

第3章 存储系统

肖梦白

xiaomb@sdu.edu.cn

<https://xiaomengbai.github.io>

现代计算机系统以存储器为中心

3.1 存储系统原理

3.2 虚拟存储器

3.3 高速缓冲存储器(Cache)

3.4 三级存储系统

3.1 存储系统原理

3.1.1 存储系统的定义

3.1.2 存储系统的层次结构

3.1.3 存储系统的频带平衡

3.1.4 并行访问存储器

3.1.5 交叉访问存储器

3.1.6 无冲突访问存储器

3.1.1 存储系统的定义

➤ 在一台计算机中，通常有多种存储器

- **种类：**主存储器、Cache、通用寄存器、缓冲存储器、磁盘存储器、磁带存储器、光盘存储器等
- **材料工艺：**ECL、TTL、MOS、磁表面、激光，SRAM，DRAM
- **访问方式：**随机访问、直接译码、先进先出、相联访问、块传送、文件组

3.1.1 存储系统的定义

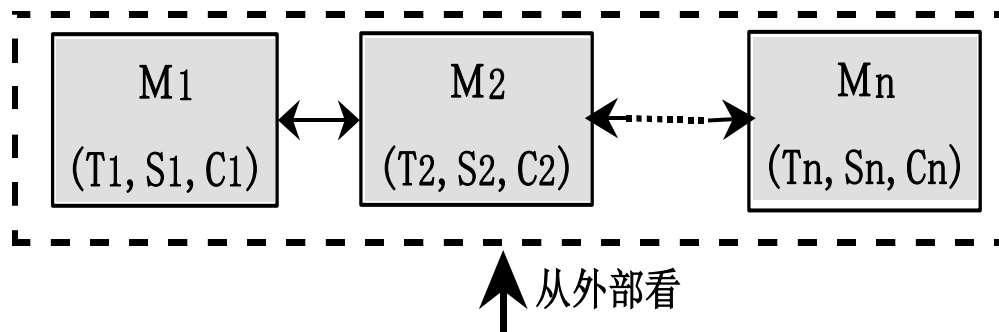
- 存储器的主要性能：速度、容量、价格
 - 速度用存储器的访问周期、读出时间、频带宽度等表示。
 - 容量用字节B、千字节KB、兆字节MB和千兆字节GB等单位表示。
 - 价格用单位容量的价格表示
- 组成存储系统的关键：把速度、容量和价格不同的多个物理存储器组织成一个存储器，这个存储器的速度最快，存储容量最大，单位容量的价格最便宜。

1. 存储系统的定义

- 两个或两个以上速度、容量和价格各不相同的存储器用硬件、软件、或软件与硬件相结合的方法连接起来成为一个存储系统。这个存储系统对应用程序员是透明的，并且，从应用程序员看，它是一个存储器，这个存储器的速度接近速度最快的那个存储器，存储容量与容量最大的那个存储器相等，单位容量的价格接近最便宜的那个存储器。
- 虚拟存储器系统：对应用程序员透明
- Cache存储系统：对系统程序员以上均透明

1. 存储系统的定义

由多个存储器构成的存储系统



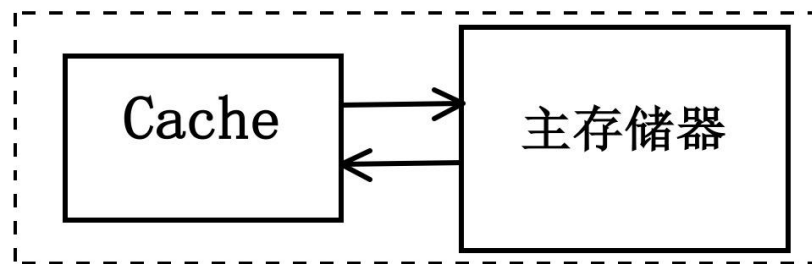
$T \approx \min (T_1, T_2, \dots, T_n)$, 用存储周期表示

$S = \max (S_1, S_2, \dots, S_n)$, 用 MB 或 GB 表示

$C \approx \min (C_1, C_2, \dots, C_n)$, 用每位的价格表示

1. 存储系统的定义

- 一般计算机系统有两种存储系统：Cache存储系统和虚拟存储系统
- Cache存储系统
 - 由Cache和主存储器构成
 - 主要目的是提高存储器速度



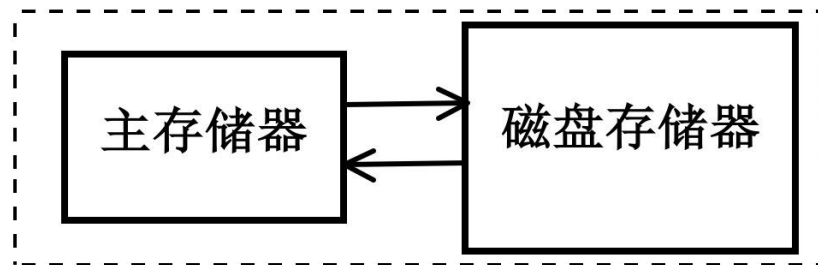
↑系统程序员看：

速度接近 Cache 的速度，
存储容量是主存的容量，
每位价格接近主存储器。

Cache 存储系统

1. 存储系统的定义

- 虚拟存储系统：由主存储器和硬盘构成
- 主要目的：扩大存储器容量



↑应用程序员看：

速度接近主存储器的速度，
存储容量是虚拟地址空间，
每位价格接近磁盘存储器。

虚拟存储系统

2. 存储系统的容量

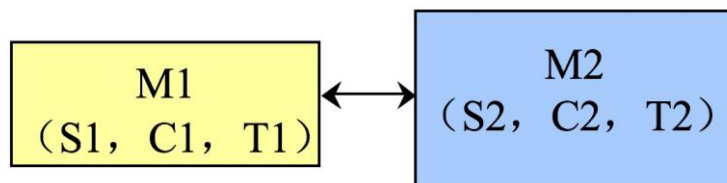
- 要求：提供尽可能大的地址空间，能够随机访问
- 方法有两种：
 - 只对系统中存储容量最大的那个存储器进行编址，其他存储器只在内部编址或不编址，例如Cache存储系统
 - 另外设计一个容量很大的逻辑地址空间，把相关存储器都映射这个地址空间中，例如虚拟存储系统

3. 存储系统的价格

➤ 计算公式:

$$C = \frac{C_1 \cdot S_1 + C_2 \cdot S_2}{S_1 + S_2}$$

➤ 当 $S_2 \gg S_1$ 时, $C \approx C_2$



(S, C, T)

由两个存储器构成的存储系统

先访问M1, 后访问M2

4. 存储系统的速度

- 表示方法：访问周期、存取周期、存储周期、存取时间等
- 命中率定义：在M1存储器中访问到的概率

$$H = \frac{N_1}{N_1 + N_2}$$

其中：N1是对M1存储器的访问次数，N2是对M2存储器的访问次数

- 访问周期与命中率的关系：

$$T = H \cdot T_1 + (1 - H) \cdot T_2$$

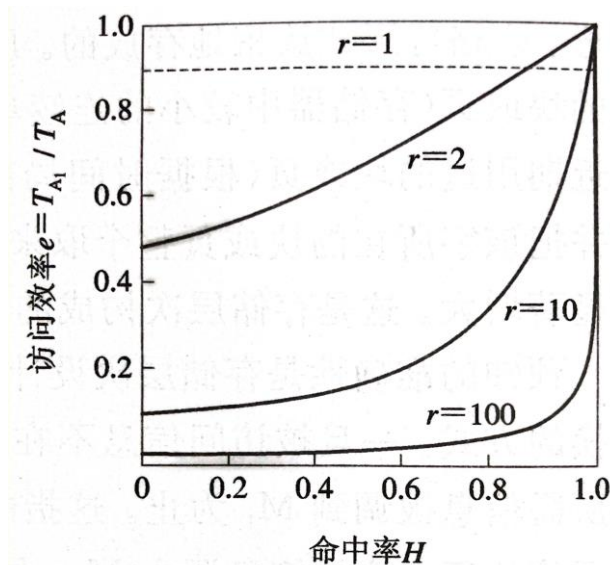
当命中率 $H \rightarrow 1$ 时， $T \rightarrow T_1$

4. 存储系统的速度

- 存储系统的访问效率：

$$e = \frac{T_1}{T} = \frac{T_1}{H \cdot T_1 + (1 - H) \cdot T_2} = \frac{1}{H + (1 - H) \cdot \frac{T_2}{T_1}} = f\left(H, \frac{T_2}{T_1}\right)$$

- 访问效率主要与命中率和两级存储器的速度之比有关



$$r = \frac{T_2}{T_1}$$

4. 存储系统的速度

- 存储系统的访问效率：

$$e = \frac{T_1}{T} = \frac{T_1}{H \cdot T_1 + (1 - H) \cdot T_2} = \frac{1}{H + (1 - H) \cdot \frac{T_2}{T_1}} = f(H, \frac{T_2}{T_1})$$

例：在虚拟存储系统中，两个存储器的速度相差特别悬殊，
例如： $T_2 = 10^5 T_1$ 。如果要使访问效率到达 $e=0.9$ ，问需要有多高的命中率？

$$0.9 = \frac{1}{H + (1 - H) \cdot 10^5}$$

$$H = 0.999999$$

4. 存储系统的速度

- 存储系统的访问效率:

$$e = \frac{T_1}{T} = \frac{T_1}{H \cdot T_1 + (1 - H) \cdot T_2} = \frac{1}{H + (1 - H) \cdot \frac{T_2}{T_1}} = f\left(H, \frac{T_2}{T_1}\right)$$

- 提高存储系统速度的两条途径:
 - 1) 提高命中率H,
 - 2) 是两个存储器的速度不要相差太大
- 第二条有时做不到(如虚拟存储器), 这时, 只能依靠提高命中率

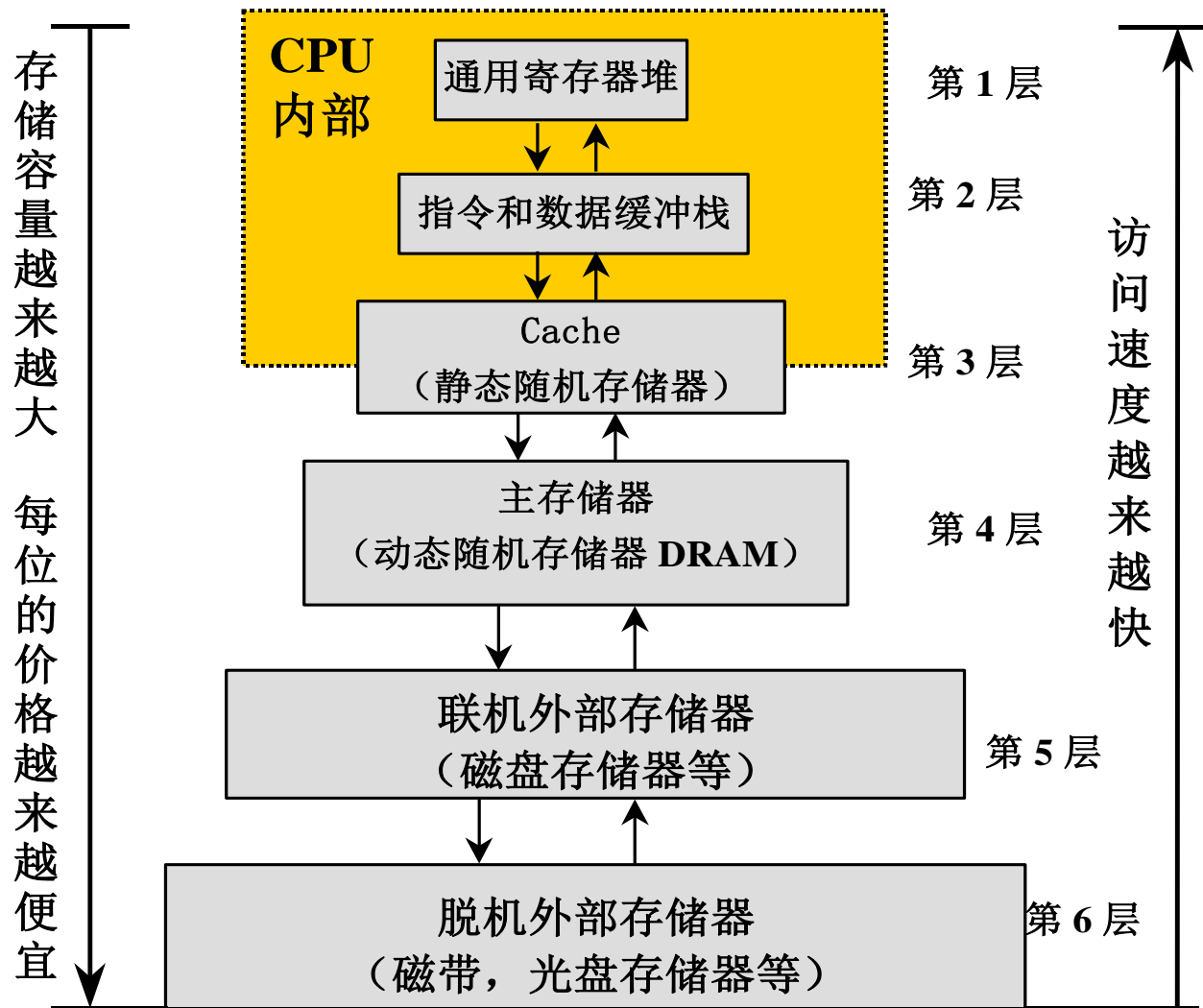
3.1.2 存储系统的层次结构

➤ 多个层次的存储器：

- 第1层：Register Files（寄存器堆）
- 第2层：Buffers（先行缓冲站）
- 第3层：Cache（高速缓冲存储器）
- 第4层：Main Memory（主存储器）
- 第5层：On-line Storage（联机存储器）
- 第6层：Off-line Storage（脱机存储器）

