存可用区的零头浪费，因为程序的大小不可能恰好就是页面大小的整数倍。产生的页内零头虽然无法利用，但其浪费比段式管理的要小得多，所以在主存空间利用率上，页式管理也优于段式管理。因此，单纯用段式管理的虚拟存储器已很少见到。

然而，相比而言，段式管理也具有页式管理所没有的若干优点，例如：段式管理中每个段独立，有利于程序员灵活实现段的链接、段的扩大/缩小和修改，而不影响到其他的

段；每段只包含一种类型的对象，如过程或是数组、堆栈、标量等集合，易于针对其特定类型实现保护；把共享的程序或数据单独构成一个段，易于实现多个用户、进程对共用段的管理，等等。如果采用页式管理，要做到这些就比较困难。因此，为取长补短，提出了将段式管理和页式管理相结合的段页式存储和管理。

段页式存储是把实(主)存机械等分成固定大小的页，程序按模块分段，每个段又分成与实(主)存页面大小相同的页。每道程序通过一个段表和相应的一组页表进行定位。段表中的每一行对应一个段。其中，“装入位”表示该段是否已装入主存。若未装入主存，则访问该段时将引起段失效故障，请求从辅存中调入页表。若已装入主存，则地址字段指出该段的页表在主存中的起始地址。“访问方式”字段指定对该段的控制保护信息。“段长”字段指定该段页表的行数。每一个段都有一个页表。页表中每一行用装入位指明此段该页是否已装入主存。若未装入主存，则访问该页时将引起页面失效故障，需从辅存调页。如果已装入主存，则用地址字段指明该页在主存中的页号。此外，页表中还可以包含一些其他信息。段页式与纯段式的主要差别是段的起点不再是任意的，而必须是主存中页面的起点。

对于多道程序来说，每道程序(用户或进程)都需要有一个用户标志号 u(转换成基号 b)以指明该道程序的段表起点存放在哪个基址寄存器中。这样，多用户虚地址就由用户标志 u、段号 s、页号 p、页内位移 d 四个字段组成。设系统中主存最多可容纳 N 道程序。图 4－9 表示采用段页式管理的定位映像机构及由多用户虚地址变换成主存实地址的过程。不少大、中型机都采用这种段页式存储。

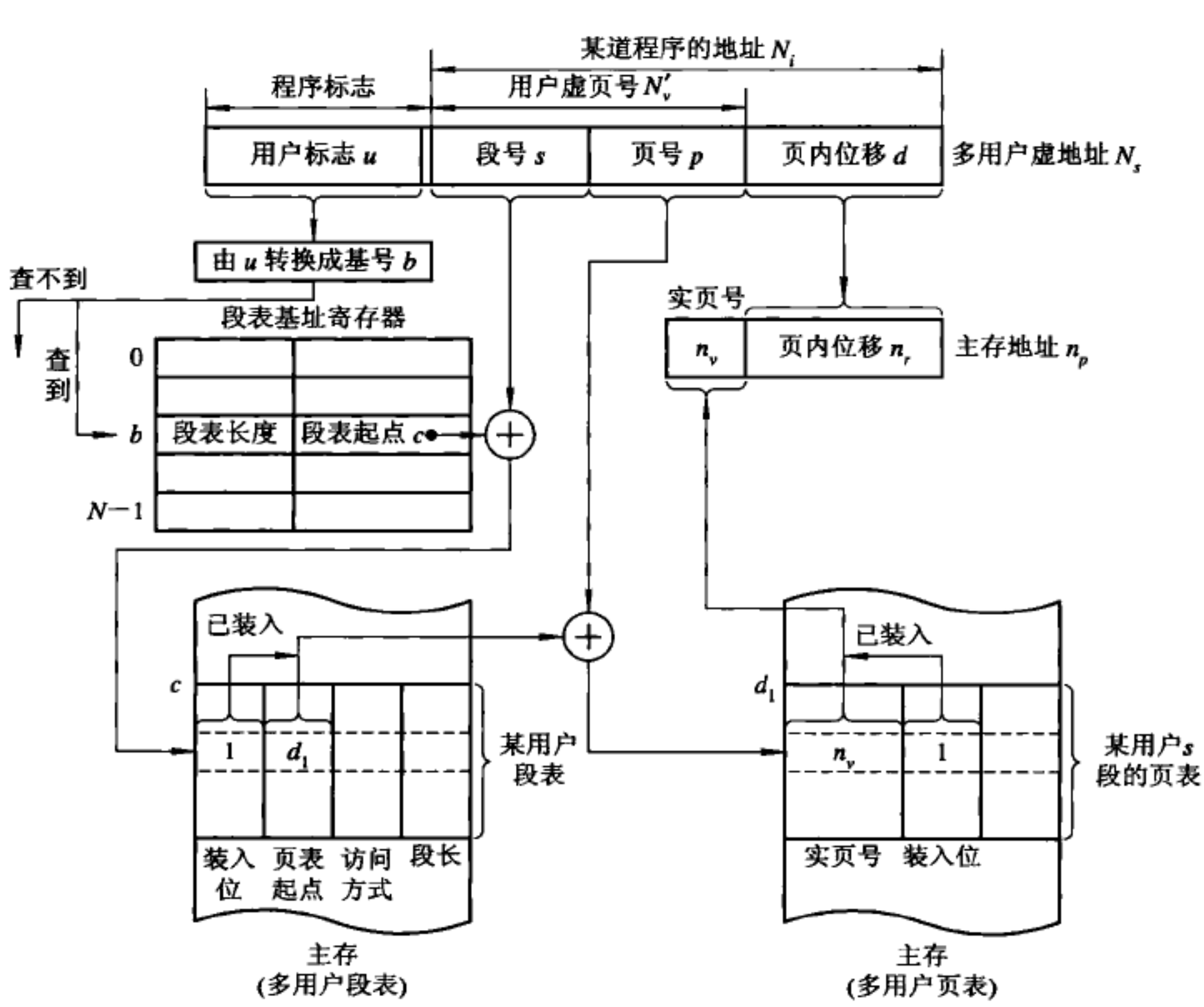


图 4－9　段页式管理的定位映像机构及其地址的变换过程

在虚拟存储器中每访问一次主存都要进行一次程序地址向实(主)存地址的转换。段页式的主要问题是地址变换过程至少需要查表两次，即查段表和页表。因此，要想使虚拟存储器的速度接近于主存，必须在结构上采取措施加快地址转换中查表的速度。

4.2.2 页式虚拟存储器的构成

1. 地址的映像和变换

前面已讲过，页式虚拟存储器是采用页式存储和管理的主存一辅存存储层次。它将主存空间和程序空间都按相同大小机械等分成页，并让程序的起点总是处在页的起点上。程序员用指令地址码 N_i 来编写每道程序，N_i 由用户虚页号 N_v' 和页内地址 N_r 组成。主存地址则分成实页号 n_v 与页内位移 n_r 两部分，其中 n_r 总是与 N_r 一样。大多数虚拟存储器中每个用户的程序空间可比实际主存空间大得多，即一般有 $N_v'>n_v$。这样，虚拟存储器系统总的多用户虚地址 N_s 就由用户标志 u、用户的虚页号 N_v' 及页内地址 N_r 三部分构成，总的虚存空间是 $2^{u+N_v'}$ 个页。可将 u 和 N_v' 合并成多用户虚页号 N_v，这时 $2^{N_v}\gg 2^{n_v}$。它们各部分的地址对应关系如图 4 - 10 所示。如果主存最多可存 N 道程序，即 N 个用户，则其他用户放在辅存中。因此，虚拟存储器工作过程中总存在着的问题是如何把大的多用户虚存空间压缩装入到小的主存空间，在程序运行时又如何将多用户虚地址 N_s 变换成主存地址 n_p。这就是本小节要讲述的地址映像和变换问题。

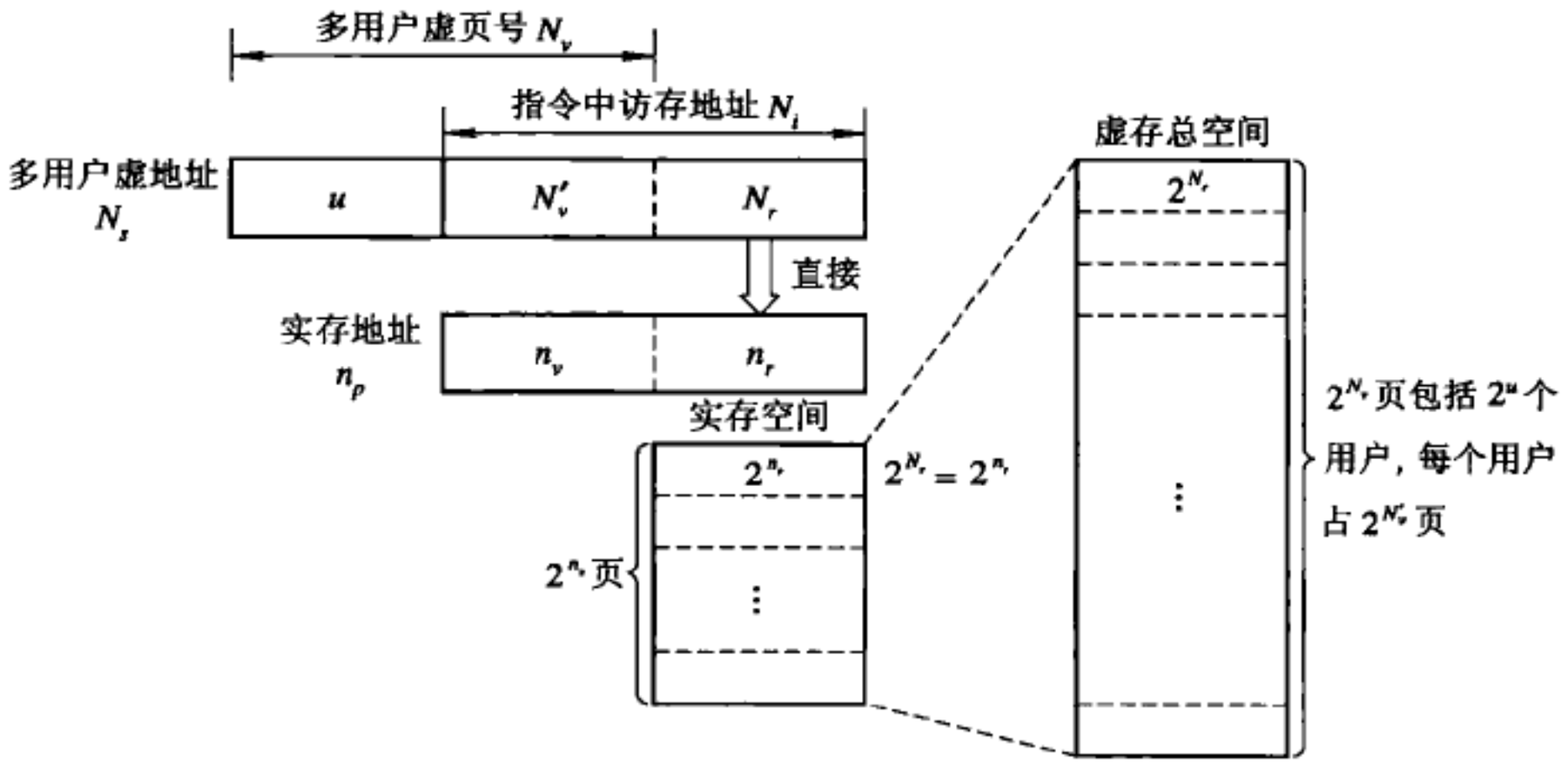


图 4 - 10　虚、实地址对应关系及空间的压缩

地址的映像是指将每个虚存单元按什么规则(算法)装入(定位于)实(主)存，建立起多用户虚地址 N_s 与实(主)存地址 n_p 之间的对应关系。对页式管理而言，就是指多用户虚页号为 N_v 的页可以装入主存中的哪些页面位置，建立起 N_v 与 n_v 的对应关系。地址的变换是指程序按照这种映像关系装入实存后，在执行中，如何将多用户虚地址 N_s 变换成对应的实地址 n_p。对页式管理而言，就是如何将多用户虚页号 N_v 变换成实页号 n_v。地址的变换与所采用的地址映像规则密切相关，因此结合在一起来讲述。

由于是把大的虚存空间压缩到小的主存空间，因此主存中的每一个页面位置应可对应多个虚页。至于能对应多少个虚页，与采用的映像方式有关。这就可能发生两个以上的虚页想进入主存同一个页面位置的页面争用(或实页冲突)的情形。一旦发生实页冲突，只能在主存中该页面位置先装入其中的一个虚页，待其退出主存后方可再装入，执行效率自然会下降。因此，映像方式的选择应考虑能否尽量减少实页冲突概率，同时应考虑辅助硬件是否少，成本是否低，实现是否方便以及地址变换的速度是否快等。

由于虚存空间远大于实存空间，页式虚拟存储器一般都采用让每道程序的任何虚页可以映像装入到任何实页位置的全相联映像，如图 4 - 11 所示。如此，仅当一个任务要求同时调入主存的页数超出 2^{n_r} 时，两个虚页才会争用同一个实页位置，这种情况是很少见的。因此，全相联映像的实页冲突概率最低。

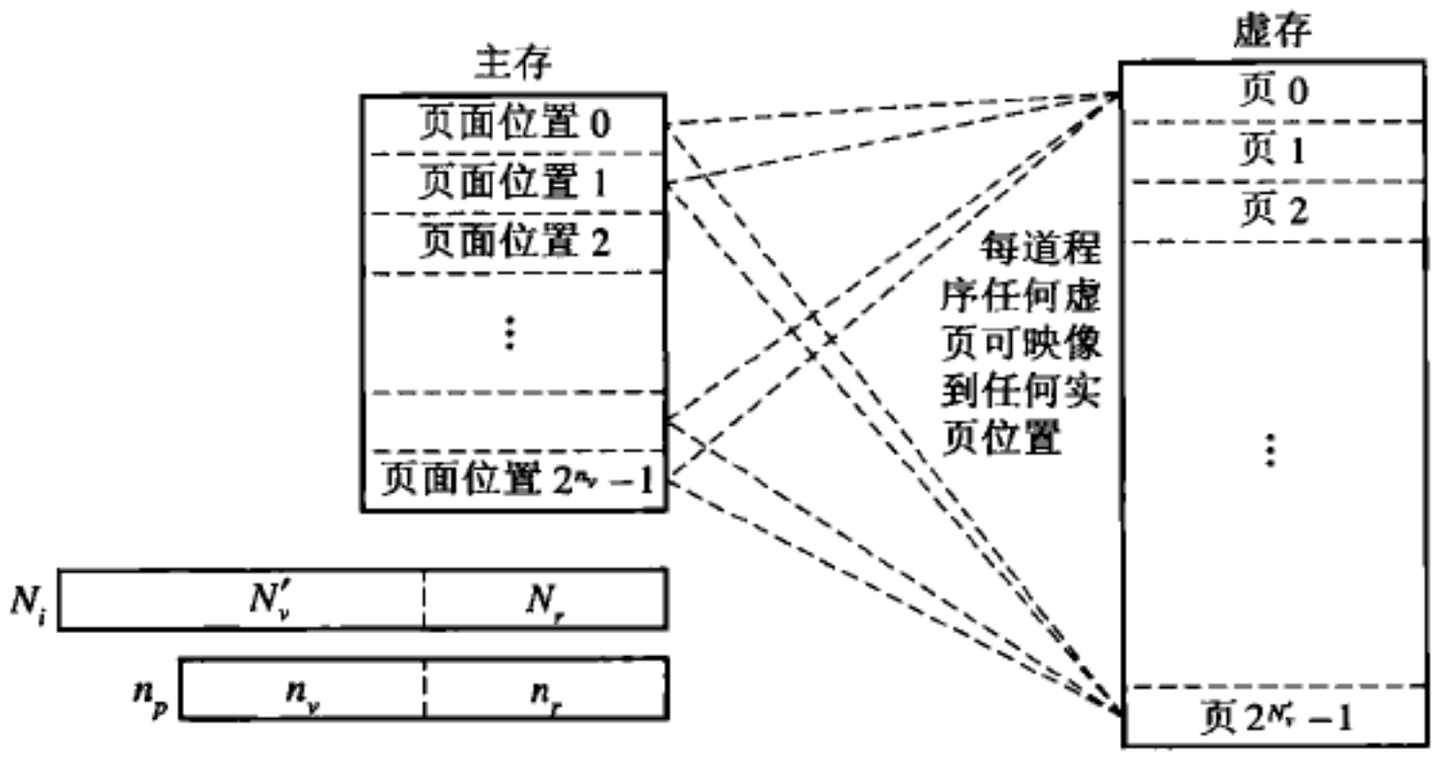


图 4 - 11　全相联映像

全相联映像的定位机构及其地址的变换过程已在 4.2.1 节中介绍过，这里不再重复。它用页表作为地址映像表，故称之为页表法。

整个多用户虚存空间可对应 2^{n} 个用户(程序)，但主存最多同时只对其中 N 个用户(N 道程序)开放。由基号 b 标识的 N 道程序中的每一道都有一个最大为 $2^{N'_v}$ 行的页表，而主存总共只有 2^{n_r} 个实页位置，因此 N 道程序页表的全部 $N\times 2^{N'_v}$ 行中，装入位为“1”的最多只有 2^{n_r} 行。由于 $N\times 2^{N'_v}\gg 2^{n_r}$，使得页表中绝大部分行中的实页号 n_r 字段及其他字段都成为无用的了，这会大大降低页表的空间利用率。

一种解决办法是将页表中装入位为“0”的行用实页号 n_r 字段存放该程序此虚页在辅存中的实地址，以便调页时实现用户虚页号到辅存实地址的变换。不过当辅存实地址的位数与用户虚页号字段的位数差别大时，就很难利用。

另一种方法是把页表压缩成只存放已装入主存的那些虚页(用基号 b 和 N'_v 标识)与实页位置(n_r)的对应关系，如图 4 - 12 所示，该表最多为 2^{n_r} 行。我们称这种方法为相联目录表法，简称目录表法。该表采用按内容访问的相联存储器构成。

按内容访问的相联存储器不同于按地址访问的随机存储器。按地址访问的随机存储器是在一个存储周期里只能按给出的一个地址访问其存储单元。相联存储器在一个存储周期中能将给定的 N_v 同时与目录表全部 2^{n_r} 个单元对应的虚页号字段内容进行比较，即进行相

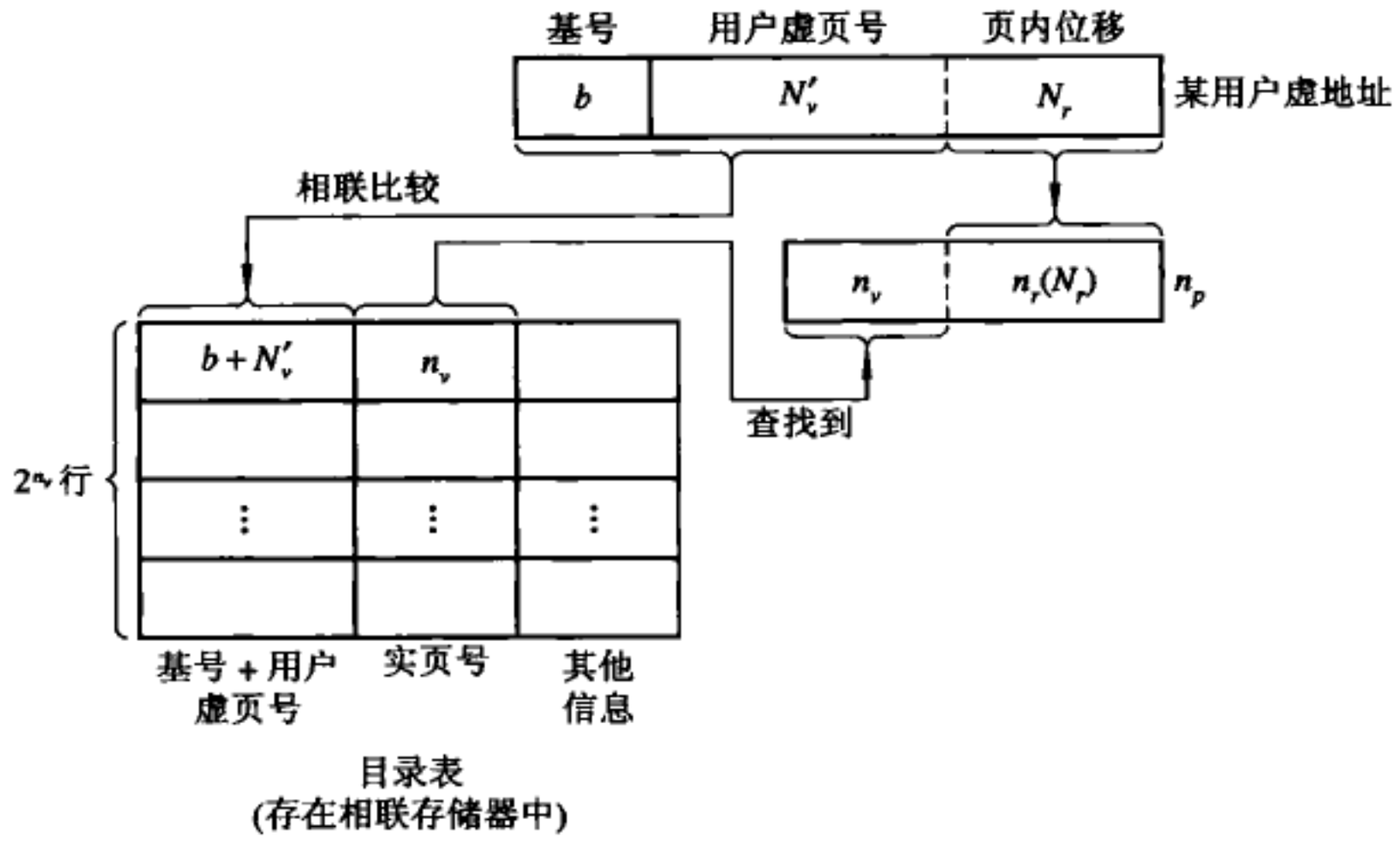


图 4-12　目录表法

联查找。如有相符的，即相联查找到了，表示此虚页已被装入主存，该单元中存放的实页号 n_v 就是此虚页所存放的实页位置，将其读出拼接上 N_r 就可形成访存实地址 n_p，该单元其他字段内容可供访问方式保护或其他工作用。如无相符的，即相联查找不到，就表示此虚页未装入主存，则发出页面失效故障，请求从辅存中调页。可见，目录表法不用设置装入位。

