

第四章 存储体系

Dr. Feng Li

fli@sdu.edu.cn

<https://funglee.github.io>

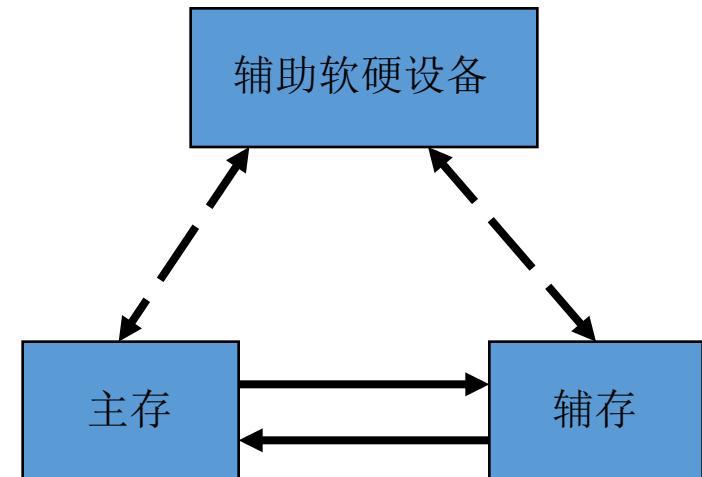
基本概念

- 存储体系是在构成存储系统的几种不同存储器 ($M_1 \sim M_n$) 之间，配上辅助软件或辅助硬件，使之从应用程序员的角度看，在逻辑上是一个整体。
- 等效访问速度接近 M_1 ，容量接近于 M_n ，每位价格接近于 M_n
- 基本的二级存储体系：虚拟存储器和 Cache 存储器

虚拟存储器

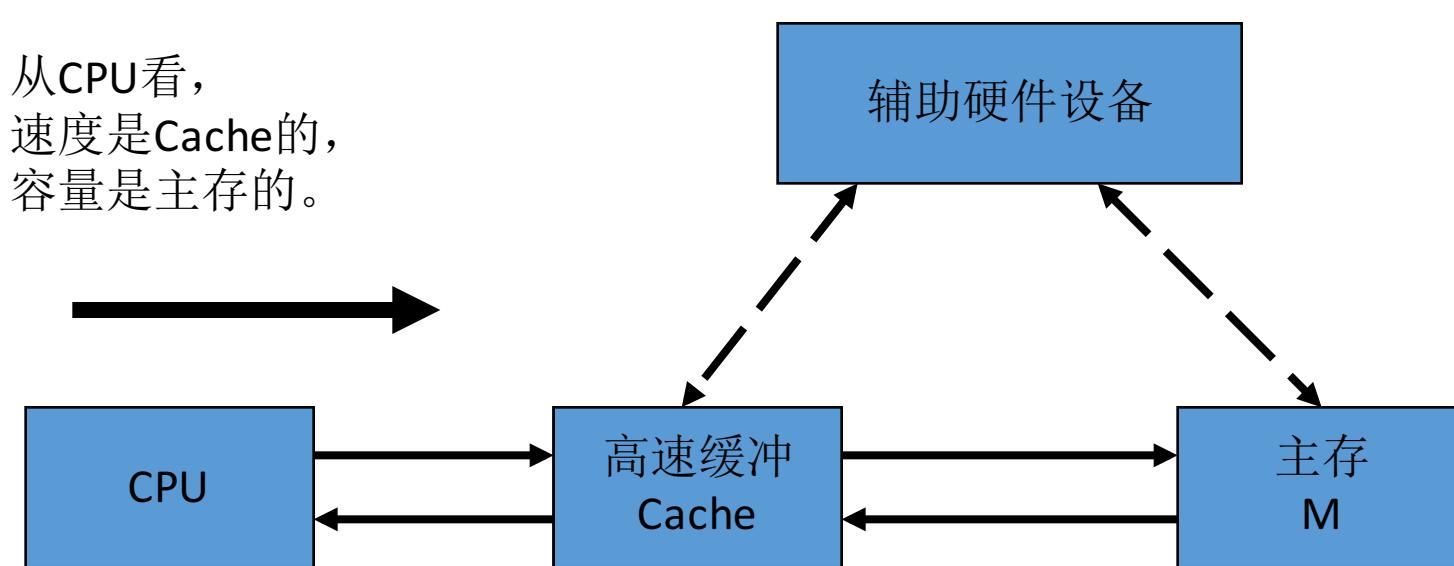
- 由于主存容量无法满足需求，因此在主存和辅存之间，增设辅助的软、硬件设备，是它们形成统一的整体，因此也称为“主存—辅存存储层次”
- 应用程序员可用机器指令的地址对整个程序统一编址，称该地址为虚地址（程序地址），而把实际主存地址称为实地址（实存地址）
- 虚存空间远大于实地址空间，需要将程序空间分割成较小的段（或者页），由系统程序按需调入物理主存，并用辅助映像表建立其虚、实地址空间的对应关系
- 虚地址访存时，由系统硬件查看虚地址所对应内容是否装入主存。若已经装入，则将虚地址变换成实存地址；若没有装入，则经辅助软、硬件将包含所访问单元的段（或页）由辅存调入主存，建立相应的映像关系

从整体上看，
速度是主存的，
容量是辅存的。



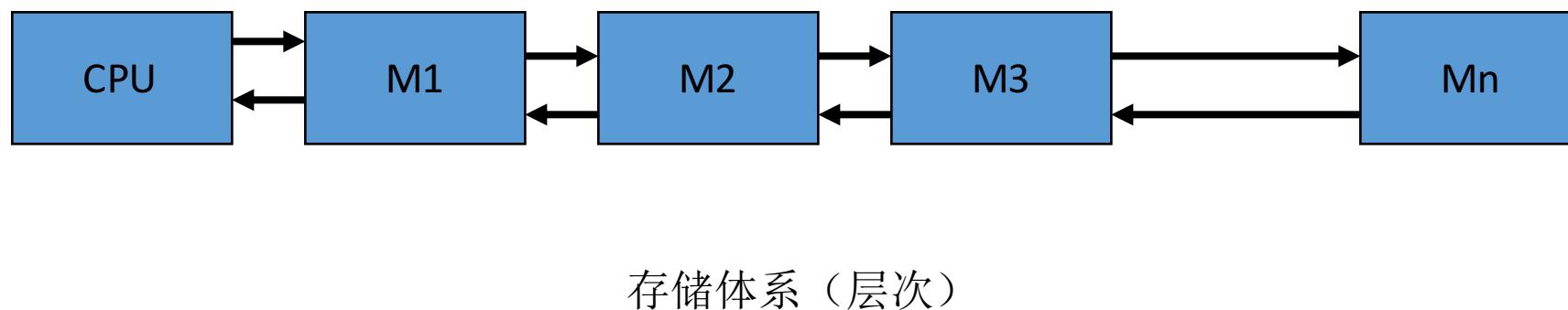
Cache存储器

- 由于主存速度满足不了要求，因而在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，称为“Cache-主存存储层次”



多级存储层次

- 从CPU的角度看，整个存储体系是一个整体，其速度接近于M1，容量接近于Mn，每位价格接近于Mn



存储体系构成的依据

- 当CPU需要某个地址的内容的时候，总是希望该内容在速度最快的 M_1 中
- 未来被访问信息的可预知性
 - 时间局部性：在最近的未来所要用到的信息很可能是现在正在使用的信息
 - 空间局部性：在最近的未来所要用到的信息很可能和现在正在使用的信息在程序空间上是邻近的
- 不必将整个程序整体存入 M_1 ，只需要将近期访问过的块（或者页）调入 M_1
- 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的算法和地址映像变换的方式

存储体系的性能参数

- c_i 为 M_i 的每位价格
- S_{M_i} 为 M_i 以位计算的存储容量
- T_{A_i} 为 CPU 访问到 M_i 中的信息所需要的时间
- 评价标准
 - 存储层次每位价格 c
 - 命中率 H : 即 CPU 产生的逻辑地址能在 M_1 中访问到的概率
 - 等效访问时间 T_A

以二级存储层次为例

- 每位价格

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

- 为使 c 接近 c_2 , 应使 $S_{M_1} \ll S_{M_2}$
- 还应限制辅助软、硬件的价格

- 命中率可以通过实验或者模拟求得
- 假设逻辑地址流在 M_1 中能访问到的次数是 R_1 ，当时在 M_2 中还未调到 M_1 中因此访问不到的次数是 R_2

$$H = \frac{R_1}{R_1 + R_2}$$

- H 与程序的地址流、所采用的地址预判算法和 M_1 的容量都有很大关系
- H 越接近 1 越好

- 存储层次的等效访问时间

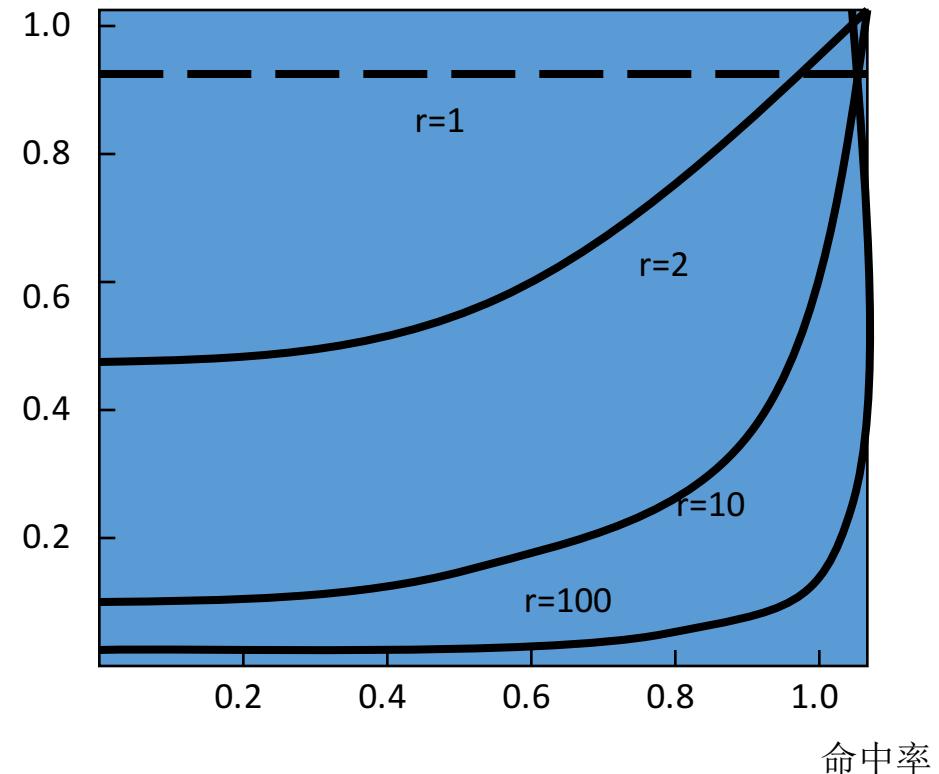
$$T_A = HT_{A_1} + (1 - H)T_{A_2}$$

- 访问效率

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

越接近1越好。其中， $r = \frac{T_{A_2}}{T_{A_1}}$ 为存储层次相邻二级的访问时间比

访问效率



- 要使访问效率 e 接近于1，当 r 越大时，要求命中率 H 越高
- 但是 H 在实际中很难提高。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，还可以减小相邻二级存储器的容量比
- 但上述策略与降低每位价格相矛盾，因此要在主存和辅存之间增加一级磁盘，使级间的 r 不会过大，有利于降低对 H 的要求，以获得同样的 e
- 总的来说，要使 e 趋近于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，以及增加的辅助软、硬件的代价等因素综合权衡，进行优化设计

存储器层次结构设计的问题

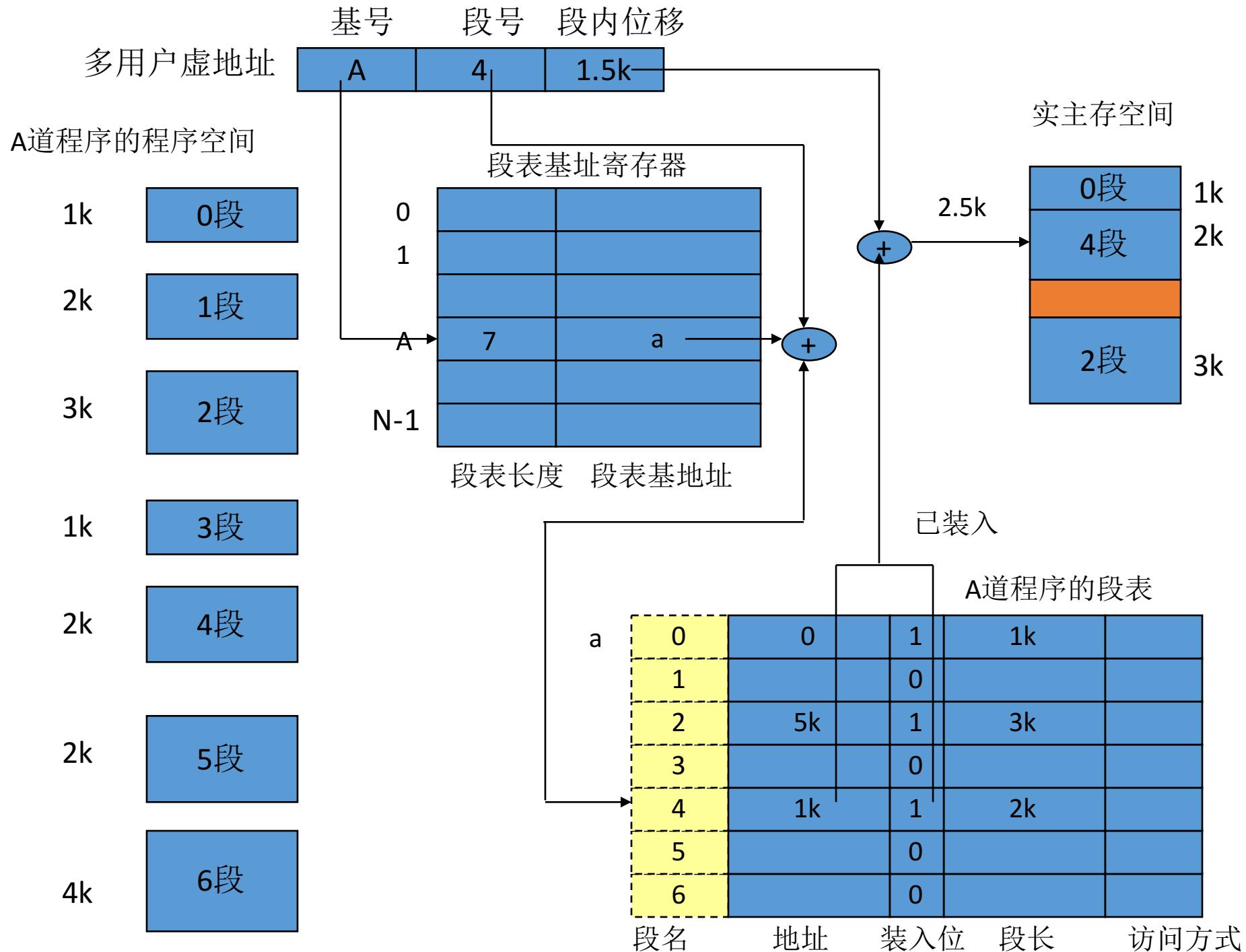
- 映像方式：低层存储器的块按什么规则装入高层存储器。
- 映像机构：映像方式的实现。如何识别和查找高层存储器的信息块。
- 替换策略：访问失效后，如何淘汰信息块，而换新块。
- 写策略：写操作时采用何种策略以保持相邻两级存储器中数据的一致性，发生写操作失效时是否将被写的块从低层存储器取入高层存储器。

虚拟存储器

- 虚拟存储器通过地址映像表机构来实现程序在主存中的定位
- 将程序分割成段或者页，用相应的映像表指明该程序某个段或者页是否装入主存
- 若已装入，指明在主存中的起始地址
- 若未装入，去辅存中调入段或者页，装入主存后再在映像表中建立好程序空间与实存空间的地址映像关系
- 程序执行时，先查映像表，将程序虚地址变成实地址后再访问主存

段式管理

- 段是程序中逻辑独立的模块
- 模块的大小个不相同，有的甚至事先无法确定
- 每个段从0开始编址
- 当某个段调入主存时，只要系统赋予一个基址，就可以有该基址和单元的段内地址形成主存内的实际地址
- 将主存按段分配的存储管理方式称为段式管理



分段方法的优势

- 使大程序分模块编制，使得多个程序员并行编程，缩短编程时间，在执行或者编译过程中对不断变化的可变长段也便于处理
- 便于多道程序共用已在主存内的程序和数据
- 由于各段是按其逻辑性组合，便于以段为单位实现存储保护

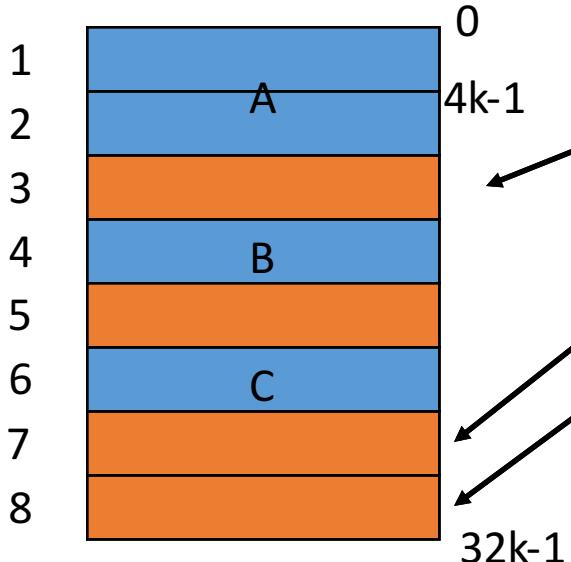
段式方法的劣势

- 由于段的长度完全取决于自身，在主存中的起点也会是随意的，这给高效地为调入段分配主存带来了困难
- 需要构造复杂的段映像表，还需要为主存系统建立一个实主存管理表，包括占用区域表和可用区域表

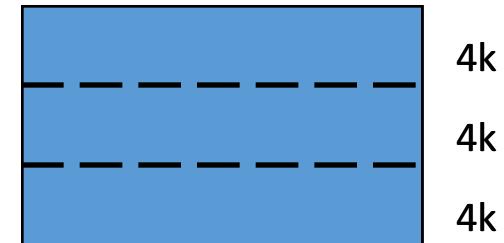
页式管理

- 页式虚拟存储器把虚拟地址空间机械地划分成一个个固定大小的块,每块称为一页,把主存储器的地址空间也按虚拟地址空间同样的大小划分为页。页是一种逻辑上的划分,它可以由系统软件任意指定。
- 虚拟地址空间中的页称为虚页, 主存地址空间中的页称为实页。
- 每个用户使用一个基址寄存器（在CPU内）, 通过用户号 U 可以直接找到与这个用户程序相对应的基址寄存器, 从这个基址寄存器中读出页表起始地址。访问这个页表地址, 把得到的主存页号 p 与虚地址中的页内偏移直接拼接起来得到主存实地址。