➤ 用 i 表示层数，则有：

- 工作周期： $T_i < T_{i+1}$
- 存储容量： $S_i < S_{i+1}$
- 单位价格： $C_i > C_{i+1}$



各级存储器的主要性能特性

存储器层次	通用寄存器	缓冲栈	Cache	主存储器	磁盘存储器	脱机存储器
存储周期	<10ns	<10ns	10~60ns	60~300ns	10~30ms	2~20min
存储容量	<512B	<512B	8KB~2MB	32MB~1GB	1GB~1TB	5GB~10TB
价格\$/KB	1200	80	3.2	0.36	0.01	0.0001
访问方式	直接译码	先进先出	相联访问	随机访问	块访问	文件组
材料工艺	ECL	ECL	SRAM	DRAM	磁表面	磁、光等
分配管理	编译器分配	硬件调度	硬件调度	操作系统	系统/用户	系统/用户
带宽(MB/S)	400~8000	400~1200	200~800	80~160	10~100	0.2~0.6

- CPU与主存储器的速度差距越来越大
- 目前相差两个数量级
- 今后CPU与主存储器的速度差距会更大

3.1.3 存储器的频带平衡

例：Pentium4的指令执行速度为8GIPS，CPU取指令8GW/s，访问数据16GW/s，各种输入输出设备访问存储器1GW/s，三项相加，要求存储器的频带宽度不低于25GW/s。如果采用PC133内存，主存与CPU速度差188倍。如果采用PC266内存，主存与CPU速度差94倍

$$\frac{25,000}{133} = 188 \quad \frac{25,000}{266} = 94$$

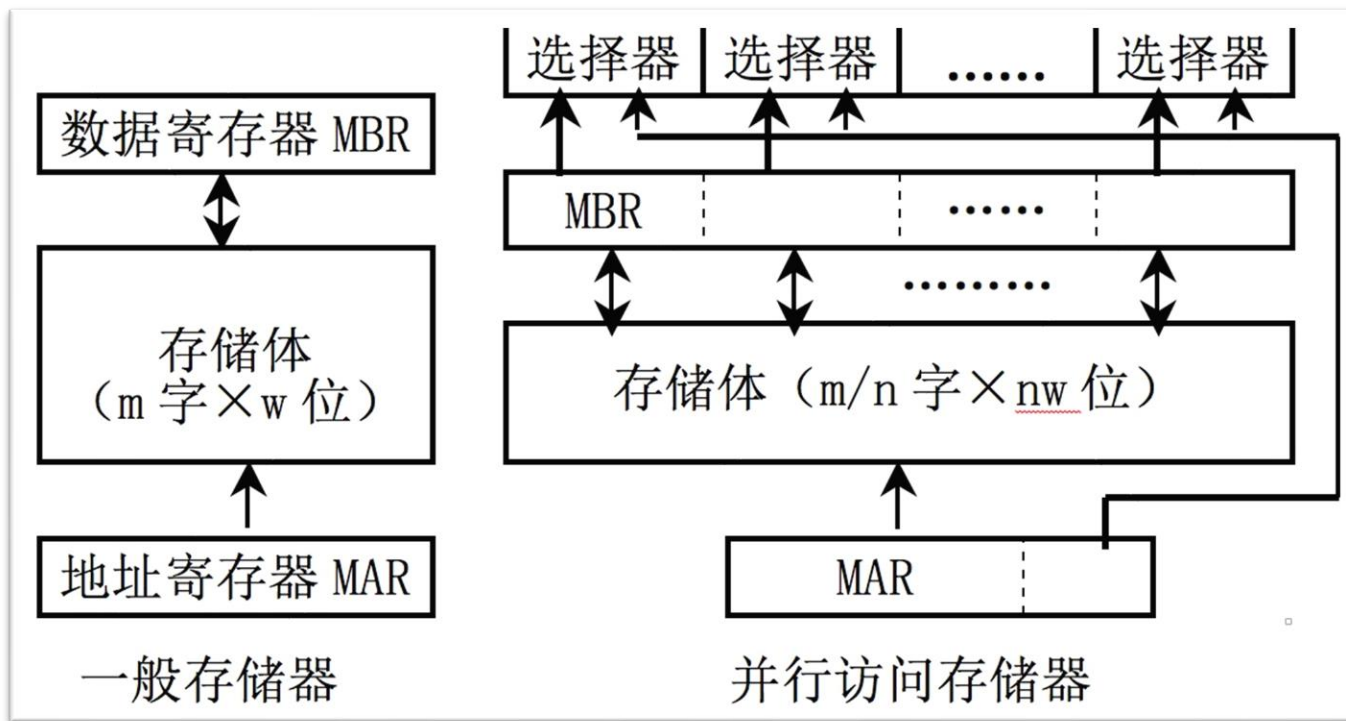
解决存储器频带平衡方法

- (1)多个存储器并行工作（本节下面介绍）
- (2)设置各种缓冲存储器（第五章介绍）
- (3)采用存储系统（本章第二、第三节介绍）

3.1.4 并行访问存储器

- 方法：把 m 字 w 位的存储器改变成为 m/n 字 $n \times w$ 位的存储器
- 逻辑实现：把地址码分成两个部分，一部分作为存储器的地址另一部分负责选择数据
- 主要缺点：访问冲突大
 - (1) 取指令冲突
 - (2) 读操作数冲突
 - (3) 写数据冲突
 - (4) 读写冲突

3.1.4 并行访问存储器



并行访问存储器结构框图

- 取指令冲突：当遇到程序转移指令且转移成功时，一个存储周期读出的 n 条指令中，后面的几条指令将无用
- 读操作数冲突：一次读出 n 个操作数，不一定都有用
- 写数据冲突：需要凑齐 n 个数据之后才能一起写入存储器；如果只写一个字，必须先把属于同一个存储字的 n 个数读到数据寄存器中，然后在地址码控制下修改其中一个字，最后把整个存储字写回存储器
- 读写冲突：当要读出的一个字和写入的一个字处在同一个存储器内时，无法在一个存储周期内完成

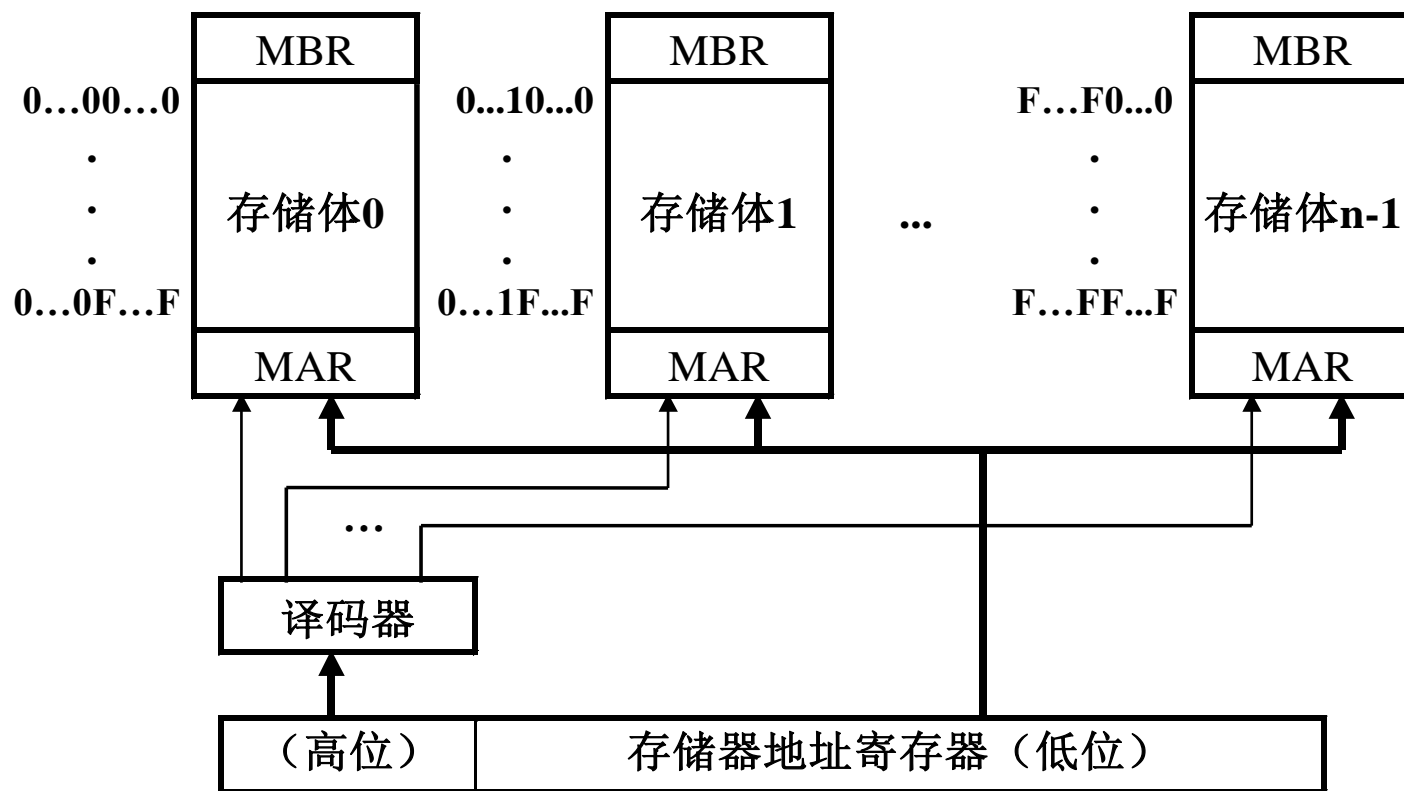
3.1.5 交叉访问存储器

1. 高位交叉访问存储器

- 主要目的：扩大存储器容量
- 实现方法：用地址码的高位部分区分存储体号
 - m : 每个存储体的容量,
 - n : 总共的存储体个数,
 - j : 存储体的体内地址, $j=0, 1, 2, \dots, m-1$
 - k : 存储体的体号, $k=0, 1, 2, \dots, n-1$
 - 存储器的地址: $A=m \times k+j$
 - 存储器的体内地址: $A_j=A \bmod m$ 。
 - 存储器的体号: $A_k = \left\lfloor \frac{A}{m} \right\rfloor$

3.1.5 交叉访问存储器

1. 高位交叉访问存储器



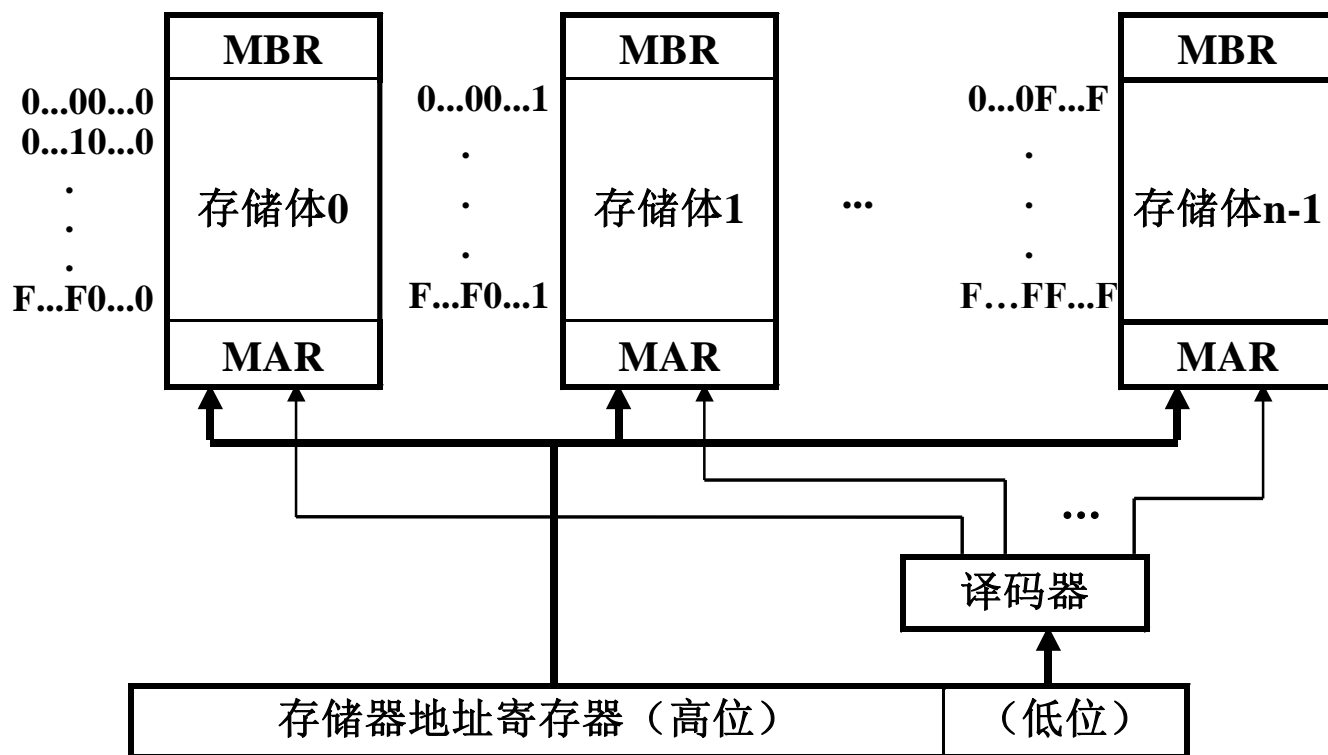
3.1.5 交叉访问存储器

2. 低位交叉访问存储器

- 主要目的：提高存储器访问速度
- 实现方法：用地址码的低位部分区分存储体号
 - m : 每个存储体的容量,
 - n : 总共的存储体个数,
 - j : 存储体的体内地址, $j=0, 1, 2, \dots, m-1$
 - k : 存储体的体号, $k=0, 1, 2, \dots, n-1$
 - 存储器地址 A 的计算公式为: $A=n \times j+k$
 - 存储器的体内地址: $A_j = \left\lfloor \frac{A}{n} \right\rfloor$
 - 存储器的体号: $A_k = A \bmod n$