尽管目录表的行数为 2^{n_v}，比起页表法的 $N\times 2^{N_v'}$ 行少得多，但主存的页数 2^{n_v} 还是很大，这样的有 2^{n_v} 行的相联存储器不仅造价很高，而且查表速度也较慢。所以，虚拟存储器一般不直接用目录表来存储全部虚页号与实页号的对应关系，但它可以被用来提高地址变换速度。

当给出的多用户虚地址 N_v 所在的虚页未装入主存时都将发生故障。发生故障的原因，可能是出现了一个从未运行过的新程序，此时将进行程序换道；也可能是已在主存中的某程序的虚页未装入主存而发生页面失效，则需到辅存中去调页。现以后者为例来讨论。

要想把该道程序的虚页调入主存，必须给出该页在辅存中的实际地址。为了提高调页效率，辅存一般是按信息块编址的，而且块的大小通常等于页面的大小。以磁盘为例，辅存实(块)地址 N_{v_d} 的格式为

N_{v_d}	磁盘机号	柱面号	磁头号	块号

这样就需要将多用户虚页号 N_v 变换成辅存实(块)地址 N_{v_d}。用类似页表的方式为每道程序(每个用户)设置一个存放用户虚页号 N_v' 与辅存实(块)地址 N_{v_d} 映像关系的表，作为外部地址变换用，称之为外页表。对应地，将前述映像 N_v' 与 n_v 的关系、用于内部地址变换的页表改称为内页表。显然，每个用户的外页表也是 $2^{N_v'}$ 项(行)，每行中用装入位表示该信息块是否已由海量存储器(如磁带)装入磁盘。当装入位为“1”时，辅存实地址字段内容有效，表示的就是该信息块(页面)在辅存(磁盘)中的实际位置。外页表的内容是在程序装入

辅存时就填好了的。

由于虚拟存储器的页面失效率一般低于1%，调用外页表进行虚地址到辅存实地址变换的机会很少，加上访辅存调页速度本来就慢，因此，外页表通常存在辅存中，只有当某道程序初始运行时，才把外页表的内容转录到已建立的内页表的实页号地址字段中，这就是前述当内页表装入位为"0"时，可以让实页号地址字段改放该虚页在辅存中的实地址的原因。而且对查找外页表的速度要求也较低，完全可用软件实现以节省硬件成本。因为程序或进程切换所需要的时间要比调页耗费的时间短得多，所以一旦发生页面失效，可以采取程序换道的做法，而不必让处理机空等调页。

图4-13为外页表的结构及用软件方法查外页表实现由多用户虚地址 N_v 到辅存实地址 N_{v_d} 的变换过程。

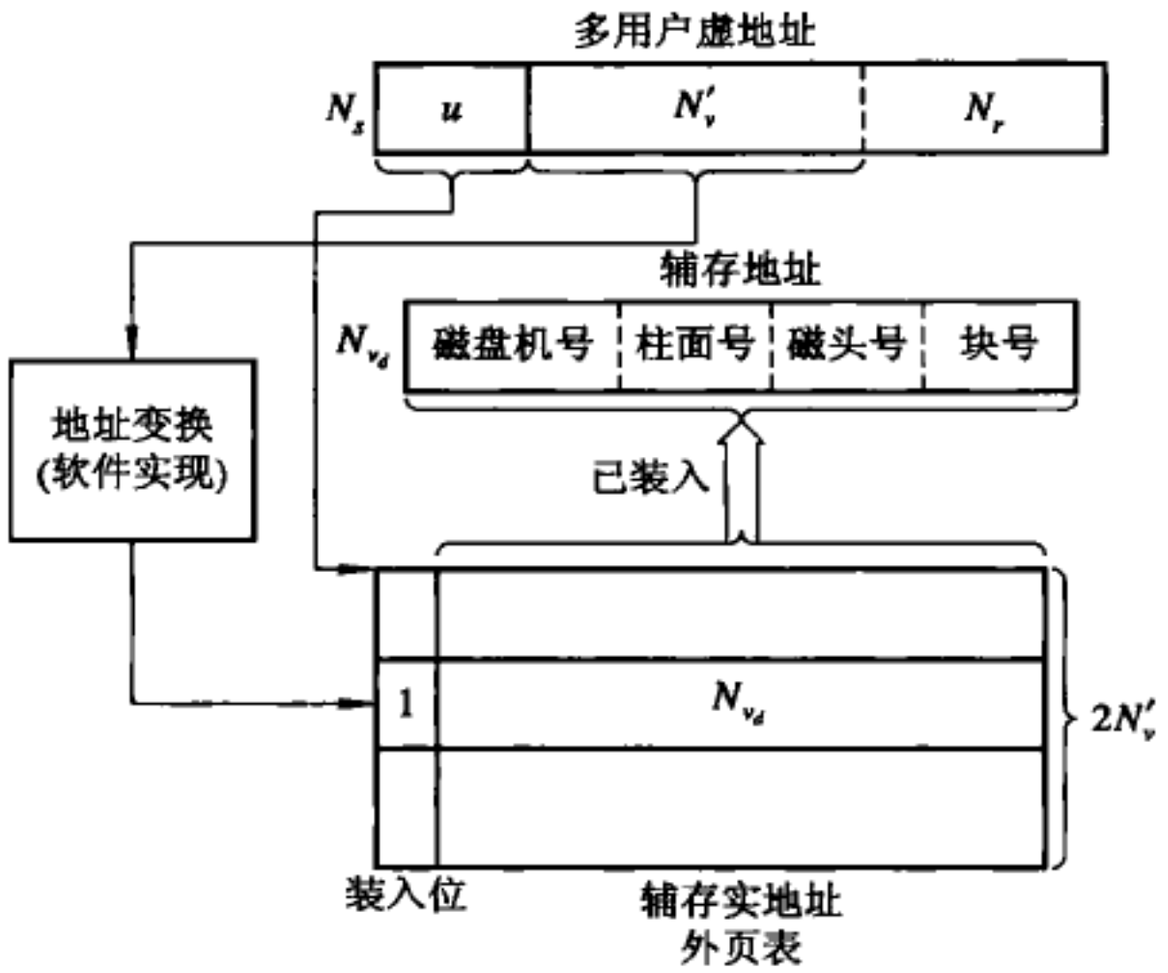


图4-13　虚地址到辅存实地址的变换

2. 页面替换算法

当处理机要用到的指令或数据不在主存中时，会产生页面失效，须去辅存中将含该指令或数据的一页调入主存。通常虚存空间比主存空间大得多，必然会出现主存已满又发生页面失效的情况，这时将辅存的一页调入主存会发生实页冲突，只有强制腾出主存中某个页才能接纳由辅存中调来的新页。选择主存中哪个页作为被替换的页，就是替换算法要解决的问题。

替换算法的确定主要看主存是否有高的命中率，也要看算法是否便于实现，辅助软、硬件成本是否低。目前已研究过多种替换算法，如随机算法、先进先出算法、近期最少使用(近期最久未用过)算法等。

随机算法(Random，RAND)采用软的或硬的随机数产生器产生主存中要被替换页的页号。这种算法简单，易于实现，但没有利用主存使用的"历史"信息，反映不了程序的局部性，使主存命中率很低，因此已不采用。

先进先出算法(First-In First-Out，FIFO)选择最早装入主存的页作为被替换的页。这种算法实现方便，只要在操作系统为主存管理所设的主存页面表中给每个实页配一个计数

器字段(参看图 4 - 14，将其中的使用位字段改成计数器字段)即可。每当一页装入主存时，让该页的计数器清零，其他已装入主存的那些页的计数器都加“1”。需要替换时，计数器值最大的页的页号就是最先进入主存而现在准备替换掉的页号。这种方法虽利用了主存使用的“历史”信息，但不一定能正确地反映出程序的局部性，因为最先进入的页很可能正是现在经常在用的页。

近期最少使用算法(Least Recently Used，LRU)选择近期最少访问的页作为被替换的页。这种算法能比较正确地反映程序的局部性。一般来说，当前最少使用的页，未来也将很少被访问。但完全按此算法实现比较困难，需要为每个实页都配一个字长很长的计数器。所以一般用其变形，即把近期最久未被访问过的页作为被替换页，将“多”和“少”变成“有”和“无”实现就方便多了。

图 4 - 14 是操作系统为实现主存管理设置的主存页面表，其中每一行用来记录主存中各页的使用状况。主存页面表不是前述的页表。页表是用于存储地址映像关系和实现地址变换的，对于用户程序空间而言，每道程序都有一个页表。而主存页面表存于主存，整个系统只有一个。主存页号是顺序的，该字段可以省去，用相对于主存页面表起点的行数表示。占用位表示主存中该页是否已被占用，“0”表示未被占用，“1”表示已被占用。至于被哪个程序的哪个段或哪个页占用，由程序号、段页号字段指明。为实现近期最久未用过算法，给表中每个主存页配一位“使用位”标志。开始时，所有页的使用位全为“0”。只要某个实页的任何单元被访问过，就由硬件自动将该页使用位置为“1”。由于是全相联映像，调入页可装入主存页面表中任何占用位为“0”的实页位置，一旦装入就将该实页之占用位置为“1”。只有当占用位都是“1”，又发生页面失效时，才有页面替换，此时只需替换使用位为“0”的页即可。

(计数器)

主存页号	占用位	程序号	段页号	使用位	程序优先位	H_s	其他信息
0							
1							
⋮							
$2^{n_2}-1$							

图 4 - 14　主存页面表

显然，使用位不能出现全为“1”，否则无法确定哪一页被替换。为避免出现这种状况，一种办法是一旦使用位要变为全“1”时立即由硬件强制全部使用位都为“0”。从概念上看，近期最少使用的“期”是从上次使用位为全“0”到这次使用位为全“0”的这段时期。此“期”的长短是随机的，故称为随机期法。另一种办法是定期置全部使用位为“0”。给每个实页再配一个“未使用过计数器”H_s(或称历史位)，定期地每隔 Δt 时间扫视所有使用位，凡使用位为“0”的将其 H_s 加“1”，并让使用位仍为“0”；而使用位为“1”的将其 H_s 和使用位均“清0”。这样，H_s 值最大的页就是最久未用的页，将被替换。可见，使用位反映一个 Δt 期内的页面使用情况，H_s 则反映了多个 Δt 期内的页面使用状况。这种方法比近期最少使用法所

耗费的计数器硬件要少得多。由于页面失效后调页时间长，加上程序换道，因此主存页面表的修改可软、硬件结合地实现。在主存页面表中还可增设修改位以记录该页进入主存后是否被修改过，如未被修改过，替换时就可不必写回辅存；否则，需先将其写回辅存原先的位置，然后才能替换。

近期最少使用和近期最久未用过两种算法都是 LRU 法，与 FIFO 法一样，都是根据页面使用的“历史”情况来预估未来的页面使用状况的。如果能根据未来实际使用情况将未来的近期里不用的页替换出去，一定会有最高的主存命中率，这种算法称为优化替换算法(Optimal，OPT)。它是在时刻 t 找出主存中每个页将要用到的时刻 t_i，然后选择其中 t_i+t 最大的那一页作为替换页。显然，这只有让程序运行过一遍，才能得到各页未来的使用情况信息，所以要实现它是不现实的。优化替换算法是一种理想算法，可以被用来作为评价其他替换算法好坏的标准，即看哪种替换算法的主存命中率最接近于优化替换算法的主存命中率。

替换算法一般是通过用典型的页地址流模拟其替换过程，再根据所得到的命中率的高低来评价其好坏的。影响命中率的因素除替换算法外，还有地址流、页面大小、主存容量等。

【例 4-1】 设有一道程序，有 1～5 页，执行时的页地址流(即依次用到的程序页页号)为

2,3,2,1,5,2,4,5,3,2,5,2

若分配给该道程序的主存有 3 页，则图 4-15 表示 FIFO、LRU、OPT 这 3 种替换算法对这 3 页的使用和替换过程。其中用 * 号标记出按所用算法选出的下次应被替换的页号。由图 4-15 可知，FIFO 算法的页命中率最低，LRU 算法的页命中率非常接近于 OPT 算法的页命中率。

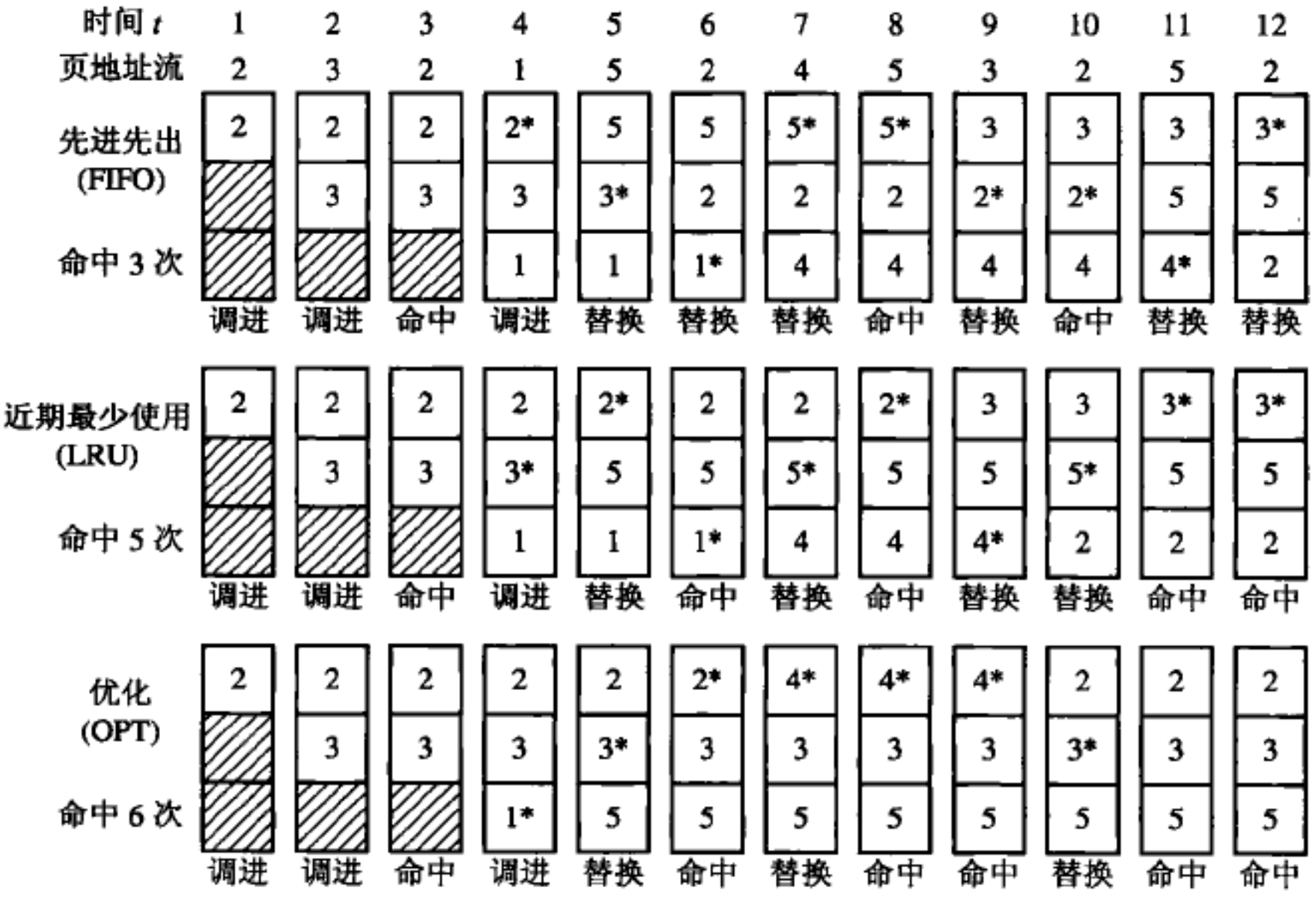


图 4-15　3 种替换算法对同一页地址流的替换过程

结论 1　命中率与所选用替换算法有关。LRU 算法要优于 FIFO 算法。命中率也与页地址流有关。

【例 4－2】　一个循环程序当所需页数大于分配给它的主存页数时，无论是 FIFO 算法还是 LRU 算法的命中率都明显低于 OPT 算法的命中率。图 4－16 就表明了这种情况。但只要将实页数增加一页，就能使这 3 种算法的命中次数都增大到 4 次。

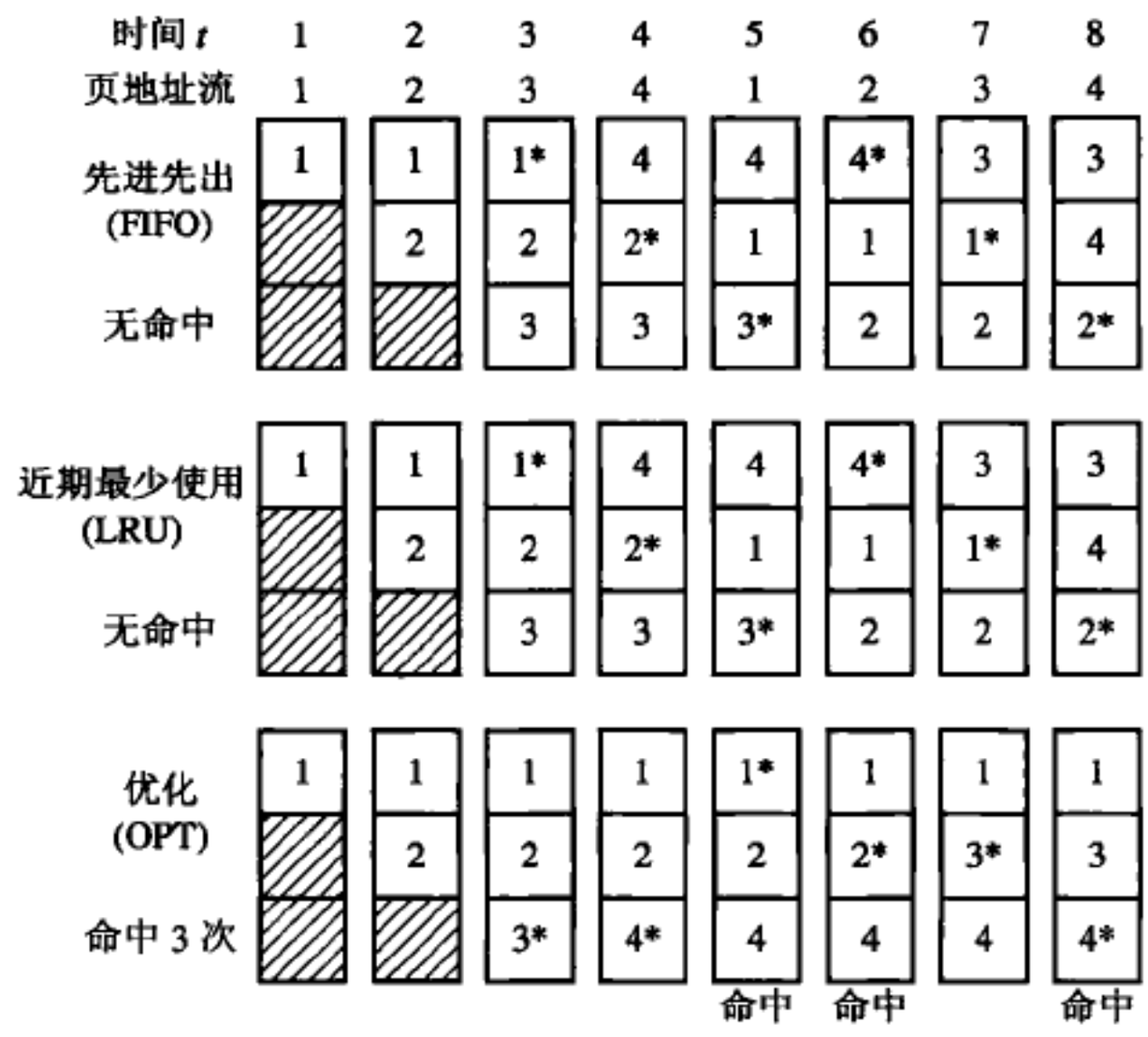


图 4－16　命中率与页地址流有关

结论 2　命中率与分配给程序的主存页数有关。

一般来说，分配给程序的主存页数越多，虚页装入主存的机会越多，命中率也就可能越高，但能否提高还和替换算法有关。FIFO 算法就不一定。由图 4－17 可知，主存页数由 3 页增至 4 页时，命中率反而由 3/12 降低到 2/12。而 LPU 算法则不会发生这种情况，随着分配给程序的主存页数的增加，其命中率一般都能提高，至少不会下降。因此，从衡量替换算法好坏的命中率高低来考虑，如果对影响命中率的主存页数 n 取不同值的情况都模拟一次，则工作量是非常大的。于是提出了使用堆栈处理技术处理的分析模型，它适用于采用堆栈型替换算法的系统，可以大大减少模拟的工作量。