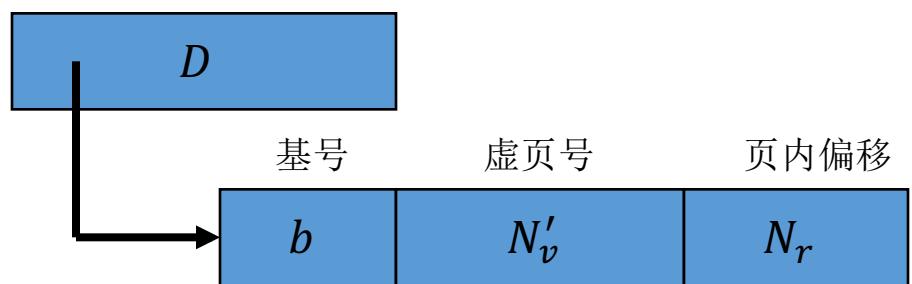
主存页号 主存空间



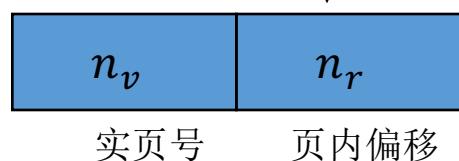
虚存页号
D道程序
程序空间



程序标志号



主存单元地址 n_p



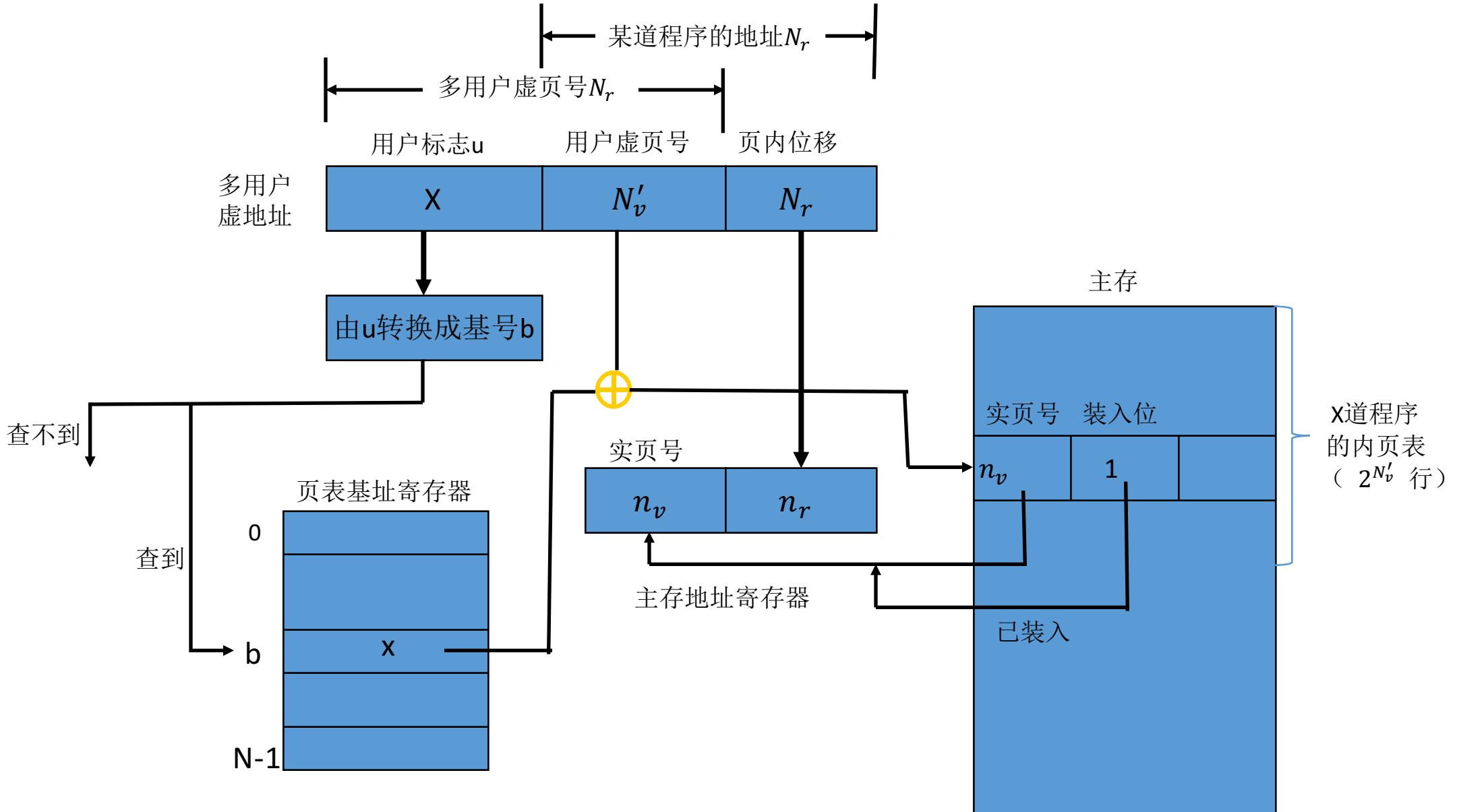
虚存页号	主存起点	装如位	访问方式	专用位
0	2	1		
1	6	1		
2	7	1		

可省略 页内位移

D道程序页表 ($2^{N'_v}$ 行)

页式管理

- 页表项简单，查找速度快；
- 页面大小固定不利于系统的效率，
 - 有些系统可调整其大小。例：MC88200 应用程序 4kb 系统程序 512kb
- 页式管理在存储空间较大时，由于页表过大，效率降低。
- 存储空间的保护困难。



页式管理的优缺点

- 优点

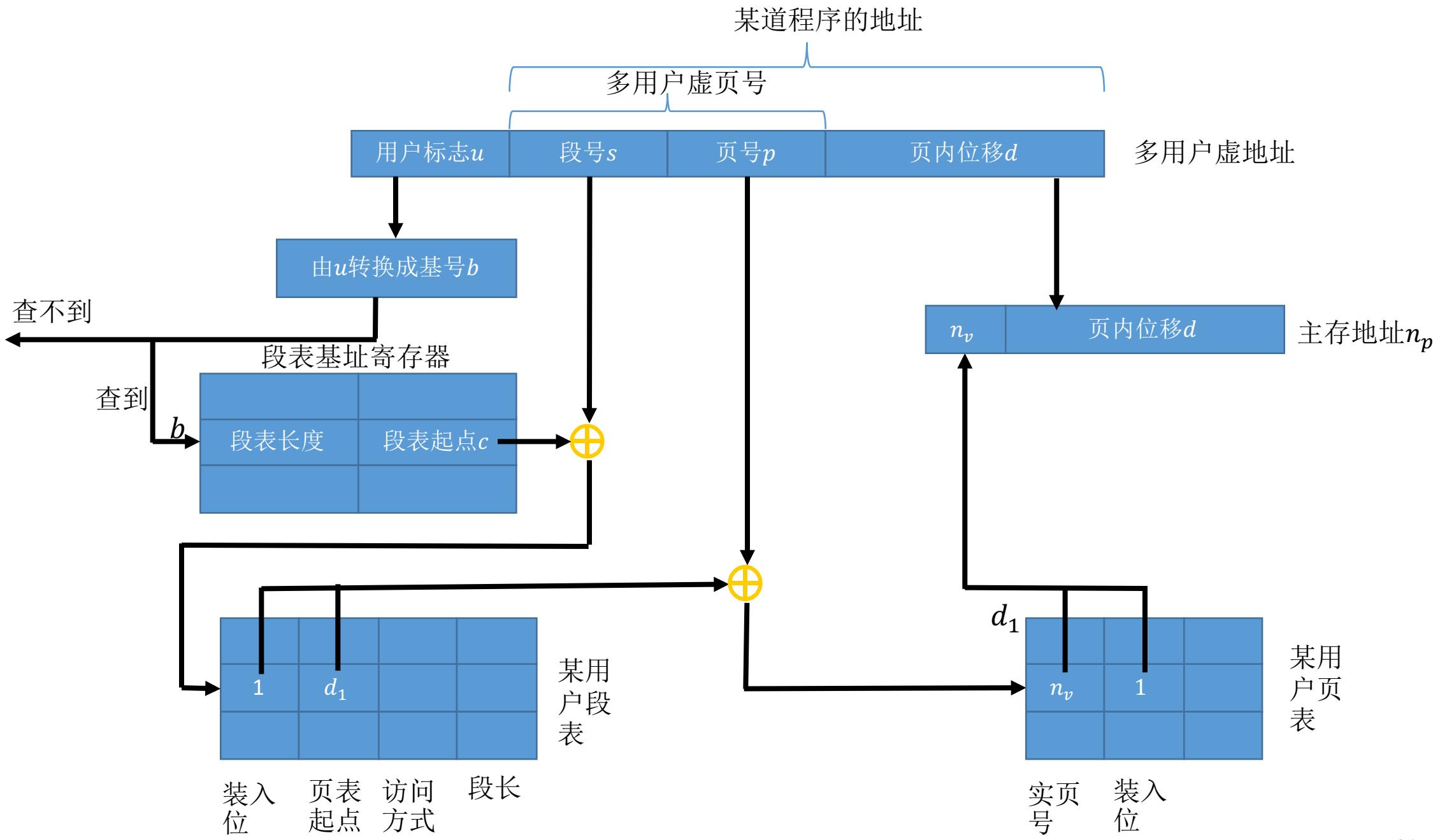
- 主存储器的利用率比较高
- 页表相对比较简单
- 地址变换的速度比较快
- 对磁盘的管理比较容易

- 缺点

- 程序的模块化性能不好
- 页表很长，需要占用很大的存储空间。

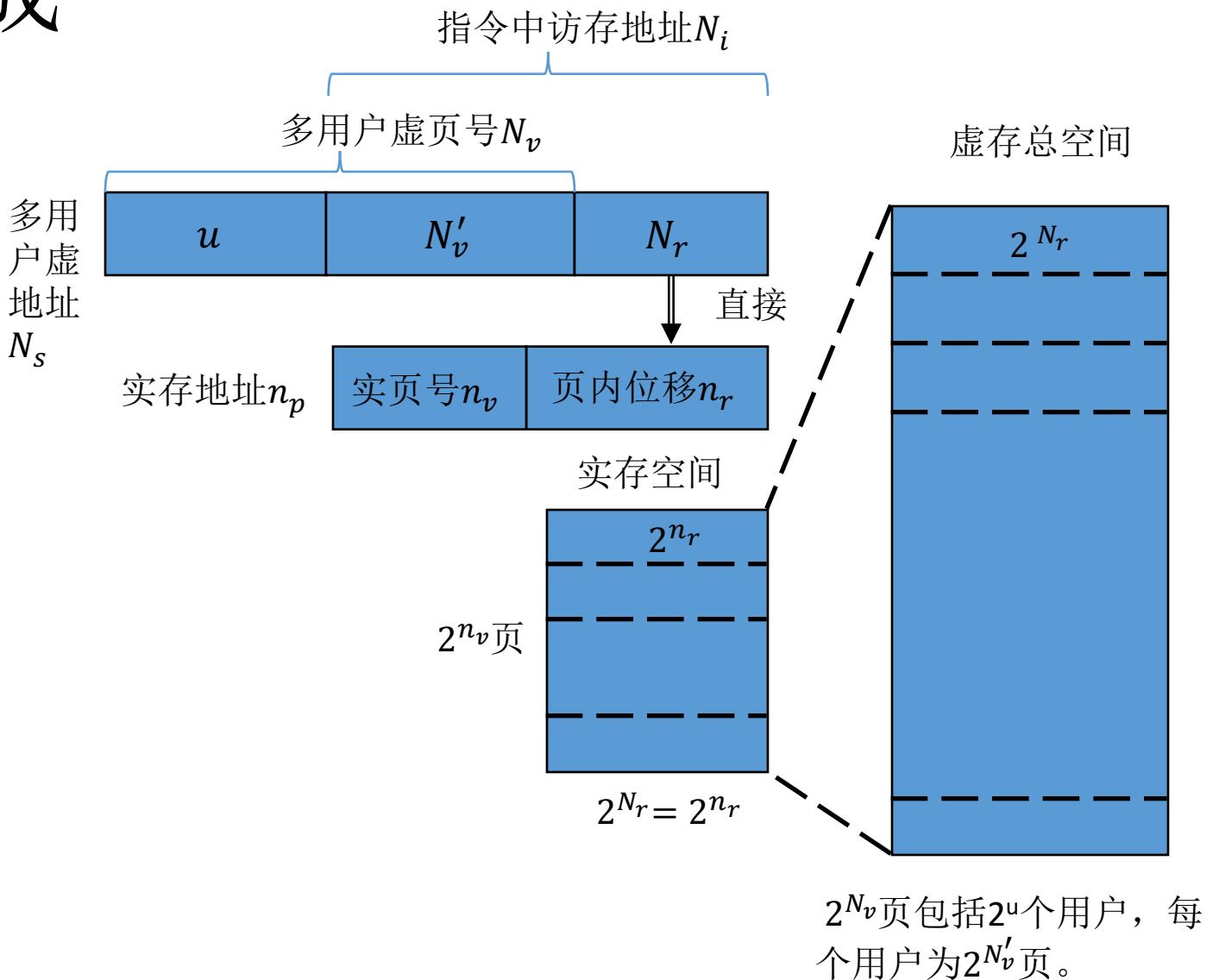
段页式管理

- 页式：对应用程序员完全透明，由系统划分。
 - 硬件较少，地址变换速度快，
 - 调入操作简单，静态连接程序；
- 段式：段独立，有利于程序员灵活实现段的连接、段的扩大/缩小和修改，而不影响其他段，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，从而易于实现多个用户、进程对共用段的管理，动态连接程序；
- 段页式：把实存机械地等分成固定大小的页，程序按模块分段，每个段又分成与主存页面大小相同的页。



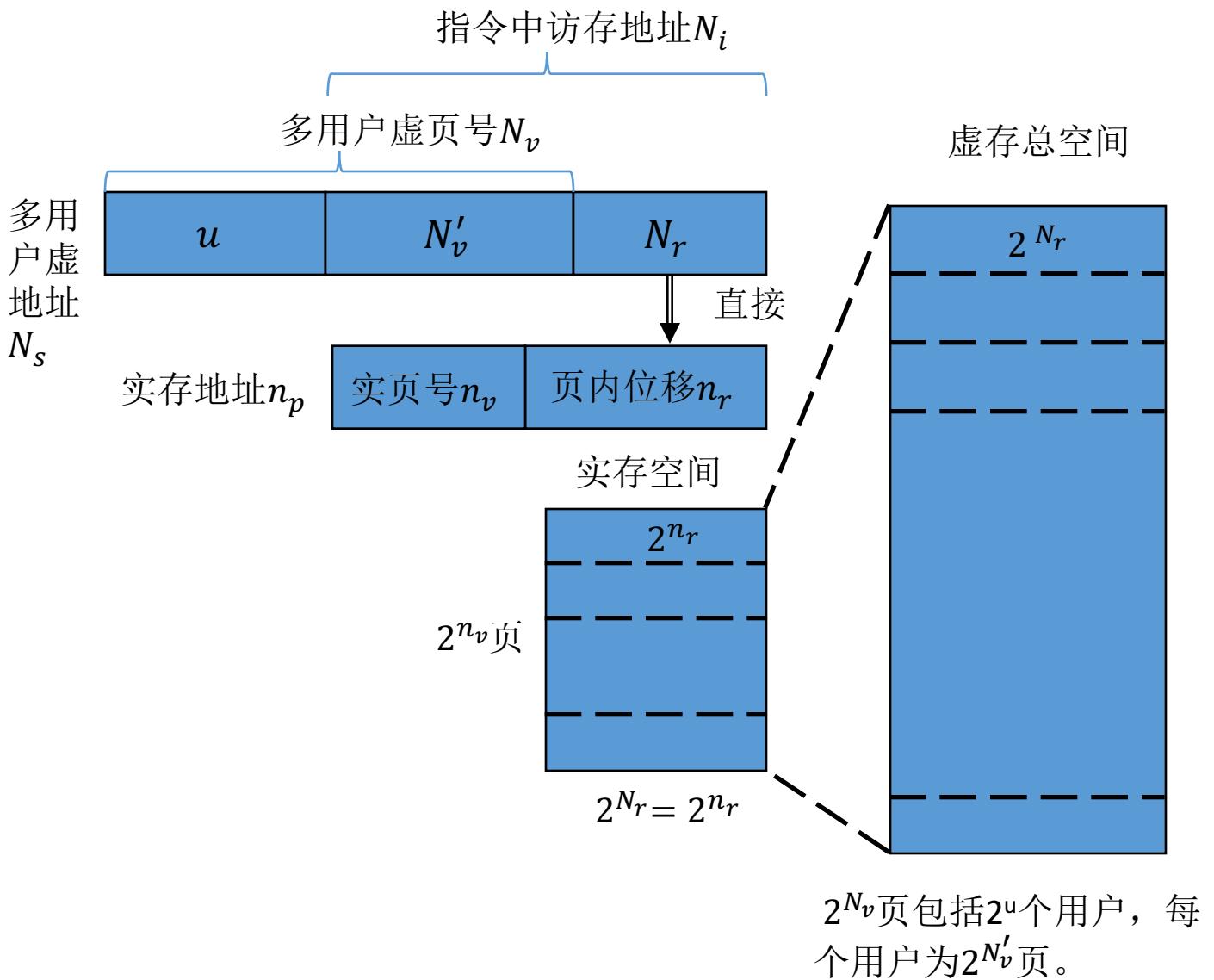
页式虚拟存储器构成

- 虚地址 $N_s = u + N'_v + N_r$
- 实地址 $n_p = n_v + n_r$
- 虚存空间 $2^u * 2^{N'_v}$
- 实存空间 2^{n_v}
- 其中 $N_v = u + N'_v$



地址映象和变换

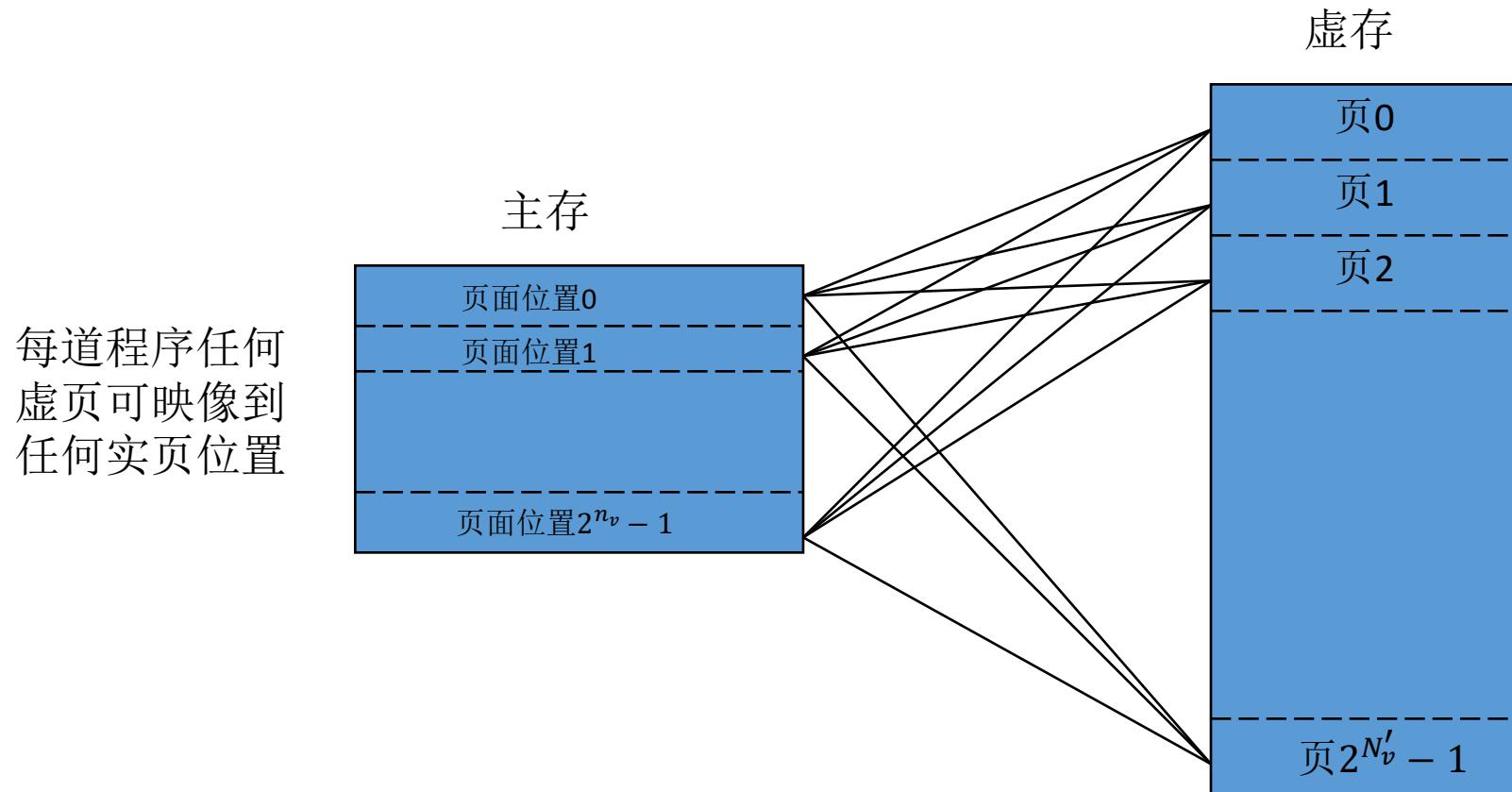
- **地址映象**: 是将每个虚存单元按某种规则(算法)装入(定位于)实存, 即建立多用户虚地址 N_s 与实存地址 n_p 之间的对应关系。对于页式管理而言, 就是指多用户虚页号为 N_v 的页可以装入主存的那个页面位置, 从而建立起 N_v 和 n_v 的对应关系
- **地址变换**: 是程序按照这种映象关系装入实存后, 在执行时, 多用户虚地址 N_s 如何变换成对应的实地址 n_p 。对于页式管理而言, 就是如何将 N_v 变换为 n_v



- 由于主存空间比虚存空间要小得多，因此一个实存页面位置可对应多个虚页
- 页面冲突：两个以上虚页想进入主存同一页面位置而产生的页面争用（或实页冲突）
- 映像方式的选择应考虑尽可能降低实页冲突的概率，同时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
- 由于虚存空间远远大于实存空间，因此页式虚拟存储器常采用全相联映像，从而使一个虚页可以装入任何实页位置。