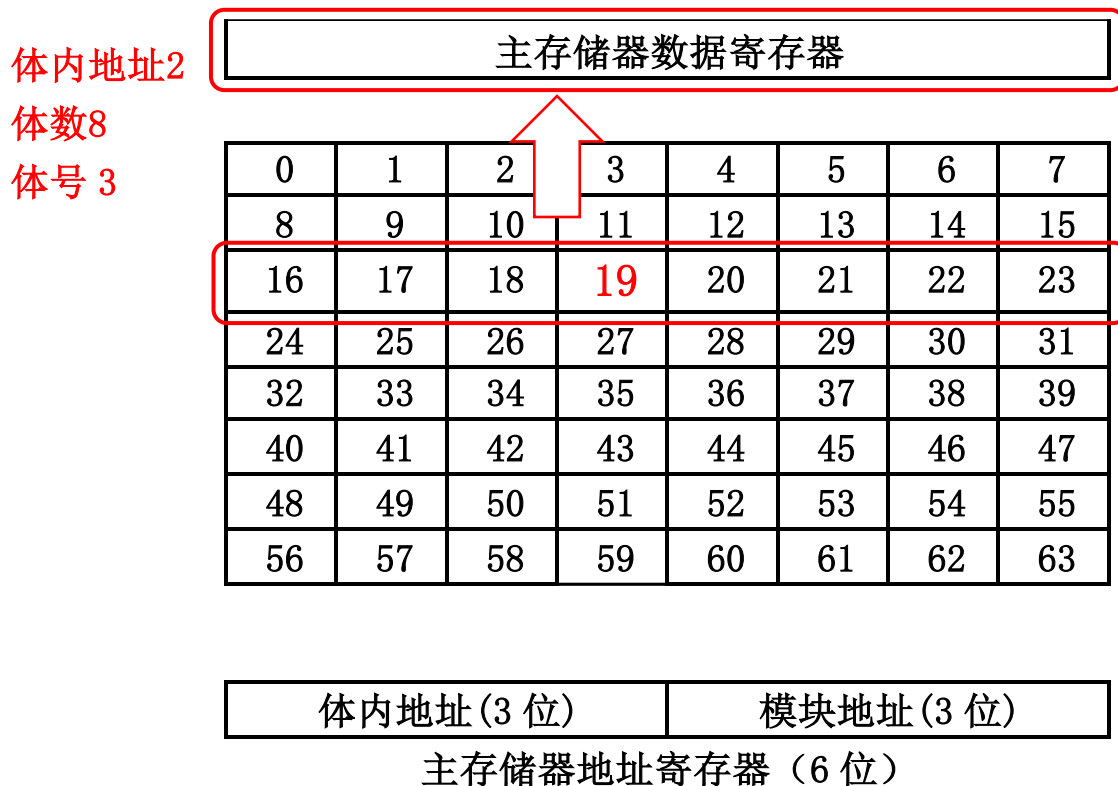
3.1.5 交叉访问存储器

2. 低位交叉访问存储器



3.1.5 交叉访问存储器

2. 低位交叉访问存储器

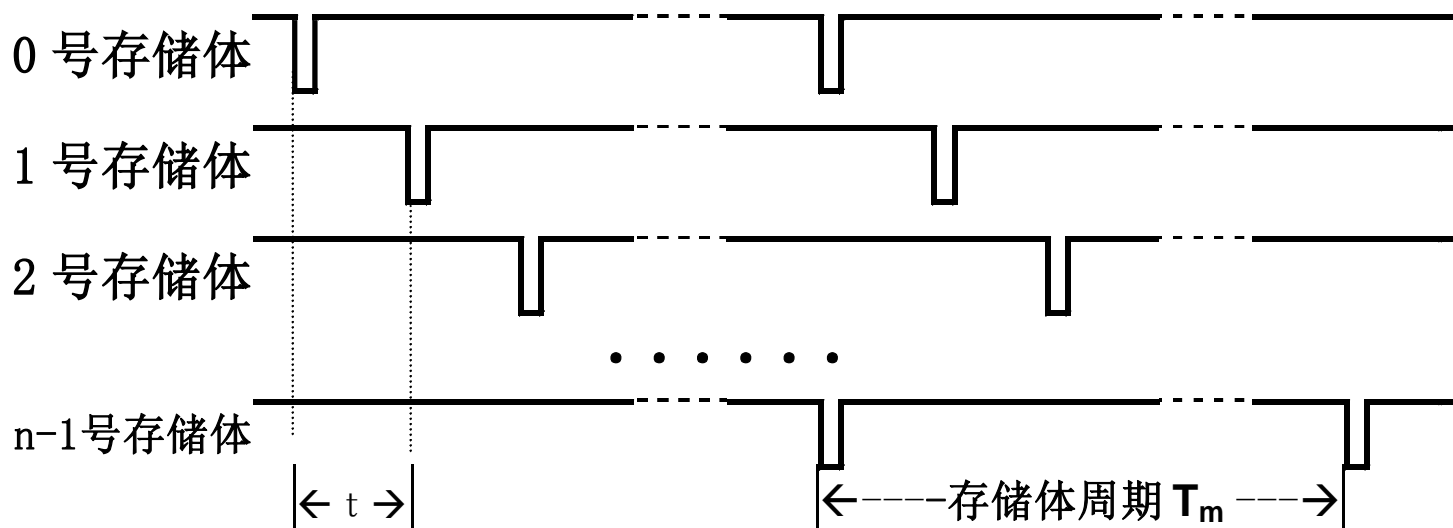


由8个存储体构成的低位交叉编址方式

3.1.5 交叉访问存储器

2. 低位交叉访问存储器

- n 个存储体分时启动
- 采用流水线方式工作的并行存储器，每存储体的启动间隔为 $t = \left\lfloor \frac{T_m}{n} \right\rfloor$ ，其中 T_m 为每个存储体的访问周期， n 为存储体个数。



3.1.5 交叉访问存储器

2. 访问冲突

- 共有 n 个存储体，每个存储周期只能取到 k 个有效字，其余 $n - k$ 个存储体有冲突。
- 假设 $p(k)$ 是 k 的概率密度函数，即 $p(1)$ 是 $k=1$ 的概率， $p(2)$ 是 $k=2$ 的概率， \dots ， $p(n)$ 是 $k=n$ 的概率。 k 的平均值为：

$$N = \sum_{k=1}^n k \cdot p(k)$$

- N 是每个存储周期能够访问到的平均有效字的个数。
- 通常把 N 称为并行存储器的加速比。

- 定义转移概率为 g ，即读出的是转移指令，且转移成功的概率。这时有：

$$p(1) = g$$

$$p(2) = (1 - p(1))g = (1 - g)g$$

$$p(3) = (1 - p(1) - p(2))g = (1 - g)^2g$$

.....

$$p(k) = (1 - g)^{k-1}g \quad (k = 1, 2, \dots, n-1)$$

.....

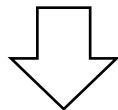
$$p(n) = (1 - g)^{n-1}$$

则:

$$N = g + 2(1-g)g + 3(1-g)^2g + \cdots + (n-2)(1-g)^{n-3}g + (n-1)(1-g)^{n-2}g + n(1-g)^{n-1}$$

$$\begin{aligned} & \nearrow n(1-g)^{n-1}(1-g) \\ & \searrow n(1-g)^{n-1}g \end{aligned}$$

$$(1-g)N = (1-g)g + 2(1-g)^2g + 3(1-g)^3g + \cdots + (n-2)(1-g)^{n-2}g + (n-1)(1-g)^{n-1}g + n(1-g)^n$$



$$N - (1-g)N =$$

$$g + (1-g)g + (1-g)^2g + (1-g)^3g + \cdots + (1-g)^{n-2}g + (1-g)^{n-1}g + 0$$

等比级数求和公式:

$$S_n = \frac{a_1 - a_n q}{1 - q}$$

$$gN = \frac{g - g(1-g)^{n-1}(1-g)}{1 - (1-g)}$$

$$N = \frac{1 - (1-g)^n}{g}$$

并行存储体个数与程序转移概率的关系

存储体个数	$g=0.01$	$g=0.1$	$g=0.2$	$g=0.3$	$g=0.4$
4	3.94	3.44	2.95	2.53	2.18
8	7.73	5.70	4.16	3.14	2.46
16	14.85	8.15	4.86	3.32	2.50
32	27.50	9.66	5.00	3.33	2.50

$$N = \frac{1 - (1 - g)^n}{g}$$

- 对于读操作数和写运算结果，随机性要比取指令大得多
- 低位交叉访问存储器的加速倍数还要比上述结论低一些

几台巨型、大型计算机的主存储器结构

机器型号	存储体个数	存储体字长	存储周期 T_m	频带宽度 B_m
IBM370/165	4	32	2,000ns	8MB/s
CDC6600	32	32	1,000ns	128MB/s
CDC7600	32	32	275ns	465MB/s
CRAY-1	16	64	50ns	2,560MB/s
ASC	8	256	160ns	1,600MB/s
STAR-100	32	512	1,280ns	1,600MB/s