什么是堆栈型替换算法呢？设 A 是长度为 L 的任意一个页地址流，t 为已处理过 $t-1$ 个页面的时间点，n 为分配给该地址流的主存页数，$B_t(n)$表示在 t 时间点、在 n 页的主存中的页面集合，L_t 表示到 t 时间点已遇到过的地址流中相异页的页数。如果替换算法满足

$$n < L_t \text{ 时，} B_t(n) \subset B_t(n+1)$$

$$n \geqslant L_t \text{ 时，} B_t(n) = B_t(n+1)$$

则属堆栈型的替换算法。

LRU 算法在主存中保留的是 n 个最近使用的页，它们又总是被包含在 $n+1$ 个最近使用的页中，所以 LRU 算法是堆栈型算法。显然，OPT 算法也是堆栈型算法，而 FIFO 算法则不是。因为从图 4－17 可以看出，FIFO 算法不具有任何时刻都能满足上述包含性质的特征。例如，$B_7(3)=\{1, 2, 5\}$，而 $B_7(4)=\{2, 3, 4, 5\}$，所以，$B_7(3) \not\subseteq B_7(4)$。由于堆栈

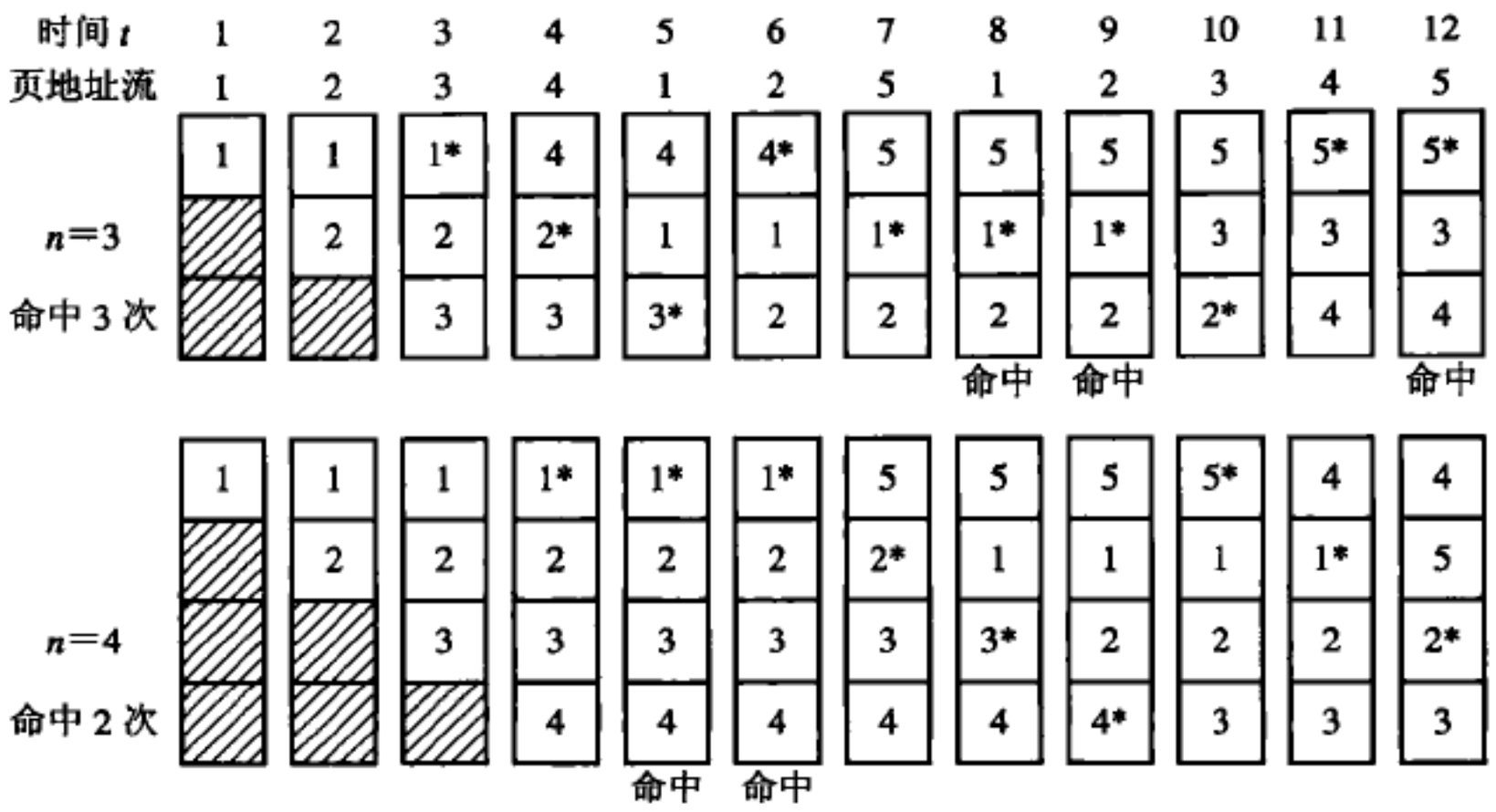


图 4 - 17　FIFO 算法的实页数增加，命中率反而有可能下降

型替换算法具有上述包含性质，因此命中率随主存页数的增加只有可能提高，至少不会下降。只要是堆栈型替换算法，只要采用堆栈处理技术对地址流模拟一次，即可同时求得对此地址流在不同主存页数时的命中率。

用堆栈模拟时，主存在 t 时间点的状况用堆栈 S_t 表示，S_t 是 L_t 个不同页面号在堆栈中的有序集，$S_t(1)$是 t 时间点的 S_t 的栈顶项，$S_t(2)$是 t 时间点的 S_t 的次栈顶项，依次类推。由于堆栈型算法的包含性，必有

$$n < L_t \text{ 时，} B_t(n) = \{S_t(1), S_t(2), \cdots, S_t(n)\}$$

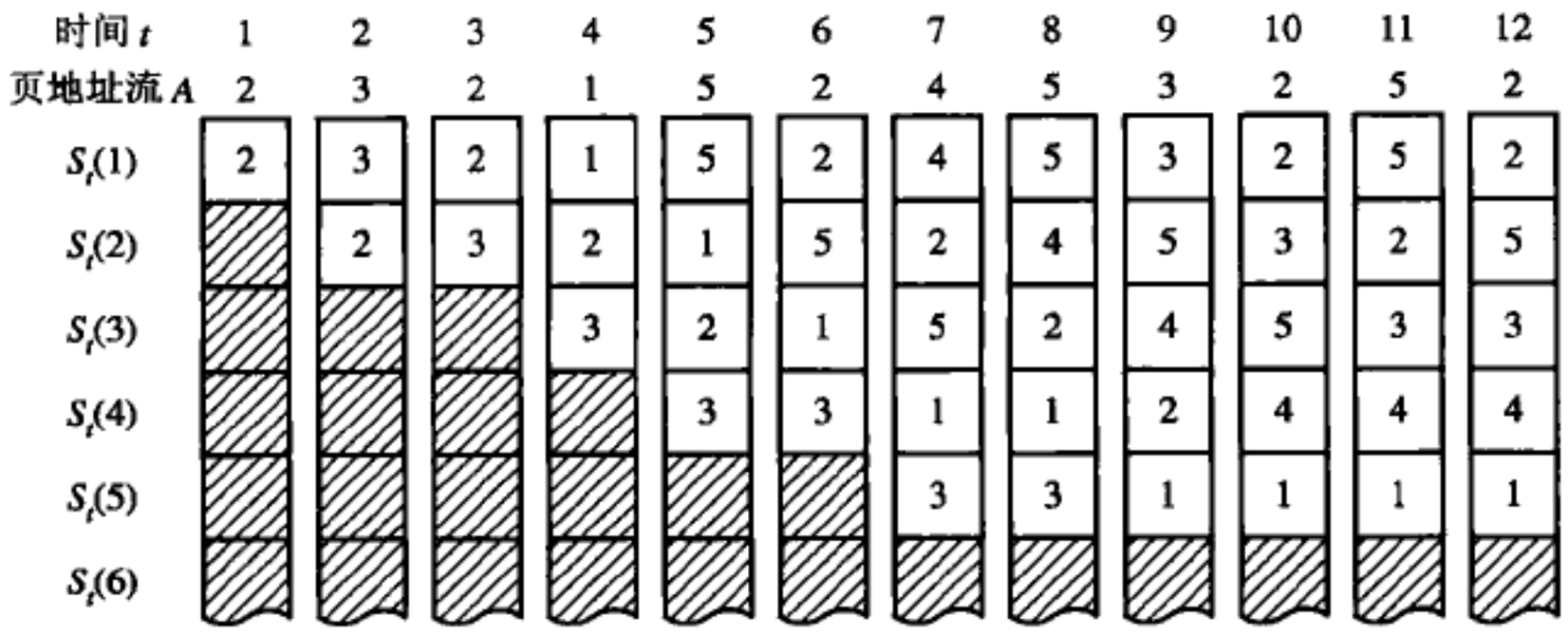
$$n \geqslant L_t \text{ 时，} B_t(n) = \{S_t(1), S_t(2), \cdots, S_t(L_t)\}$$

这样，容量为 n 页的主存中，页地址流 A 在 t 时间点的 A_t 页是否命中，只需看 S_{t-1} 的前 n 项中是否有 A_t，若有则命中。所以，经过一次模拟处理获得 $S_t(1)$，$S_t(2)$，…，$S_t(L_t)$之后，就能同时知道不同 n 值时的命中率，从而为该道程序确定所分配的主存页数提供依据。

不同的堆栈型替换算法，其 S_t 各项的改变过程不同。LRU 算法是把主存中刚被访问过的页号置于栈顶，而把最久未被访问过的页号置于栈底。设 t 时间点被访问的页为 A_t，若 $A_t \notin S_{t-1}$，则把 A_t 压入栈顶使之成为 $S_t(1)$，S_{t-1} 各项都下推一个位置；若 $A_t \in S_{t-1}$，则将它由 S_{t-1} 中取出压入栈顶成为 $S_t(1)$，在 A_t 之下的各项位置不动，而 A_t 之上的各项都下推一个位置。

【例 4 - 3】 图 4 - 18 是图 4 - 15 页地址流采用 LRU 算法进行堆栈处理的 S_t 变化过程。

由图 4 - 18 的 S_t 可确定对应这个页地址流，主存页数 n 取不同值时的命中率。只要取栈顶的前 n 项，若 $A_t \in S_{t-1}$ 则为命中，若 $A \notin S_{t-1}$ 则为不命中。例如，对 $n=4$，其 $S_5 = \{5, 1, 2, 3\}$，因为 $A_6 = 2 \in S_5$，所以命中；但对 $n=2$，其 $S_5 = \{5, 1\}$，因为 $A_6 = 2 \notin S_5$，所以不命中。这样就可算出各个 n 值的命中率 H^*，如表 4 - 1 所示。可见，LRU 算法的命中率随分配给该道程序的主存页数 n 的增加而单调上升，至少不会下降，这是堆栈型算法所共有的特点。而 FIFO 这种非堆栈型算法由图 4 - 17 可知，不具备这一特点。虽然虚页在主存的机会多了，命中率总趋势应随 n 的增加而增大，但从某个局部看，n 的增加有时会使命中率降低。



n=1												
n=2		命中										命中
n=3		命中		命中		命中				命中	命中	
n=4		命中		命中		命中			命中	命中	命中	
n=5		命中		命中		命中	命中		命中	命中	命中	

图 4－18　图 4－15 页地址流使用 LRU 算法进行堆栈处理的 S_t 变化过程

表 4－1　不同实页数 n 的命中率 H

n	1	2	3	4	5	>5
H	0	0.17	0.42	0.50	0.58	0.58

结论：由于堆栈型替换算法有随分配给该道程序的实页数 n 的增加，命中率 H 会单调上升这一特点，因此可对 LRU 算法加以改进，提出使系统性能更优的动态算法。即根据各道程序运行中的主存页面失效率，由操作系统动态调节分配给各道程序的实页数。当主存页面失效率超过某个值时就自动增加分配给该道程序的主存页数以提高其命中率；而当主存页面失效率低于某个值时就自动减少分配给该道程序的主存页数，以便释放出这部分主存页面位置供给其他程序用，从而使整个系统总的主存命中率和主存利用率得到提高。我们称此算法为页面失效频率(PFF)算法。显然它是立足于主存页数增加一定会使命中率单调上升，至少不下降这一基本点上的。

3. 虚拟存储器工作的全过程

参看图 4－19，在页式虚拟存储器中每当用户用虚地址访问主存时，都必须查找内页表，将多用户虚地址变换成主存的实地址①、②。在查找内页表时，如果对应该虚页的装入位为“1”，就取出其主存页号 n_v，拼接上页内位移 N_r，形成主存实地址 n_p 后访主存③。如果对应该虚页的装入位为“0”，表示该虚页未在主存中，就产生页面失效④，程序换道，经异常处理从辅存中调页。这时需查找外页表，完成外部地址变换⑤。在查找外页表时，若该虚页的装入位为“0”，表示该虚页尚未装入辅存，则产生辅存缺页故障(异常)，由海量存储器调入⑧；若查找外页表时，该虚页的装入位为“1”，就将多用户虚地址变换成辅存中的实块号 N_{v_d}，告诉 I/O 处理机到辅存中去调页⑥，而后经 I/O 处理机送入主存⑦。在多道程序机器中，CPU 的运行与 I/O 处理机的调页是并行进行的。

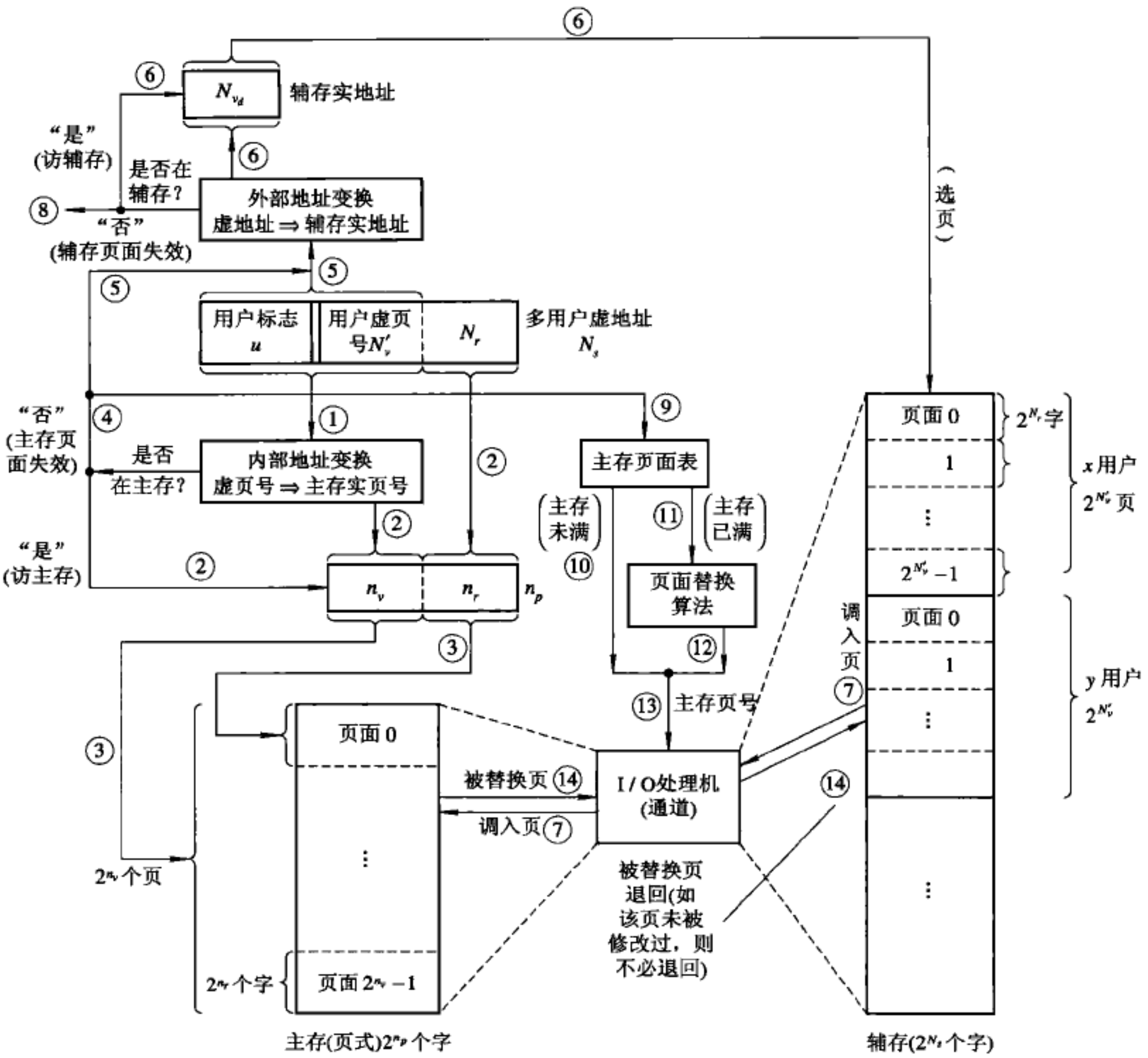


图 4－19　页式虚拟存储器工作的全过程

一旦发生页面失效，还需要确定调入页应进入主存中哪一页位置，这就需要操作系统查找主存页面表⑨。若占用位为"0"，表示主存未装满，因为是全相联映像，所以此时只需找到任何占用位为"0"的一个页面位置即可⑩。若占用位全为"1"，表示主存已装满，就需要通过替换算法寻找替换页⑪、⑫。不管是哪种情况，均将确定好的主存页号送入 I/O 处理机⑬，由I/O 处理机完成页的调入⑦。在页面替换时，如果被替换的页调入主存后一直未经改写，则不需送回辅存；如果已被修改，则需先将它送回辅存原处⑭，再把调入页装入主存⑦。装入主存的页是否改写过可用主存页面表中的修改位来指明。每次调入一页或访问过一页，应记录或修改有关页表和主存页面表的内容。

4.2.3　页式虚拟存储器实现中的问题

1. 页面失效的处理

对页面失效的处理是设计好页式虚拟存储器的关键之一。

前面已讲过，页面的划分只是对程序和主存空间进行机械等分。对于按字节编址的存

储器完全可能出现指令或操作数横跨在两页上存储的情况。特别是对于字符串数据、操作数多重间接寻址，这种跨页现象更为严重。如果当前页在主存中，而跨页存放的那一页不在主存中，就会在取指令、取操作数或间接寻址等访存过程中发生页面失效。就是说，页面失效会在一条指令的分析或执行的过程中产生。一般地，中断都是在每条指令执行的末尾安排有访中断微操作，检验系统中有无未屏蔽的中断请求，以便对其响应和处理。页面失效如果也用这种办法就会造成死机，因为不调页，指令就无法执行到访中断微操作，从而就不可能对页面失效给予响应和处理。因此，页面失效不能按一般的中断对待，应看作是一种故障，必须立即响应和处理。

页面失效后还应解决如何保存好故障点现场以及故障处理完后如何恢复故障点现场，以便能继续执行这条指令。目前多数机器都采用后援寄存器法，把发生页面失效时指令的全部现场都保存下来。待调页后再取出后援寄存器内容恢复故障点现场，以便继续执行完该指令。也有的机器同时采用一些预判技术。例如，在执行字符串指令前预判字符数据首尾字符所在页是否都在主存中，如果是，才执行，否则，发页面失效请求，等到调页完成后才开始执行此指令。替换算法的选择也是很重要的。不应发生让指令或操作数跨页存放的那些页轮流从主存中被替换出去的“颠簸”现象。因此，给一道程序分配的主存页数应有某个下限。假设指令和两个操作数都跨页存储，那这条指令的执行至少需分配 6 个主存页面。另外，页面也不能过大，以使多道程序的道数、每道程序所分配到的主存页数都能在一个适当的范围内。页面过大会使主存中的页数过少，从而出现大量的页面失效，严重降低虚拟存储器的访问效率和等效速度。

应认真解决页面失效的处理，这是操作系统和系统结构设计共同需要解决的问题。

2. 提高虚拟存储器等效访问速度的措施

要想使虚拟存储器的等效访问速度提高到接近于主存的访问速度并不容易。从存储层次的等效访问速度公式可以看出，要达到这样的目标既要有很高的主存命中率，又要有尽可能短的访主存时间。高的主存命中率受很多因素影响，包括页地址流、页面调度策略、替换算法、页面大小、分配给程序的页数(主存容量)等，有些前面已提过了，后面将对影响虚拟存储器性能的某些因素作进一步分析。这里先就缩短访主存时间讲述结构设计中可采取的措施。

由虚拟存储器工作的全过程可以看出，每次访主存时，都要进行内部地址变换，其概率是 100%。从缩短访主存的时间看，只有内部地址变换快到使整个访存速度非常接近于不用虚拟存储器时，虚拟存储器才能真正实用。

页式虚拟存储器的内部地址变换靠页表进行，页表容量很大，只能放在主存中，每访主存一次，就要加访一次主存查表。如果采用段页式，查表还需加访主存两次。这样，为存、取一个字，需经 2 次或 3 次访主存才能完成，其等效访问速度只能是不用虚拟存储器的1/2 或 1/3。有的小机器可用单独的小容量快速随机存储器或寄存器组成存放页表。如 Xerox Sigma7 处理机的虚拟存储空间容量为 2^{17} 个字，页面大小为 2^9 个字，用 2^8 个字、每个字为 9 位的寄存器组来存放页表，一定程度缩短了内部地址变换的时间。在大多数规模较大的机器上，是靠硬件上增设“快表”来解决的。