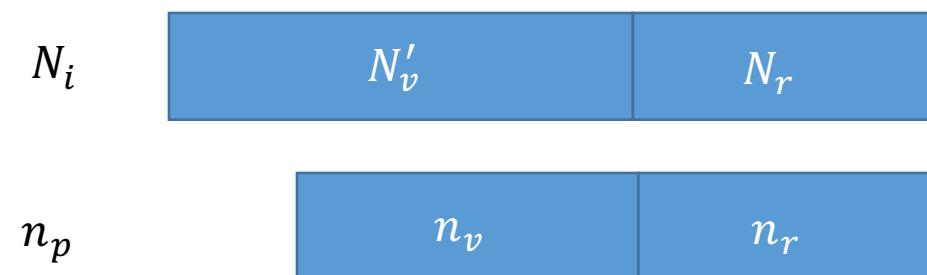
全相联映像（页表法）

任何虚页可以映象装入到任何实页位置。冲突概率最低。



全相联映像的页表法

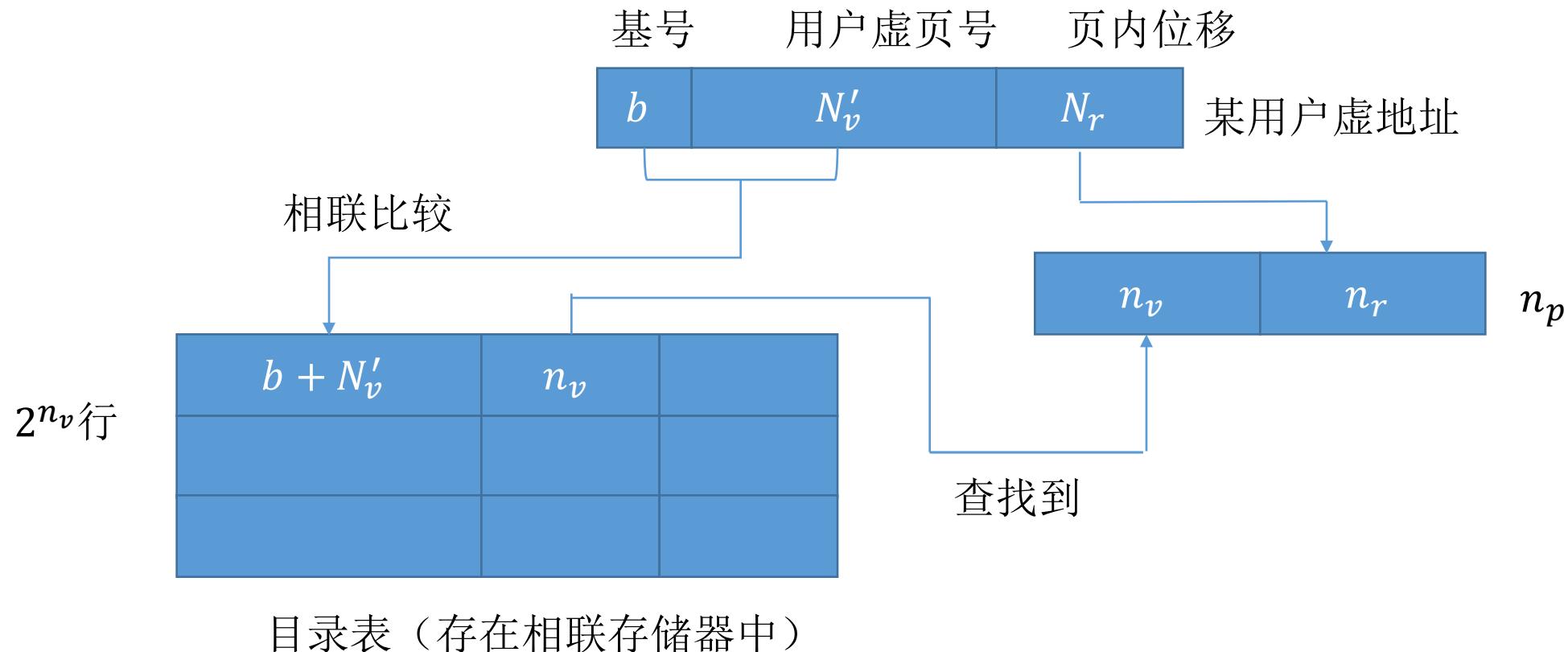
- 多用户虚存空间可以有 2^u 个用户，但实存空间只对 N 个用户（程序）开放
- 每道程序有 $2^{N'_v}$ 行的页表，而主存总共有 2^{n_v} 个实页位置
- 由于 $N \times 2^{N'_v} \gg 2^{n_v}$ ，页表中绝大多数行中的实页号 n_v 字段及其他字段都无用，大大降低了页表空间利用率



- 由于装入位为“1”的行最多为 2^{n_v} , 而 $N \times 2^{N'_v} \gg 2^{n_v}$, 使得页表中绝大部分行中的实页号 n_v 字段及其他字段都成为无用的了, 大大降低了页表的空间利用效率

全相联映像的相联目录表法

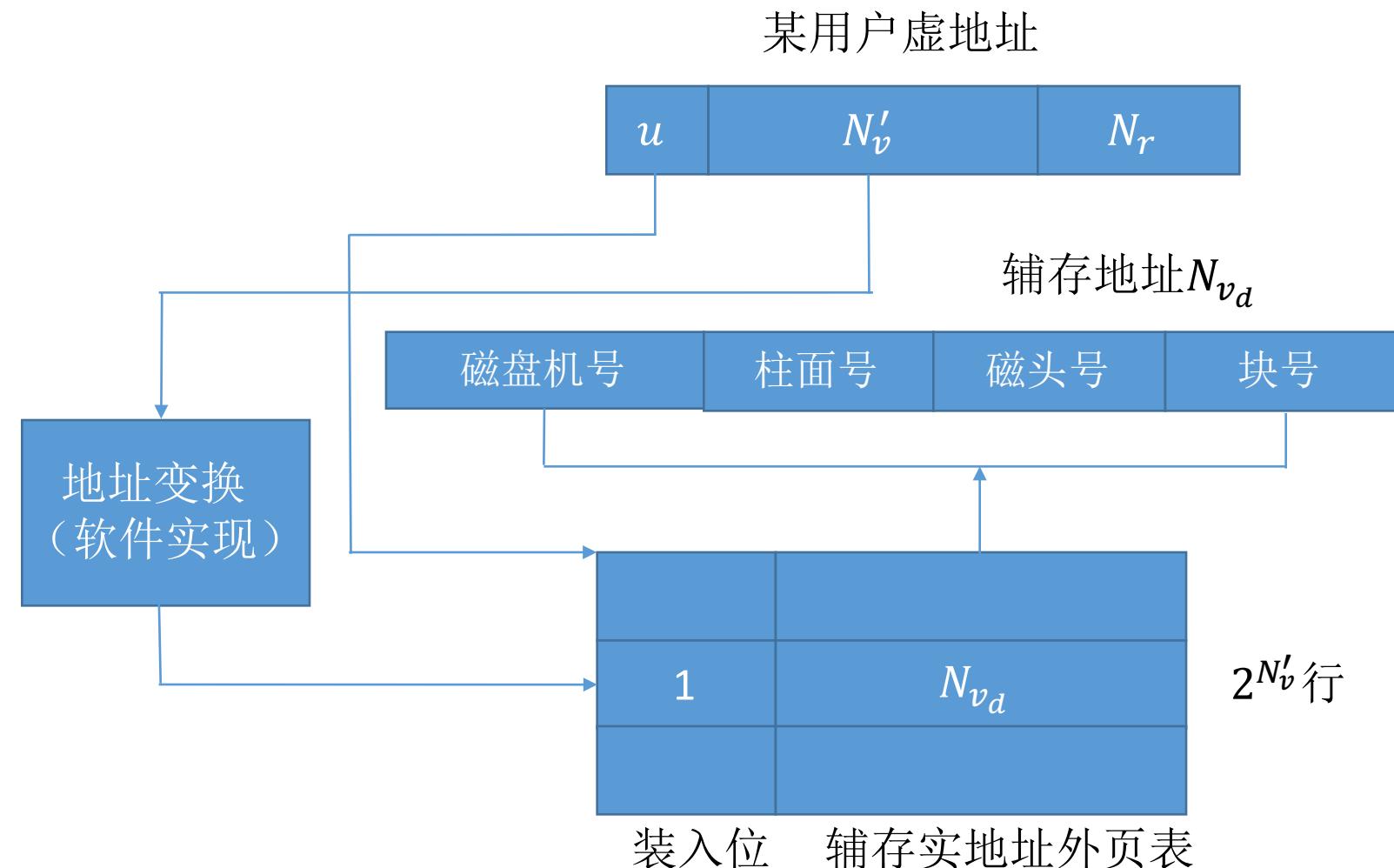
- 把页表压缩成只存放已装入主存的那些虚页号（用基号 b 和 N'_v 标记）与实页位置 n_v 的对应关系



- 相联目录表采用按内容访问的相联存储器构成
- 按地址访问的随机存储器是在一个存储周期内只能按给出的一个地址访问其存储单元，而相联存储器在一个存储周期内能将给定的 N_v 同时与目录表的全部 2^{n_v} 个单元对应的虚页号字段内容进行比较，即进行相联查找
- 若多用户虚地址 N_s 所在的虚页未装入主存，则需要进行程序换道或者调页

调页流程

- 为提高调页效率，辅存按信息块编址，块的大小通常等于页的大小
- 外页表：描述用户虚页号 N'_v 与辅存实地址 N_{v_d} 之间的映像关系



替换算法

- 页面替换发生时间：

当发生页面失效时，要从磁盘中调入一页到主存。如果主存所有页面都已经被占用，必须从主存储器中淘汰掉一个不常使用的页面，以便腾出主存空间来存放新调入的页面。

- 替换算法的确定

- 主存的命中率
- 是否便于实现，软、硬件成本

替换算法

- 随机算法（Random， RAND）：用软的或硬的随机数产生器来形成主存中要被替换页的页号。
 - 简单，易于实现
 - 没有利用历史信息
 - 命中率低，很少使用
- 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。
 - 操作系统为主存页面表中给每个实页配置一个计数器字段
 - 每当一页装入主存时，让该页的计数器清零，其它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
 - 虽然利用历史信息，但不一定反映出程序的局部性

替换算法

- 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。
 - 配备计数器
 - 比较正确反映程序的局部性。
- 优化替换算法（Optimal Replacement Algorithm, OPT）:是在时刻 t 找出主存中每个页将要用到时刻 t_i ，然后选择其中 $t_i - t$ 最大的那一页作为替换页。
 - 理想化算法

主存页面表

- 主存页面表存储于主存，整个系统只有一个
- 主存当中每一页

占用位	程序号	段页号	使用位	程序优先位	历史位 H_s	其他信息
-----	-----	-----	-----	-------	-----------	------

- 周期性查询所有使用位，凡使用位为“0”的将其历史位 H_s 加“1”，而使用位为“1”的将其 H_s 和使用位均清“0”
- H_s 值最大的页就是最久未使用页
- 由软件和硬件相结合实现其修改
- 可增设修改位，判断被替换时是否需要写回辅存

举例1

- 设有一道程序，有1至5共5页，执行时的地址流为：

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

分别采用FIFO、LRU、OPT算法。

时间 t	1	2	3	4	5	6	7	8	9	10	11	12	实际命中次数
页地址流	2	3	2	1	5	2	4	5	3	2	5	2	
先进先出算法 (FIFO 算法)	2	2	2	2*	5	5	5*	5*	3	3	3	3*	3 次
近期最少使用算法 (LRU 算法)													5 次
最优替换算法 (OPT 算法)													6 次

三种页面替换算法对同一个页地址流的调度过程

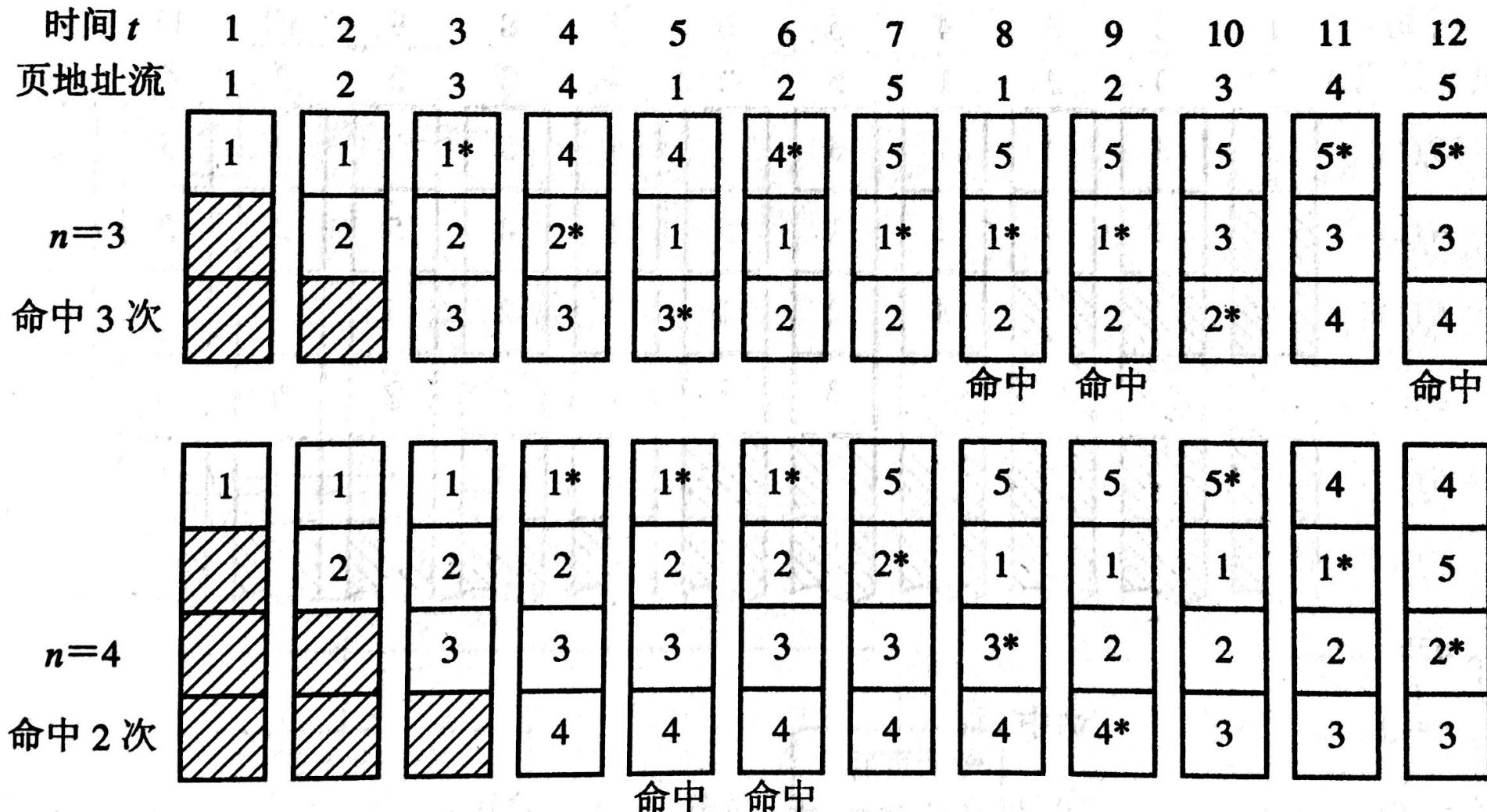
举例

- 一个循环程序，依次使用P1, P2, P3, P4四个页面，分配给这个程序的主存页面数为3个。FIFO、LRU和OPT三种页面替换算法对主存页面的调度情况如下图所示。在FIFO和LRU算法中，总是发生下次就要使用的页面本次被替换出去的情况，这就是“颠簸”现象。

时间 t	1	2	3	4	5	6	7	8	实际命中次数
页地址流	P1	P2	P3	P4	P1	P2	P3	P4	
先进先出算法 (FIFO 算法)	1	1	1*	4	4	4*	3	3	0 次
		2	2	2*	1	1	1*	4	
			3	3	3*	2	2	2*	
	调入	调入	调入	替换	替换	替换	替换	替换	
最久没有使用算法 (LRU 算法)	1	1	1*	4	4	4*	3	3	0 次
		2	2	2*	1	1	1*	4	
			3	3	3*	2	2	2*	
	调入	调入	调入	替换	替换	替换	替换	替换	
最优替换算法 (OPT 算法)	1	1	1	1	1*	1	1	1	3 次
		2	2	2	2	2*	3*	3	
			3*	4*	4	4	4	4*	
	调入	调入	调入	替换	命中	命中	替换	命中	

说明

- 命中率与替换算法有关
 - LRU算法要优于FIFO算法
- 命中率与地址流有关
 - 例如：一个循环程序， FIFO、LRU的命中率明显低于OPT。
- 命中率与分配给程序的主存页数有关。
 - 主存页数增加， LRU命中率提高，至少不会下降，而FIFO不一定。



FIFO算法的实页数增加，命中率反而有可能下降

堆栈型替换算法的定义

- A 是长度为 L 的任意一个页地址流
- t 为已经处理过 $t - 1$ 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 页主存中的页面的集合
- L_t 表示到 t 时间点已经遇到过的地址流中相异页的页数
- 如果替换算法满足

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

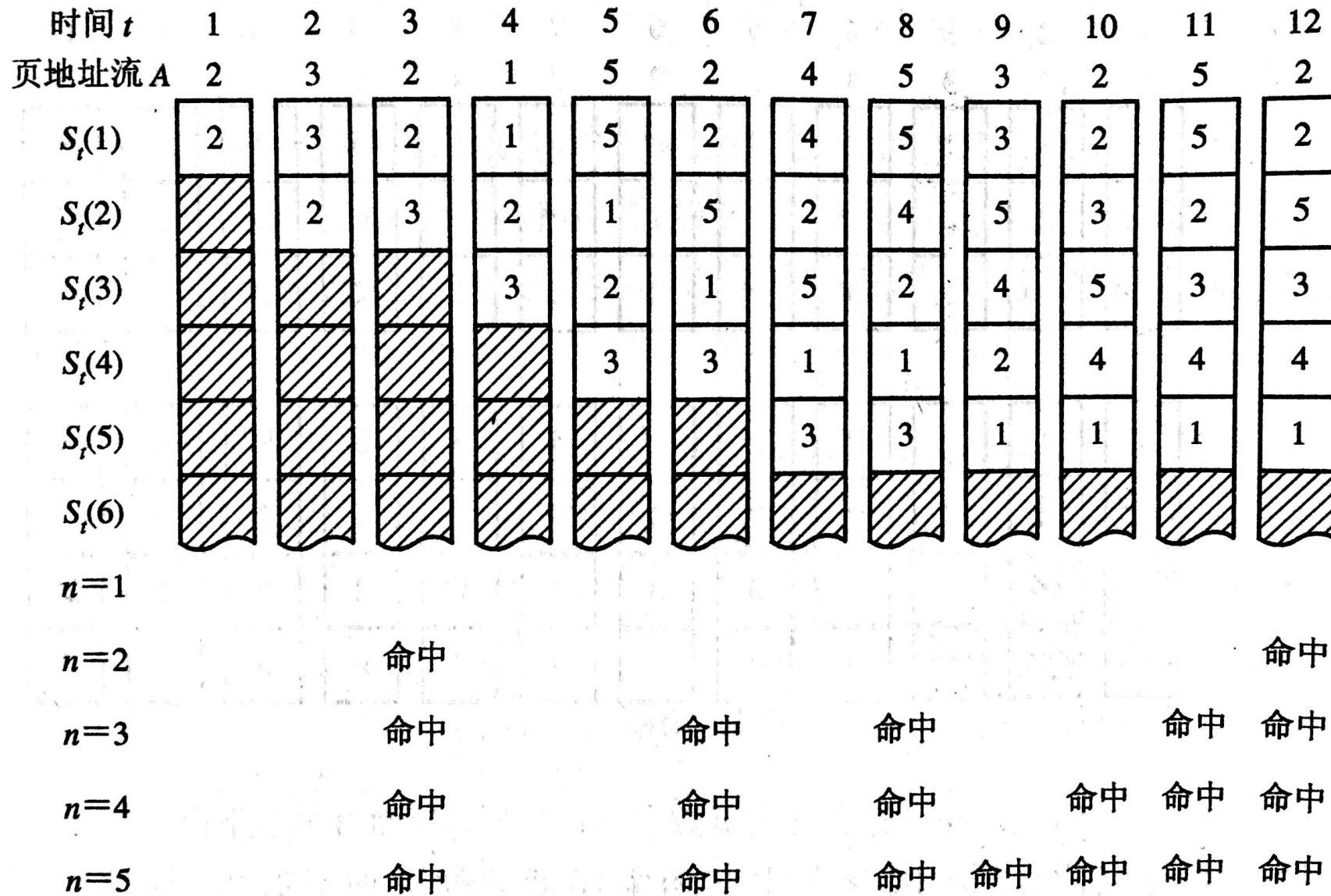
$$n \geq L_t \text{ 时, } B_t(n) = B_t(n + 1)$$