3.2 虚拟存储器

3.2.1 虚拟存储器工作原理

3.2.2 地址的映象和变换方法

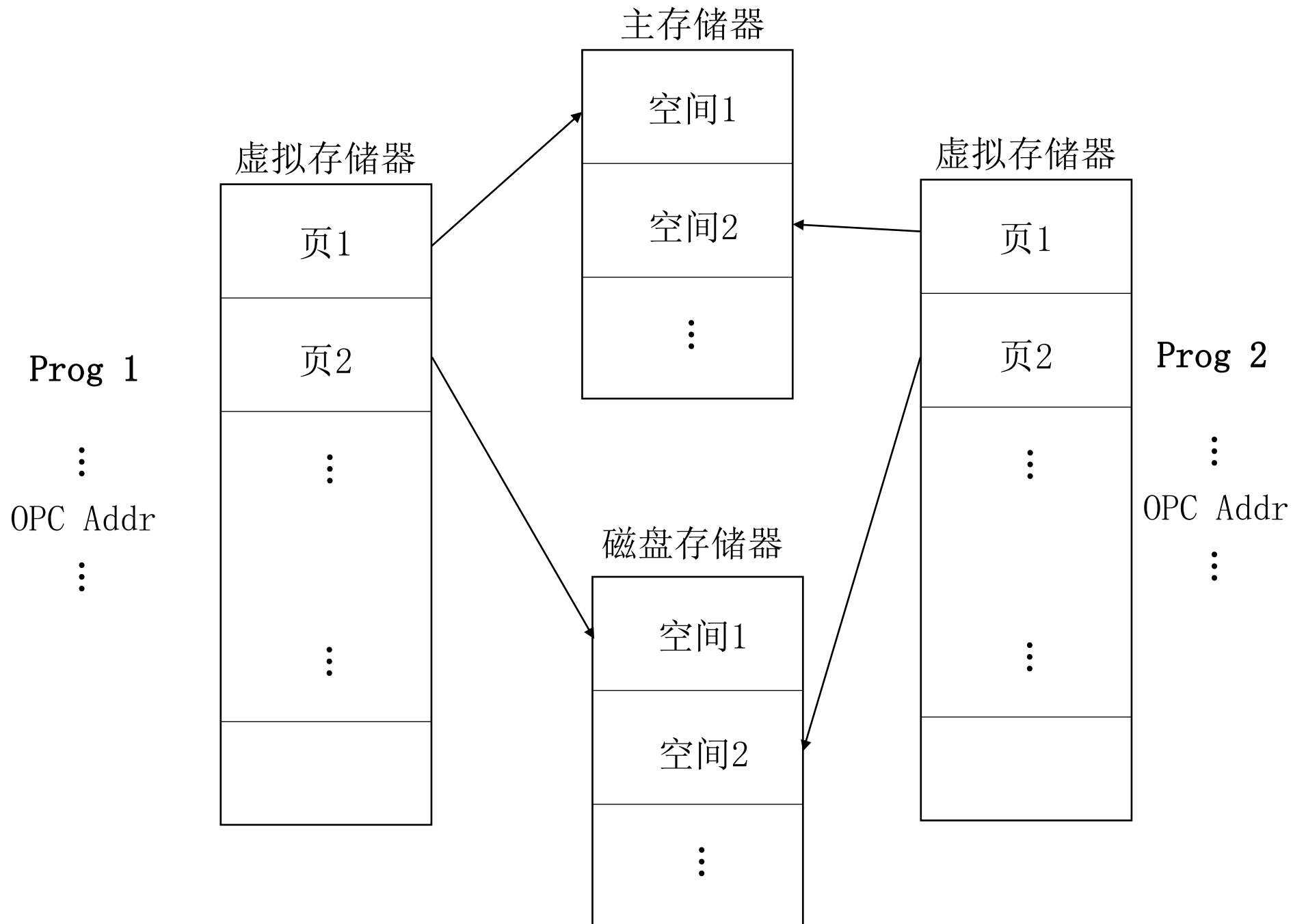
3.2.3 加快内部地址变换的方法

3.2.4 页面替换算法及其实现

3.2.5 提高主存命中率的方法

3.2.1 虚拟存储器工作原理

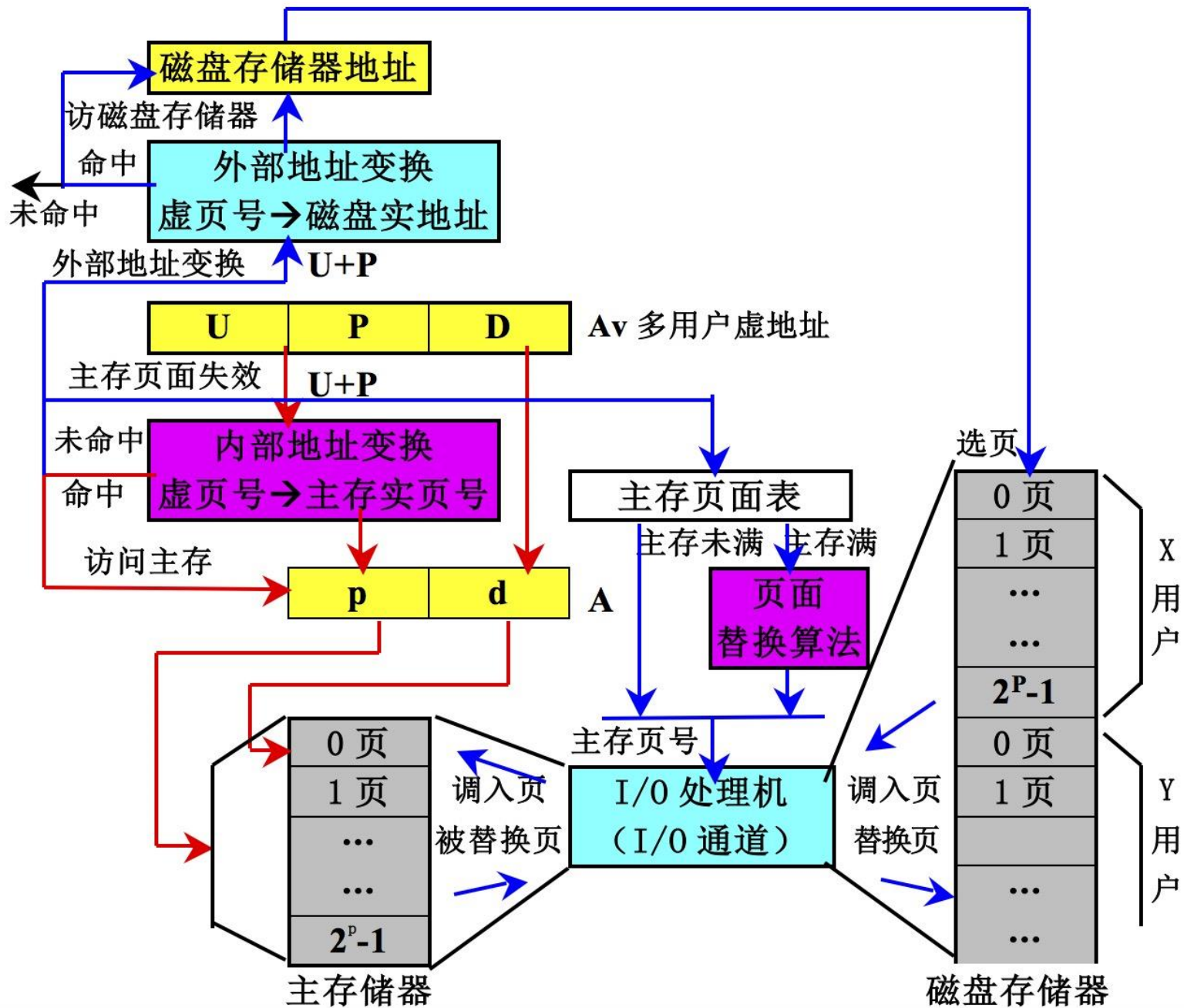
- 也称为虚拟存储系统、虚拟存储体系等
- 其概念由英国曼彻斯特大学的Kilbrn等人于1961年提出
- 到70年代广泛应用于大中型计算机系统
- 目前，许多微型机也使用虚拟存储器
- 把主存储器、磁盘存储器和虚拟存储器都划分成固定大小的页
 - 主存储器的页称为实页
 - 虚拟存储器中的页称为虚页





内部地址变换:

- 多用户虚拟地址 A_v 变换成主存实地址 A
- 多用户虚拟地址中的页内偏移 D 直接作为主存实地址中的页内偏移 d
- 主存实页号 p 与它的页内偏移 d 直接拼接起来就得到主存实地址 A 。



3.2.2 地址的映象与变换

➤ 三种地址空间:

- 虚拟地址空间
- 主存储器地址空间
- 辅存地址空间

➤ 地址映象: 把虚拟地址空间映象到主存地址空间

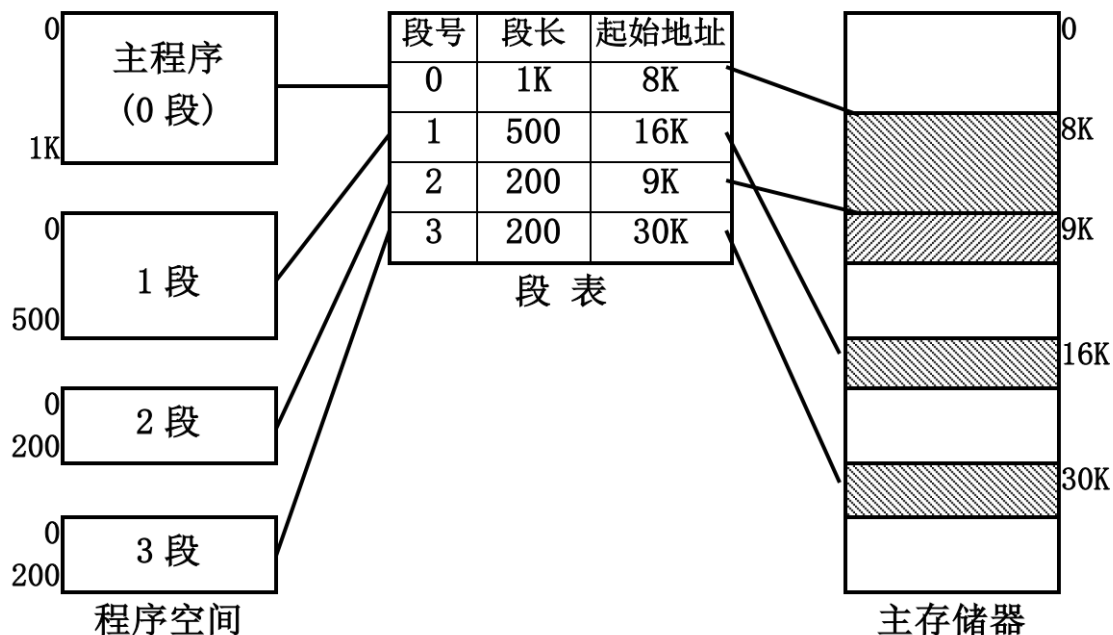
➤ 地址变换: 在程序运行时, 把虚地址变换成主存实地址

➤ 三种虚拟存储器:

- 页式虚拟存储器
- 段式虚拟存储器
- 段页式虚拟存储器

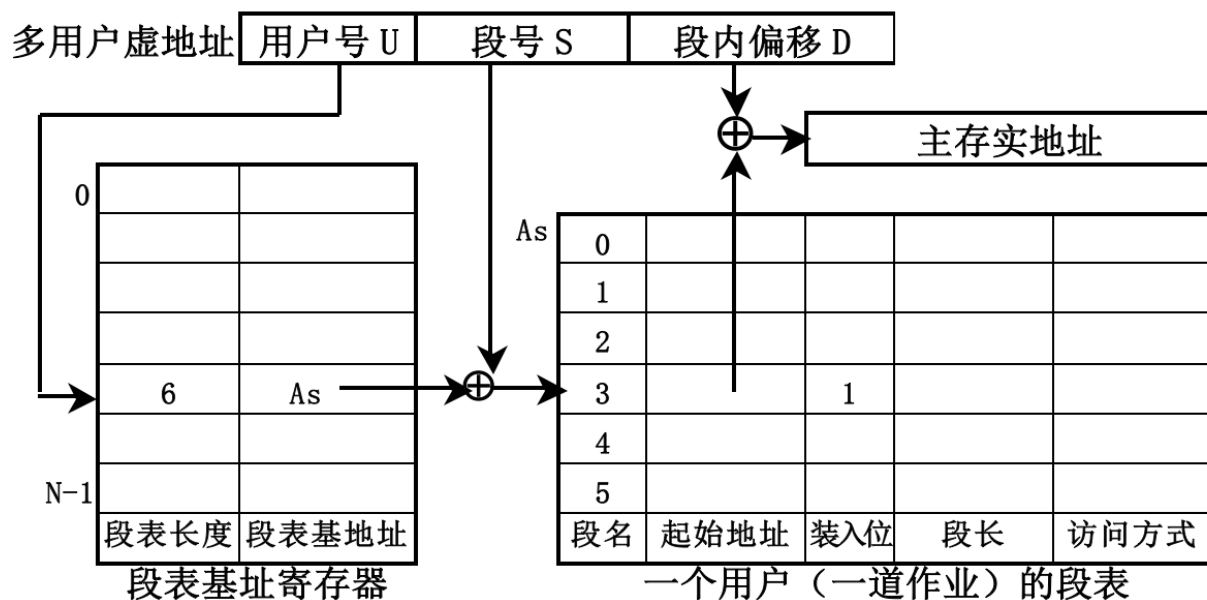
1. 段式虚拟存储器

地址映象方法：每个程序段都从0地址开始编址，长度可长可短，可以在程序执行过程中动态改变程序段的长度。



1. 段式虚拟存储器

地址变换方法：由用户号找到基址寄存器，读出段表起始地址，与虚地址中段号相加得到段表地址，把段表中的起始地址与段内偏移D相加就能得到主存实地址。



1. 段式虚拟存储器

➤ 主要优点：

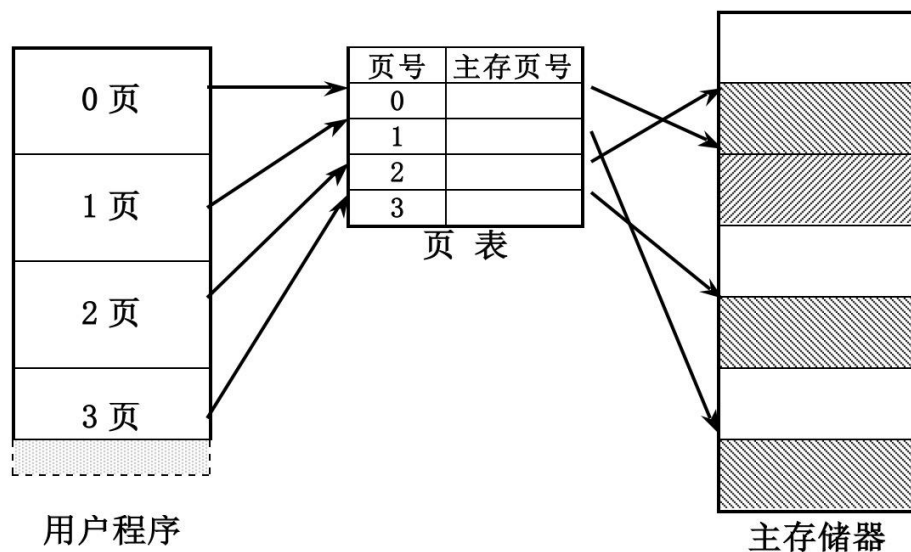
- 1) 程序的模块化性能好。
- 2) 便于程序和数据共享。
- 3) 程序的动态链接和调度比较容易。
- 4) 便于实现信息保护。

➤ 主要缺点：

- 1) 地址变换所花费的时间长，两次加法
- 2) 主存储器的利用率往往比较低。

2. 页式虚拟存储器

- **地址映像**：页式虚拟存储器把虚拟地址空间划分成一个个固定大小的块，每一块称为一页（page），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。
- 虚拟地址空间中的页称为虚页，主存地址空间的页称为实页。