由于程序存在局部性，因此对页表内各行的使用不是随机的。在一段时间里实际可能只用到表中很少的几行。这样，用快速硬件构成比全表小得多，例如(8～16)行的部分目录

表存放当前正用的虚实地址映像关系，那么其相联查找的速度将会很快。我们称这个部分目录表为快表。将原先存放全部虚、实地址映像关系的表称为慢表。快表只是慢表中很小一部分副本。这样，从虚地址到主存实地址的变换可以用图 4 - 20 的办法来实现。

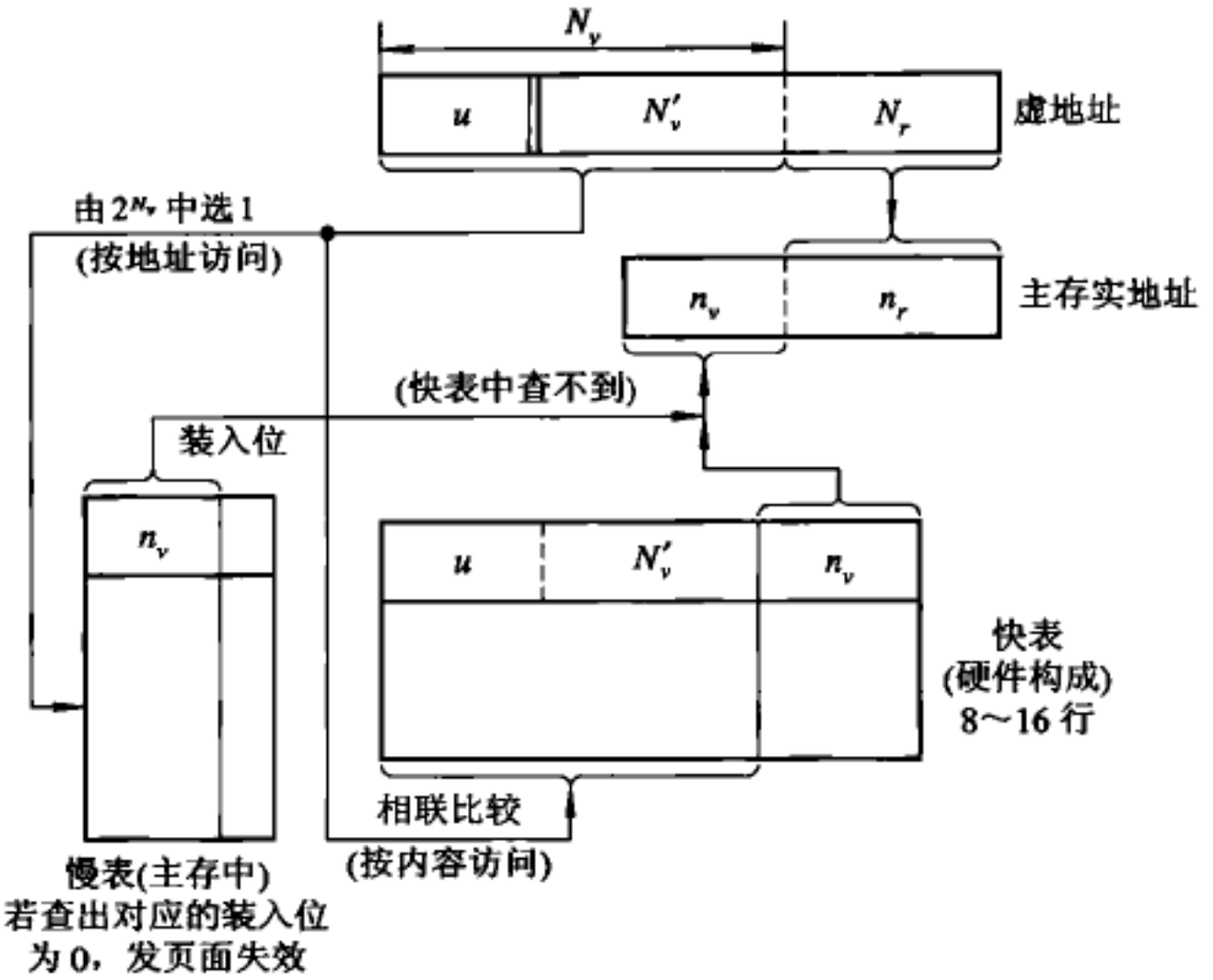


图 4 - 20　经快表与慢表实现内部地址变换

查表时，由虚页号 $u+N_v'$(即 N_v)同时去查快表和慢表。当快表中有此虚页号时，就能很快找到对应的实页号 n_v，将其送入主存地址寄存器访存，并立即使慢表的查找中止，这时访主存的速度几乎未下降。如果在快表中查不到，则经一个访主存时间，从慢表中查到的实页号 n_v 就会送入主存地址寄存器并访存，同时将此虚页号与实页号的对应关系送入快表。这里，也需要用替换算法去替换快表中已不用的内容。

如果快表的命中率不高，系统效率就会显著下降。快表如果用堆栈型替换算法，则快表容量越大，其命中率就会越高。但容量越大，会使相联查找的速度越低，所以快表的命中率和查表速度有矛盾。若快表取 8～16 行，每页容量为 1～4 K 字，则快表容量可反映主存中 8～64 K 个单元，其命中率应该是较高的。于是快表和慢表实际构成了一个两级层次，其所用的替换算法一般也是 LRU 算法。

为了提高快表的命中率和查表速度，可以用高速按地址访问的存储器来构成更大容量的快表，并用散列(Hashing)方法实现按内容查找。散列方法的基本思想是让内容 N_v 与存放该内容的地址 A 之间有某种散列函数关系，即让快表的地址 $A=H(N_v)$。参看图4 - 21，当需要将虚、实地址 N_v 与 n_v 的映像关系存入快表存储器中时，只需将 N_v 对应的 n_v 等内容存入快表存储器的 $A=H(N_v)$单元中。查找时按现给的 N_v 经同样的散列函数变换成 A 后，再按地址 A 访问快表存储器，就可能找到存放该 N_v 所对应的 n_v 及其余内容。散列函数变换必须采用硬化实现才能保证必要的速度。在快表中增设 N_v 字段是为解决多个不同的 N_v 可能散列到同一个 A 的散列冲突。在快表的 A 单元中除了存入当时的 n_v，也存入当时的 N_v。这样在地址变换时用现行 N_v 经散列函数求得 A、查到 n_v 并访主存的同时，再将同行中原保存的 N_v 读出与现行虚地址中的 N_v 作比较。若相等，就让按 n_v 形成的主存实地址进行的访存继续下去；若不等，就表明出现了散列冲突，即 A 地址单元中的 n_v 不是现行虚地址对应的实页号，这时就让刚才按 n_v 形成的主存实地址进行的访存中止，经过一个

主存周期，用从慢表中读得的 n_v 再去访存。可以看出，这种按地址访问构成的快表，其容量可比前述使用相联存储器片子构成的快表容量 8～16 行要大，例如可以是 64～128 行，这不仅提高了快表的命中率，而且仍具有很高的查表速度。加上这种判相等与访主存是同时并行的，还可以进一步缩短地址变换所需的时间。

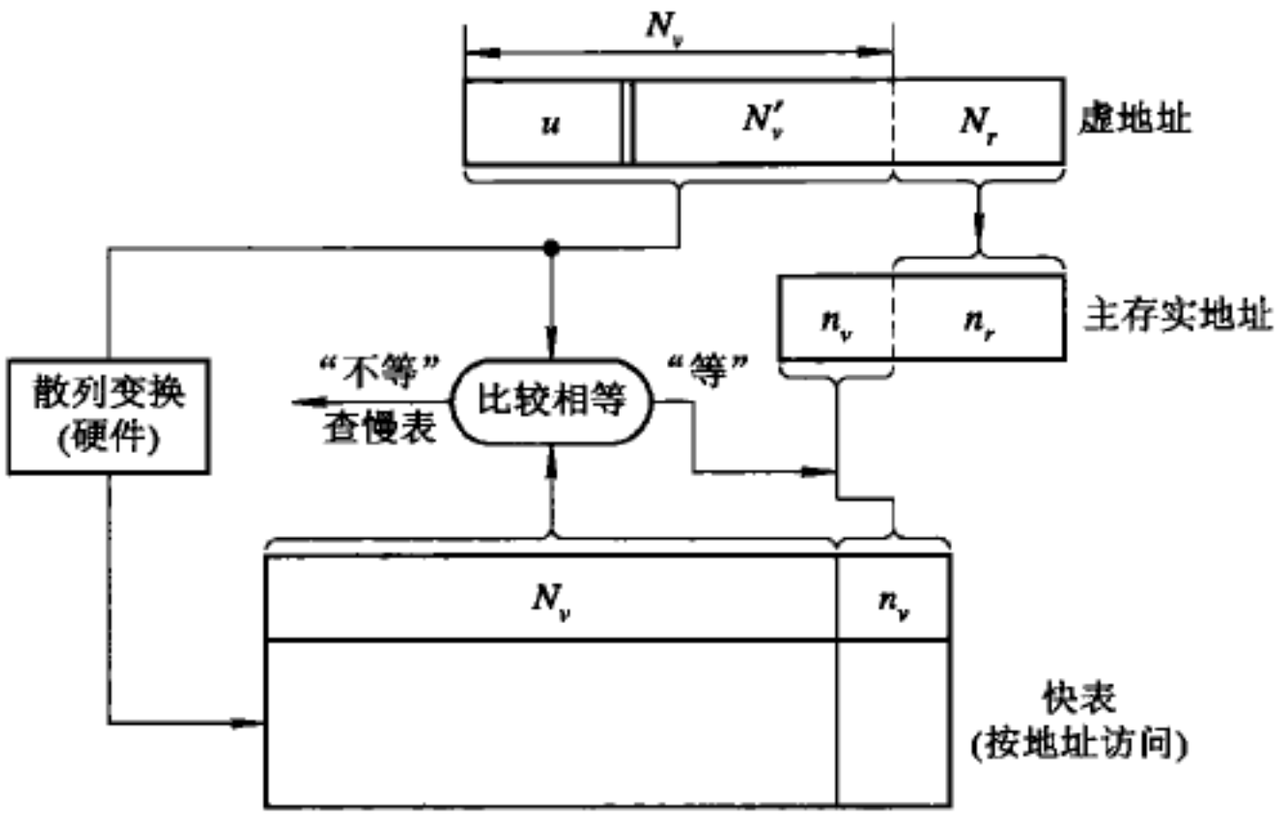


图 4 - 21　快表中增加 N_v 比较以解决散列冲突引起的查错

若在快表的每个地址 A 单元中存放多对虚页号与实页号的映像关系，则还可进一步降低散列冲突引起的不命中率。例如图 4 - 22 所示的 IBM 370/168 虚拟存储器的快表，每一行用的是两对虚、实地址映像关系。用两套外部的相等比较电路比较，看哪一个相符就送哪一个的 n_v。只有当 A 单元的两个虚页号都不相符时，才是不命中，才需经慢表途径获得 n_v。

图 4 - 22　IBM 370/168 虚拟存储器快表示意图

此外，散列变换(压缩)的入、出位数差愈小，散列冲突的概率就愈低。因此，可以考虑缩短被变换的虚地址位数，但如简单去掉 u 字段是不行的，需采取其他措施。

【例 4－4】 IBM 370/168 的设计者考虑到 24 位的 u 可对应 2^{24} 个任务，但实际在较长一段时间只有几个任务在运行，远比 2^{24} 小得多，即 u 的变化概率要比 N_v' 的变化概率低得多。因此，只需把这几个 u 值存在几个(如 6 个)相联寄存器中，从而只要 3 位 ID 就可区分。相当于只用 6 行高速相联寄存器组就把 24 位 u 压缩成 3 位的 ID 了。地址变换时，先用虚地址中的 u 到相联寄存器组中查找，找到相同的 ID 值(3 位)后，再与 N_v'(12 位)拼接，使得需相联比较的位数由 $u+N_v'=36$ 位缩短成 ID$+N_v'=15$ 位，从而既能缩短相联比较的位数，缩小散列变换的入、出位数差，又仍能达到在任务切换时不会出错的目的。这样，在快表内可以同时存有多达 6 个任务的部分页表，而且在任务切换时，操作系统不用作废整个快表或某行内容，使快表对操作系统和系统程序员是透明的。只有当某个 u 值与 6 行相联寄存器组的哪一行都不相等时，才发现已切换到这 6 个任务之外的新任务，这时才按前述替换算法选择一个 ID 标志分配给此新任务。随着新任务的执行，被替换掉的任务在快表中的各行也被新任务逐渐取代。只要替换算法选择得好，常用任务(或主任务)的那部分页表内容被替换掉的概率便会很低，避免了因快表内容不适当的作废引起的虚、实地址变换速度的下降。现在不少机器的虚拟存储器所用的快表结构与 IBM 370/168 的基本相同。

上述查相联寄存器组、散列压缩、查快表等都用硬件实现。因此，用于虚拟存储器的内部地址映像和变换的快、慢表对应用程序员和系统程序员都是透明的。

3. 影响主存命中率和 CPU 效率的某些因素

命中率是评价存储体系性能的重要指标。程序地址流、替换算法以及分配给程序的实页数不同都会影响命中率。

论点 1 页面大小 S_p、分配给某道程序的主存容量 S_1 与命中率 H 的关系如图 4－23 所示。当 S_1 一定时，随着 S_p 的增大，命中率 H 先逐渐增大，到达某个最大值后又减小。若增大 S_1，可普遍提高命中率，达到最高命中率时的页面大小 S_p 也可以增大一些。

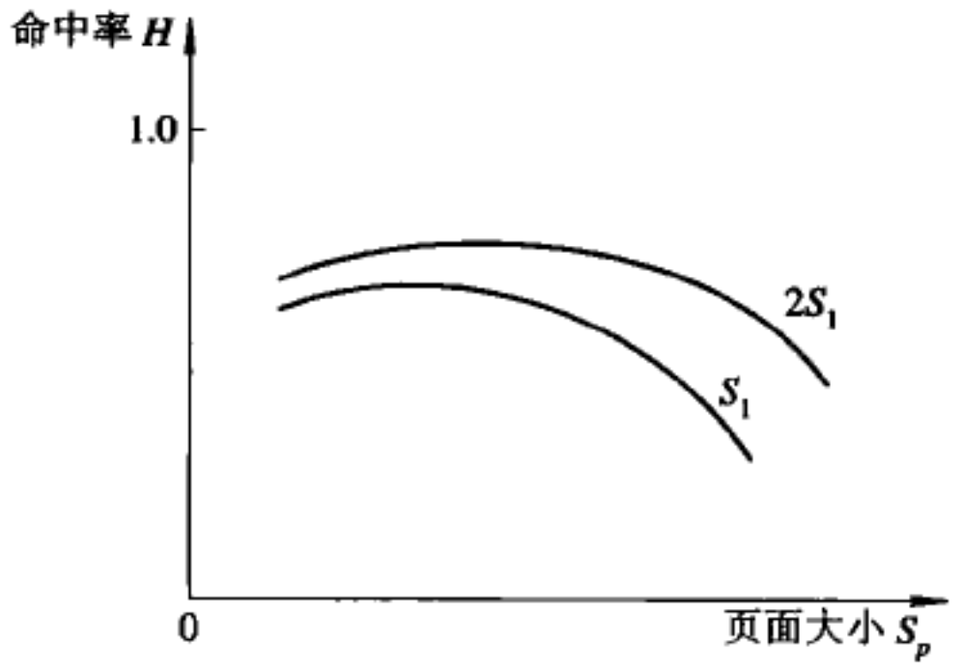


图 4－23 页面小大 S_p、容量 S_1 与命中率 H 的关系

对上述现象可作如下分析。假设程序执行过程中，相邻两次访存的逻辑地址间距为 d_r，若 d_r 比 S_p 小，随着 S_p 的增大，相邻两次访存的地址处于同一页内的概率将会增加，从这点看，H 随 S_p 的增大而上升；而若 d_r 比 S_p 大，两个地址肯定不会在同一页。如果该地址所在页也在主存，则也会命中。从这点看，H 会随分配给该道程序的实页数的增加而

上升。这对采用堆栈型替换算法是必然的。若分配给该道程序的主存容量固定，那么增大页面必使总页数减少。这样，虽然同页内的访问命中率会上升，但对于两个地址分属不同页的情况，就会使命中率下降。程序运行时是两种情况的综合。当 S_p 较小时，增大 S_p 的过程中，前一种因素起主要作用。因此，综合来看，H 随 S_p 的增大而上升。当达到某个最大值后，随着页数的显著减少，后一种因素起主要作用，这就导致增大 S_p 反而使 H 下降，而且偶然性访问某些页的页面失效率也会上升。当然，如果分配给该道程序的容量 S_1 增大，则可延缓后一种因素使 H 下降的情况发生。

论点 2 分配给某道程序的容量 S_1 的增大也只是在开始时对 H 提高有明显作用。

图4 - 24 的实线反映了用堆栈型替换算法时 H 与 S_1 的关系。由图可知，一开始随 S_1 的增大，H 明显上升，但到一定程度后，H 的提高就渐趋平缓，而且最高也不会到 1。当 S_1 过分增大时，主存空间的利用率会因程序的不活跃部分所占比例增大而下降。如果采用 FIFO 算法替换，由于它不是堆栈型算法，随着 S_1 的增大，H 总的趋势也是上升的，但是从某个局部看，可能会有下降，如图 4 - 24 中虚线所示。这种现象同样会体现在 S_p、S_1 与 H 的关系上。

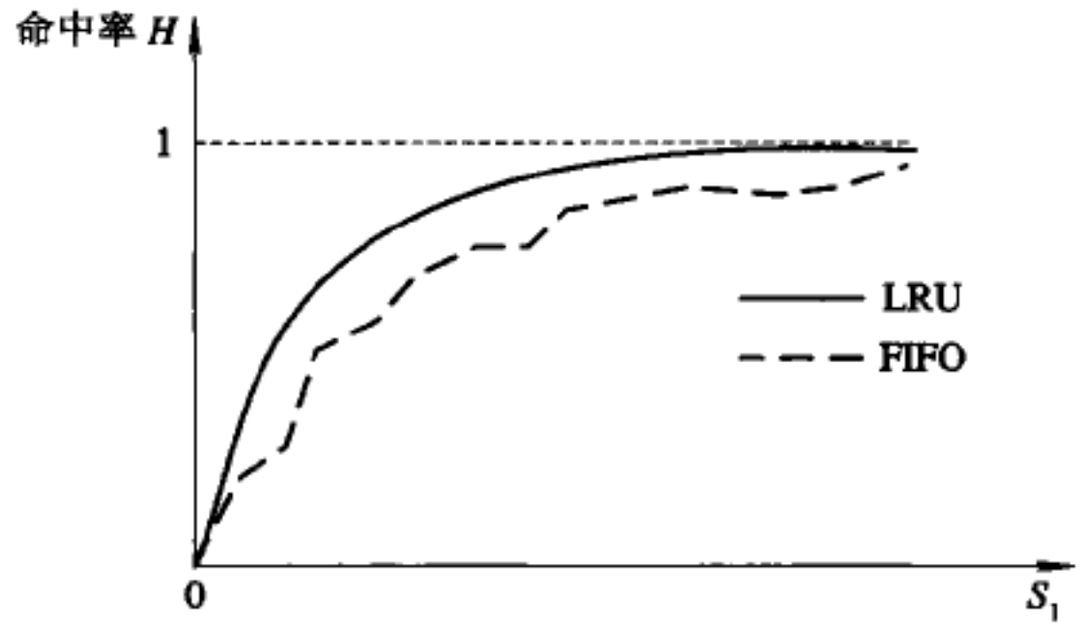


图 4 - 24 H 与 S_1 的关系

由以上分析得出结论，不要让 S_1 过大。S_p 和 S_1 的选择应折中权衡，只要 H 高到不会再明显增大时就可以了。目前多数机器取 1～4 KB 的页面大小。尽管页面取大会使页内零头浪费增大，降低主存的空间利用率，但页表行数的减小又缓解了主存空间利用率的下降。同时，页面取大有利于提高辅存的调页效率，减小操作系统为页面替换所花的辅助开销，还可降低指令、操作数和字符串的跨页存储概率。

主存命中率也与所用的页面调度策略有关。大多数虚拟存储器都采用请求式调页，仅当页面失效时才把所需页调入主存。针对程序存在的局部性，可改用预取工作区调度策略。所谓工作区，是指在时间 t 之前一段时间 Δt 内已访问过的页面集合。程序的局部性使工作区随时间 t 缓慢变化。可以在启动某道程序重新运行之前，先将该程序上次运行时所用的虚页集合调入主存。这种预取工作区的方法可以免除在程序启动后出现大量的页面失效，使命中率有所提高。但应看到这种调度策略不见得一定比请求式的页面调度策略好，因为可能会把许多不用的虚页也调入主存，所以是否采用应根据具体情况决定。

以上主要是围绕某道程序讨论的，如果要讨论如何提高多道程序运行时 CPU 的效率，则还要考虑一些其他因素。

例如，对分时系统，分配给每道程序的 CPU 时间片大小会影响对虚拟存储器的使用。如果分配给 CPU 的时间片较小，就应尽量减少页面失效的次数，不然所给时间的大部分就会消

耗在调页上。但此时对 S_i 的要求却可降低，因为在短的 CPU 时间内，来得及使用的主存容量会较少。同理，页面也不能选得过大，不然会出现连一页也没用完，就得换道了。