则属于堆栈型替换算法

堆栈型算法的基本特点

- 随着分配给程序的主存页面数增加， 主存的命中率也提高， 至少不下降。
 - LRU、 OPT都是堆栈型算法， 保存的是 n 个最近使用的页， 它们又总是包含在 $n+1$ 个最近使用的页
 - FIFO是非堆栈型算法
- 提出使系统性能可以更优的动态算法
 - 根据各个页面失效率达到某个值， 就自动增加分配给该道程序的主存页数

- 主存在 t 时间点的状况用 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， $S_t(2)$ 为次栈顶项，因此
- $n < L_t$ 时， $B_t(n) = \{S_t(1), S_t(2), \dots, S_t(n)\}$
- $n \geq L_t$ 时， $B_t(n) = \{S_t(1), S_t(2), \dots, S_t(L_t)\}$
- 容量为 n 页的主存中，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
- 经过一次模拟处理获得 $S_t(1), S_t(2), \dots, S_t(L_t)$ 之后，就能同时知道不同 n 值时的命中率
- 不同的堆栈型替换算法，其 S_t 各项的改变过程不同

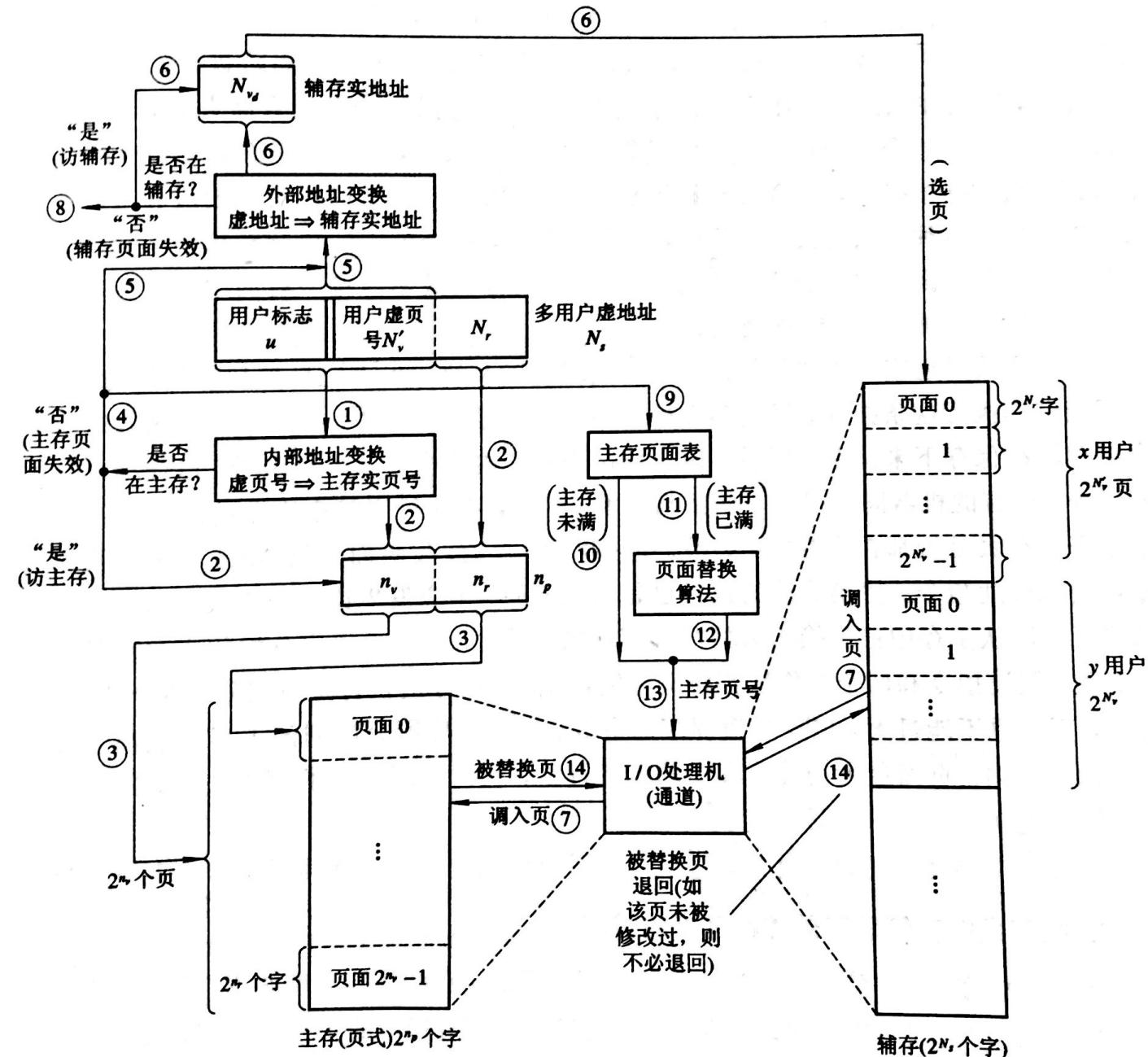


使用LRU算法进行堆栈处理的 S_t 变化过程

页面失效频率算法

- 对于堆栈型替换算法，对着分配给程序的实页数 n 的增加，命中率 H 会单调上升
- 根据各道程序运行过程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
- 当主存页面失效率低于某个阈值时候就自动减少分配给该道程序的主存页面数，以便释放出这部分主存页面位置供给其他程序使用，从而使整个系统总的主存命中率和主存利用率得到提高

虚拟存储器工作的全过程

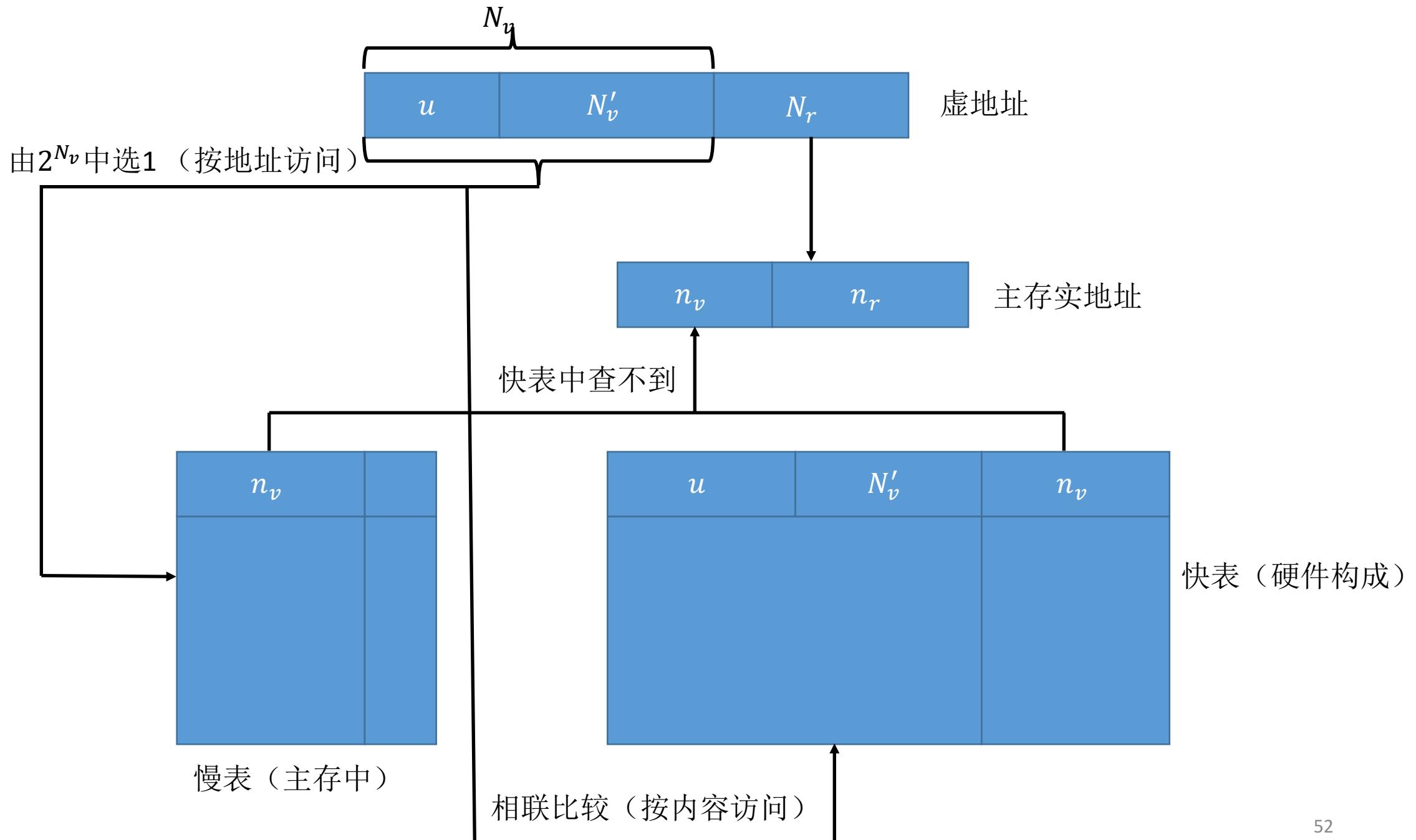


页式虚拟存储器实现中的问题

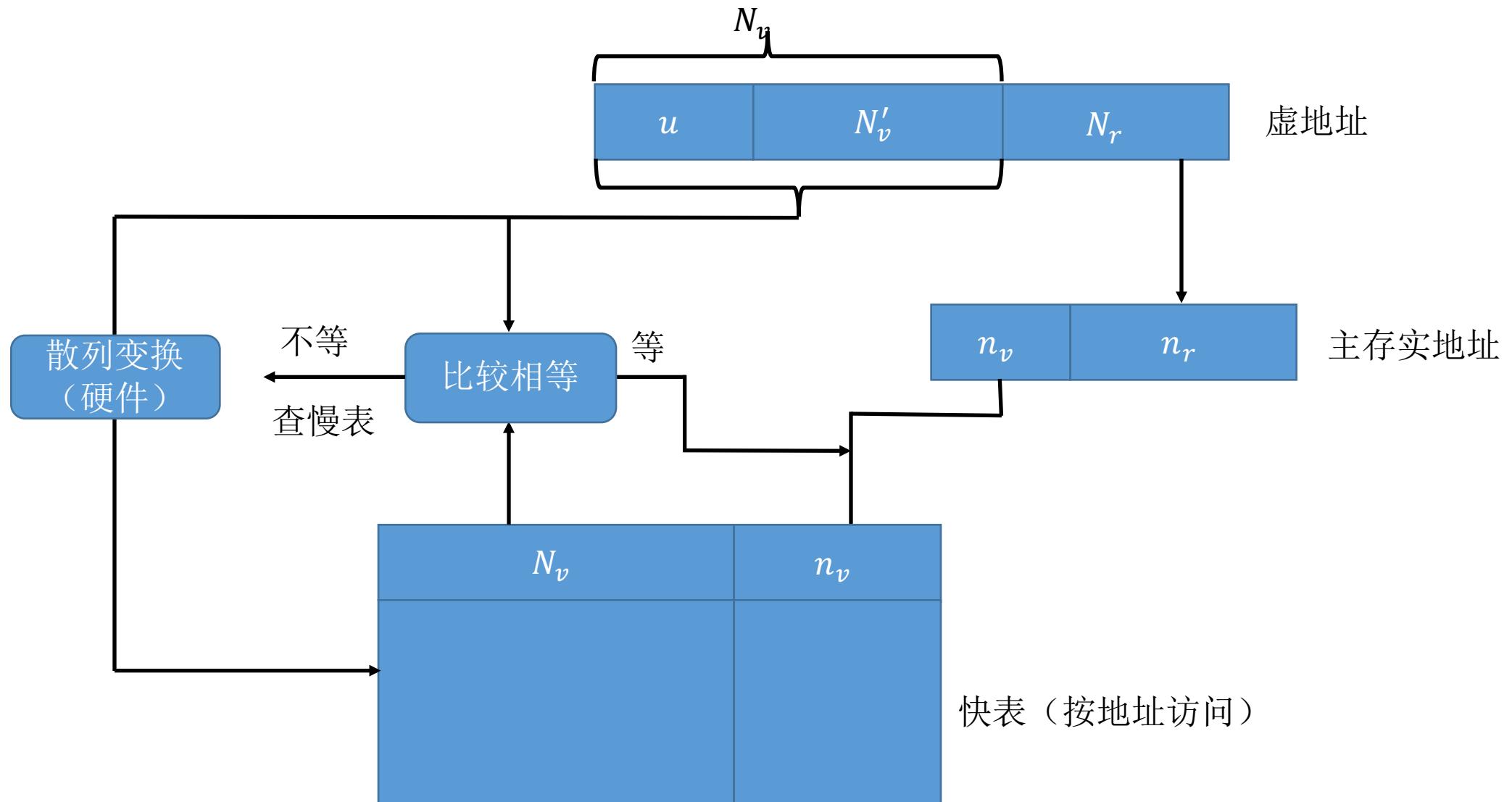
- 页面失效的处理
 - 由于指令或者操作数可能跨页存放，页面失效会在一条指令的分析或执行过程中发出。
 - 页面失效时一种故障，不是一般的中断；
 - 注意保护现场,采用后援寄存器技术、预判技术；
 - 选择合适的替换算法；

页式虚拟存储器实现中的问题

- 提高虚拟存储器等效访问速度的措施
 - 要求：提高命中率，加快访存时间；
 - 命中率受很多因素影响，如：地址流、页面调度策略、替换算法、页面大小、主存容量等。
 - 访存时间
 - 快表：硬件构成
 - 慢表：主存中
 - 快表-慢表存储层次的替换算法一般采用LRU法。

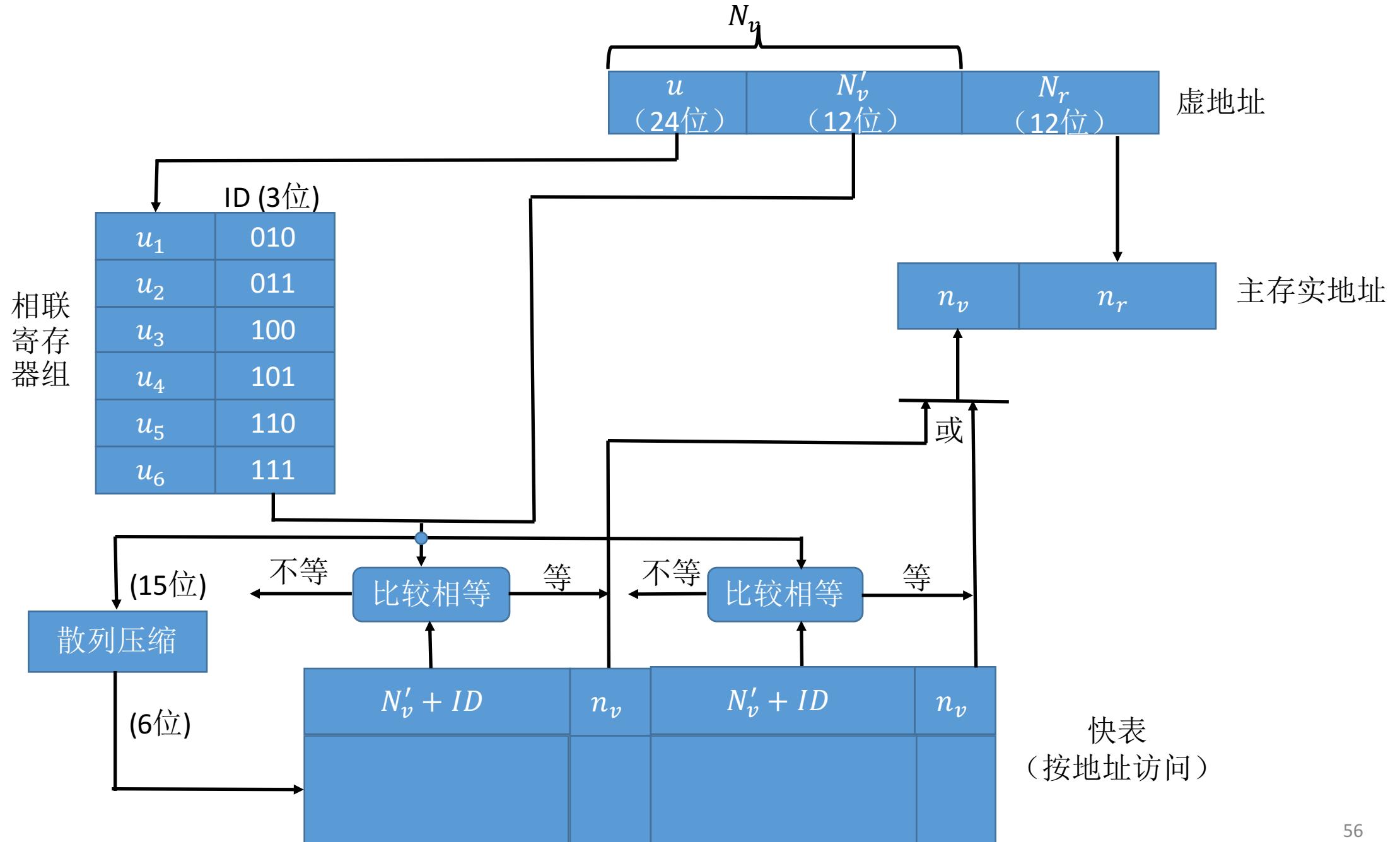


- 为提高快表的命中率和查表速度，可以用高速按地址访问存储器构成更大容量的快表
- 散列函数
 - 目的：把相联访问变成按地址访问，从而加大快表容量
 - 散列（Hashing）函数： $A = H(N_v)$
 - 采用散列变换实现快表按地址访问
 - 避免散列冲突：采用相等比较器
 - 地址变换过程：相等比较与访问存储器同时进行



虚拟存储器举例

- IBM370/168计算机的虚拟存储器快表结构及地址变换过程。虚拟地址长36位，页面大小为4KB，每个用户最多占用4K个页面，最多允许16个用户，但同时上机的用户数一般不超过6个。
- 采用了两项新的措施：
 - 采用两个相等比较器
 - 用相联寄存器组把24位用户号U压缩成3位



页式虚拟存储器实现中的问题

- 影响主存命中率CPU效率的某些因素
 - 程序地址流、替换算法、实页数、页面调度策略
 - 页面大小通常为1KB到4KB。

S_1 ：某道程序的主存容量

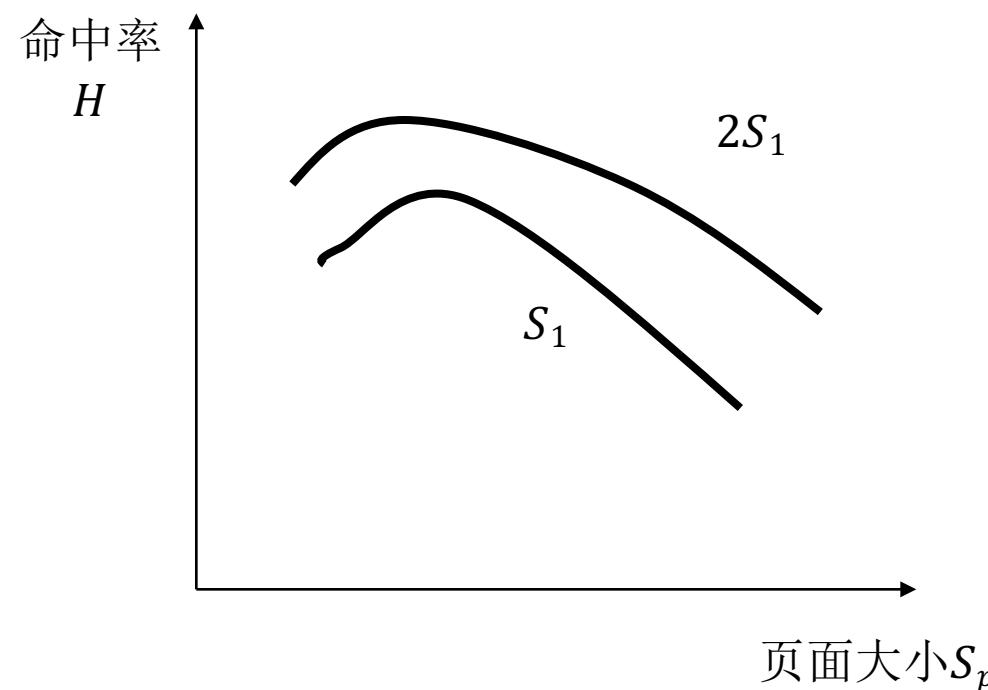
S_p ：页面大小

S_1 一定时：

S_p 由小增大， H 逐渐增大

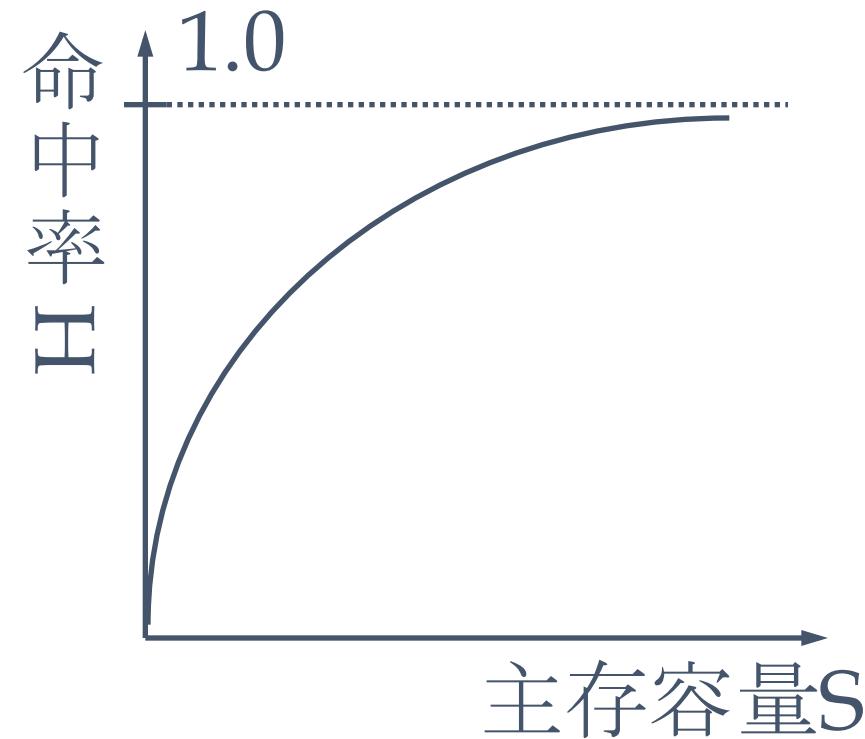
增达到最大值，下降。

增大 S_1 可增大 S_p 。



主存容量与命中率的关系

- 主存命中率 H 随着分配给该程序的主存容量 S 的增加而单调上升。
- 在 S 比较小的时候， H 提高得非常快。随着 S 的逐渐增， H 提高的速度逐渐降低。当 S 增加到某一个值之后， H 几乎不再提高。