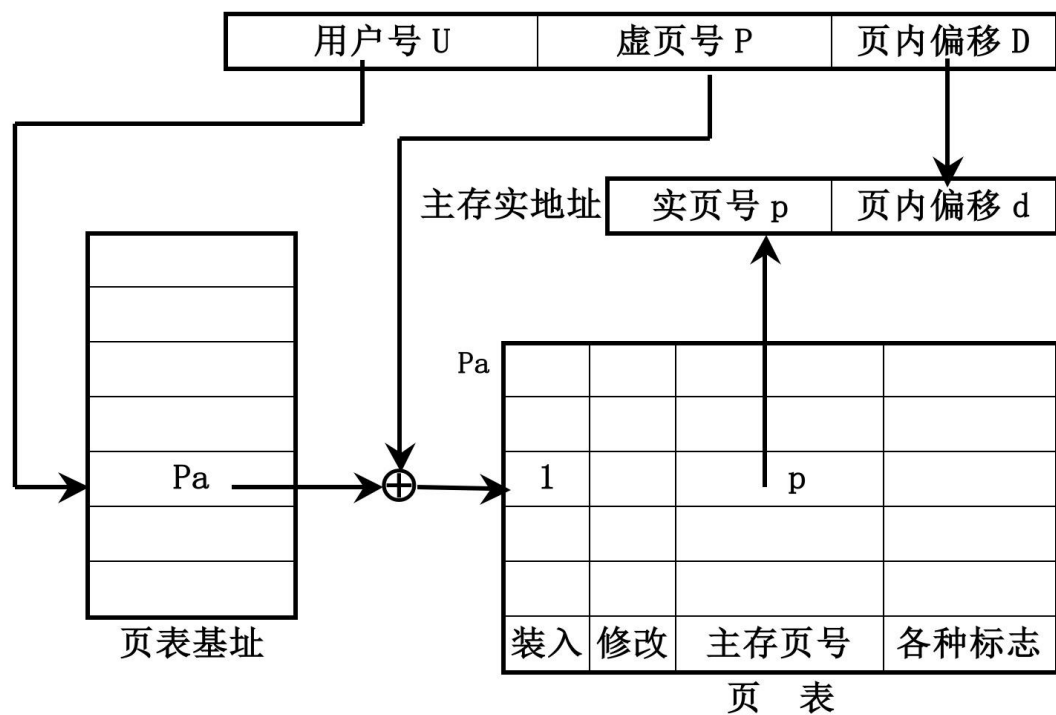


页式虚拟存储器的地址映像

全相联地址映像：用户的每一页都可以映像到主存储器的任意一页的位置

2. 页式虚拟存储器

➤ 地址变换



页式虚拟存储器的地址变换

2. 页式虚拟存储器

➤ 主要优点：

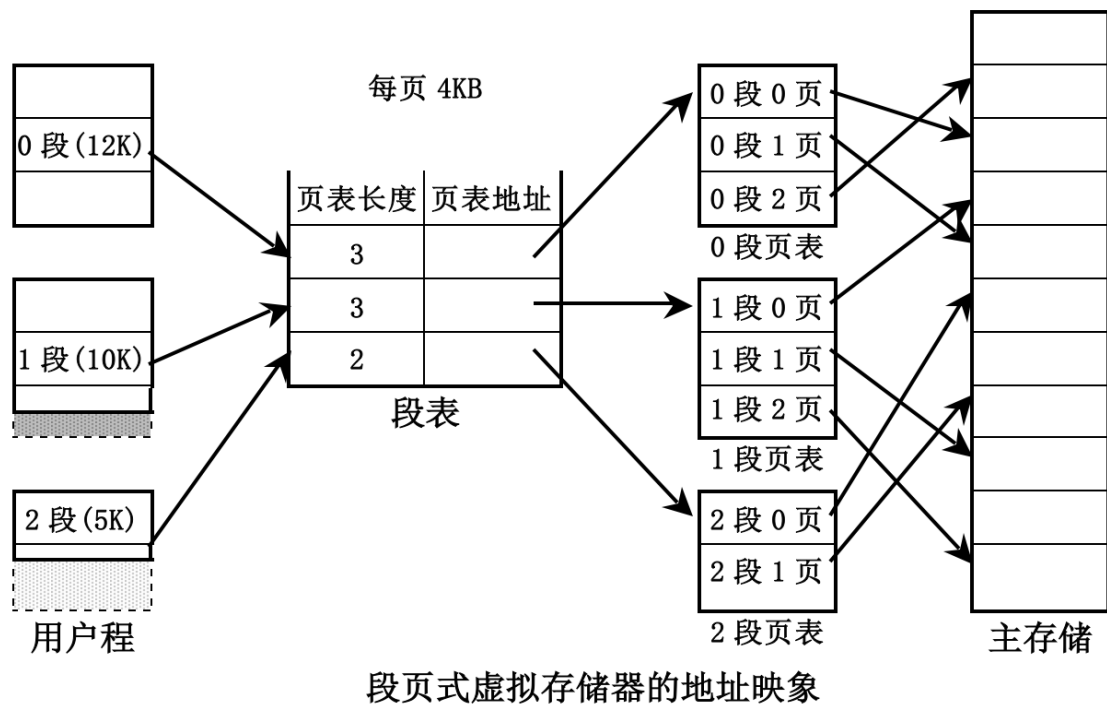
- 1) 主存储器的利用率比较高
- 2) 页表相对比较简单
- 3) 地址变换的速度比较快
- 4) 对磁盘的管理比较容易

➤ 主要缺点：

- 1) 程序的模块化性能不好
- 2) 页表很长，需要占用很大的存储空间（例如：虚拟存储空间4GB，页大小1KB，则页表的容量为 4M字，16MB）

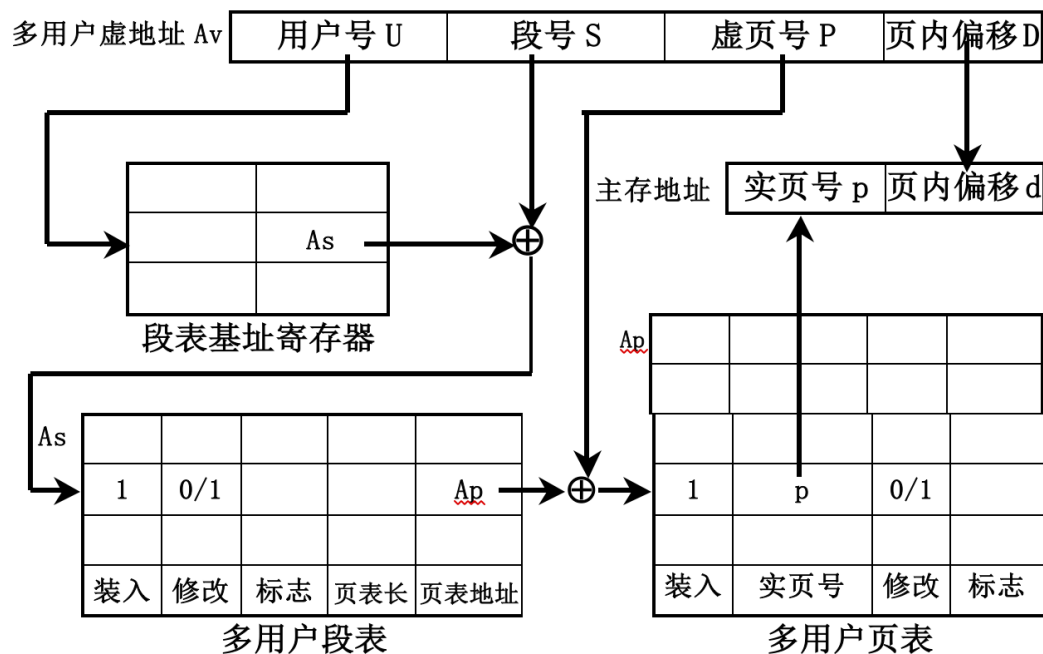
3. 段页式虚拟存储器

- 用户按段写程序, 每段分成几个固定大小的页
- 地址映象方法: 每个程序段在段表中占一行, 在段表中给出页表长度和页表的起始地址, 页表中给出每一页在主存储器中的实页号。



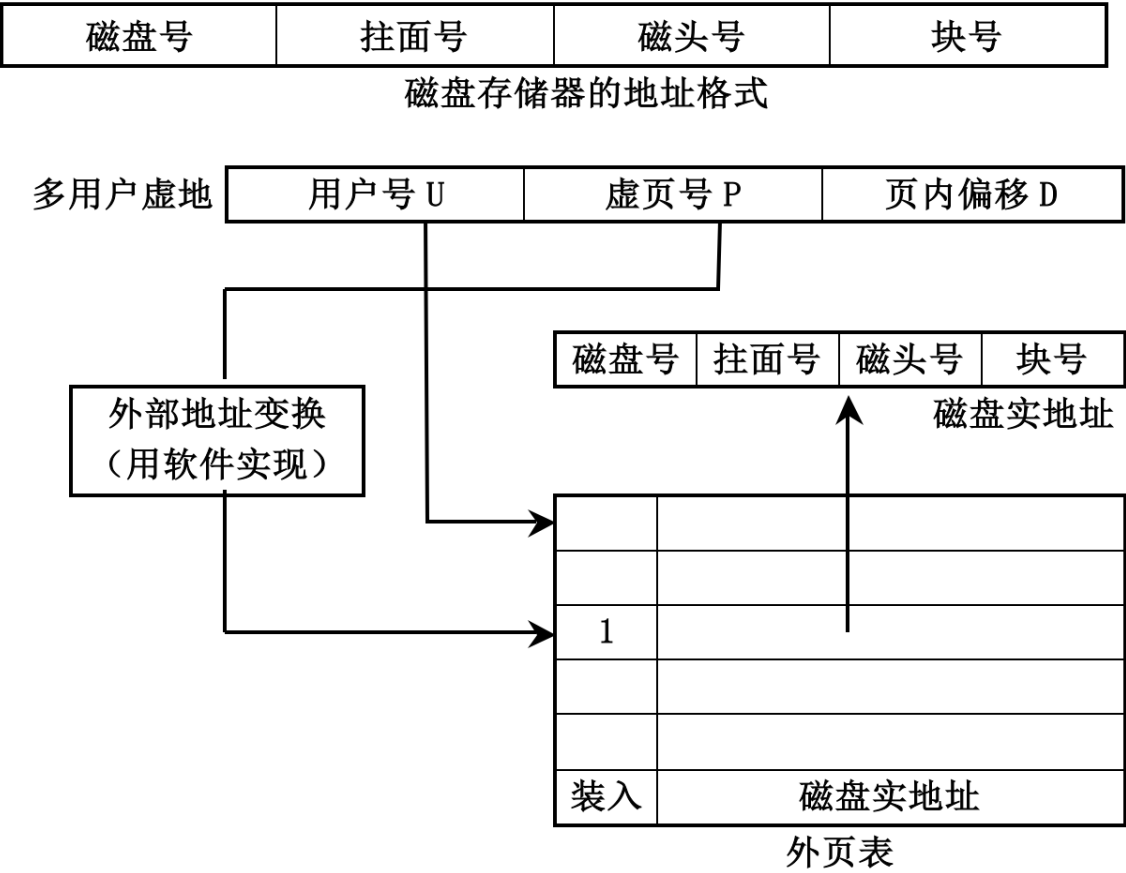
3. 段页式虚拟存储器

- 地址变换方法：先查段表，得到页表起始地址和页表长度，再查页表找到要访问的主存实页号，把实页号p与页内偏移d拼接得到主存实地址。



4. 外部地址变换

- 每个程序有一张外页表，每一页或每个程序段，在外页表中都有对应的一个存储字。



3.2.3 加快内部地址变换的方法

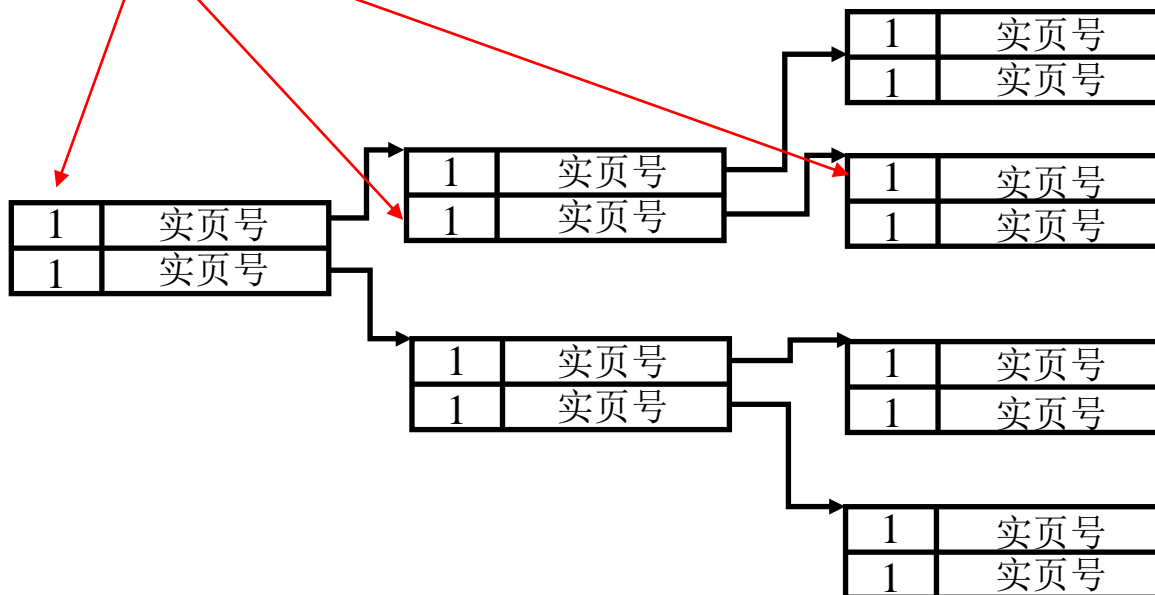
➤ 造成虚拟存储器速度降低的主要原因：

- 1) 要访问主存储器必须先查段表或页表。如果段表和页表都在主存中，则主存储器的访问速度要降低2至3倍
- 2) 当页表和段表的容量超过了一个页面的大小时，它们有可能被映像到主存不连续的页面上，使得按照地址查找主存实页号的办法无法成立。