又如，多道程序的道数取多少，也会影响到 CPU 的效率。道数太少，由于调页时 CPU 可能没有可以运行的程序而不得不停下来等待，使效率降低；反之当道数太多时，每道程序占有的主存页数太少，会产生频繁的页面失效。为此提出了多种多道程序系统优化 CPU 效率的调页模型。下面列举三种。一种认为应遵循所谓 50%准则，即如果调页操作能使辅存约有一半时间在忙着的，CPU 的利用率视为最大。另一种认为当调页时间近似等于页面失效间隔的平均时间时，CPU 的利用率最高。第三种认为每道程序的页面分配量应选择成能使页面失效间隔的平均时间达到最大值。按照这些见解可以提出调整道数(并行作业数)的算法以及使系统吞吐量最大的存储管理策略。

4.3 高速缓冲存储器

4.3.1 工作原理和基本结构

高速缓冲(Cache)存储器是指为弥补主存速度的不足，在处理机和主存之间设置一个高速、小容量的 Cache，构成 Cache—主存存储层次，使之从 CPU 来看，速度接近于 Cache，容量却是主存的。

Cache 存储器的基本结构如图 4 - 25 所示。

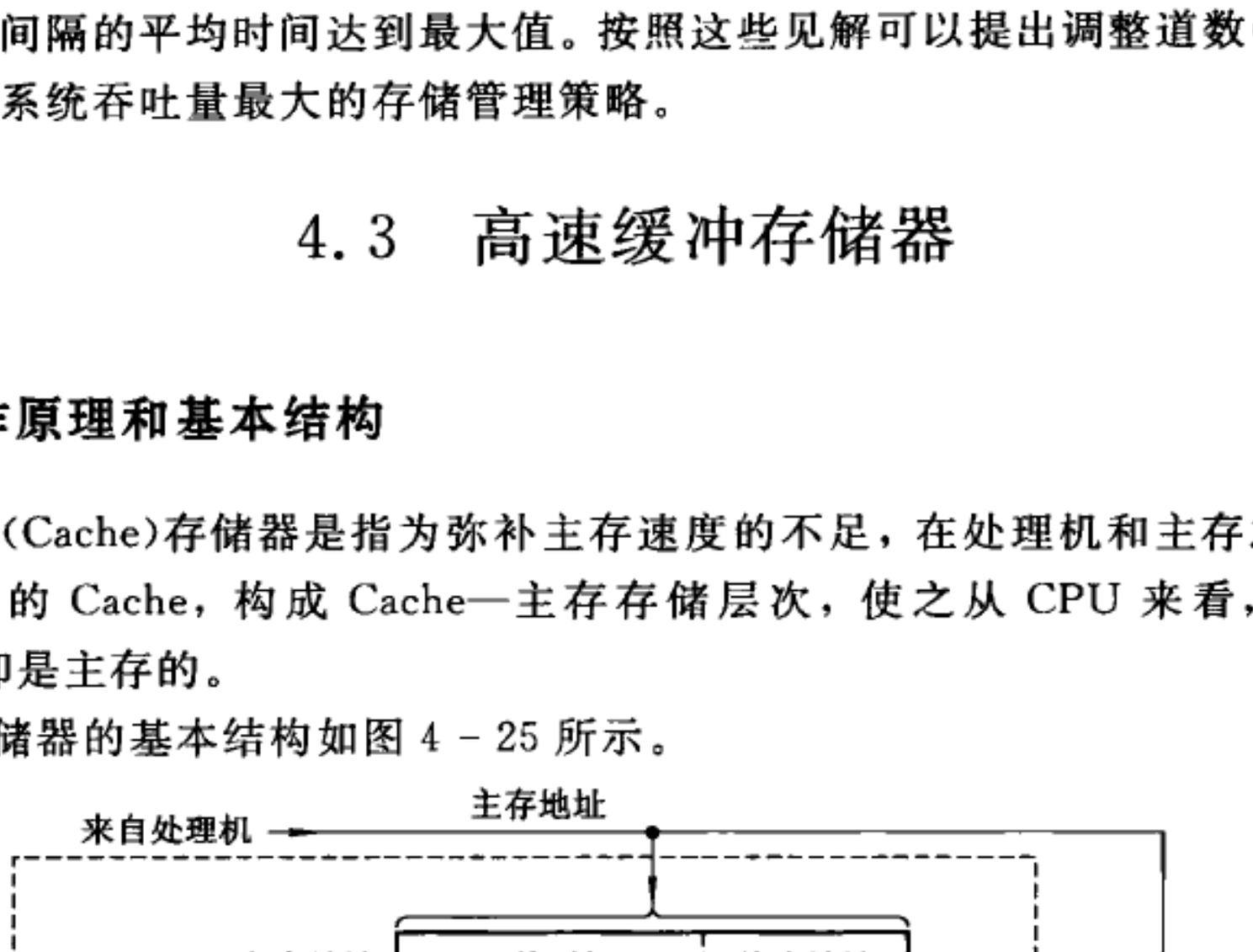


图 4 - 25 Cache 存储器的基本结构

将 Cache 和主存机械等分成相同大小的块(或行)。每一块由若干个字(或字节)组成。从存储层次原理上讲，Cache 存储器中的块和虚拟存储器中的页具有相同的地位，但块的大小要比页的大小小得多，一般只是页的几十分之一或几百分之一。每当给出一个主存字地址进行访存时，都必须通过主存—Cache 地址映像变换机构判定该访问字所在的块是否

已在 Cache 中。如果在 Cache 中(Cache 命中)，主存地址经地址映像变换机构变换成 Cache 地址去访 Cache，Cache 与处理机之间进行单字信息传送；如果不在 Cache 中(Cache 不命中)，产生 Cache 块失效，这时就需要从访存的通路中把包含该字的一块信息通过多字宽通路调入 Cache，同时将被访问字直接从单字通路送往处理机。如果 Cache 已装不进了，发生了块冲突，就要将该块替换掉被选上的块，并修改地址映像表中有关的地址映像关系及 Cache 各块的使用状态标志等信息。

目前，访 Cache 的时间一般是访主存时间的 1/4～1/10，只要 Cache 的命中率足够高，就能以接近于 Cache 的速度来访问大容量主存。

可见，Cache 存储器和虚拟存储器在原理上是类似的，所以虚拟存储器中使用的地址映像变换及替换算法基本上也适用于 Cache 存储器。只是由于对 Cache 存储器的速度要求更高，因此在构成、实现以及透明性等问题上有其自己的特点。

前面已讲过，Cache 与主存的速度差不到 1/10，比主存与辅存之间的速度差小两个数量级，加上 Cache 存储器的速度要比虚拟存储器的高得多，希望能与 CPU 的速度相匹配，为此 Cache 本身一般采用与 CPU 同类型的半导体工艺制成。此外，Cache—主存间的地址映像和变换，以及替换、调度算法全得用专门的硬件实现。这样，Cache 存储器不仅对应用程序员是透明的，就是对系统程序员也是透明的，结构设计时必须解决好因为这种透明带来的问题。

由图 4 - 25 可知，从送入主存地址到 Cache 的读出或写入完成实际包括查表地址变换和访 Cache 两部分工作，这两部分工作所花费的时间基本相近，例如都是 30 ns。那么，可以让前一地址的访 Cache 和后一地址的查表变换在时间上重叠，流水地进行(见第 5、6 章)。虽然从送入主存地址到访 Cache 完成需要 60 ns，是处理机拍宽 30 ns 的 2 倍，但经流水后，CPU 仍可每隔 30 ns 完成一次对 Cache 的访问。实际上，访问一次 Cache 存储器往往要经过很多子过程。多个请求源同时访问 Cache 存储器时，首先要经优先级排队，然后访目录表进行地址变换，接着访 Cache，从 Cache 中选择所需的字和字节，修改 Cache 中块(行)的使用状态标志等，这些子过程流水地处理使 Cache 存储器能在一个周期为多条指令和数据服务，进一步提高 Cache 存储器的吞吐率。像 Amdahl470 V/7 和 IBM 3033 的 Cache 存储器都采用了流水技术。而且，IBM 3033 还采用异步流水。当某次访问 Cache 失效时，可以保存其请求，并让之后的其他访问 Cache 的请求继续进行，从而使各请求对 Cache 访问的完成次序可不同于它们进入的次序。

为了能更好地发挥 Cache 的高速性，减少 CPU 与 Cache 之间的传输延迟，应让 Cache 在物理位置上尽量靠近处理机或者就放在处理机中。对共用主存的多处理机系统，如果每个处理机都有它自己的 Cache，让处理机主要与 Cache 交换，就能大大减少使用主存的冲突，提高整个系统的吞吐量。

在处理机和 Cache、主存的联系上也不同于虚拟存储器的处理机和主存、辅存之间的联系方式。在虚拟存储器中，处理机和辅存之间没有直接的通路，因为辅存的速度相对主存的差距很大。一旦发生页面失效，由辅存调页的时间是毫秒级。为使处理机在这段时间内不致于白等，一般采用切换到其他程序的办法。但当 Cache 存储器发生 Cache 块失效时，由于主存调块的时间是微秒级，显然不能采用程序换道。为了减少处理机的空等时间，除了 Cache 到处理机的通路外，在主存和处理机之间还设有直接的通路，如图 4 - 25 所示。这样，Cache 块失效时，就不必等主存把整块调入 Cache 后，再由 Cache 把所需的字送入

处理机，而是让 Cache 调块与处理机访主存取所需的字重叠地进行，这就是通过直接通路实现读直达；同样，也可以实现 CPU 直接写入主存的写直达。这样，Cache 既是 Cache 存储器中的一级，又是处理机和主存间的一个旁视存储器。

为了加速调块，一般让每块的大小等于在一个主存周期内由主存所能访问到的字数，因此在有 Cache 存储器的主存系统都采用多体交叉存储器。

【例 4－5】 IBM 370/168 的主存是模 4 交叉，每个分体是 8 B 宽，所以 Cache 的每块为 32 B；CRAY－1 的主存是模 16 交叉，每个分体是单字宽，所以其指令 Cache(专门存放指令的 Cache)的块容量为 16 个字。

另外，主存是被机器的多个部件所共用的，应把 Cache 的访主存优先级尽量提高，一般应高于通道的访主存级别，这样在采用 Cache 存储器的系统中，访存申请响应的优先顺序通常安排成 Cache、通道、写数、读数、取指。因为 Cache 的调块时间只占用 1～2 个主存周期，所以这样做不会对外设访主存带来太大的影响。

4.3.2 地址的映像与变换

对 Cache 存储器而言，地址的映像就是将每个主存块按什么规则装入 Cache 中；地址的变换就是每次访 Cache 时怎样将主存地址变换成 Cache 地址。

映像规则的选择除了看所用的地址映像和变换硬件是否速度高、价格低和实现方便外，还要看块冲突概率是否低、Cache 空间利用率是否高。

所谓块冲突，是指出现了主存块要进入 Cache 中的块位置已被其他主存块占用了。

1. 全相联映像和变换

主存中任意一块都可映像装入到 Cache 中任意一块位置，如图4－26 所示。

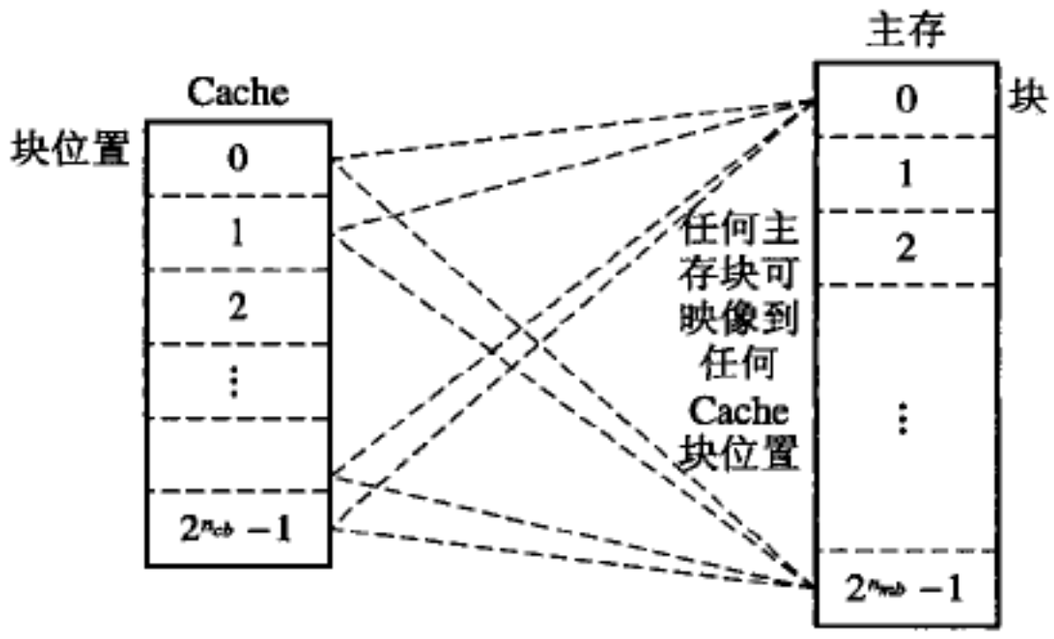


图 4－26　全相联映像规则图

为加快主存—Cache 地址的变换，不宜用类似虚拟存储器的(虚)页表法来存放主存—Cache 的地址映像关系，因为(虚)块表要用容量达 $2^{n_{mb}}$ 项的随机访问存储器，代价大，速度慢，所以都采用类似图 4－12 所示的目录表硬件方式实现。

全相联映像的主存—Cache 地址变换过程如图 4－27 所示。给出主存地址 n_m 访存时，将其主存块号 n_{mb} 与目录表中所有各项的 n_{mb} 字段同时相联比较。若有相同的，就将对应行的 Cache 块号 n_{cb} 取出，拼接上块内地址 n_{mr} 形成 Cache 地址 n_c，访 Cache；若没有相同的，表示该主存块未装入 Cache，发生 Cache 块失效，由硬件调块。

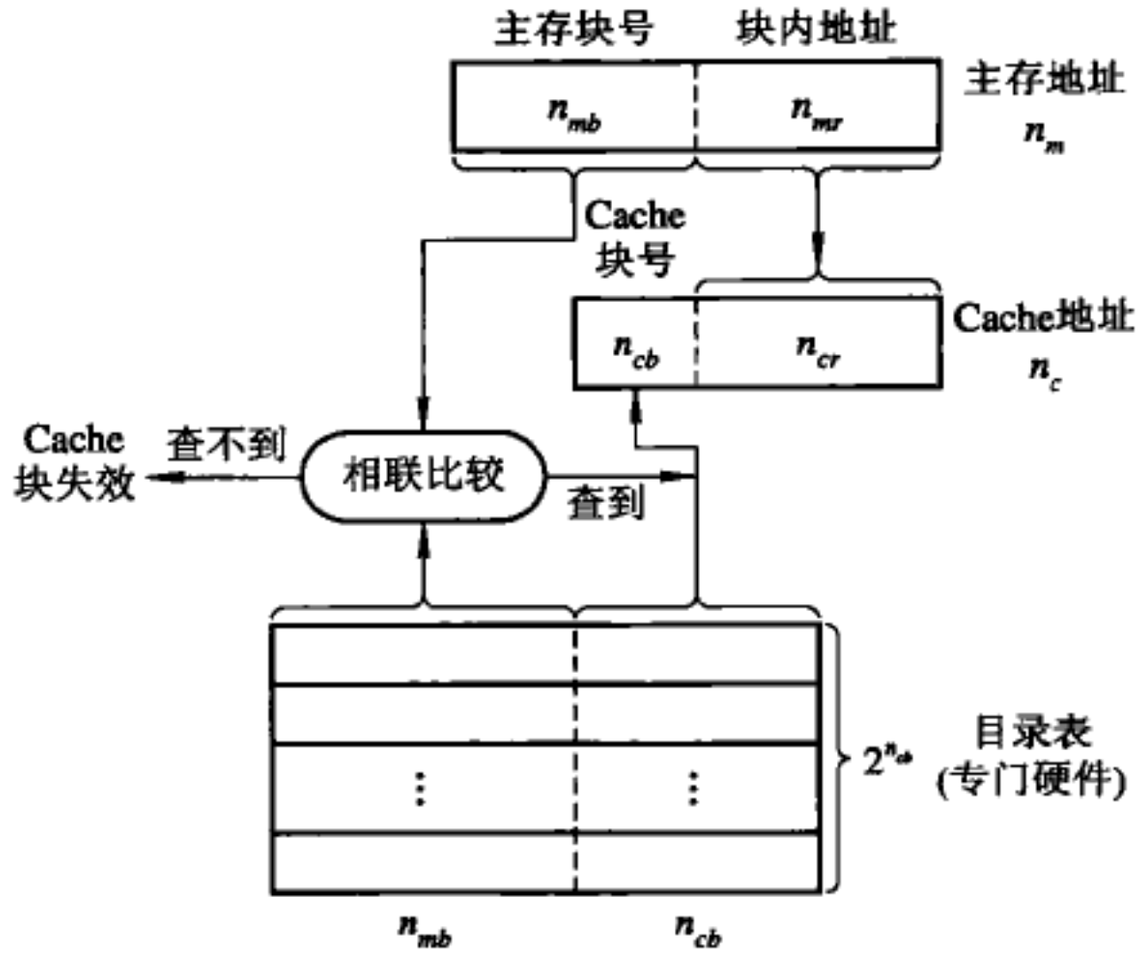


图 4 - 27　全相联映像的地址变换过程

全相联映像法的优点是块冲突概率最低，只有当 Cache 全部装满才可能出现块冲突，所以 Cache 的空间利用率最高。但要构成容量为 $2^{n_{cb}}$ 项的相联存储器，其代价太大，而且 Cache 容量很大时，其查表速度很难提高。那么，能否缩小和简化映像表机构，以加快相联查找呢？为此提出了直接映像规则。

2. 直接映像及其变换

把主存空间按 Cache 大小等分成区，每区内的各块只能按位置一一对应到 Cache 的相应块位置上，即主存第 i 块只能唯一映像到 $i \bmod 2^{n_{cb}}$ 块位置上，如图 4 - 28 所示。

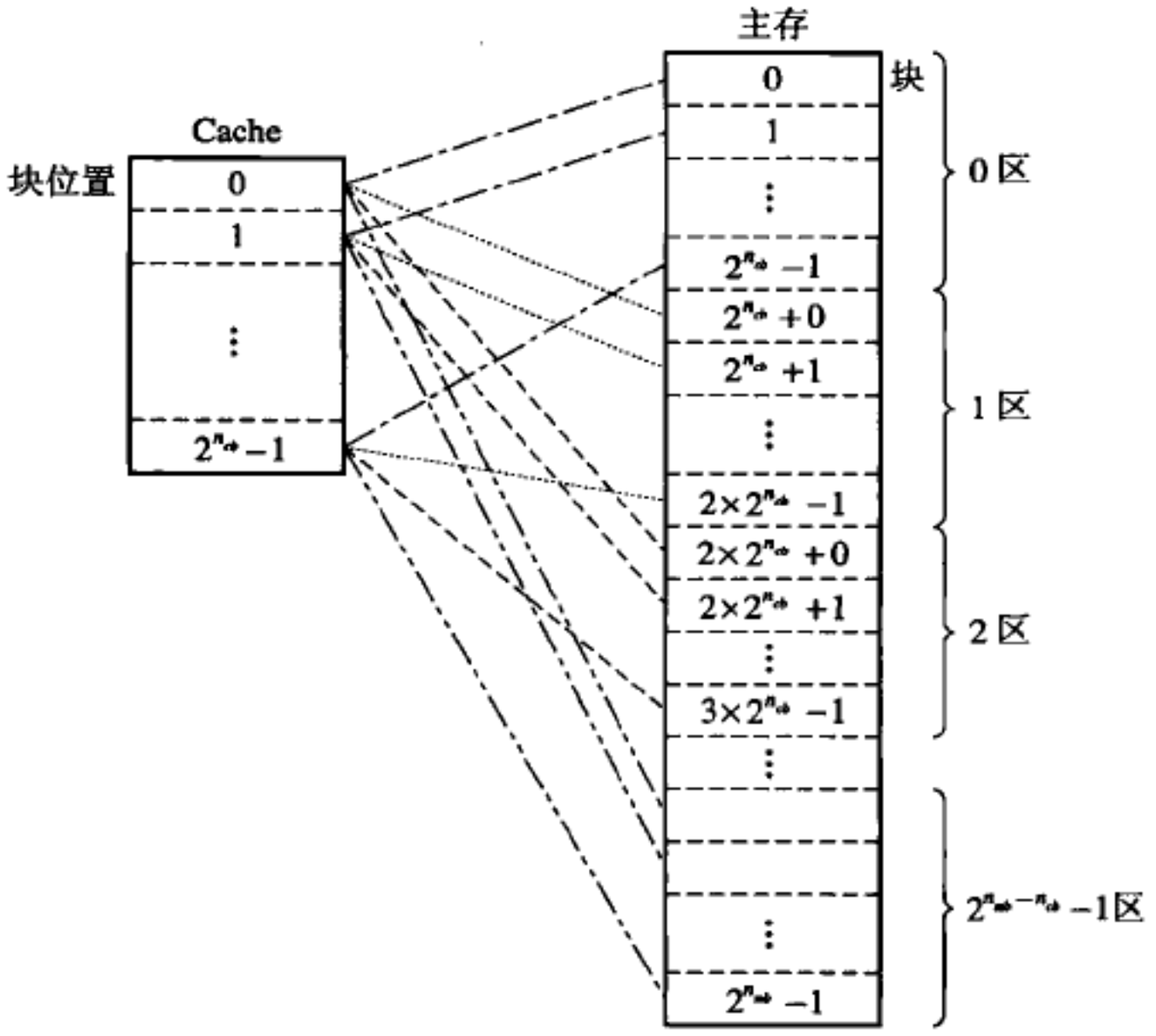


图 4 - 28　直接映像规则

按直接映像的规则，装入 Cache 中某块位置的主存块可以来自主存不同的区。为了区分装入 Cache 中的块是哪一个主存区的，需用一个按地址访问的表存储器来存放 Cache 中每一块位置目前是被主存中哪个区的块所占用的区号。因此，表存储器为 $2^{n_{cb}}$ 项，每项的区号标志字段为 $n_{mb}-n_{cb}$ 位宽，相当于可表示主存中 $2^{n_{mb}-n_{cb}}$ 个不同的区号。所以，当主存中第 i 块信息按直接映像规则装入 Cache 中第 j 块时，应将第 i 块在主存中的区号装入第 j 块对应的区号标志字段中。