页面调度方式与命中率的关系

- 请求式：
 - 当使用到的时候，再调入主存
- 预取式：
 - 在程序重新开始运行之前，把上次停止运行前一段时间内用到的页面先调入到主存储器，然后才开始运行程序。
 - 可以避免在程序开始运行时，频繁发生页面失效的情况。
 - 如果调入的页面用不上，浪费了调入的时间，占用了主存资源。

高速缓冲存储器（Cache）

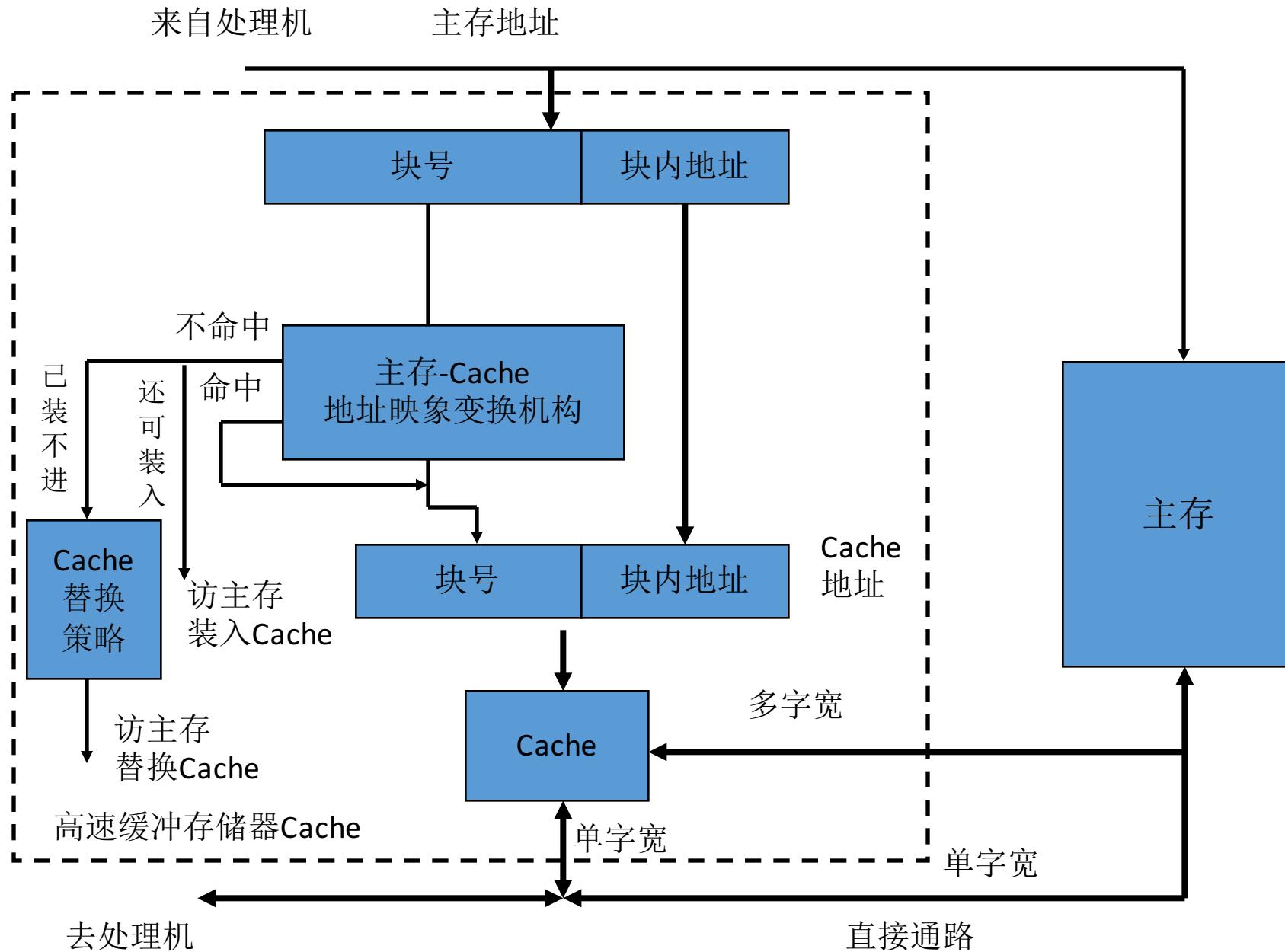
- 工作原理和基本结构
- 地址映象与变换
- Cache存储器的LRU替换算法的硬件实现
- Cache存储器的透明性及性能分析

为什么要使用Cache?

- 用以弥补主存速度的不足。
- CPU速度与主存速度相差很大（例如，一般的DRAM的工作速度比CPU慢100倍以上。）
- Cache工作速度很高，可以将其集成到CPU内。高性能CPU通常用两级Cache,一级在CPU内，其容量比较小，速度很快，第二级在主板上，容量比较大，速度比第一级低5倍左右。
- Cache全部用硬件调度对所有程序员都是透明的。
- Cache与主存储器之间以块为单位进行数据交换。块的大小通常以在主存储器的一个存储周期内可以访问到的数据长度为限。

Cache存储系统与虚拟存储系统比较

存储系统	Cache	虚拟存储器
要达到的目标	提高速度	扩大容量
实现方法	全部硬件	软件为主 硬件为辅
两级存储器速度比	3~10倍	10^5 倍
页(块)大小	1~16字	1KB~16KB
等效存储容量	主存	辅存
透明性	对系统和 应用程序员	仅对应用 程序员
不命中时处理方式	等待主存储器	任务切换



基本结构

- 把主存和Cache机械等分成相同大小的块（行），块比页小得多；
- 访问Cache的时间是访问主存时间的 $1/4$ 到 $1/10$ ；
- Cache和CPU是同类型的半导体器件；
- Cache-主存间的地址映像和变换，以及替换、调度算法用硬件实现，对应用程序员透明，也对系统程序员透明；

基本结构（续）

- 从送入主存地址到Cache的读出（或写入）包括查表地址变换和访Cache两部分，这两部分所花费的时间基本相近，可以在时间上重叠，流水地进行
- Cache在物理位置上靠近CPU，不在主存，减少传输延迟；
- 除Cache到处理机的通路外，还设有主存到处理机的通路，因此，Cache既是Cache-主存存储层次中的一级，又是处理机和主存的一个旁视存储器；
- 有Cache的主存系统都采用多体交叉存储器；
- 应尽量提高Cache的访主存的优先级；

地址映象与变换

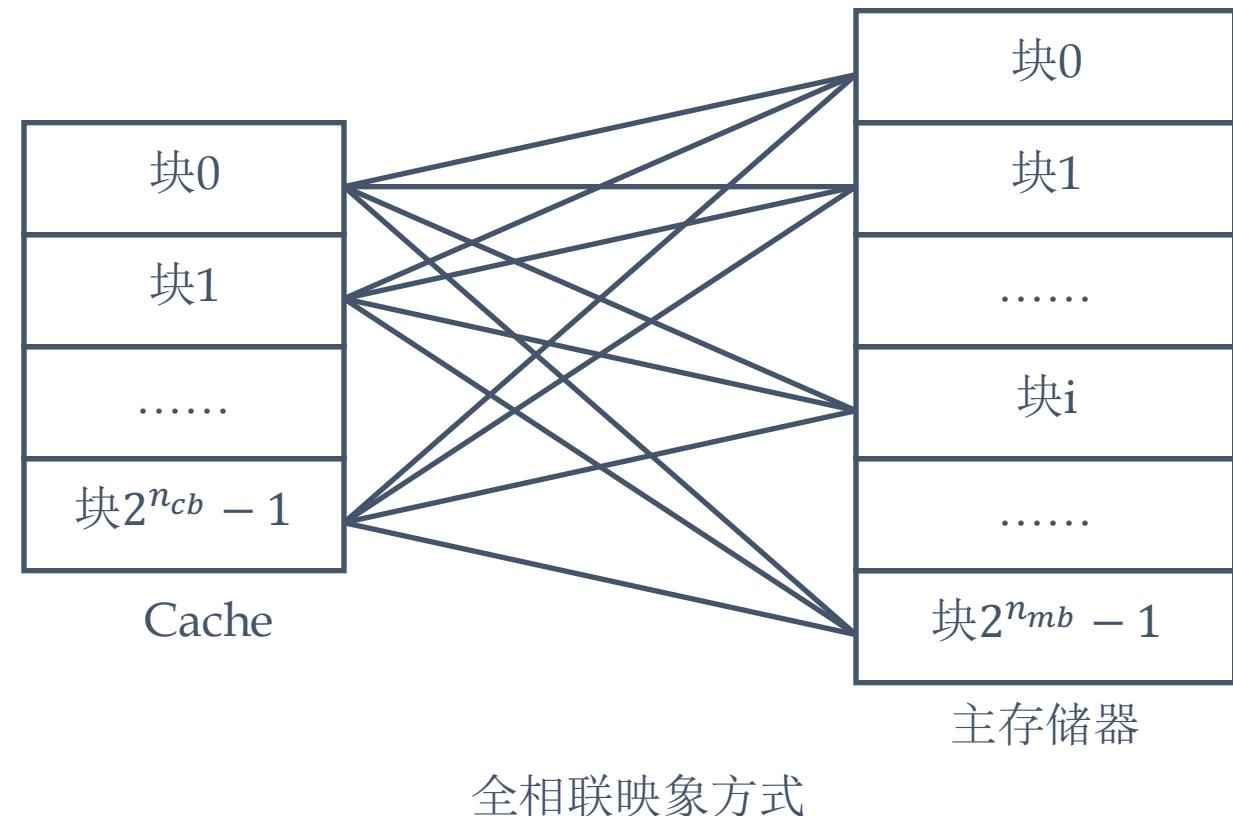
- 地址映象：是将每个主存块按某种规则（算法）装入（定位于）Cache，并建立主存地址与Cache地址之间的对应关系。
- 地址变换：是主存块按照这种映象关系装入Cache后，每次访Cache，如何将主存地址变换成Cache地址。
- 在选取地址映象方法要考虑的主要因素：
 - 地址变换的硬件容易实现；
 - 地址变换的速度要快；
 - Cache空间利用率要高；
 - 发生块冲突的概率要小