➤ 解决方法：多级页表

- 由页表基址寄存器指出第一级页表的基地址，再用第一级页表各单元中的地址字段指出第二级页表的基地址，如此下去，构成一个树型结构的多级页表。在最后一级页表中给出主存实页号信息。

虚页号: 010



3.2.3 加快内部地址变换的方法

➤ 多级页表

- 页表级数的计算公式：

$$g = \left\lceil \frac{\log_2 N_v - \log_2 N_p}{\log_2 N_p - \log_2 N_d} \right\rceil$$

- N_v 为虚拟存储空间大小，
 N_p 为页面的大小，
 N_d 为一个页表存储字的大小

例如：虚拟存储空间大小 $N_v = 4GB$ ，页的大小 $N_p = 1KB$ ，每个页表存储字占用 $N_d = 4$ 个字节。计算得到页表的级数：

$$g = \left\lceil \frac{\log_2 4G - \log_2 1K}{\log_2 1K - \log_2 4} \right\rceil = \left\lceil \frac{32 - 10}{10 - 2} \right\rceil = 3$$

通常仅把1级页表和2、3级页表中的一小部分驻留在主存中

- 第一级页表为1页，存储容量1KB，可以有256个存储字 $256 \cdot 256 \cdot 1KB = 64MB$
- 其中的64个存储字指向第二级页表的64个页面，每个页表256个存储字，那么第二级页表有16K个存储字 $64MB \cdot 64 = 4GB$
- 类似的，第三级页表有4M个存储字，这4M个存储字可以用来存放虚拟空间的4M个页面的信息

$$N_p = 4KB?$$

3.2.3 加快内部地址变换的方法

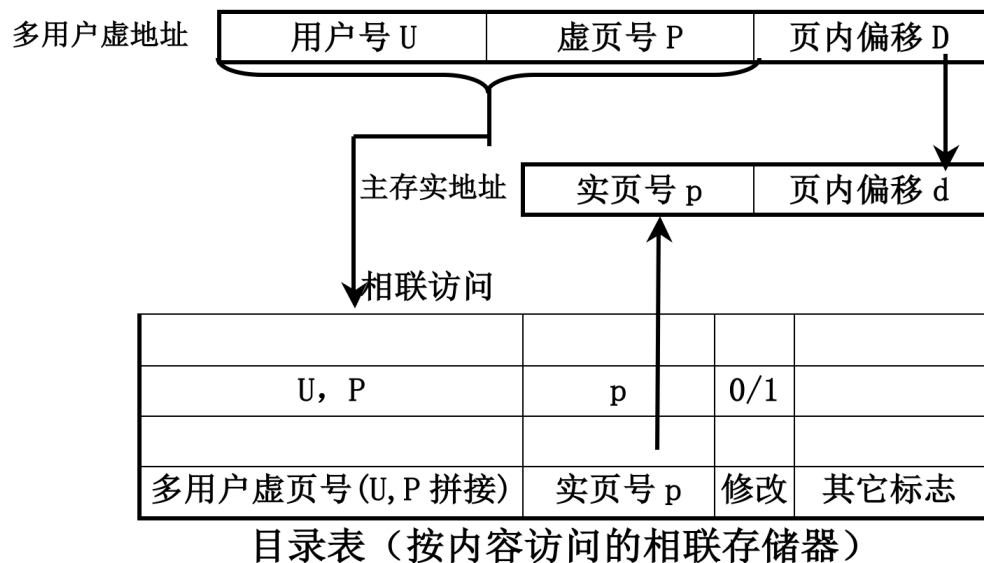
➤ 多级页表

- 一级页表必须驻留在主存储器中
- 对于二、三级页表，只需要把目前正在运行中的程序的相关页表，或者把已经调入到主存中的程序的相关页表驻留在主存中，绝大部分页表可以放在磁盘存储器中
- 当一个用户程序被调入到主存中时，在把相关页表也同时装入到主存中，并且修改页表中的装入位和主存地址等字段
- 然而，采用多级页表（包括段表）使得访问主存的次数又要增加，需要加快查表速度。

3.2.3 加快内部地址变换的方法

1 目录表法

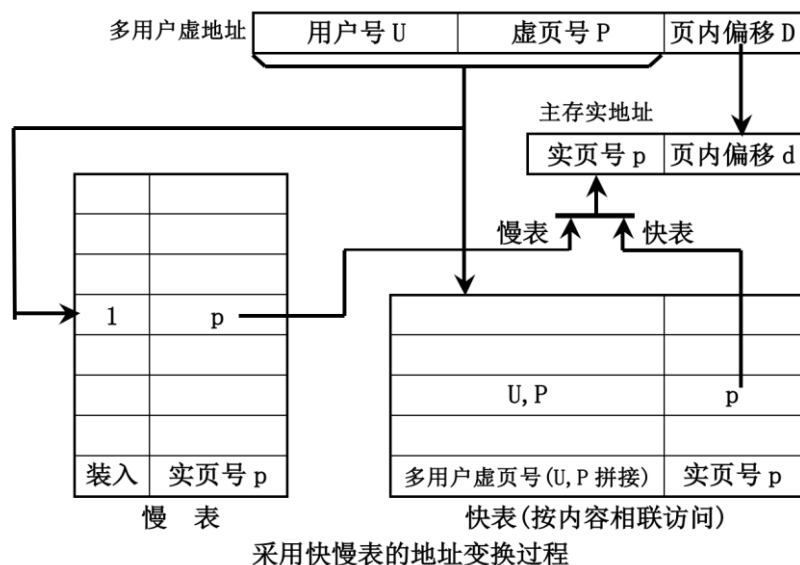
- 压缩页表的存储容量，用一个容量比较小的高速存储器来存放页表，从而加快页表的查表速度
- 页表只为已经装入到主存中的那些页面建立虚页号与实页号之间的对应关系
 - 随着主存容量的增加，目录表的容量也必须增加，所以可扩展性较差。当主存容量增加到一定数量，目录表的造价就会很高，查表速度也会降低



3.2.3 加快内部地址变换的方法

2 快慢表

- 由于程序访问的时间局部性，在一段时间内，对页表的访问只是局限在少数几个存储字内。因此，可以大大缩小目录表的存储容量（8-16个存储字），加快访问速度（与CPU中通用寄存器速度相当）
- 用快表（Translation Lookaside Buffer, TLB）保存最近访问的虚页到实页的映射

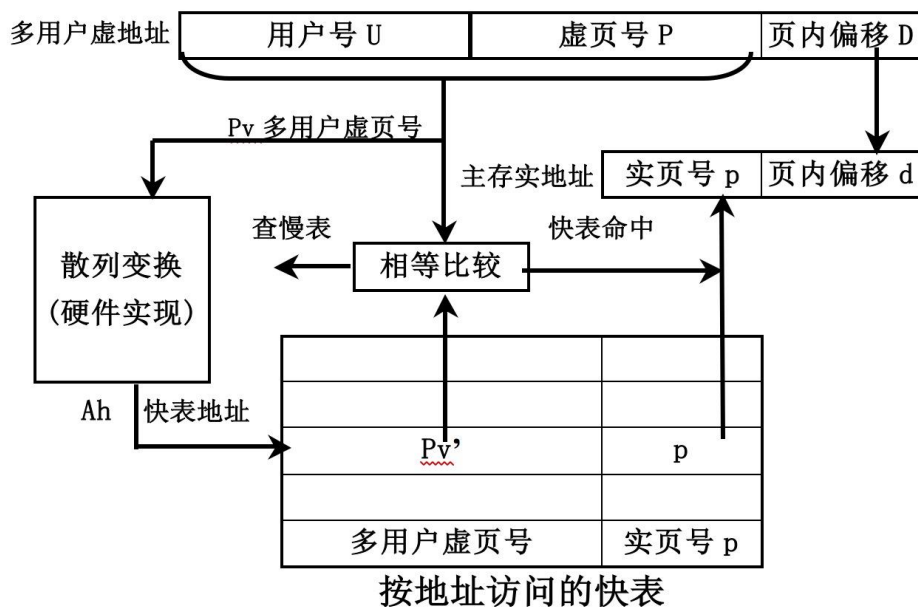


- ❑ 快表：高速硬件实现，采用相联方式访问。
- ❑ 慢表：当快表中查不到时，从主存的慢表中查找；慢表按地址访问；用软件实现。
- ❑ 快表与慢表也构成一个两级存储系统。
- ❑ 主要存在问题：相联访问实现困难，速度低

3.2.3 加快内部地址变换的方法

3 散列函数

- 把按内容的相联访问变为按地址访问
- 散列函数: $Ah = H(Pv)$

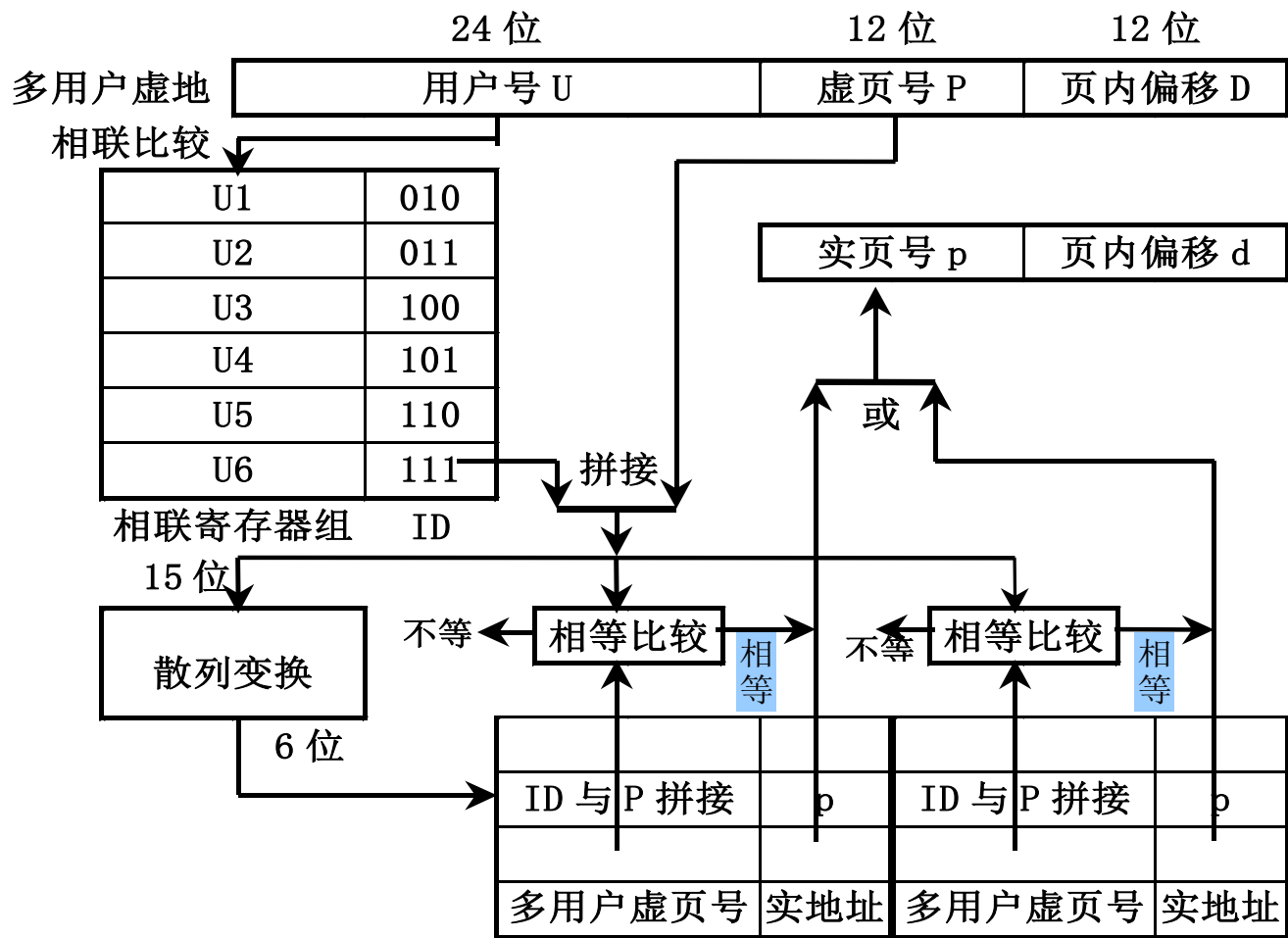


- 采用散列变换实现快表按地址访问
- 避免散列冲突：采用相等比较器
- 地址变换：相等比较与访问存储器同时进行

3.2.3 加快内部地址变换的方法

4 虚拟存储器举例

- IMB370/168计算机的虚拟存储器中的快表结构及地址变换过程。虚拟地址长48位，页面大小为4KB，每个用户最多占用4K 个页面，最多允许16M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：
 - 1) 采用两个相等比较器，以减少散列冲突
 - 2) 用一个相联寄存器组，把24位多户号压缩成3位，以缩短快表长度



快表（按地址访问，有两组）

IBM370/168 计算机的虚拟存储器快表结构

3.2.4 页面替换算法及其实现

- **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。
如果主存储器的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。
- **评价页面替换算法好坏的标准：**一是命中率要高，二是算法要容易实现。
- **页面替换算法的使用场合：**
 - 1) 虚拟存储器中，主存页面的替换，一般用软件实现。
 - 2) Cache中的块替换，一般用硬件实现。
 - 3) 虚拟存储器的快慢表中，快表存储字的替换，用硬件实现。
 - 4) 虚拟存储器中，用户基地址寄存器的替换，用硬件实现。
 - 5) 在有些虚拟存储器中，目录表的替换。

3.2.4 页面替换算法及其实现

➤ 主要页面替换算法

- 1) 随机算法 (Random, RAND)：采用软件或者硬件随机数产生器产生主存中要被替换的页号。
 - 算法简单，容易实现；没有利用历史信息，没有反映程序的局部性，命中率低
- 2) 最优替换算法 (Optimal Replacement Algorithm, OPT)：选择将来最久没有被访问的页作为被替换页面
 - 是一种理想算法，仅用作评价其它页面替换算法好坏的标准。

3.2.4 页面替换算法及其实现

➤ 主要页面替换算法

- 3) 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
 - 能够比较正确反映程序的局部性
 - 把最久未被访问的页作为被替换页
- 4) 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
 - 容易实现，利用了历史信息；没有反映局部性，最先调入的页面，很可能也是要使用的页面

- **Least Frequently Used (LFU)**

- Counts how often an item is needed. Those that are used least often are discarded first. We store the value of how many times it was accessed. While running an access sequence, we will replace a block which was used least number of times from our cache. E.g., if A was used (accessed) 5 times and B was used 3 times and others C and D were used 10 times each, we will replace B.