直接映像的主存—Cache 地址变换过程如图 4 - 29 所示。处理机给出主存地址 n_m。访主存时，截取与 n_c 对应的部分作为 Cache 地址访 Cache，同时取 n_{cb} 部分作为地址访问区号标志表存储器，读出原先所存的区号标志与主存地址对应的区号部分进行比较。若比较相等，表示 Cache 命中，让 Cache 的访问继续进行，并中止访主存；如比较不等，表示 Cache 块失效，此时让 Cache 的访问中止，而让主存的访问继续进行，并由硬件自动将主存中该块调入 Cache。

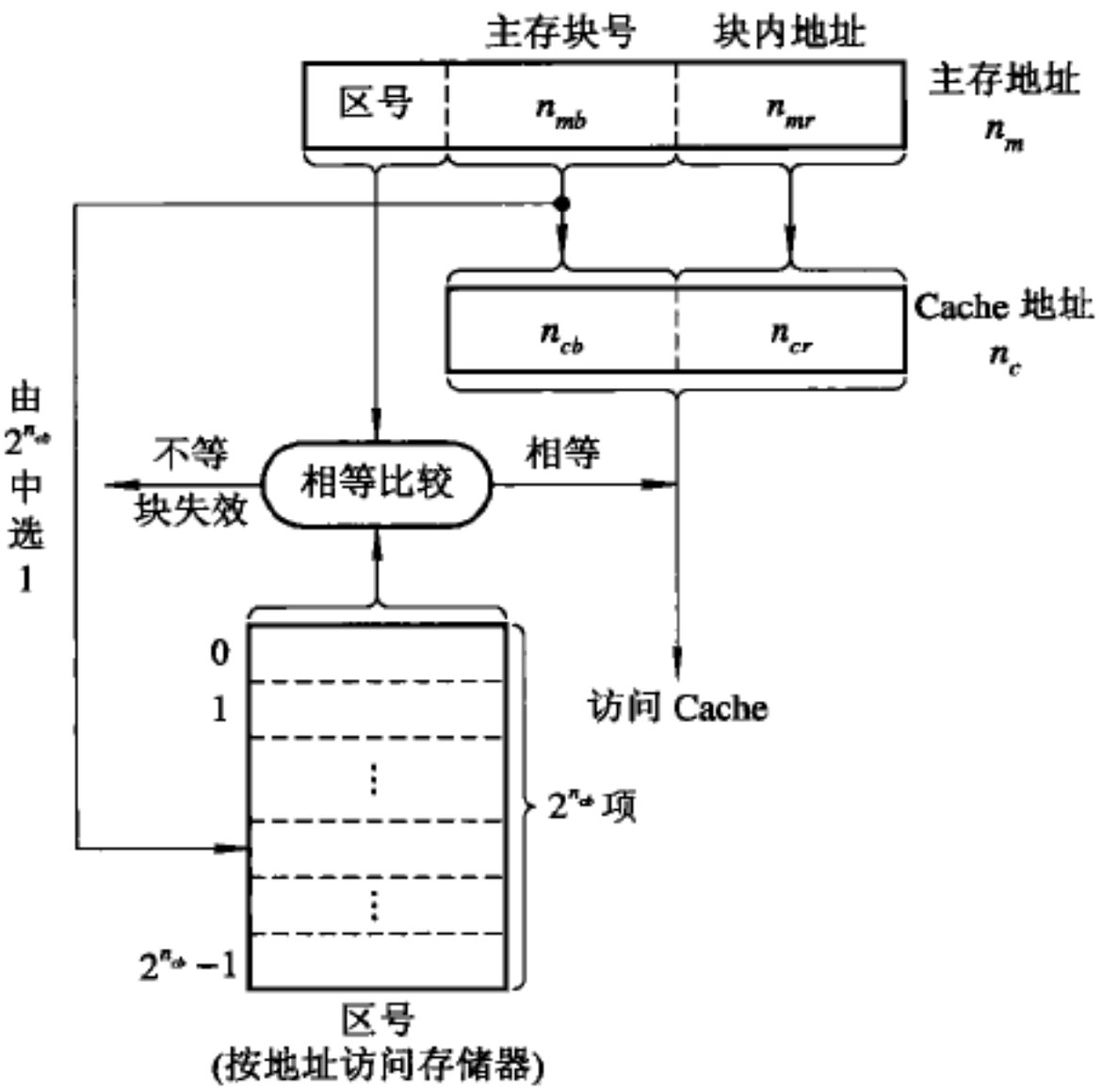


图 4 - 29　直接映像的地址变换过程

直接映像法的优点是节省所需硬件，只需容量较小的按地址访问的区号标志表存储器和少量外比较电路，成本很低。访问 Cache 与访问区号表、比较区号是否相符的操作是同时进行的，当 Cache 命中时意味着省去了地址变换的时间。其致命缺点是 Cache 的块冲突概率很高。只要有两个或两个以上经常使用的块恰好被映像到 Cache 同一块位置，就会使 Cache 命中率急剧下降，即使此时 Cache 中有大量空闲块也无法利用，所以 Cache 的空间利用率很低。因此，目前已很少使用直接映像规则了。

3. 组相联映像及其变换

全相联映像和直接映像的优、缺点正好相反，那么能否将两者结合，采用一种映像规则，既能减少块冲突概率，提高 Cache 空间利用率，又能使地址映像机构及地址变换速度比全相联的简单和快速呢？组相联映像就是其中的一种。

用简例来说明这种规则。如图 4 - 30 所示，将 Cache 空间和主存空间都分成组，每组为 S 块($S=2^s$)。Cache 共有 $2^{n_{cb}}$ 个块，分成 Q 组($Q=2^q$)，整个 Cache 是一区。主存分成与 Cache 同样大小的 2^{n_d} 个区，其地址按区号、组号、组内块号、块内地址分成对应的 4 个字段。主存地址的组号、组内块号分别用 q、s' 字段表示，它们的宽度和位置与 Cache 地址的 q、s 是一致的。

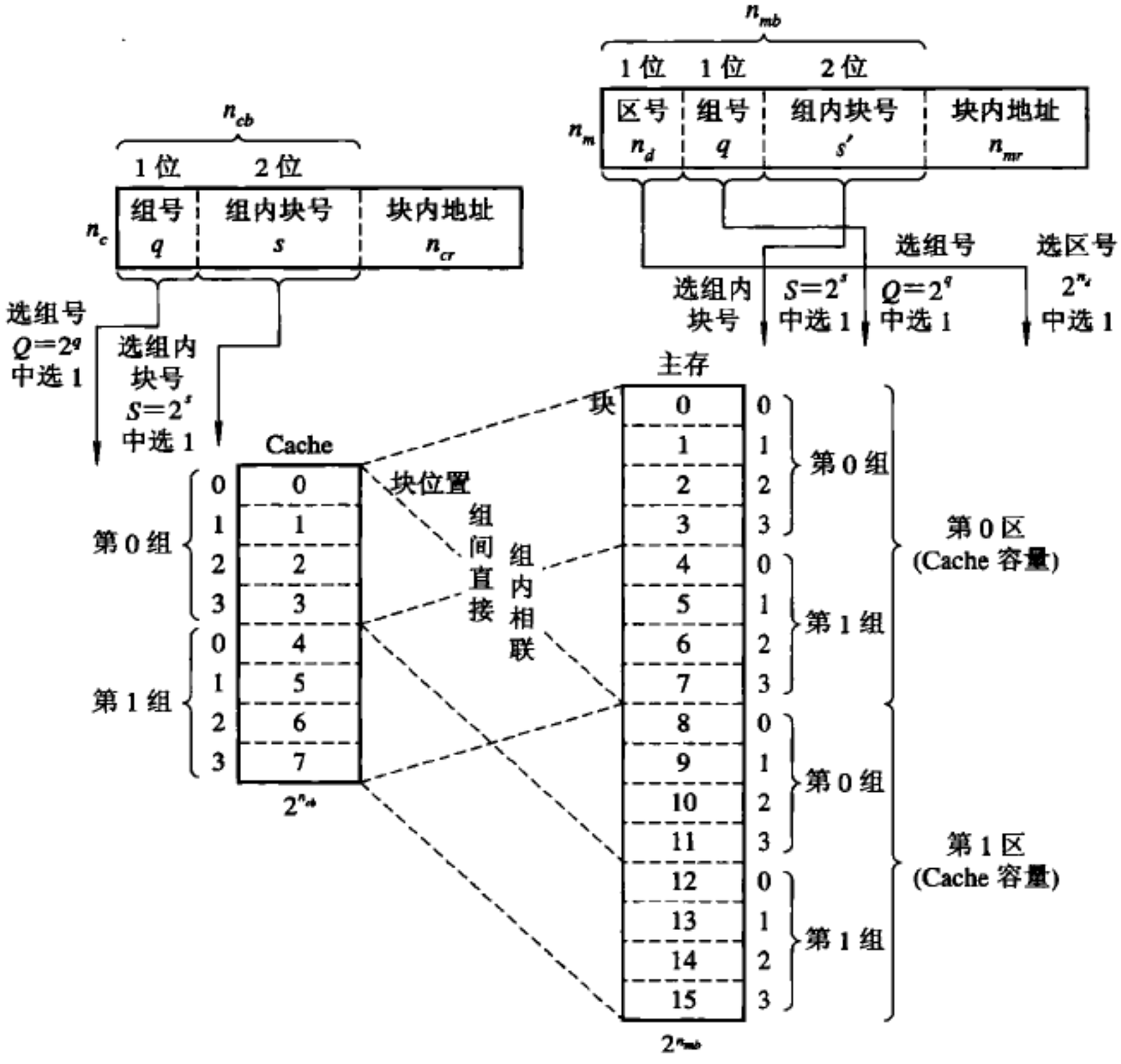


图 4 - 30　组相联映像规则

组相联映像指的是各组之间是直接映像，而组内各块之间是全相联映像。图中，n_d、q 都是 1 位，s 是 2 位，即主存的第 0 组只能进入 Cache 的 0 组，第 1 组只能进入 Cache 的 1 组。组内的各个块，如主存的 0，1，2，3 及 8，9，10，11 块可进入 Cache 的第 0，1，2，3 中任意一块，但不能进入 Cache 的 4，5，6，7 块。从图中看出，组相联映像是介于全相联映像与直接映像之间的。它的 Cache 块冲突概率要比直接映像的低得多。例如，当主存 0 块已在 Cache 的 0 块中，若要调入主存第 8 块，则对直接映像来说就要发生块冲突，而对组相联映像来说，第 8 块仍可进入 Cache 的 1、2、3 块中的任意一块位置。只有当 Cache 中第 0 组各块都被占用时，才出现块冲突，即使第 1 组中有空块也无用。显然，Cache 中第 0 组各块都被占用的概率要小得多，因此，大大降低了块冲突概率，同时也就大大提高了 Cache 空间的利用率。S 值越大，Cache 块冲突概率越低，当然仍比全相联的要高。

当组相联映像的 S 值大到等于 Cache 的块数(即 $s=n_{cb}$)时就成了全相联映像，而当 S 值小到只有 1 块(即无 s 字段)时就变成了直接映像。因此全相联映像和直接映像只是组相

联映像的两个极端。在 Cache 空间大小及块的大小都已定的情况下，Cache 的总块数就定了，但结构设计时仍可对 S 和 Q 值进行选择。Q 和 S 的选取主要依据于对块冲突概率、块失效率、映像表复杂性和成本、查表速度等的折中权衡。组内块数 S 愈多，块冲突概率和块失效率愈低，映像表越复杂，成本越高，查表速度越慢，所以通常通过在典型工作负荷下进行模拟来确定。

组相联映像比全相联映像在成本上要低得多，而性能上仍可接近于全相联映像，所以获得了广泛应用。

【例 4-6】 Intel i860 的数据 Cache 和 Motorola 88110 的指令 Cache 均采用 128 组，每组 2 块；Amdahl 470 V/6 采用 256 组，每组 2 块；VAX-11/780 采用 512 组，每组 2 块；Intel 80486 和 Honeywell 66/60 均采用 128 组，每组 4 块；Amdahl 470 V/8 采用 512 组，每组 4 块；而 Amdahl 470 V/7 和 IBM 370/168-3 均采用 128 组，每组 8 块；IBM 3033 则采用 64 组，每组 16 块。

前面在全相联中讲过的目录表法同样可用于实现组内的全相联，此时目录表的行数可从全相联的 $2^{n_{cb}}$ 减少到 2^s。因为各组间是直接映像，所以组号 q 可照搬而不参与相联比较。实现时对应每一组都有一个目录表，共有 2^q 个目录表。每个目录表只需 2^s 行、n_d+2s 位宽，其中参与相联比较的位数为 n_d+s，它们比全相联目录表的 $2^{n_{cb}}$ 行、$n_{mb}+n_{cb}$ 位宽、n_{mb} 位参与相联比较的位数都要小得多，这均使查表速度得到提高。

组相联的地址变换原理如图 4-31 所示。先由 q 在 2^q 组中选出一组，对该组再用 n_d+s' 进行相联查找，若在 2^s 行中查不到相符的，表示主存该块不在 Cache 中；如果查到有相符的，则将表中相应的 s 拼上 q 和 n_{mr} 就是 Cache 地址 n_c。

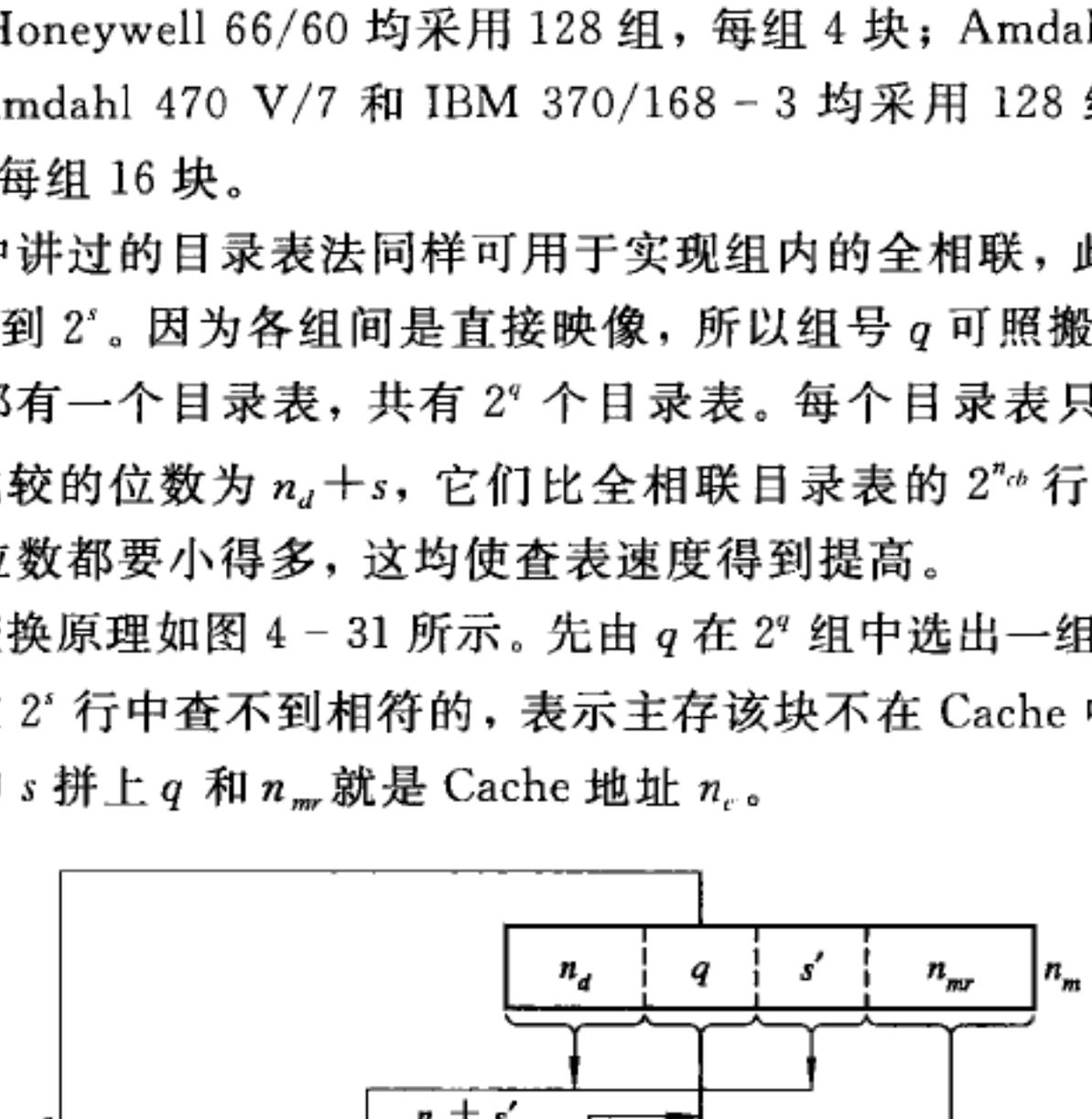


图 4-31　组相联地址变换原理

组相联地址映像机构也可以采用按地址访问与按内容访问混合的存储器实现，其存储总容量应为 $2^{n_{cb}}$ 行。办法之一是使用 3.1.2 节中讲过的单体多字并行存储器，如图 4-32 所示。先由 q 从 2^q 中选出一个单元，由该单元同时读出 2^s 个字，分别通过 S 套外比较电路

与主存地址的 n_d+s' 同时比较。将其中比较符合的 s 取出拼上 q 和 n_{mr} 即为 Cache 地址 n_c。如果都不相符，表示该块不在 Cache 中，出现块失效。显然，这种方法的 S 值不能很大(如图 4 - 32 中 S 为 4)。

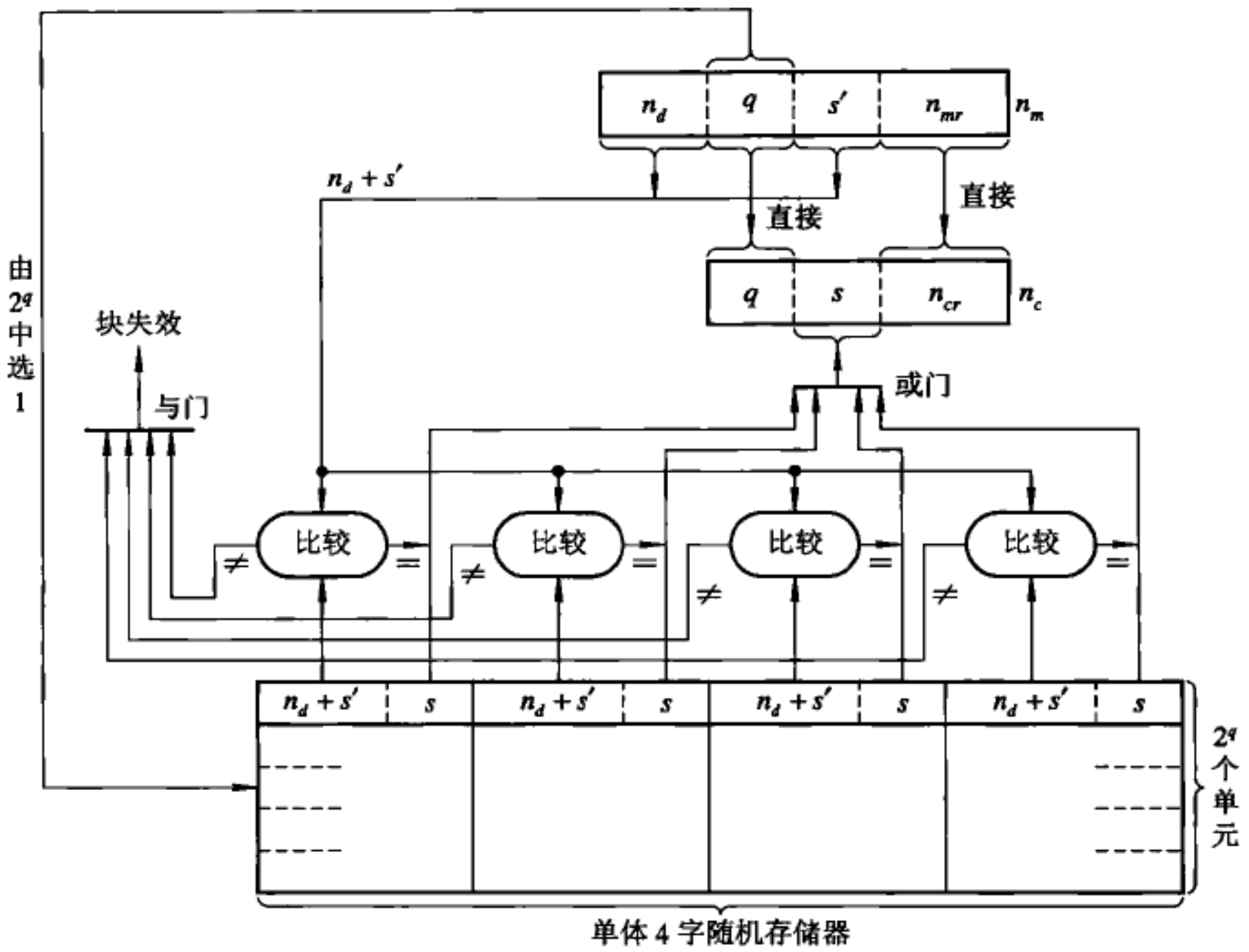


图 4 - 32　组相联地址变换的一种实现方式

应当强调的是，采用组相联并不是操作系统或存储层次的要求，只是在全相联的速度满足不了要求时才不得已而采用的，以便于实现，尽管这样做会增加一些 Cache 块冲突概率和降低一些 Cache 空间利用率。随着半导体集成电路技术的发展，组内块数 S 还可增大，以进一步降低 Cache 块冲突概率。