四种方式

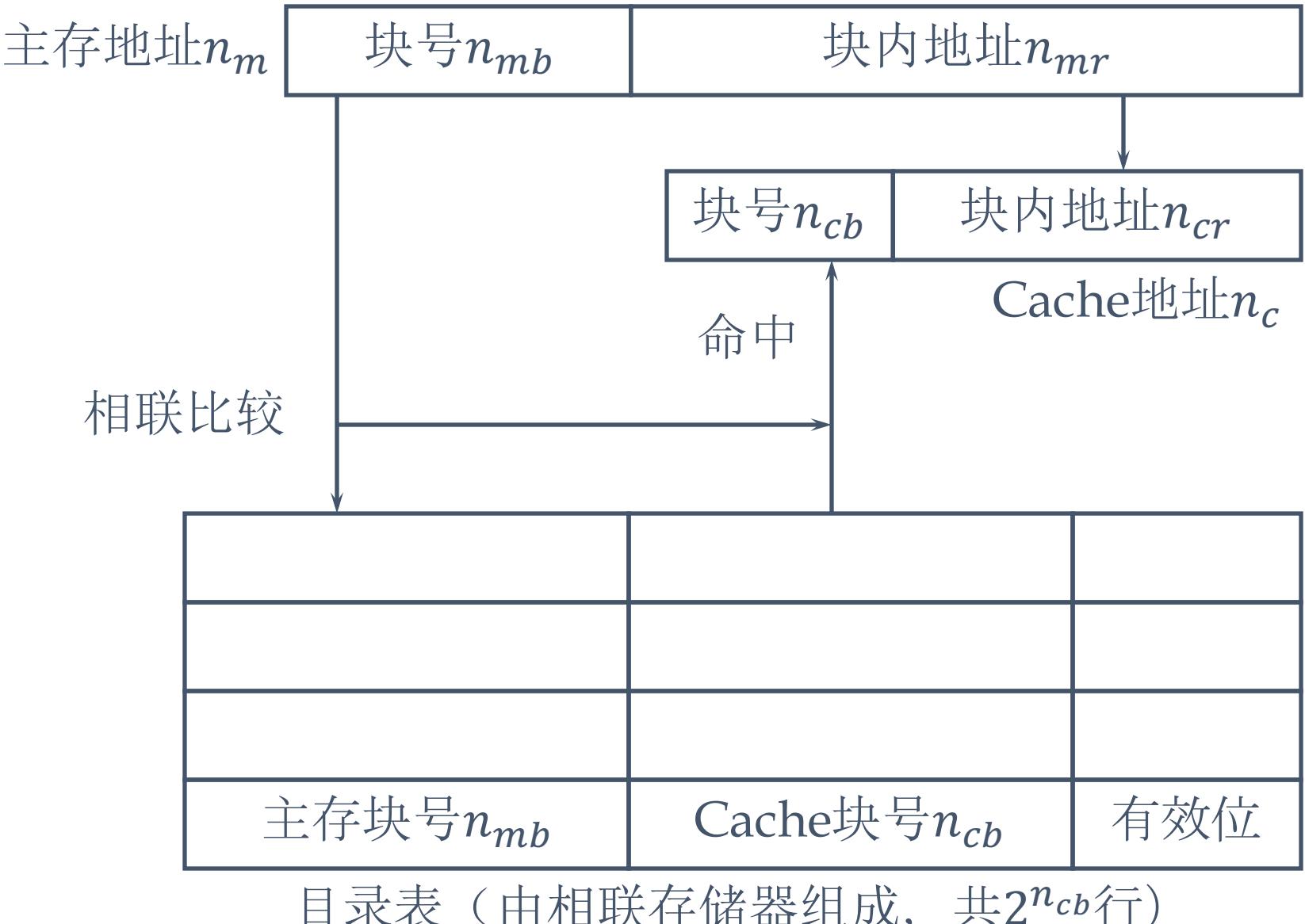
- 全相联映象与变换
- 直接映象与变换
- 组相联映像与变换
- 段相联映象

全相联映象与变换

- 定义及规则
 - 映象规则：主存中的任意一块都可以映象到Cache中的任意一块。
- 硬件方式实现

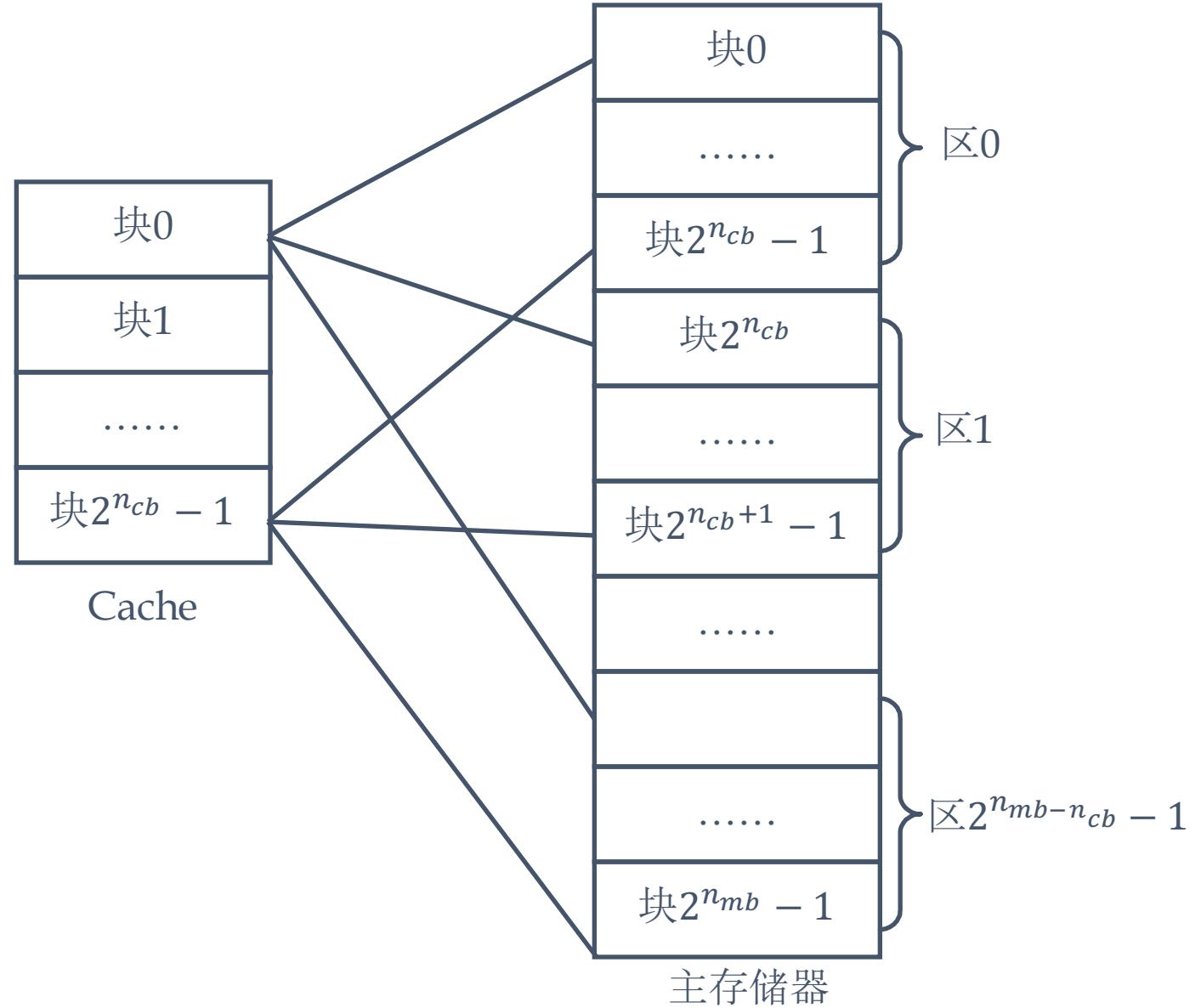


- 相联目录表法
- 特点：
 - 冲突概率低
 - 空间利用率高
 - 地址变换快

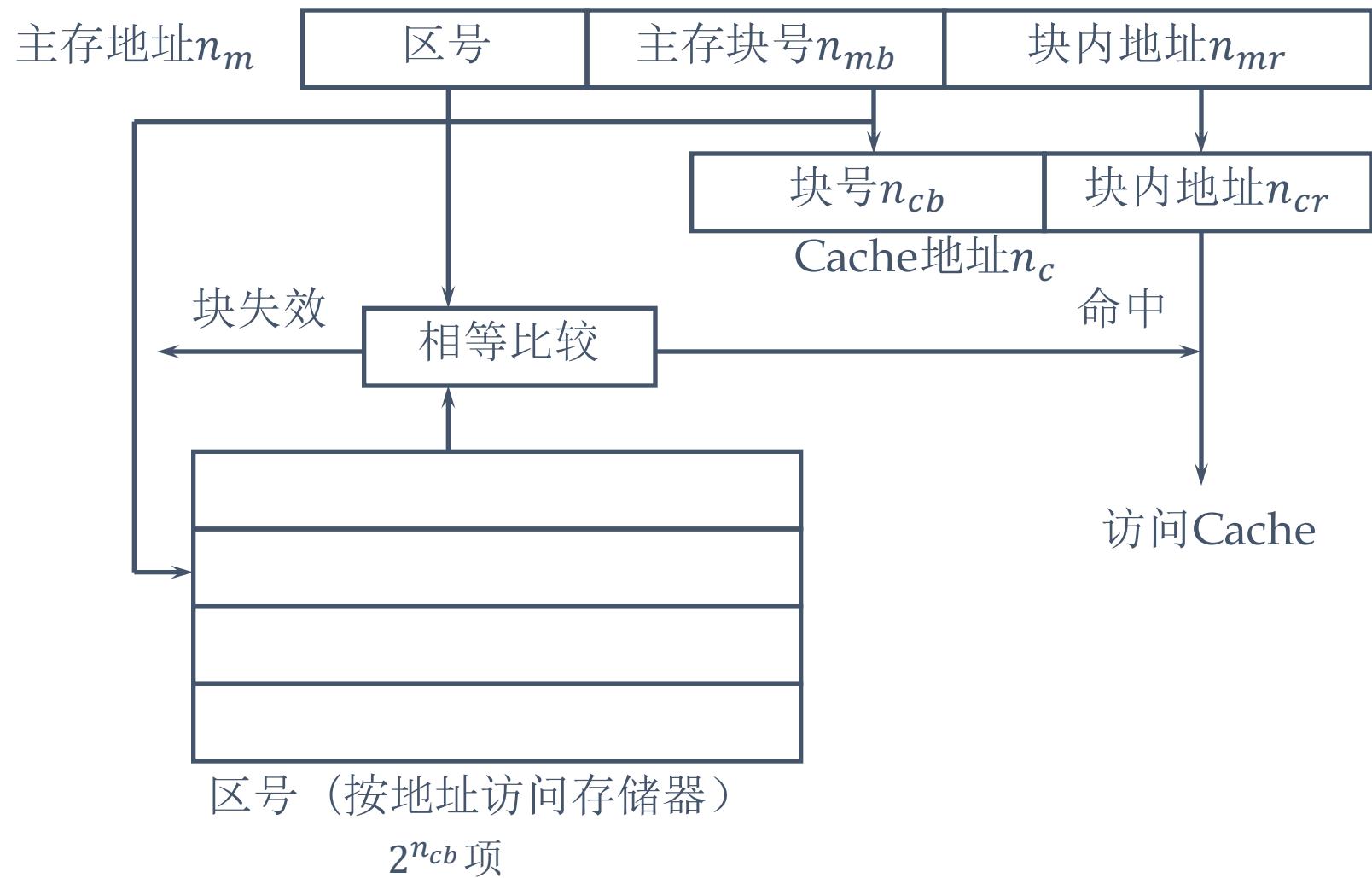


直接映象与变换

- 定义及规则
 - 映象规则：主存中一块只能映象到Cache的一个特定的块中。
 - 主存第 i 块只能映像到 $i \bmod 2^{n_{cb}}$ 块位置上
 - 整个Cache地址与主存地址的低位部分完全相同。

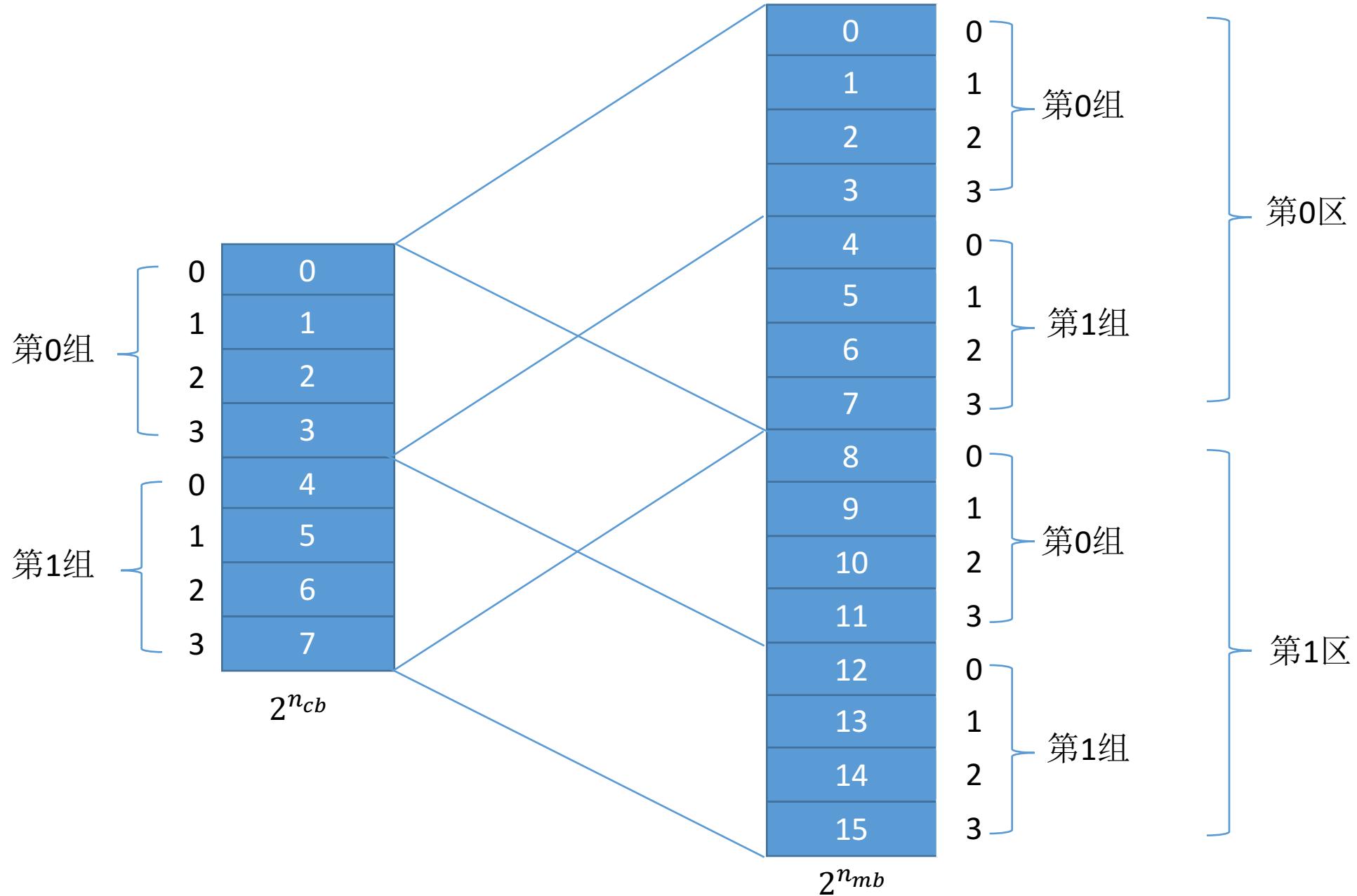


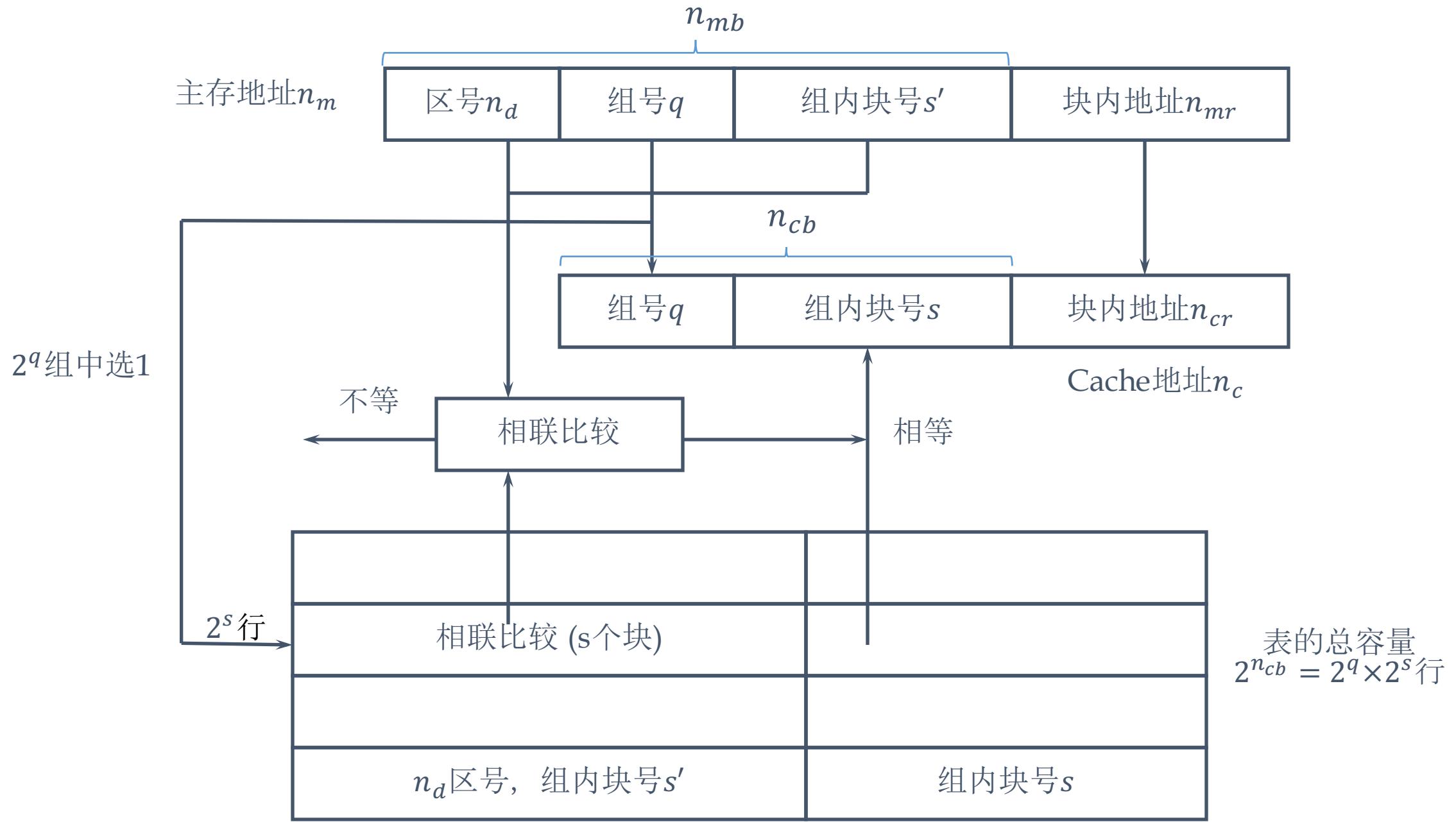
- 装入Cache中的某块位置的主存块可以来自主存的不同区。为了区分装入Cache中的块是哪一个主存区的，需要用一个按地址方位的表存储器来存放Cache中每一块位置目前是被主存中的哪个区的块所占用的区号
- 特点：
 - 硬件简单
 - 冲突概率高
 - 出现大量空闲块
 - 很少使用。



组相联映像与变换

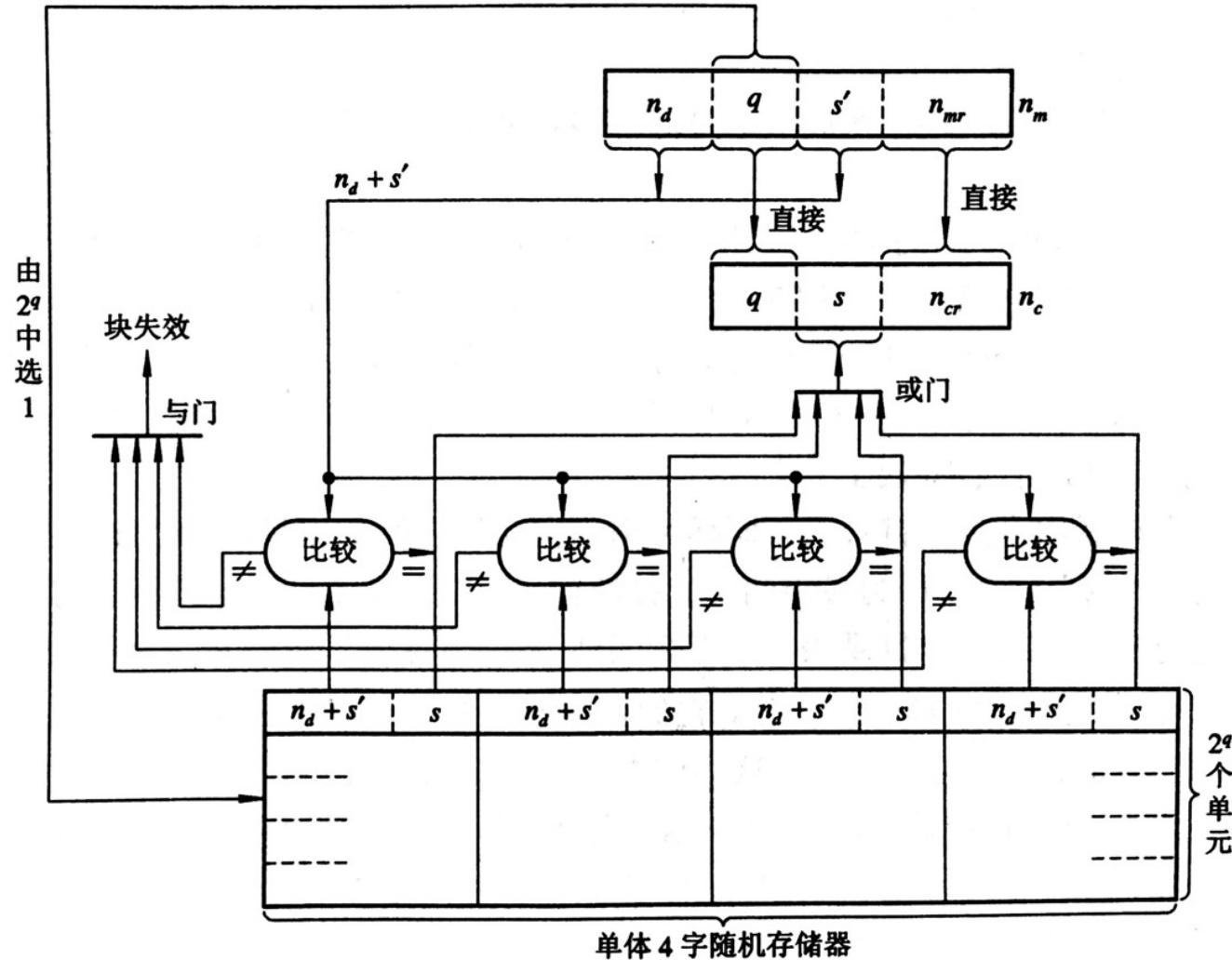
- 定义及规则：各组之间是直接映象，组内各块间是全相联映象。
- 将Cache空间和主存空间都分成组，每组 S 块（ $S=2^s$ ）
- Cache分成 Q 组（ $Q=2^q$ ），整个Cache作为一个区
- 主存分成和Cache同样大小的 2^{n_d} 个区，每个区包含 Q 组





- 组相联映象方式的优点：
 - 块的冲突概率比较低
 - 块的利用率大幅度提高
 - 块失效率明显降低
- 组相联映象方式的缺点：实现难度和造价要比直接映象方式高
- 地址变换过程：用主存地址的组号按地址访问块表存储器
- 把读出来的一组区号和块号与主存地址中的区号和块号进行相联比较
 - 如果有相等的，表示Cache命中
 - 如果没有相等的，表示Cache没有命中

- 采用按地址访问和按内容访问混合的存储器实现组相联地址映像机构
- 采用单体多字并行存储器
- 每组的块数 s 不能很大
- 尽管采用组相联会增加一些Cache块冲突的概率和降低Cache的空间利用效率，但是该方法提高了速度
- 随着半导体集成电路技术的发展， s 可以增大，以进一步降低Cache块冲突的概率



段相联映象

- 减少相联目录表的容量，降低成本，提高地址变换速度。
- 组间全相联，组内直接映象。

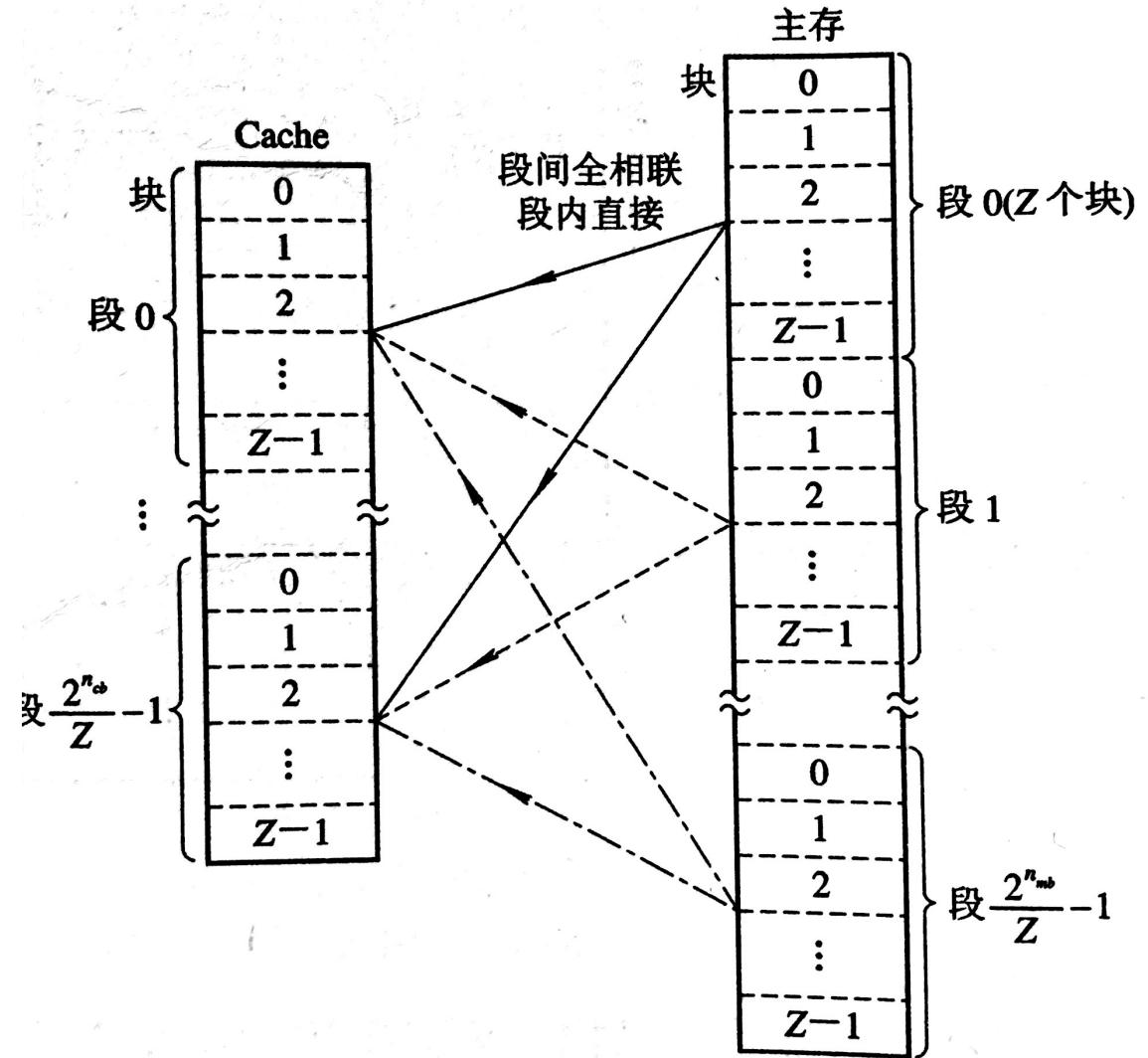


图 4 - 33 具有每段 Z 个块的段相联映像

替换算法的实现

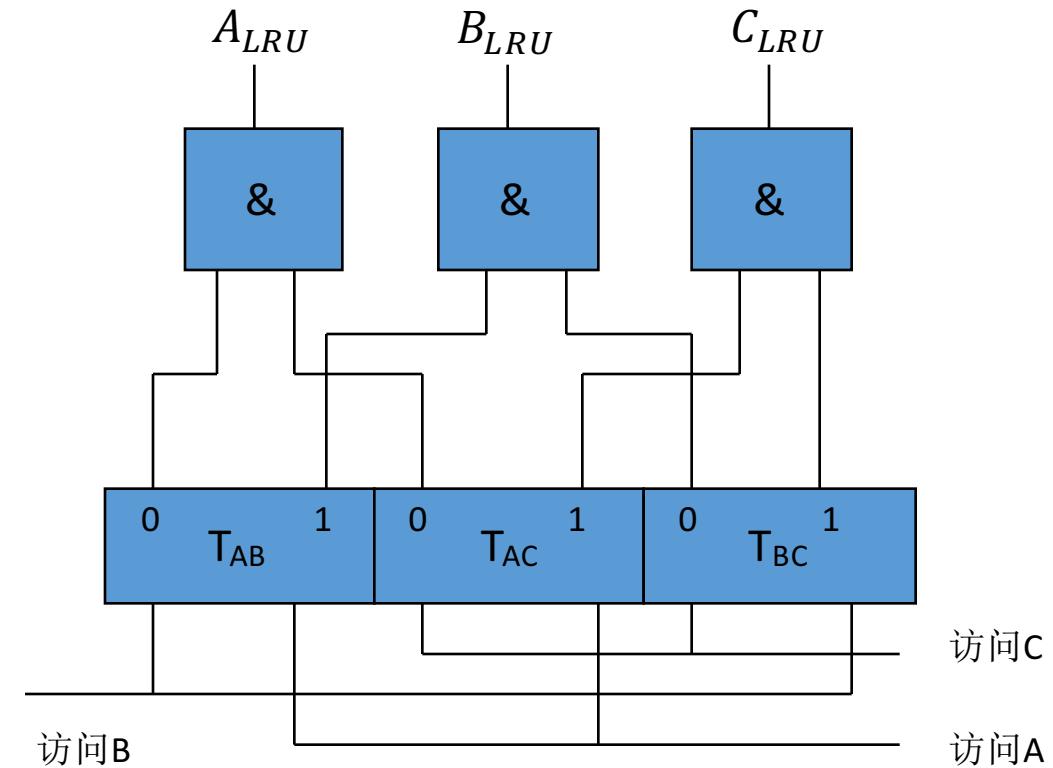
- 常采用LRU算法， LRU算法是堆栈型算法
- 由于Cache的调块时间是微秒级， 不能采用程序换道
- 替换算法全部采用硬件途径实现