- **Least Recently Used (LRU)**


- Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes.

一个程序共有5个页面组成，在程序执行过程中，页面地址流如下：





P1, P2, P1, P5, P4, P1, P3, P4, P2, P4

假设分配给这个程序的主存只有3个页面。

- 1) 给出用FIFO、LRU和OPT三种页面替换算法对这3个主存页面的调度情况表，并统计页面命中次数。
- 2) 计算这LRU页面替换算法的页面命中率。

时间 t	1	2	3	4	5	6	7	8	9	10	实际命中次数
页地址流	P1	P2	P1	P5	P4	P1	P3	P4	P2	P4	
先进先出算法 (FIFO 算法)	1	1	1	1*	4	4	4*	4*	2	2	2 次
		2	2	2	2*	1	1	1	1*	4	
				5	5	5*	3	3	3	3*	
	调入	调入	命中	调入	替换	替换	替换	命中	替换	替换	
最久没有使用算法 (LRU 算法)	1	1	1	1	1*	1	1	1*	2	2	4 次
		2	2	2*	4	4	4*	4	4	4	
				5	5	5*	3	3	3*	3*	
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中	
最优替换算法 (OPT 算法)	1	1	1	1	1	1*	3*	3*	3	3	5 次
		2	2	2	2*	2	2	2	2	2	
				5*	4	4	4	4	4	4	
	调入	调入	命中	调入	替换	命中	替换	命中	命中	命中	

三种页面替换算法对同一个页地址流的调度过程

时间 t	1	2	3	4	5	6	7	8	9	10	实际命中次数
页地址流	P1	P2	P1	P5	P4	P1	P3	P4	P2	P4	
先进先出算法 (FIFO 算法)	1	1	1	1*	4	4	4*	4*	2	2	2 次
		2	2	2	2*	1	1	1	1*	4	
				5	5	5*	3	3	3	3*	
	调入	调入	命中	调入	替换	替换	替换	命中	替换	替换	
最久没有使用算法 (LRU 算法)	1	1	1	1	1*	1	1	1*	2	2	4 次
		2	2	2*	4	4	4*	4	4	4	
				5	5	5*	3	3	3*	3*	
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中	
最优替换算法 (OPT 算法)	1	1	1	1	1	1*	3*	3*	3	3	5 次
		2	2	2	2*	2	2	2	2	2	
				5*	4	4	4	4	4	4	
	调入	调入	命中	调入	替换	命中	替换	命中	命中	命中	

三种页面替换算法对同一个页地址流的调度过程

- 1) FIFO、LRU和OPT的页面命中次数分别为2次、4次和5次
- 2) LRU页面替换算法的页面命中率为： $H_p = 4/10 = 0.4$

一个循环程序，依次使用P1，P2，P3，P4页面，分配给它的主存页面数只有3个。在FIFO和LRU算法中，发生“**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	

3.2.4 页面替换算法及其实现

➤ 堆栈型替换算法

- 定义：对任意一个程序的页地址流作两次主存页面数分配，分别分配 m 个主存页面和 n 个主存页面，并且有 $m \leq n$ 。如果在任何时刻 t ，主存页面集合 B_t 都满足关系： $B_t(m) \subseteq B_t(n)$ ，则这类算法称为堆栈型替换算法。
- 堆栈型算法的基本特点是：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算法是堆栈型替换算法

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	P1	P2	P3	P4	P1	P2	P5	P1	P2	P3	P4	P5	
主存页面数 N=3	1	1	1*	4	4	4*	5	5	5	5	5*	5*	3 次
	▨	2	2	2*	1	1	1*	1*	1*	3	3	3	
	▨	▨	3	3	3*	2	2	2	2	2*	4	4	
	调入	调入	调入	替换	替换	替换	替换	命中	命中	替换	替换	命中	
主存页面数 N=4	1	1	1	1	1*	1*	5	5	5	5*	4	4	2 次
	▨	2	2	2	2	2*	2*	1	1	1	1*	5	
	▨	▨	3	3	3	3	3	3*	2	2	2	2*	
	▨	▨	▨	4	4	4	4	4	4*	3	3	3	
	调入	调入	调入	调入	命中	命中	替换	替换	替换	替换	替换	替换	

FIFO 算法在主存页面数增加时命中率反而下降

3. 2. 4 页面替换算法及其实现

➤ 页面替换算法的实现

- 虚拟存储器中的页面替换算法一般用软件实现

实页号	占用位	程序号	段页号	使用位 (计数器)	程序优 先级	历史位Hb (未使用计数器)	其他
0							
1							
2							
P-1							

主存页面表（P为主存页面数）

3.2.4 页面替换算法及其实现

➤ 页面替换算法的实现

- FIFO：将主存页面表中的“使用位”字段设置成一个计数器。每当有一个页面装入主存储器时，让该页面的计数器清零，而其他已经装入主存储器的页面所对应的计数器加1。需要替换时，计数器的值最大的页面就是最先进入主存的页面。

实页号	占用位	程序号	段页号	使用位 (计数器)	程序优 先级	历史位Hb (未使用计数器)	其他
0							
1							
2							
P-1							

3.2.4 页面替换算法及其实现

➤ 页面替换算法的实现

- LRU：一般情况下，只需要一个使用位。所有页面的初始使用位均为“0”。当页面被访问后，使用位设置为“1”。进行替换时，找出使用位为“0”的页面作为被替换页面。
- 不允许所有页面的使用位全部为1
 - ① 当所有页面使用位都为1时，把所有页面的使用位清“0”
 - ② 每隔一个固定的时间，把所有页面的使用位清“0”
 - ③ 在主存页面表设置一个历史位Hb。每隔一段时间扫描所有页面的使用位，如果使用位为“0”，则把对应的Hb加“1”；否则，将Hb清“0”。同时，将使用位清“0”。因此，Hb越大，说明对应的页面越久没有被访问过，应该成为最先被替换掉的页面。

实页号	占用位	程序号	段页号	使用位 (计数器)	程序优 先级	历史位Hb (未使用计数器)	其他
-----	-----	-----	-----	--------------	-----------	-------------------	----

3.2.5 提高主存命中率的方法

➤ 影响主存命中率的主要因素：

- 1) 程序在执行过程中的页地址流分布情况。
- 2) 所采用的页面替换算法。
- 3) 页面大小。
- 4) 主存储器的容量
- 5) 所采用的页面调度算法

3.2.5 提高主存命中率的方法

➤ 页面大小与命中率的关系：页面大小为某个值时，命中率达到最大。

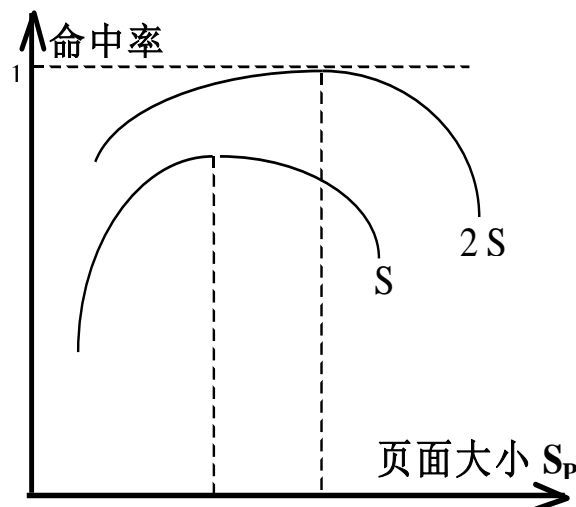
- 假设 A_t 和 A_{t+1} 是相邻两次访问主存的逻辑地址，

$$d = |A_t - A_{t+1}|$$

- 如果 $d < S_p$ ，随着 S_p 增大， A_t 和 A_{t+1} 在同一页面的可能性增加，即命中率 H 随着 S_p 的增大而提高。
- 如果 $d > S_p$ ， A_t 和 A_{t+1} 一定不在同一个页面内。随着 S_p 增大，主存页面数减少，页面替换更加频繁。 H 随着 S_p 的增大而降低。

3.2.5 提高主存命中率的方法

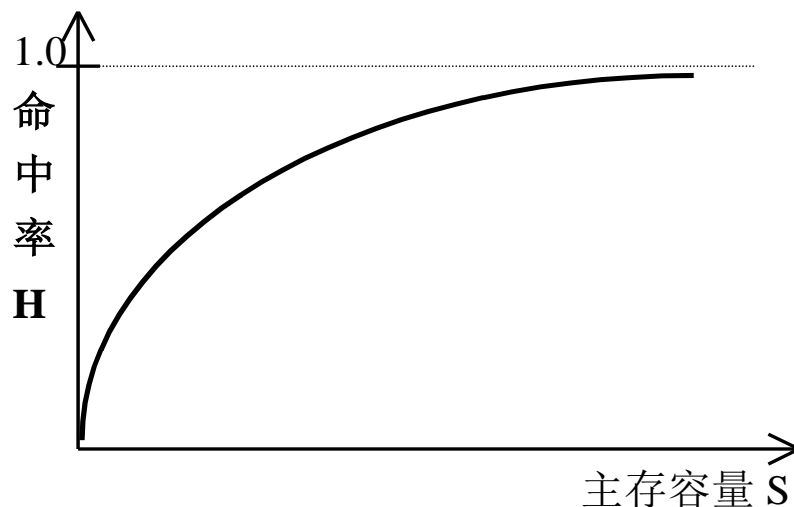
- 页面大小与命中率的关系：页面大小为某个值时，命中率达到最大。
- 当 S_p 比较小的时候，前一种情况是主要的， H 随着 S_p 的增大而提高
 - 当 S_p 达到某一个最大值之后，后一种情况成为主要的， H 随着 S_p 的增大而降低。
 - 当页面增大时，造成的浪费也要增加
 - 当页面减小时，页表和页面表在主存储器中所占的比例将增加



3.2.5 提高主存命中率的方法

➤ 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 的逐渐增加， 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。



3.2.5 提高主存命中率的方法

➤ 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。其主要优点是可以避免在程序开始运行时，频繁发生页面失效的情况；其主要缺点：如果调入的页面用不上，浪费了调入的时间，占用了主存的资源。

3.3 高速缓冲存储器

3.3.1 基本工作原理

3.3.2 地址映象与变换方法

3.3.3 Cache替换算法及其实现

3.3.4 Cache存储系统的加速比

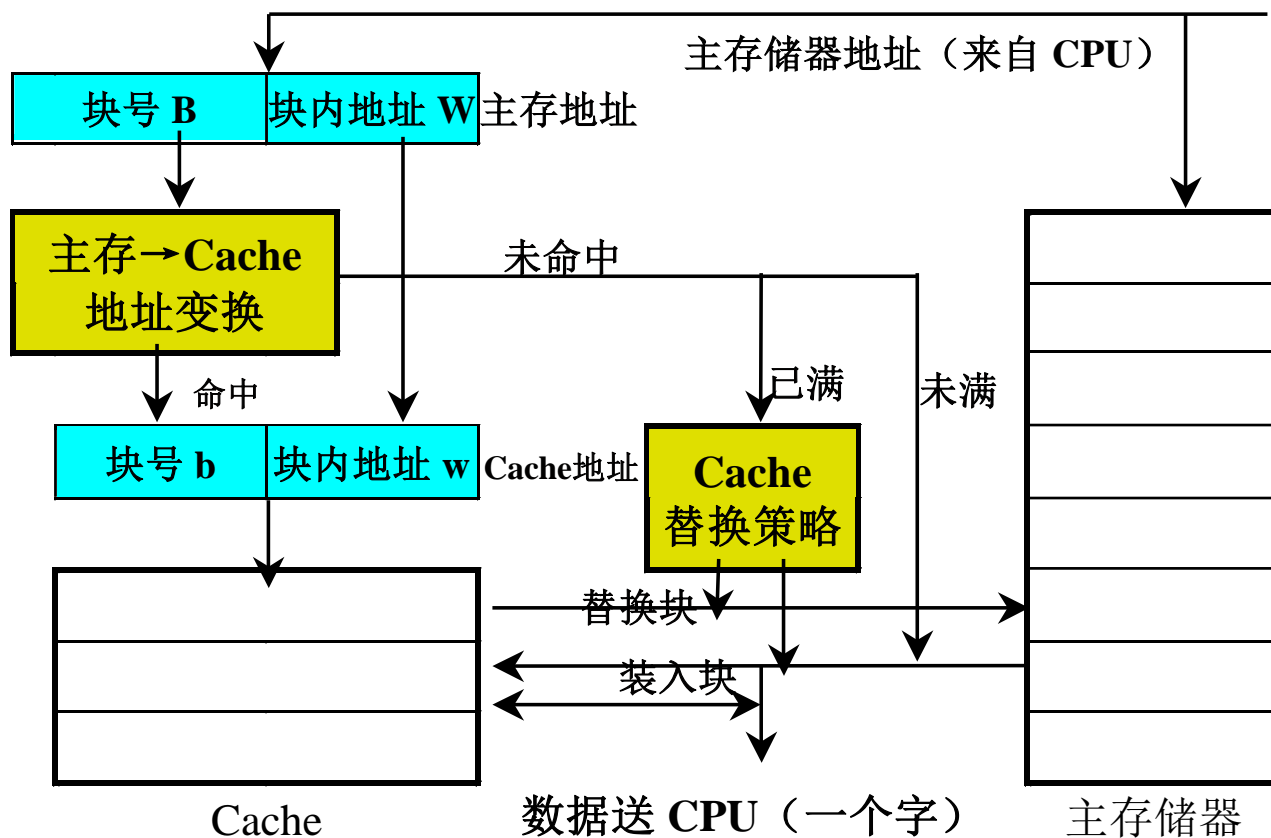
3.3.5 Cache的一致性问题

3.3.6 Cache的预取算法

Cache 存储系统与虚拟存储系统的比较

存储系统	Cache 存储系统	虚拟存储系统
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	主存储器	虚拟存储器
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