在全相联、直接、组相联映像的基础上还可以有各种变形，段相联就是一例。段相联实质上是组相联的特例。它采用组间全相联、组内直接映像。为了与组相联映像加以区别，将这种映像方式称为段相联。就是说，段相联映像是把主存和 Cache 分成具有相同的 Z 块的若干段，段与段之间采用全相联映像，而段内各块之间却采用直接映像。如图 4 - 33 所示，主存中段 0、段 1、…、段 $(2^{n_{mb}}/Z)-1$ 中的第 i 块可以映

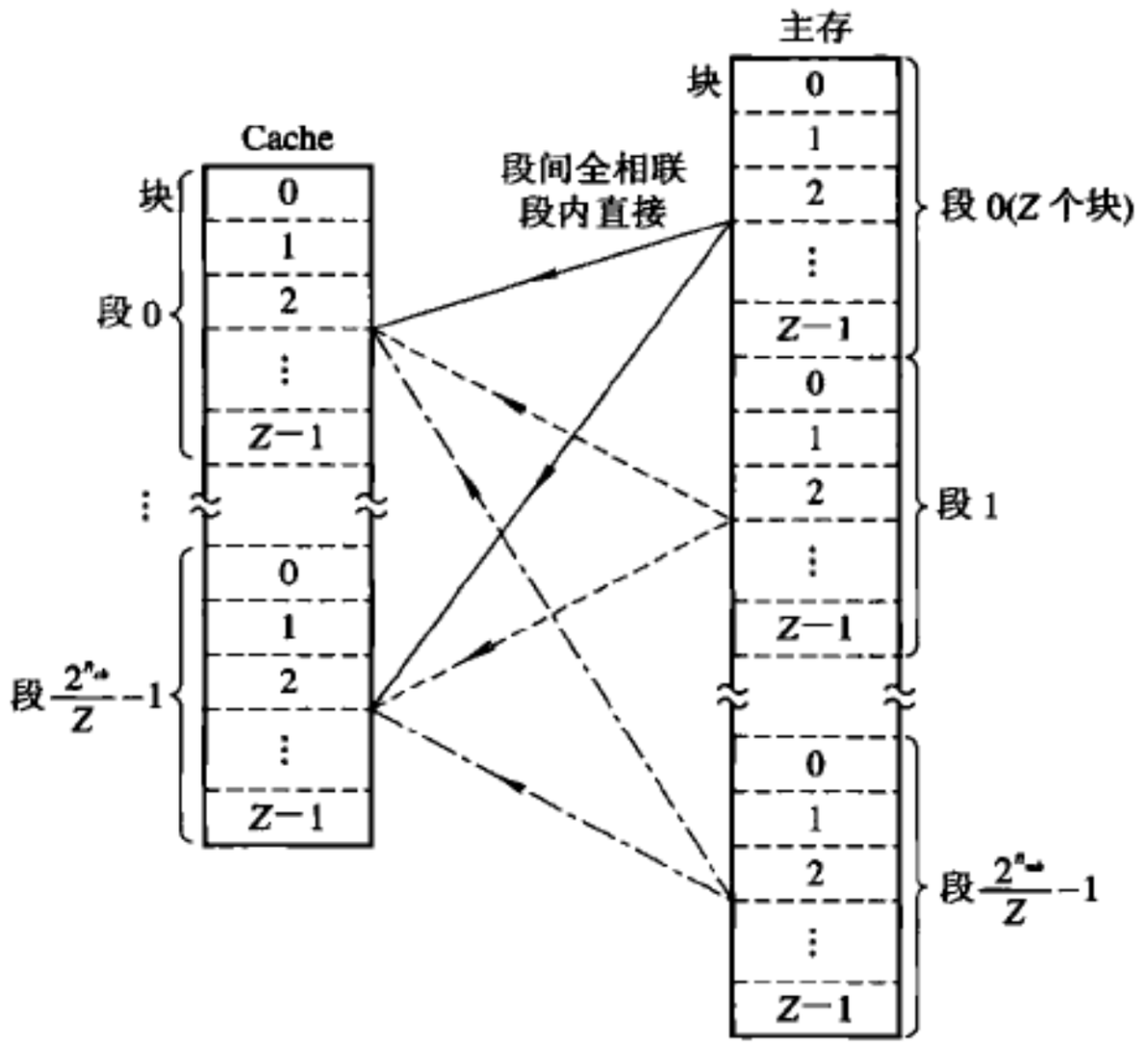


图 4 - 33　具有每段 Z 个块的段相联映像

像装入 Cache 中段 0、段 1、……、段($2^{n_{cb}}/Z$)－1 中的第 i 块位置。

显然，采用段相联映像的目的也和采用组相联映像的目的一样，主要是减小相联目录表的容量，降低成本，提高地址变换的速度。相联目录表由 Z 个组成，每个目录表的行数由 $2^{n_{cb}}$ 行减为 $2^{n_{cb}}/Z$。当然，其 Cache 块冲突概率将比全相联的高。

4.3.3 Cache 存储器的 LRU 替换算法的硬件实现

当因 Cache 块失效而将主存块装入 Cache 又出现 Cache 块冲突时，就必须按某种替换策略选择 Cache 中的一块替换出去。Cache 存储器的替换算法与虚拟存储器的一样，也是用 FIFO 算法或 LRU 算法，其中 LRU 算法最为常用。

在 4.3.1 节中已讲过，Cache 的调块时间是微秒级的，不能采用程序换道。为了减少处理机空等的时间，Cache 存储器中的替换算法只能由全硬件实现。本节介绍 LRU 算法的比较对法。

比较对法的基本思路是让组内各块成对组合，用一个触发器的状态表示该比较对内两块访问的远近次序，再经门电路就可找到 LRU 块。如有 A、B、C 3 块，组成 AB、AC、BC 3 对。各对内块的访问顺序分别用“对触发器”T_{AB}、T_{AC}、T_{BC}表示。T_{AB}为“1”，表示 A 比 B 更近被访问过；T_{AB}为“0”，表示 B 比 A 更近被访问过。T_{AC}、T_{BC}也有类似定义。这样，若访问过的次序为 ABC，即最近被访问过的为 A，最久未被访问的是 C，则这三个触发器状态是 $T_{AB}=1$，$T_{AC}=1$，$T_{BC}=1$。如果访问过的次序是 BAC，C 为最久未被访问过，则有 $T_{AB}=0$、$T_{AC}=1$，$T_{BC}=1$。因此 C 作为最久未被访问过的替换块的话，用布尔代数式表示必有

$$C_{LRU} = T_{AB}T_{AC}T_{BC} + \overline{T}_{AB}T_{AC}T_{BC} = T_{AC}T_{BC}$$

同理可得

$$B_{LRU} = T_{AB}\overline{T}_{BC}$$

$$A_{LRU} = \overline{T}_{AB}\overline{T}_{AC}$$

因此，LRU 算法完全可用与门、触发器等硬件组合实现，如图 4－34 所示。

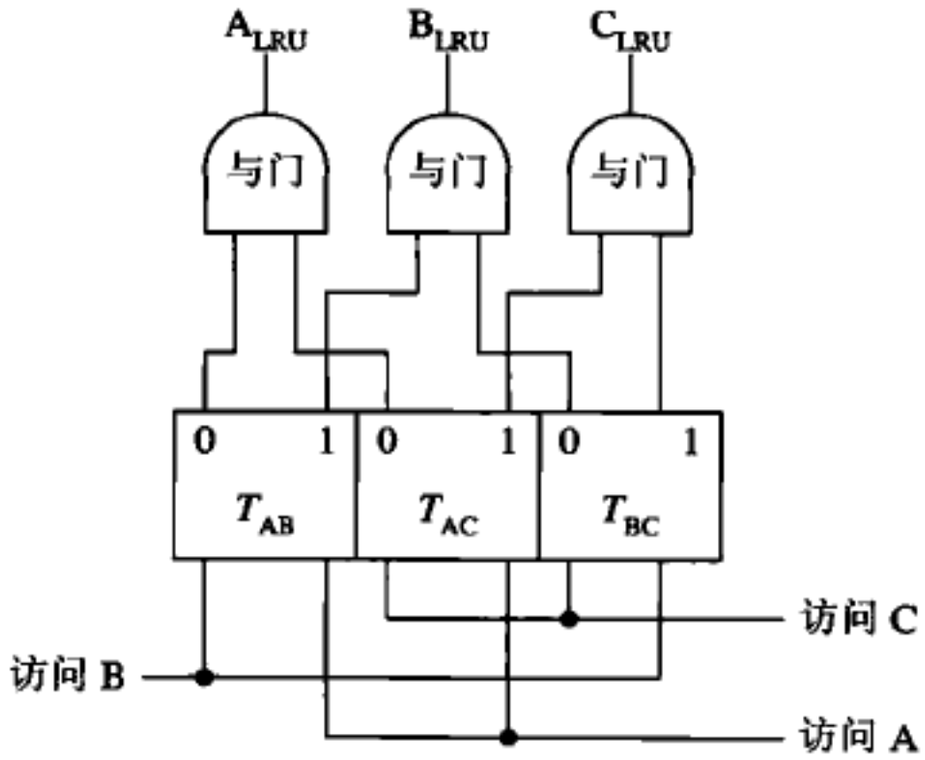


图 4－34　用比较对法实现 LRU 算法

每次访问某块时，应改变与该块有关的比较对触发器的状态。以上述 A、B、C 3 块为例，每次访问 A 后需改变与 A 有关的比较对触发器的状态，置 T_{AB}、T_{AC}为“1”，以反映 A

比 B 更近、A 比 C 更近访问过；同理，访问 B 后，置 T_{AB}为“0”、T_{BC}为“1”；访问 C 后，置 T_{AC}、T_{BC}为“0”。据此可定出各比较对触发器的输入控制逻辑如图 4－34 所示。

现在来分析比较对法所用的硬件。由于每块均可能作为 LRU 块，其信号需用与门产生，所以有多少块，就得有多少个与门；每个与门接收与它有关的比较对触发器来的输入，例如 A_{LRU}与门要有从 T_{AB}、T_{AC}来的输入，B_{LRU}要有从 T_{AB}、T_{BC}来的输入，而与每块有关的比较对触发器数为块数减 1，所以与门的输入端数是块数减 1。若 P 为块数，两两组合，则比较对触发器数为 $C_P^2 = P(P-1)/2$。表 4－2 列出了比较对法块数 P 与门数、门的输入端数及比较对触发器数的关系。

表 4－2　比较对触发器数、门数、门的输入端数与块数的关系

块数	3	4	8	16	64	256	…	P
比较对触发器数	3	6	28	120	2016	32 640	…	$\frac{P(P-1)}{2}$
门数	3	4	8	16	64	256	…	P
门的输入端数	2	3	7	15	63	255	…	$P-1$

从表 4－2 可以看出，比较对触发器的个数会随块数的增多以极快的速度增加，门的输入端数也线性增加，这在工程实现上会带来麻烦，所以比较对法只适用于组内块数较少的组相联映像 Cache 存储器中。在块数少时，它比较容易实现。若组内块数超过 8，则所需比较对触发器个数就多得不能承受了。不过这时也还可以用多级状态位技术来减少所用的比较对触发器个数。

【例 4－7】 IBM 3033，组内块数为 16，可分成群、对、行 3 级。先分成 4 群，选 LRU 群需 6 个比较对触发器。每群再分成两对，由一位触发器的状态选 LRU 对，这样 4 个群需 4 位。而每对中的 LRU 行又需用一位触发器的状态指示，这又要 8 位。所以，全部触发器数就成了 6(选群)＋4(选对)＋8(选行)，共 18 个，比单级的 120 个比较对触发器要少得多，但这是以牺牲速度为代价的。就是在组内块数为 8 时，若采用对、行二级，也能使触发器数由 28 个减少到 6＋4 共 10 个，IBM 370/168－3 就是如此。

4.3.4　Cache 存储器的透明性及性能分析

1. Cache 存储器的透明性分析及解决办法

由于 Cache 存储器的地址变换和块替换算法是由全硬件实现的，因此 Cache 存储器对应用程序员和系统程序员都是透明的，而且 Cache 对处理机和主存之间的信息交往也是透明的。对于 Cache 透明所带来的问题和影响必须仔细分析，并采取相应的办法来妥善解决。

虽然 Cache 是主存的一部分副本，主存中某单元的内容却可能在一段时期里与 Cache 中对应单元的内容不一致。例如，中央处理机写 Cache，修改了 Cache 中某单元的内容，但主存中对应此单元的内容没有改变。这时如果 CPU、I/O 处理机和其他处理机要经主存交换信息，那么这种主存内容跟不上 Cache 对应内容变化的不一致就会造成错误。同样，I/O 处理机或其他处理机把新的内容送入主存某个区域，而 Cache 中对应此区域的副本内容却仍是原来的。这时，如果 CPU 要从 Cache 中读取信息，也会因这种 Cache 内容跟不上主存

对应内容变化的不一致而造成错误。因此，必须采取措施解决好由于读/写过程中产生的Cache和主存对应内容不一致的问题。

解决因中央处理机写Cache使主存内容跟不上Cache对应内容变化造成不一致问题的关键是选择好更新主存内容的算法。一般可有写回法和写直达法两种。写回法也称为抵触修改法。它是在CPU执行写操作时，只将信息写入Cache，仅当需要替换时，才将改写过的Cache块先写回主存，然后再调入新块。因此，在主存－Cache的地址映像表中需为Cache中每个块设置一个“修改位”，作为该块装入Cache后是否被修改过的标志。只要修改过，就将该标志位置成“1”。这样在块替换时，根据该块的修改位是否为“1”，就可以决定替换时是否需要先将该块存回主存。写直达法也称存直达法。它利用Cache存储器在处理机和主存之间的直接通路，每当处理机写入Cache的同时，也通过此通路直接写入主存。这样在块替换时，不必先写回主存就可调入新块。写回法把开销花在每次要替换的时候，写直达法则把开销花在每次写Cache时都要增加一个比写Cache时间长得多的写主存时间。

写回法和写直达法都需要有少量缓冲器。写回法中缓冲器用于暂存将要写回的块，使之不必等待被替换块写回主存后才开始进行Cache取。写直达法中则用于缓冲由写Cache所要求的要写回主存的内容，使CPU不必等待这些写主存完成就能往下运行。缓冲器由要存的数据和要存入的目标地址组成。在写直达系统中容量为4的缓冲器就可以显著改进其性能，IBM 3033就是这样用的。需要注意的是，这些缓冲器对Cache和主存是“透明”的。在设计时，要处理好可能由它们所引起的错误(如另一个处理机要访问的主存单元的内容正好仍在缓冲器中)。

据对典型程序的统计，在所有访存中约有10%～34%，甚至更多的是写操作。虽然写回法是写回整个块，不是只写回一个字或两个字，但写回法几乎总是使主存的通信量比写直达法的要小得多。例如，设Cache不命中率为3%，块的大小为32个字节，主存模块宽8个字节，写操作占16%，且所有Cache块的30%需要写回操作，则写主存次数与总的访主存次数之比，写直达法为16%，而写回法仅为3.6%(0.03×0.30×32/8)。处理机的不少写入是暂存中间结果，采用写回法有利于省去许多将中间结果写入主存的无谓开销。但是，写回法增加了Cache的复杂性，需要设置修改位以确定是否需要写回以及控制先写回后才调入的执行顺序。而且写回法在块替换前，仍然会存在主存内容与Cache内容的不一致。

从可靠性上讲，写回法不如写直达法好。写直达法在Cache出错时可以由主存来纠正，因此Cache中只需有一位奇偶校验位。写回法则由于有效的块只在Cache中，因此需要在Cache中采用纠错码，即需要在Cache中增加更多的冗余信息位来提高其内容的可靠性。

很难对写直达法和写回法进行明确的选择。写直达法需要花费大量缓冲器和其他辅助逻辑来减少CPU为等待写主存完成所耗费的时间，而写回法的实现成本要低得多。目前，采用写回法的有Amdahl的所有机器、IBM 3081；采用写直达法的有IBM 370/168、IBM 3033、PDP－11/70、VAX－11/780、Honeywell 66/60及66/80等。

采用写回法还是写直达法与系统应用有关。单处理机系统的Cache存储器，多数用写回法以节省成本。共享主存的多处理机系统，为保证各处理机经主存交换信息时不出错，

则多用写直达法。

如果由多个处理机共享主存交换信息改成共享 Cache 交换信息，信息的一致性就能得到保证，但目前多个中央处理机共享 Cache 尚有不少困难。一是要求 Cache 的容量必须大大增加才行；二是要让共享 Cache 在物理位置上与多个 CPU 都靠得很近来减少其间的延时也很困难，这都会降低 Cache 的速度。此外，Cache 的频宽尚难以支持两个以上 CPU 的同时访问。因此，共享 Cache 的办法目前只限于用在单 CPU、多 I/O 处理机系统上。例如，Amdahl 470 机的 CPU 与 I/O 处理机就是共用的同一个 Cache，让 Cache 在物理位置上靠近 CPU，而与其他 I/O 处理机距离较远，虽然这会使访问延迟增大，但由于是输入/输出数据的传送，延迟大些影响并不大。

对于共享主存的多 CPU 系统，绝大多数还是使各个 CPU 都有自己的 Cache。在这样的系统中由于 Cache 的透明性，仅靠采用写直达法并不能保证同一主存单元在各 Cache 中的对应内容都一致。例如，处理机 A 和处理机 B 通过各自的 Cache a 和 Cache b 共享主存，如图 4 - 35 所示。当处理机 A 写入 Cache a 的同时，采用写直达法也写入了主存，如果恰好 Cache b 中也有此单元，则其内容并未改变，此时若处理机 B 也访问此单元时读到的就会是原先的内容而出错。因此，还需要采取措施保证让各个 Cache 有此单元的内容都一致才行。

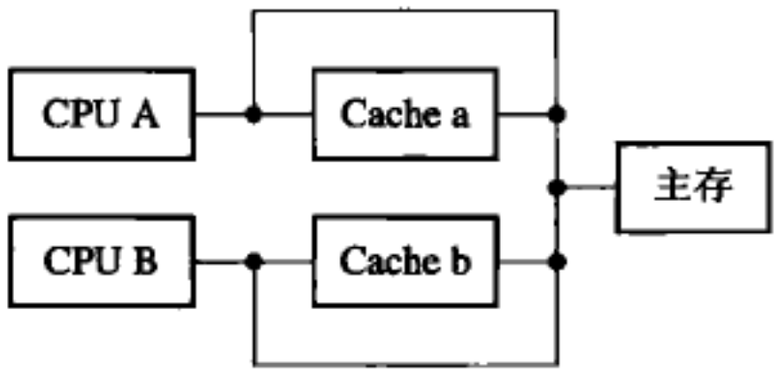


图 4 - 35　每个处理机都有 Cache 的共享主存多处理机系统

一种解决办法就是采用播写法。所谓播写法，是指任何处理机要写入 Cache 时，不仅写入自己 Cache 的目标块和主存中，还把信息或者播写到所有 Cache 有此单元的地方，或者让所有 Cache 有此单元的块作废(以便下次访问时按缺块处理，从主存中调入)。采用作废的办法可以减少播送的信息量，IBM 370/168、IBM 3033 都采用的是这种办法。

另一种办法是控制某些共享信息(如信号灯或作业队等)不得进入 Cache。还有一种办法是目录表法，即在 CPU 读、写 Cache 不命中时，先得查在主存中的目录表，以判定目标块是否在别的 Cache 内，以及是否正在被修改等，然后再决定如何读、写此块。

Cache 内容跟不上主存内容变化问题的一种解决办法是，当 I/O 处理机未经 Cache 往主存写入新内容的同时，由操作系统经专用指令清除整个 Cache。这种办法的缺点是 Cache 存储器对操作系统和系统程序员不透明了，因此并不好。另一种解决办法是当 I/O 处理机往主存某个区域写入新内容时，由专用硬件自动地将 Cache 内对应此区域的副本作废，从而保持了 Cache 的透明性。CPU、I/O 处理机共享同一 Cache 也是一种解决办法。

总之，结构设计必须解决好 Cache 存储器的透明性带来的问题。

2. Cache 的取算法

由于 Cache 的命中率对机器速度影响很大，采用什么样的取算法可以提高命中率是 Cache 存储器设计中的重要问题。

Cache 所用的取算法基本上是按需取进法，即在 Cache 块失效时才将要访问的字所在块取进。适当选择好 Cache 的容量、块的大小、组相联的组数和组内块数，是可以保证有较高的命中率的。如再采用在信息块要用之前就预取进 Cache 的预取算法，还可能进一步提高命中率。

为了便于硬件实现，通常在访问主存第 i 块(不论是否已取进 Cache)时，只预取顺序的第 $i+1$ 块。至于何时取进该块，可有恒预取和不命中时预取两种方法。恒预取是只要访问到主存第 i 块，不论 Cache 是否命中，恒预取第 $i+1$ 块。不命中时预取则是只当访问主存第 i 块在 Cache 不命中时，才预取主存中第 $i+1$ 块。Amdahl 470 V/8 采用的就是不命中时预取。