替换算法的设计要考虑的问题

- 如何对每次访问进行记录（适用位法、堆栈法、比较对法所用的记录方法都不同）
- 如何根据所纪录的信息来判定近期内哪一块是最久没有被访问过的
- Cache替换算法全部用硬件实现

LRU的比较对法

- 让各个块成对组合，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。
- 假设有A、B、C三块，组成AB、AC、BC三对，其访问顺序分别用对触发器 T_{AB} 、 T_{AC} 、 T_{BC} 表示
- 若A比B更近被访问，则 $T_{AB} = 1$ ，否则， $T_{AB} = 0$ 。同理，可定义 T_{AC} 和 T_{BC}
- 若访问过的次序为ABC或者BAC，则C是最久未被访问的块，即当 $T_{AB} = 1, T_{AC} = 1, T_{BC} = 1$ 或者 $T_{AB} = 0, T_{AC} = 1, T_{BC} = 1$ 时， $C_{LRU} = 1$



$$C_{LRU} = T_{AC} T_{BC}$$

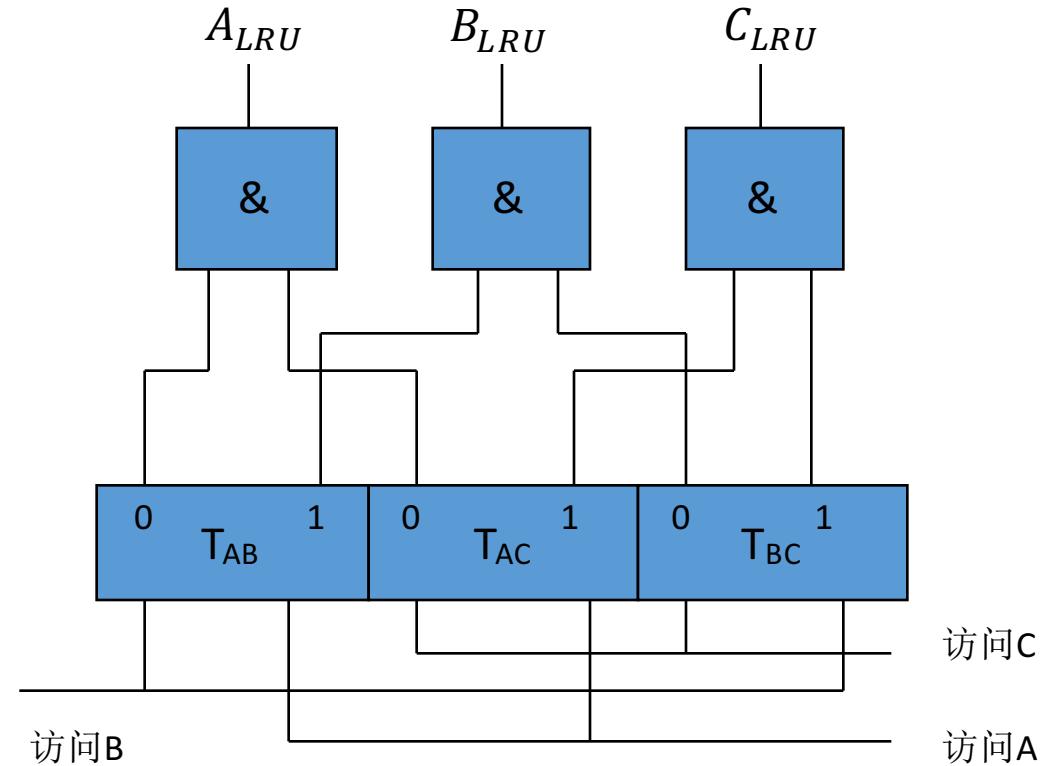
$$B_{LRU} = T_{AB} \overline{T_{BC}}$$

$$A_{LRU} = \overline{T_{AB}} \overline{T_{AC}}$$

- 假设 P 为块数
- 比较触发器的个数

$$C_P^2 = P(P - 2)/2$$

- 门数 P
- 门的输入端 $P - 1$



$$C_{LRU} = T_{AC} T_{BC}$$

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

$$A_{LRU} = \overline{T_{AB}} \overline{T_{AC}}$$

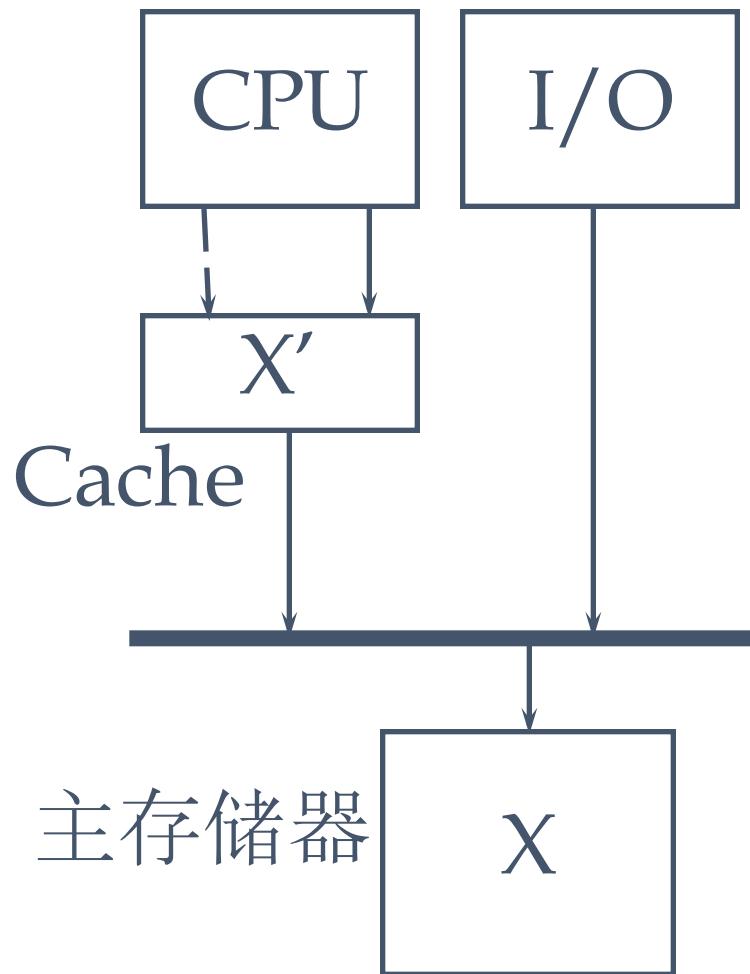
- 比较对触发器的个数会随块数的增多以极快的速度增加，门的输入端数也线性增加，这会在工程实现上带来麻烦，所以比较对法只适用于组内块数较少的组相联映像Cache存储器中
- 若组内块数 ≥ 8 时，则所需要的比较对触发器的个数就多得不能承受了
- 可以用多级状态位技术来减少所用的比较对触发器个数
 - 如IBM 3033中，组内块数为16，可分成群、对、行3级
 - 4个群需要6个比较对触发器；每个群分成2对，需要一位触发器指示状态，因此4个群需要4位；每对中的行需要1位触发器进行状态指示，共需要8位
 - 因此，全部触发器的个数为 $6+4+8=18$ ，比单级的120个要少得多
 - 但是多级状态位技术牺牲了速度

Cache存储器的透明性及性能分析

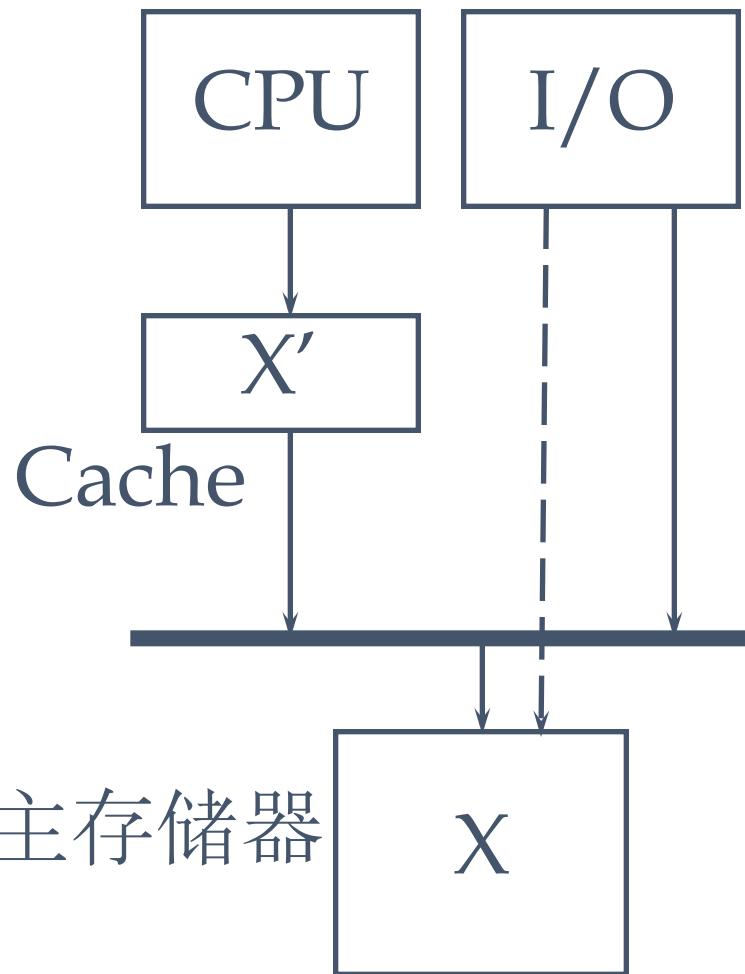
- Cache的透明性
- Cache的取算法
- Cache存储器性能分析

Cache的透明性和一致性问题

- 由于Cache存储器的地址变换和块替换算法全由硬件实现，因此Cache-主存存储层次对应用程序员和系统程序员都是透明的。
- Cache是主存的一部分副本，主存中某单元的内容可能在一段时期里与Cache中所对应单元的内容不一致
 - 由于CPU写Cache，修改了Cache中某单元的内容，但没有立即写主存来更新对应内容，若此时CPU、I/O处理机或其他处理机要经过主存交换信息，会造成错误
 - I/O处理机或其他处理机将新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，若此时CPU从Cache中读取信息，也会因内容不一致造成错误



(a) CPU写Cache



(b) I/O写主存

Cache与主存不一致的两种情况

Cache的透明性

- 解决因CPU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
- 写回法（抵触修改法， WB）：是在CPU执行写操作时，信息只写入Cache，仅当需要被替换时，才将以被写入过的Cache块先送回主存，然后再调入新块。
- 写直达法（直达法， WT）：利用Cache—主存存储层次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。

- 写回法和写直达法都需要少量缓冲器
 - 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
 - 写直达法则用于缓存由写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
 - 缓冲器对于Cache和主存是透明的，在设计时要处理好可能引起的错误（例如，另一个处理机要访问的主存单元的内容正好在缓冲器中）

- 在与主存的通信量方面,写回法少于写直达法

假设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存次数之比

- 写直达法: 16%
- 写回法: $3\% * 30\% * 32/8 = 3.6\%$

- 可靠性，写直达法优于写回法
 - 写直达法在Cache出错时可以由主存来纠正，因此Cache只需要一位奇偶校验位
 - 写回法则由于有效块只在Cache中，因此需要在Cache中采用纠错码，利用冗余信息来提高内容可靠性
- 控制的复杂性：写直达法比写回法简单
- 硬件实现的代价：写回法要比写直达法好
- 采用何种算法与适用场合有关
 - 单处理机（节省成本）：写回法
 - 共享主存的多处理机（保证信息交换可靠）：写直达法

- 若多个处理机共享Cache交换信息，则信息的一致性可以得到保证。但是该方法很难实现
 - 需要增大Cache容量
 - 需要共享的Cache在物理上靠近多个CPU以减少时延，但会降低速度
 - Cache的频宽很难支持两个以上CPU的同时访问
 - 共享Cache目前仅限于单CPU、多I/O处理机系统上

- 共享主存多CPU系统
 - 播写法：任何处理机要写入Cache时，不仅要写入自己的Cache的目标块和主存中，还要把信息播写到所有Cache有此单元的地方，或者让所有Cache有此单元的块作废
 - 控制某些共享信息（如作业队列等）不得进入Cache
 - 目录表法

Cache内容滞后于主存内容

- 当I/O处理机未经Cache向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
- 当I/O处理机未经Cache向主存某区域写入新内容时，由专用硬件自动将Cache内对应区域的副本作废

Cache的取算法

- 按需取进法：出现Cache块失效时，才将要访问的字所在的块（行）取进。
- 预取法
 - **恒预取**：只要访问到主存第 i 块的某个字，不论Cache是否命中，恒预取第 $i + 1$ 块。
 - **不命中时预取**：仅当访问第 i 块不命中时，才预取命第 $i + 1$ 块。
- 采用预取法并非能提高命中率，其他因素
 - 块的大小
 - 预取开销

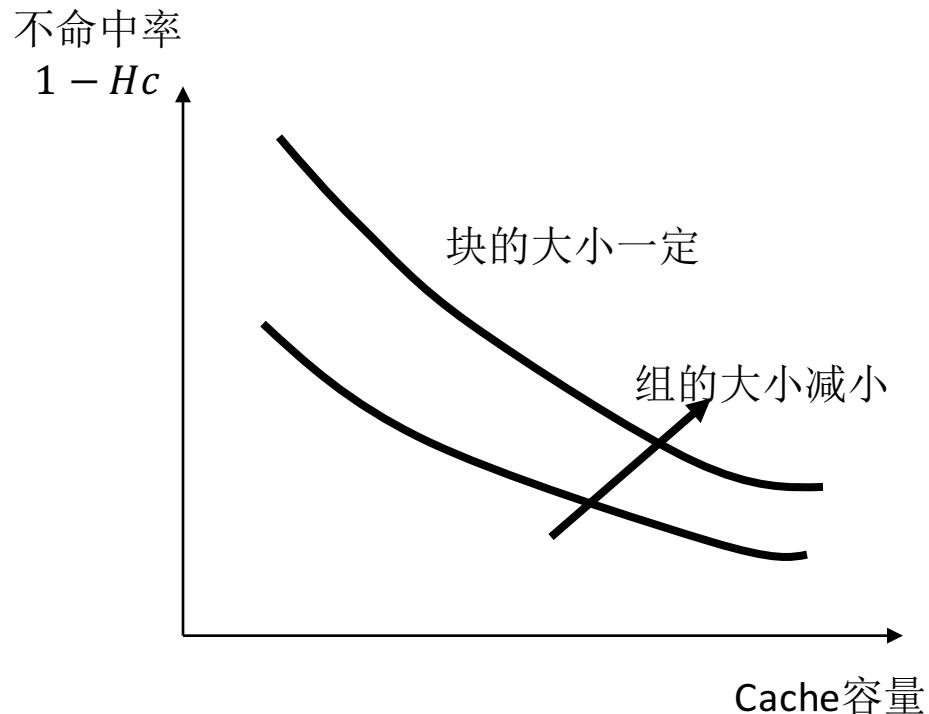
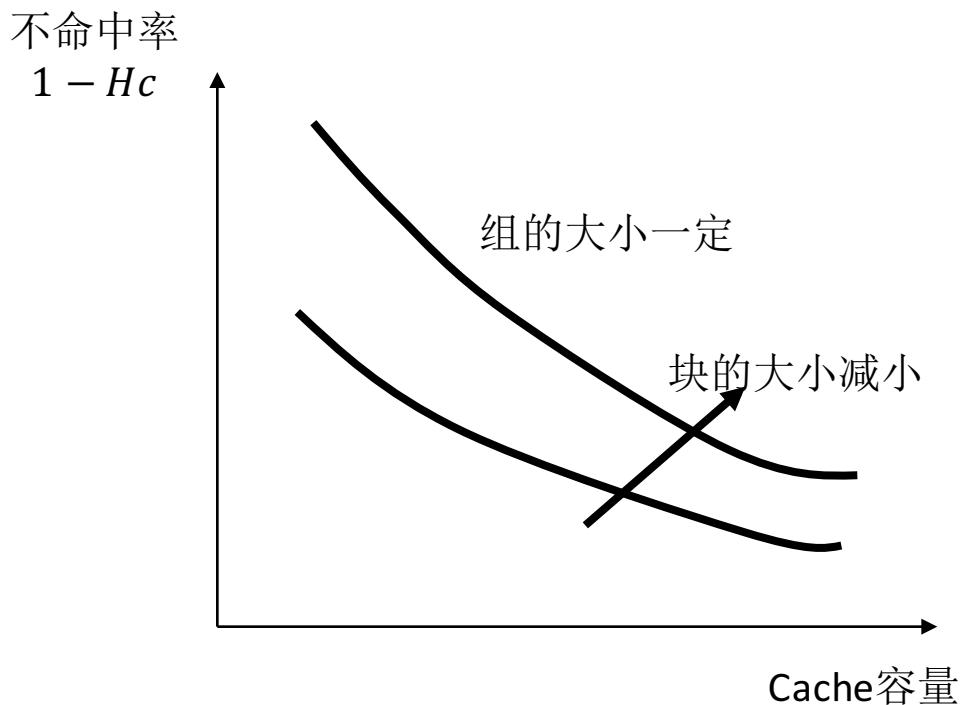
说明

- 采用缓冲器技术是减少预取干扰的好办法
- 模拟结果表明
 - 恒预取法使不命中率降低75%--80%
 - 不命中率时预取法使不命中率降低30%--40%
 - 但前者所引起的Cache、主存间传输量的增加要比后者大得多。

Cache存储器性能分析

- 不命中率与Cache的容量、组的大小和块的大小的关系
- Cache-主存存储层次的等效速度与命中率的关系推导
- Cache的容量对机器速度的关系

块的大小、组的大小与Cache容量对Cache命中率的影响



块的大小、组的大小及Cache容量增大时都能提高命中率

Cache-主存存储层次的等效速度与命中率的关系推导

设： t_c 为 Cache 的访问时间， t_m 为主存周期， H_c 为访 Cache 命中率。

则： Cache 的等效存储周期

$$t_a = H_c t_c + (1 - H_c) t_m$$

因为： 主存与 CPU 之间有直接通路，因此 CPU 对第二级的访时间就是 t_m 。

速度提高倍数是：

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

因为 H_c 总小于 1，可以令 $H_c = \alpha/(\alpha+1)$

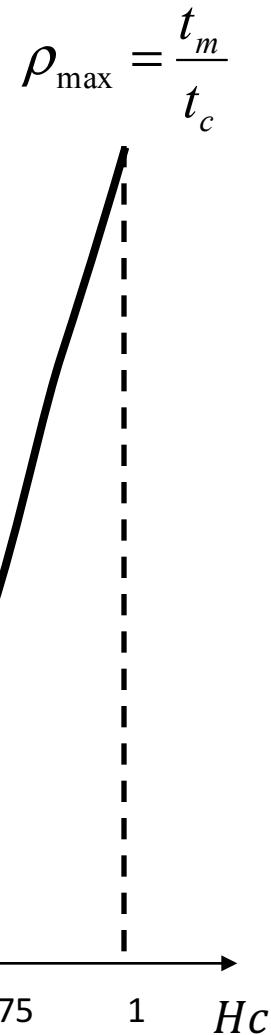
$$\rho = \frac{1}{1 - \left(1 - \frac{t_c}{t_m}\right) * \frac{\alpha}{\alpha+1}} = (\alpha+1) * \frac{t_m}{t_m + \alpha t_c} < \alpha+1$$

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- 主存存储层次后，速度能提高的最大值是有限的，不会超过 $\alpha+1$

举例

- $Hc = 0.5, \alpha = 1$
 ρ 的最大值<2
- $Hc = 0.75, \alpha = 3$
 ρ 的最大值<4
- $Hc = 1$

ρ 的期望值



举例

- 由于Cache的命中率一般比0.9大的多，可达0.996，因此 ρ 接近于所期望的 t_m/t_c
- H_c 受Cache容量的影响很大。
 - 容量为4kb时， $H_c = 0.93$
 - 容量为8kb时， $H_c = 0.97$

举例

- 因此在 $t_m/t_c = 0.12$ 时
 - 4KB的Cache，速度的倍数是

$$\rho_{4k} = \frac{1}{1 - (1 - t_c/t_m)H_c} = \frac{1}{1 - 0.88 * 0.93} \approx 5.5$$

- 8KB的Cache，速度的倍数是

$$\rho_{8k} = \frac{1}{1 - 0.88 * 0.97} \approx 6.85$$

- 增加4KB容量，带来层次速度的提高：

$$\frac{\rho_{8k} - \rho_{4k}}{\rho_{4k}} = \frac{6.85 - 5.5}{5.5} \approx 0.24(24\%)$$

- 改进Cache存储器的等效访问速度
 - 若等效访问速度与Cache本身的速度差很远，则说明Cache的命中率较低，应调整组的大小、块的大小、替换算法以及增大Cache容量，以提高Cache命中率
 - 若等效访问速度已经接近于Cache本身的速度，则应更换更高速的Cache片子

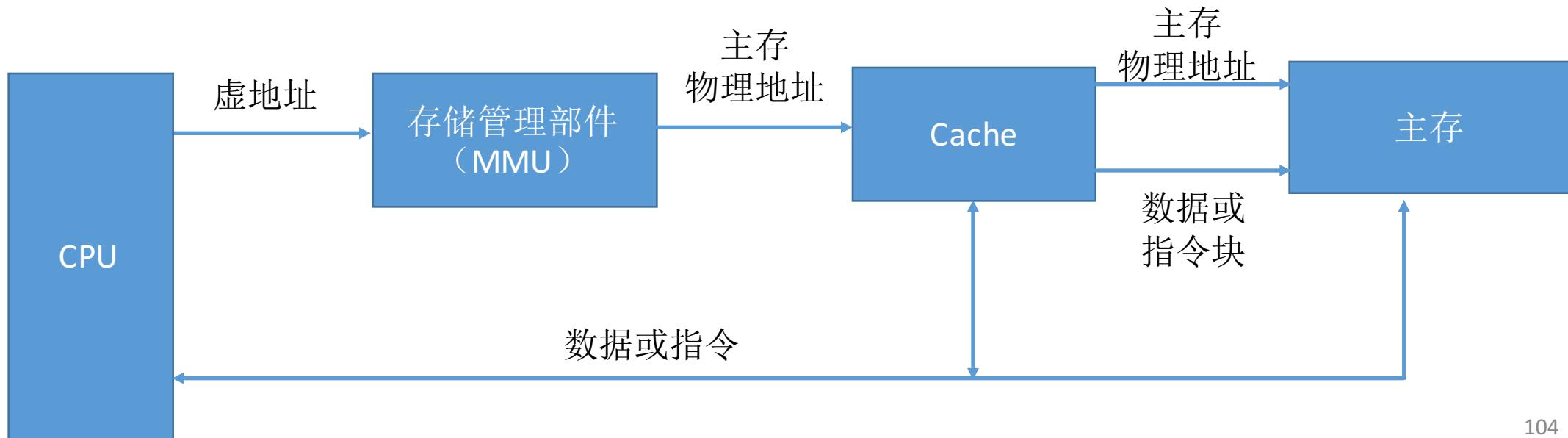
三级存储体系

计算机系统同时拥有虚拟存储器和Cache存储器，程序采用虚地址访存，要求速度接近于Cache，容量接近于辅存

- 物理地址Cache
- 虚地址Cache
- 全Cache技术

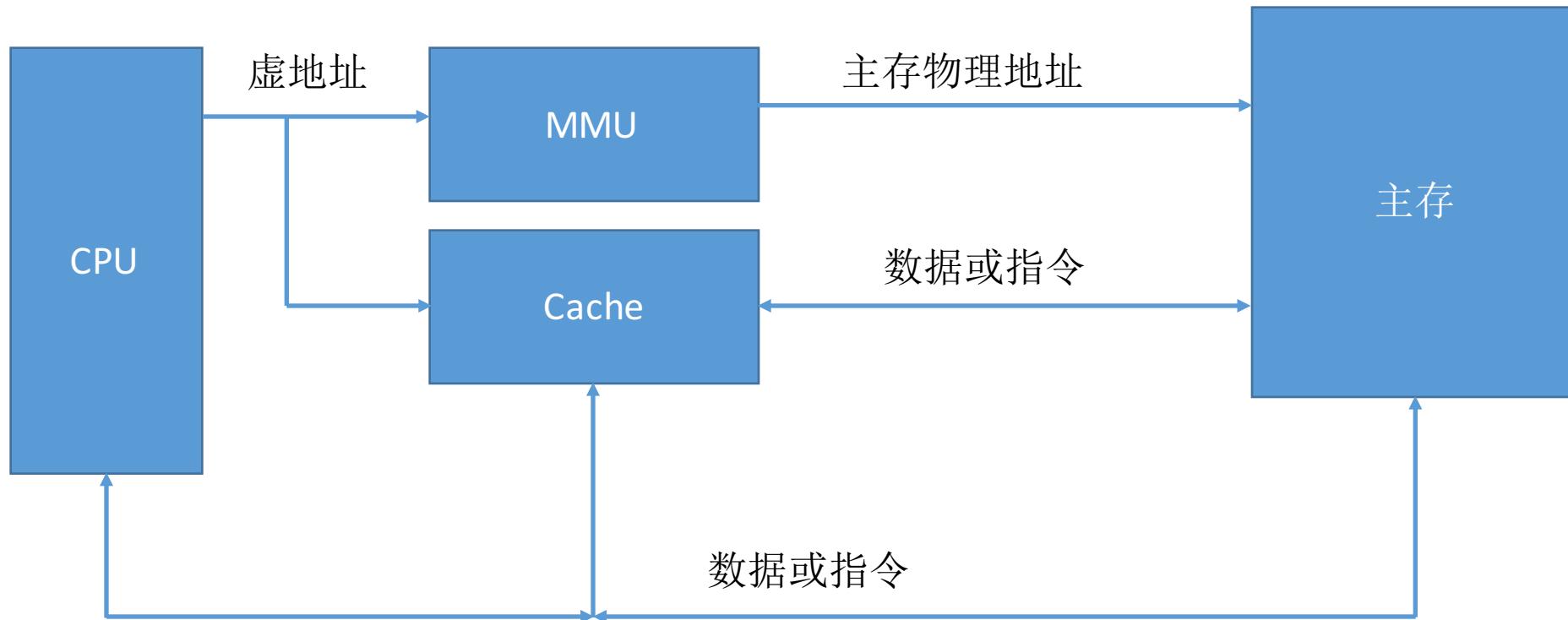
物理地址Cache

- 由“Cache—主存”和“主存—辅存”两个独立的存储层次组成
- 当CPU要访问存储器时，给出一个虚拟地址，由存储管理部件（MMU: Memory Management Unit）中的地址变换部件把该虚地址变换成主存物理地址，然后用主存物理地址访问Cache
- 若要访问的数据或者指令在Cache中找到，则Cache命中；否则，Cache失效，用该物理地址访问主存储器，取出数据或指令装入Cache和送往CPU

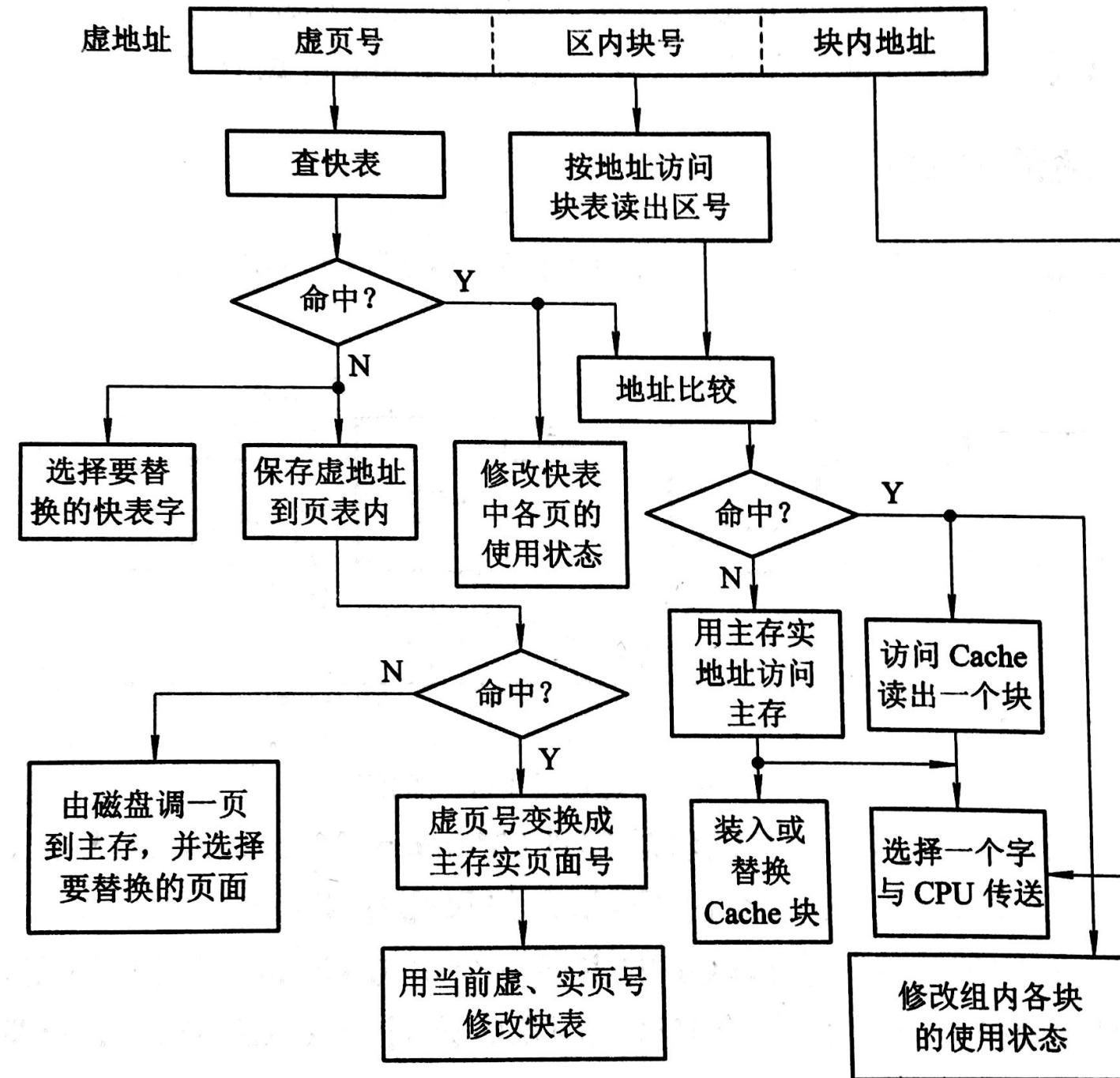


虚地址Cache

- 如果对 Cache的访问采用主存实地址，则需要先将虚拟地址转换为主存实地址，这必然增加了访问Cache的时间花费
- 虚地址Cache将Cache—主存—辅存直接构成三级存储层次，采用虚地址访问Cache

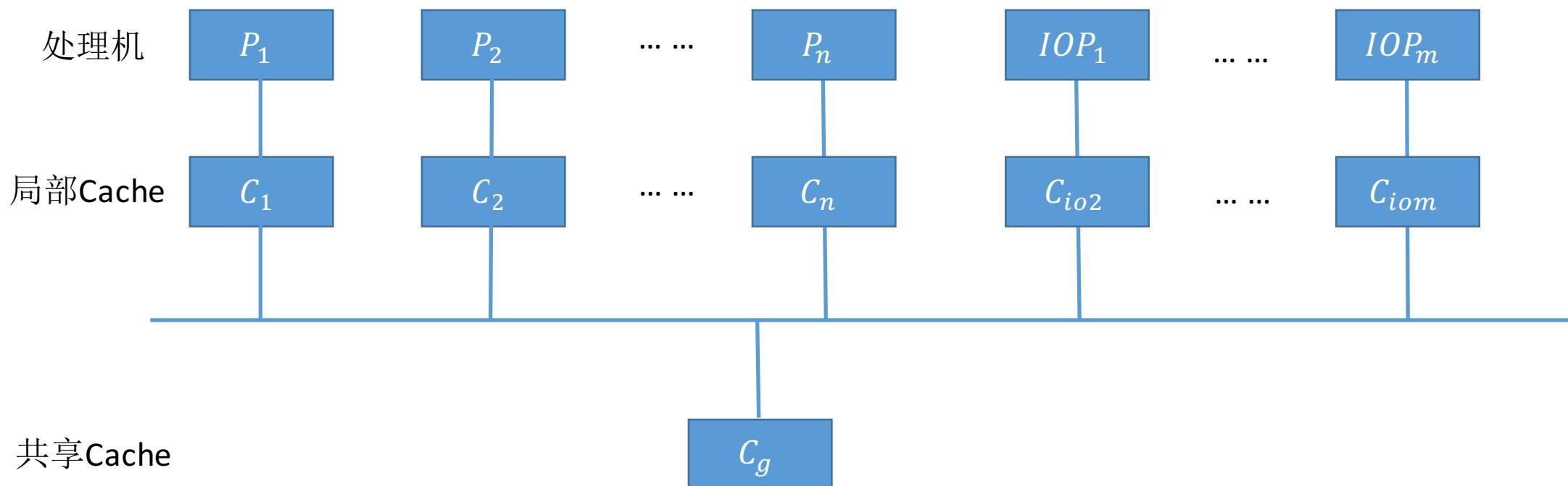


- 虚存采用组相联的映像和变换方式，虚存的一个页恰好是主存的一个区
- 用虚地址的区内块号B按地址访问Cache的块表，读出主存区号E（即页号P）和对应的Cache块号b
- 在访问Cache块表的同时，用虚地址的虚页号访问快表
- 若快表命中，块表中读出的主存区号E和从快表中得到的主存实页号P相等，则Cache命中，将虚地址中的区内块号B直接作为Cache地址中的组号g，从块表中读出Cache的组内块号b，把虚地址中的块内地址w直接作为Cache地址中的块内地址w，拼接成Cache地址，送往CPU；若Cache不命中，则用主存实地址页号P拼接上虚拟地址中的的区内块号B和块内地址w，得到主存实地址去访问主存储器，将读出的目标字送往CPU，并将目标字所在的块送入Cache
- 若快表没有命中，则通过软件的方法查询主存中的慢表，其后的工作过程与页式虚存或段页式虚存类似



全Cache技术

- “Cache-辅存” 存储系统



存储系统的保护

- 在多用户计算机系统中，为使系统正常工作： 1) 应防止由于一个用户程序出错而破坏主存中其他用户的程序或系统软件； 2) 还要防止一个用户程序不合法地访问未分配给它的的主存区域
- 系统结构需要为主存的使用提供存储保护
 - 区域保护
 - 访问方式的保护

区域保护

- 若每个程序占用主存的一个或几个连续区域时，可采用界限寄存器的方式，由系统软件置定上、下界寄存器，从而划定每个用户程序的区域，禁止越界访问
- 对于虚拟存储器，一个用户的各个页离散地分布于主存内，需要采用页表保护和键式保护等方式

页表保护

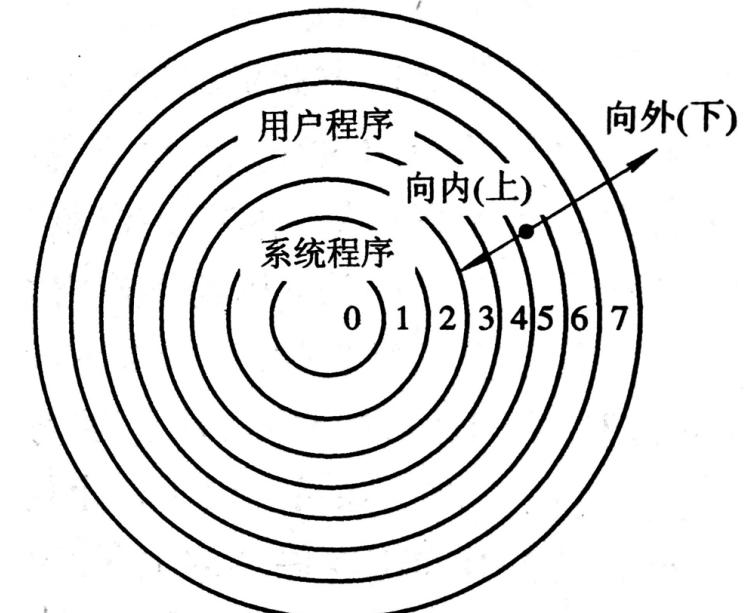
- 每个程序都有自己的页表，每个虚页都对应着一个实页
- 虚地址的错误只能影响其对应的实页，虚页号的错误可以由页表本身防范
- 为便于保护，还可在段表中的每行内不仅设置页表的起点，还设置段长（即页数），使得段内虚页号大于段长的情况可以引发越界中断
- 页表保护是在没有形成主存实地址前进行保护，若地址形成环节由于软、硬件的故障形成了不属于本程序区域的错误地址，则页表保护无法发挥作用

键式保护

- 由操作系统根据当前的主存使用分配状况给主存的每个页配一个键，称为存储键（相当于一把锁）
- 所有页的存储键存在于主存相应的快速寄存器内，每个用户的各个实页的存储键相同
- 操作系统给每个用户分配一个访问键（相当于一把钥匙），存在处理机的程序状态字（PSW）或控制寄存器中
- 程序每次访问主存前，需要核对主存地址所在页的存储键与该程序的访问键是否相符，只有相符，才准许访问

环式保护：对正在执行的程序本身进行保护

- 把系统程序和用户程序按其重要性及对整个系统能否正常工作的影响程度分层，环号越大，等级越低
- 程序运行之前，先由操作系统定好程序各页的环号，并置入页表，而后把该道程序的开始环号送入处理器内的现行环号寄存器，并把操作系统规定的该程序的上限环号也置入相应寄存器
- 若 P_i 是某一时刻属 i 层各页的集合，则当进程执行 $P \in P_i$ 页内程序时，允许访问 $F \in P_j$ ($j \geq i$)
- 若 $j < i$ ，则需由操作系统环控制例行程序判定这个内向访问是否允许和是否正确之后才能访问，否则出错，进入保护处理



访问方式的保护

- 对主存信息的使用可分为读 (R)、写 (W) 和执行 (E) 三种

$\overline{R \vee W \vee E}$ ——不允许进行任何访问(如专用的系统表格);

$R \vee W \vee E$ ——可以进行任何访问;

$R \wedge \overline{W \vee E}$ ——只能进行读访问(如对各个用户都用到的表格常数);

$(R \vee W) \wedge \overline{E}$ ——只能按数据进行读、写(例如阵列数据当然不能作为指令执行);

$\overline{R \vee W} \wedge E$ ——只能执行, 不能作为数据使用(如某个专门的程序);

$\overline{R \vee E} \wedge W$ ——只能进行写访问(如用户对操作系统缓冲器的写入);

$(R \vee E) \wedge \overline{W}$ ——不准写访问。

小结

- 存储系统与存储体系
- 并行主存频宽的分析
- 存储体系的性能参数
- 虚拟存储器的管理方式
- 四种地址映像与变换的含义与区别
- Cache的工作原理