采用预取法并非一定能提高命中率，它还和块的小大及预取开销的大小有关。若块太小，预取的效果会不明显。从预取需要出发，希望块尽量大。但若块太大就会预取进不用的信息，因 Cache 容量有限，反而将正用或近期就要用的信息给挤出去，使命中率降低。模拟结果表明，块的大小不宜超过 256 个字节。要预取就要有访主存、将其取进 Cache 的访 Cache、被替换块写回主存等的预取开销，它们将增加主存和 Cache 的负担，干扰和延缓程序的执行。所以预取法的效果不能只从命中率的提高来衡量，还应考虑为此所花费的开销是否值得。

模拟的结果是恒预取可使不命中率降低 75%～80%，而不命中时预取的不命中率只降低 30%～40%。但前者在 Cache、主存间增加的传输量要比后者大得多。

3. Cache 存储器的性能分析

和虚拟存储器中类似，评价 Cache 存储器的性能主要是看命中率的高低，而命中率与块的大小、块的总数(即 Cache 的总容量)、采用组相联时组的大小(组内块数)、替换算法和地址流的簇聚性等有关。

不命中率与 Cache 的容量、组的大小和块的大小的关系如图 4 - 36 所示。块的大小、组的大小及 Cache 容量增大时都能提高命中率。Cache 的容量在不断增大，现已达几百 KB 到几 MB。但 Cache 的块不可能太大，否则调块时 CPU 空等的时间太长。块的大小一般取成是多体交叉主存的总的宽度，使调块可在一个主存周期内完成。这样，Cache 的块数极多，不会出现如虚拟存储器中主存命中率随页面大小增大先升高而后降低的现象，也就是说，随着块的增大，Cache 不命中率总是呈下降趋势。

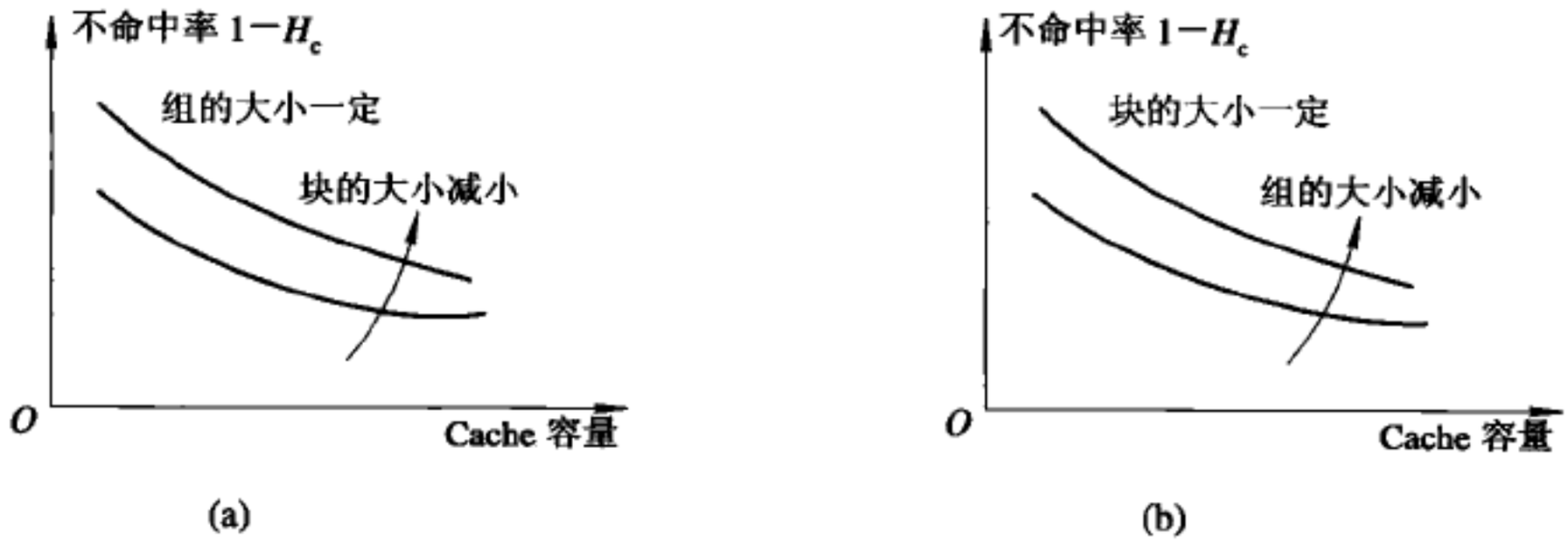


图 4 - 36　块的大小、组的大小与 Cache 容量对 Cache 命中率的影响

至于替换算法及程序的不同对命中率的影响与虚拟存储器的情况类似，绝大多数Cache存储器都采用LRU算法替换。

下面分析Cache存储器的等效访问速度与命中率的关系。

设t_c为Cache的访问时间，t_m为主存周期，H_c为访Cache的命中率，则Cache存储器的等效存储周期$t_a=H_ct_c+(1-H_c)t_m$。与虚拟存储器不同的是，一旦Cache不命中，主存与CPU经直接通路传送，所以CPU对第二级的访问时间是t_m，而不是调块时间再加一个访Cache的时间。这样，采用Cache存储器比之于处理机直接访问主存的等效访问速度提高的倍数为

$$\rho=\frac{t_m}{t_a}=\frac{t_m}{H_ct_c+(1-H_c)t_m}=\frac{1}{1-(1-t_c/t_m)H_c}$$

因为H_c总小于1，可令$H_c=\alpha/(\alpha+1)$，代入上式得

$$\rho=\frac{1}{1-\left(1-\frac{t_c}{t_m}\right)\frac{\alpha}{\alpha+1}}=(\alpha+1)\frac{t_m}{t_m+\alpha t_c}<\alpha+1$$

就是说，不管Cache本身的速度有多高，只要Cache的命中率有限，那么采用Cache存储器后，等效访问速度能提高的最大值是有限的，不会超过$\alpha+1$倍。

例如，$H_c=0.5$，相当于$\alpha=1$，则不论其Cache速度有多高，其ρ的最大值一定比2小；$H_c=0.75$，相当于$\alpha=3$，则ρ的最大值一定比4小；$H_c=1$，$\rho=\rho_{max}=t_m/t_c$，这是ρ可能的最大值。由此可得出ρ的期望值与命中率H_c的关系如图4-37所示。

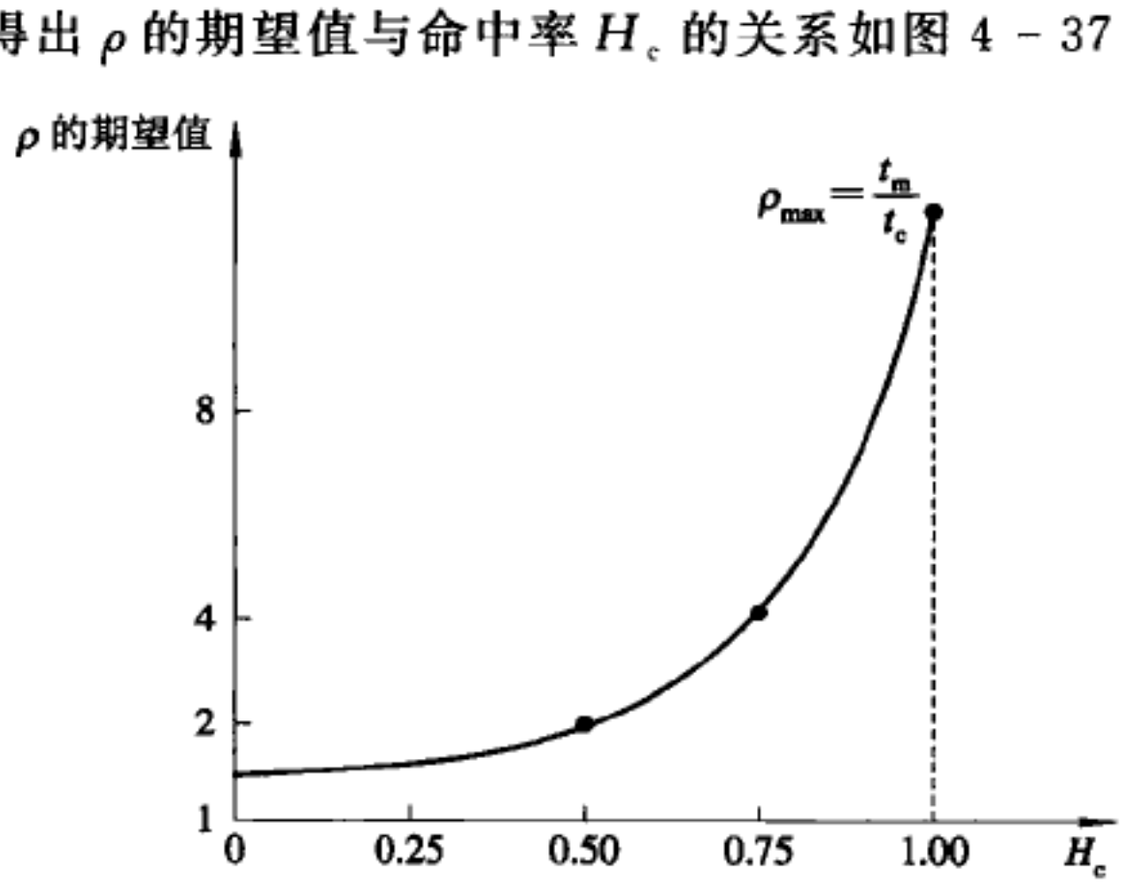


图4-37　ρ的期望值与H_c的关系

由于Cache的命中率一般比0.9大得多，可达0.996，因此采用Cache存储器能使ρ接近所期望的t_m/t_c。

结论：Cache本身的速度与容量都会影响Cache存储器的等效访问速度。如果对Cache存储器的等效访问速度不满意，需要改进的话，就要作具体分析，看现在Cache存储器的等效访问速度是否已接近于Cache本身的速度。如果差得较远，说明Cache的命中率低，这时就不应该用更高速的Cache片子来替换现有的Cache片子，而应该从提高Cache命中率着手，包括调整组的大小、块的大小、替换算法以及增大Cache容量等，否则速度是无法提高的。相反，如果Cache存储器的等效访问速度已经非常接近于Cache本身的速度却还不能满足速度要求，就只有更换成更高速的Cache片子。否则，任何其他途径也是不会

有什么效果的。因此，不能盲目设计和改进，否则花了很大代价，反而降低了 Cache 存储器的性能价格比。

4.4 三级存储体系

目前，多数的计算机系统既有虚拟存储器又有 Cache 存储器。程序用虚地址访存，要求速度接近于 Cache，容量是辅存的。这种三级存储体系，可以有 3 种形式。

4.4.1 物理地址 Cache

它是由“Cache—主存”和“主存—辅存”两个独立的存储层次组成的。图 4 - 38 就是这种形式。

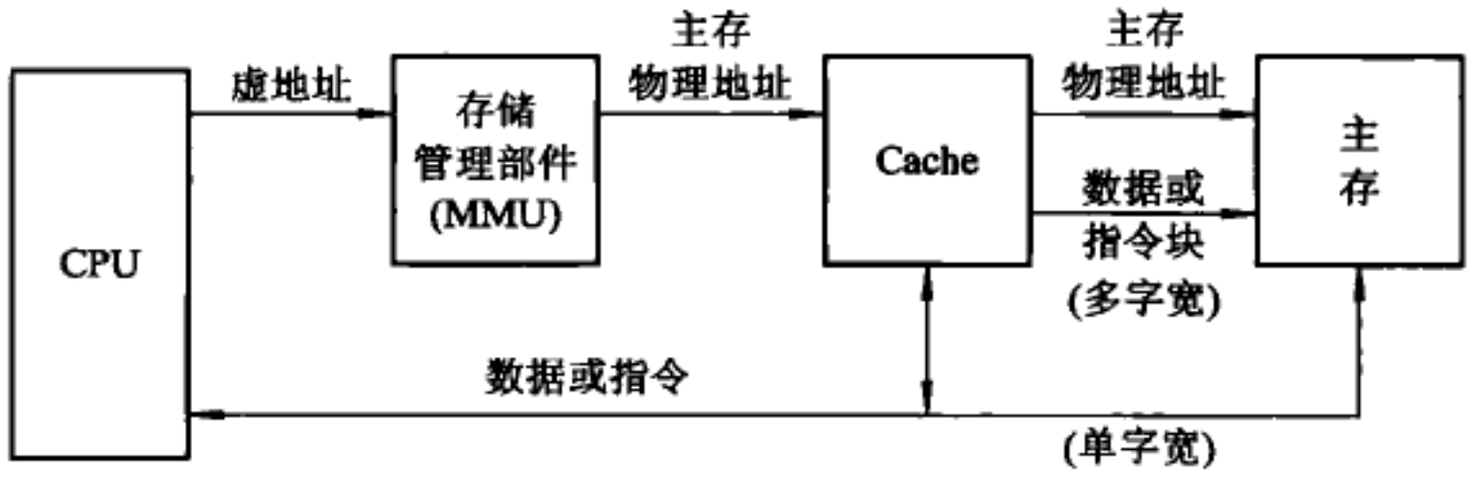


图 4 - 38 物理地址 Cache 的组成

CPU 用程序虚地址访存，经存储管理部件(Memory Management Unit，MMU)中的地址变换部件变换成主存物理地址访 Cache。如果命中 Cache，就访 Cache，如不命中 Cache，就将该主存物理地址的字和该字的主存一个块与 Cache 某相应块交换，而所访问的字直接与 CPU 交换。Intel 公司的 i486 和 DEC 公司的 VAX 8600 等机器都采用这种方式。

这种方式需要将主存物理地址变换成 Cache 地址才能访 Cache，这将增大访 Cache 的时间，至少要增加一个查主存快表的时间。为弥补这个不足，许多系统就改为直接用虚地址访 Cache，这就是虚地址 Cache 形式。

4.4.2 虚地址 Cache

虚地址 Cache 将 Cache—主存—辅存直接构成三级存储层次，其组成形式如图 4 - 39 所示。

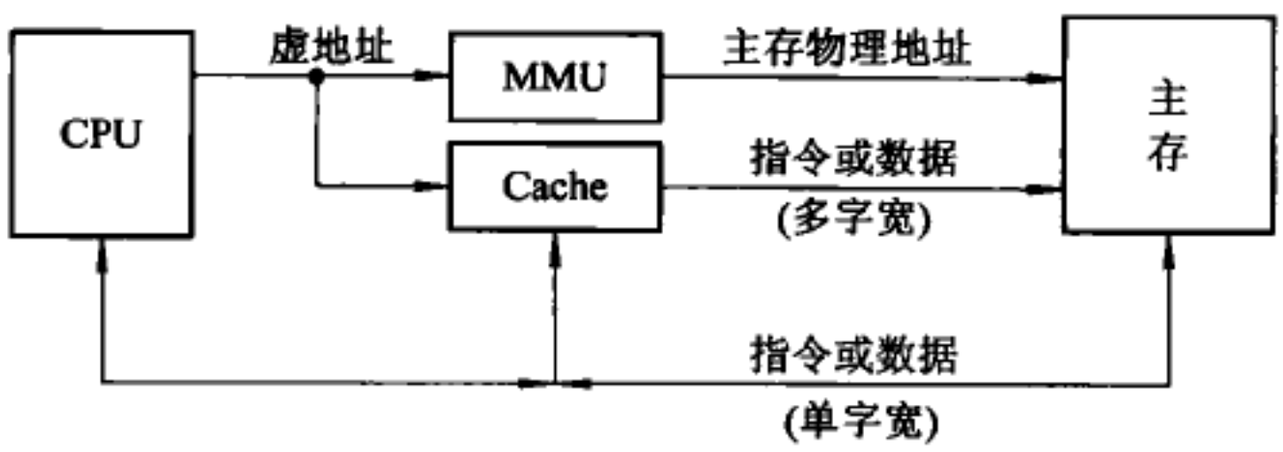


图 4 - 39 虚地址 Cache 的组成

CPU 访存时，直接将虚地址送存储管理部件 MMU 和 Cache。如果 Cache 命中，数据、指令就直接与 CPU 交换。如果 Cache 不命中，就由存储管理部件将虚地址变换成主存物理地址访主存，将含该地址的数据块或指令块与 Cache 交换的同时，将单个指令和数据与 CPU 交换。Intel 公司的 i860 就采用这种形式。

用虚地址直接访 Cache 的方法其地址变换过程如图 4 - 40 所示。

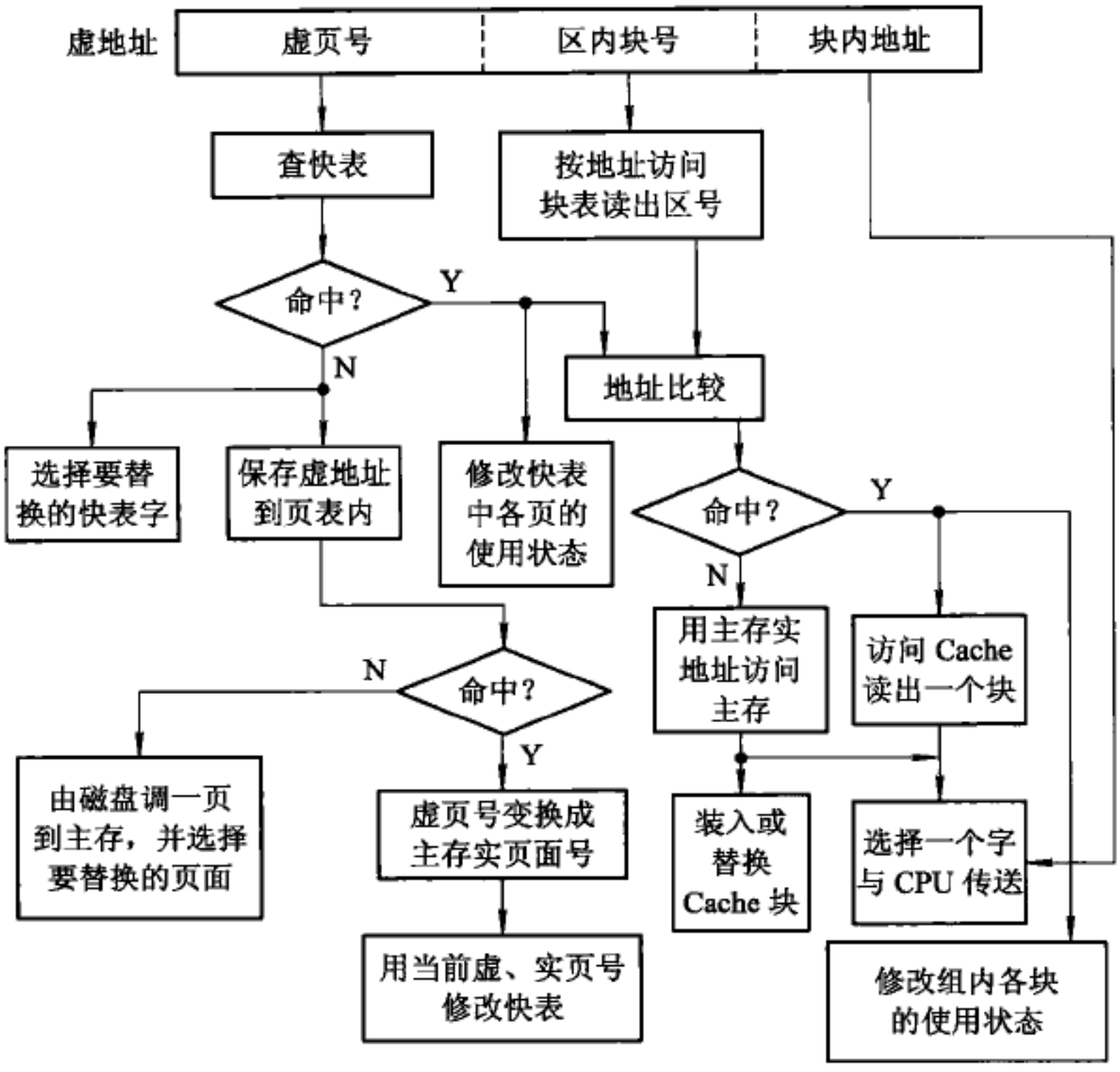


图 4 - 40　一种虚地址 Cache 的地址变换过程

在图 4 - 40 中，虚存采用位选择组相联映像和地址变换方式。为加快地址变换，可让虚存的一个页恰好是主存的一个区，直接用虚地址的区内块号按地址访问 Cache 的块表，从块表中读出主存的区号和对应的 Cache 块块号。这里，主存的区号就是虚页号。在访问 Cache 块表同时，用虚地址的虚页号访问快表。

如在快表中命中，就将从块表中读出的主存区号与从快表中得到的主存实页号进行全等比较。若比较相等，则 Cache 命中，此时，把虚地址中的区内块号直接作为 Cache 地址中的组号，从快表的相应单元中读出 Cache 的组内块号，把虚地址中的块内地址直接作为 Cache 地址中的块内地址。将上述得到的组号，组内块号、块内地址拼接成 Cache 地址访问 Cache 中的字送往 CPU。若 Cache 不命中，则直接用虚地址作为主存实地址访主存，将访主存的字送往 CPU。同时将含此字的一个块从主存中读出装入到 Cache 中。如果 Cache 已满，还需用某种 Cache 块替换算法，先把不用的一块替换到主存中去。

如果在快表中未命中，则要通过软件去查找存放在主存中的慢表，其后的工作过程与页式虚存或段页式虚存类似。