3.3.1 基本工作原理



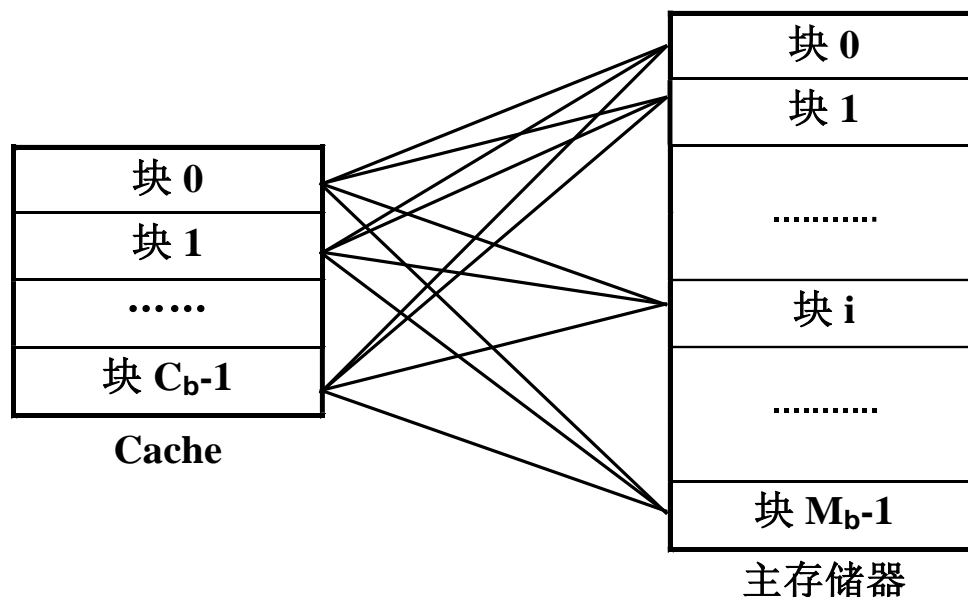
3.3.2 地址映象与变换方法

- **地址映象**：把主存中的程序按照某种规则装入到Cache中，并建立主存地址与Cache地址之间的对应关系。
- **地址变换**：当程序已经装入到Cache之后，在程序运行过程中，把主存地址变换成Cache地址。
- 在选取地址映象方法要考虑的主要因素：
 - 地址变换的硬件实现容易、速度要快
 - Cache空间利用率要高
 - 发生块冲突的概率要小

3.3.2 地址映象与变换方法

1. 全相联映象及其变换

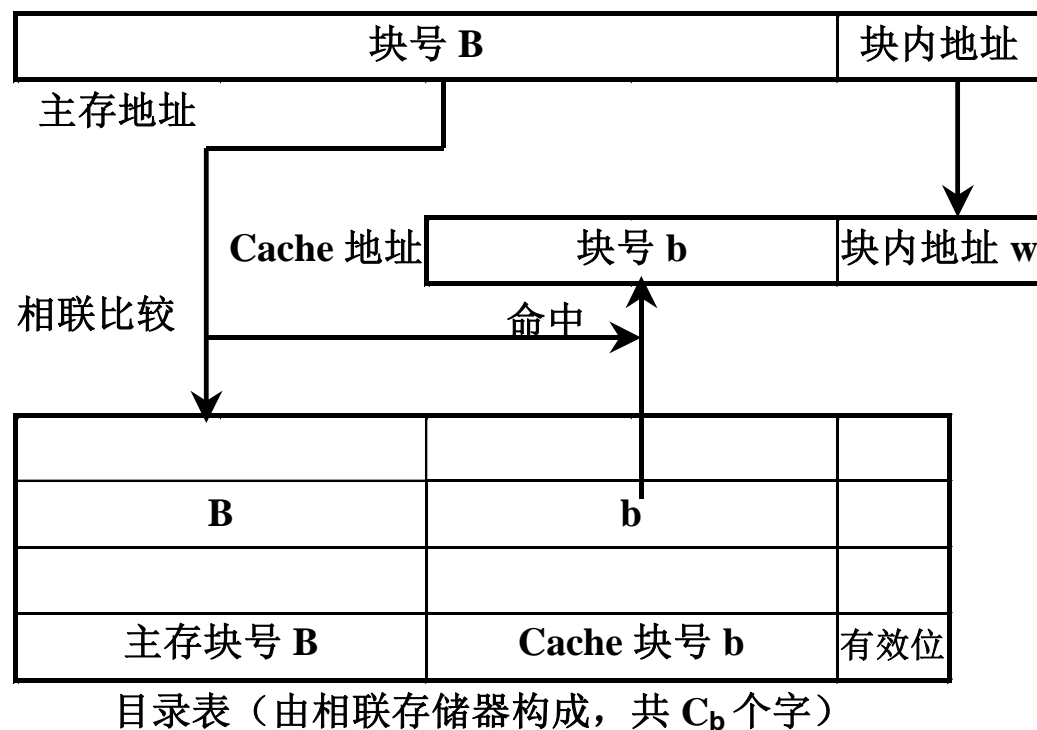
- **映象规则：**主存的任意一块可以映象到Cache中的任意一块。（映象关系有 $C_b \times M_b$ 种）



3.3.2 地址映象与变换方法

1. 全相联映象及其变换

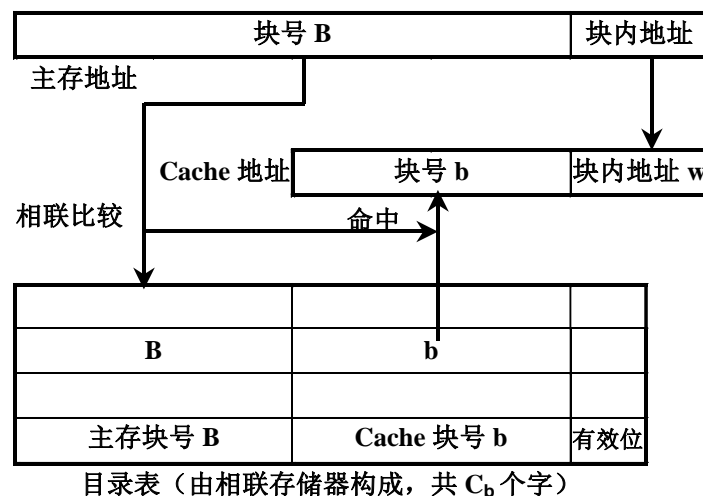
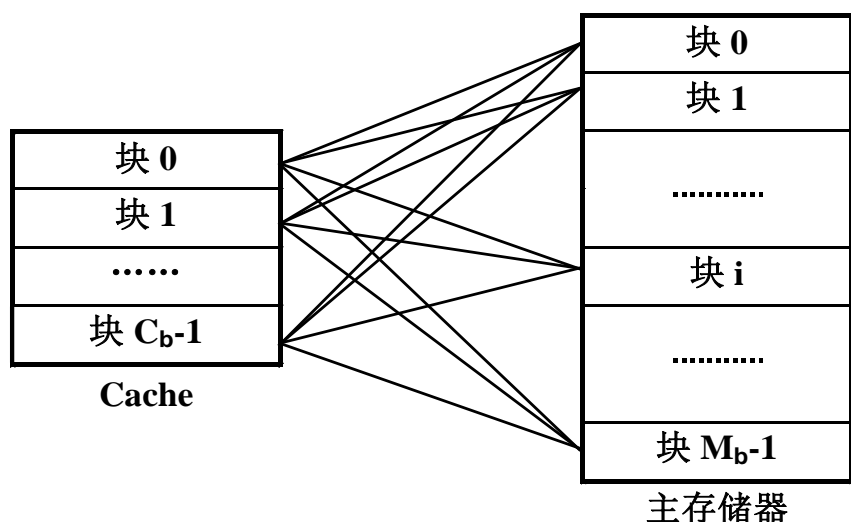
➤ 地址变换



3.3.2 地址映象与变换方法

1. 全相联映象及其变换

- 全相联映像的优点是块冲突概率低，空间利用率高
- 相联存储器成本高，查表速度慢



3.3.2 地址映象与变换方法

2. 直接映象及其变换

- 映象规则：主存储器中一块只能映象到Cache的一个特定的块中。
- Cache地址的计算公式：

$$b = B \bmod C_b$$

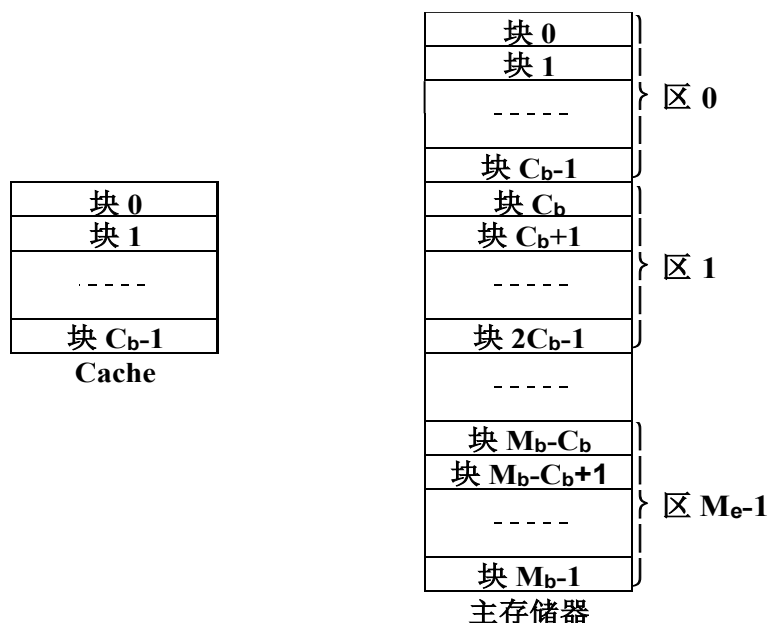
其中： b 为Cache块号， B 是主存块号， C_b 是Cache块数。

- 实际上，Cache地址与主存储器地址的低位部分完全相同。

3.3.2 地址映象与变换方法

2. 直接映象及其变换

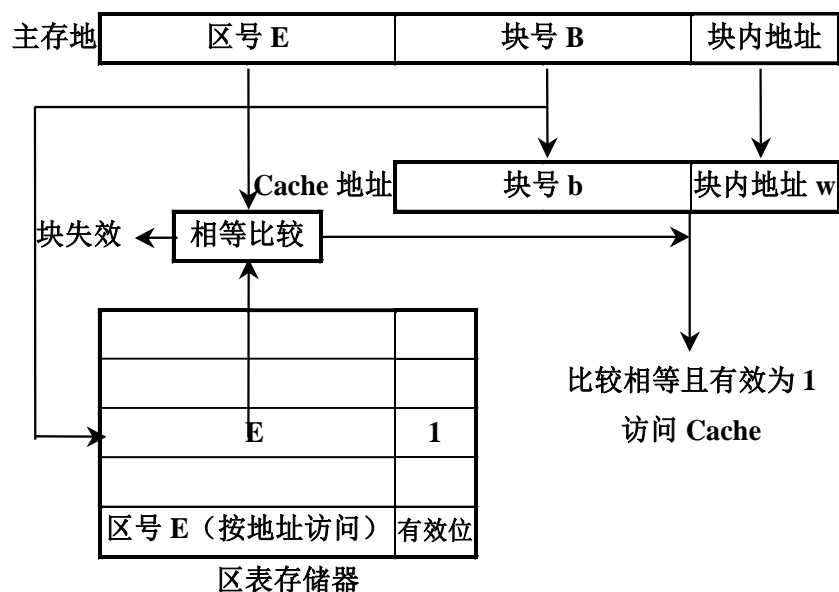
- 映象规则：主存储器中一块只能映象到Cache的一个特定的块中。
- Cache地址的计算公式： $b = B \bmod C_b$
- 实际上，Cache地址与主存储器地址的低位部分完全相同。



直接映象方式的地址映象规则

3.3.2 地址映象与变换方法

2. 直接映象及其变换

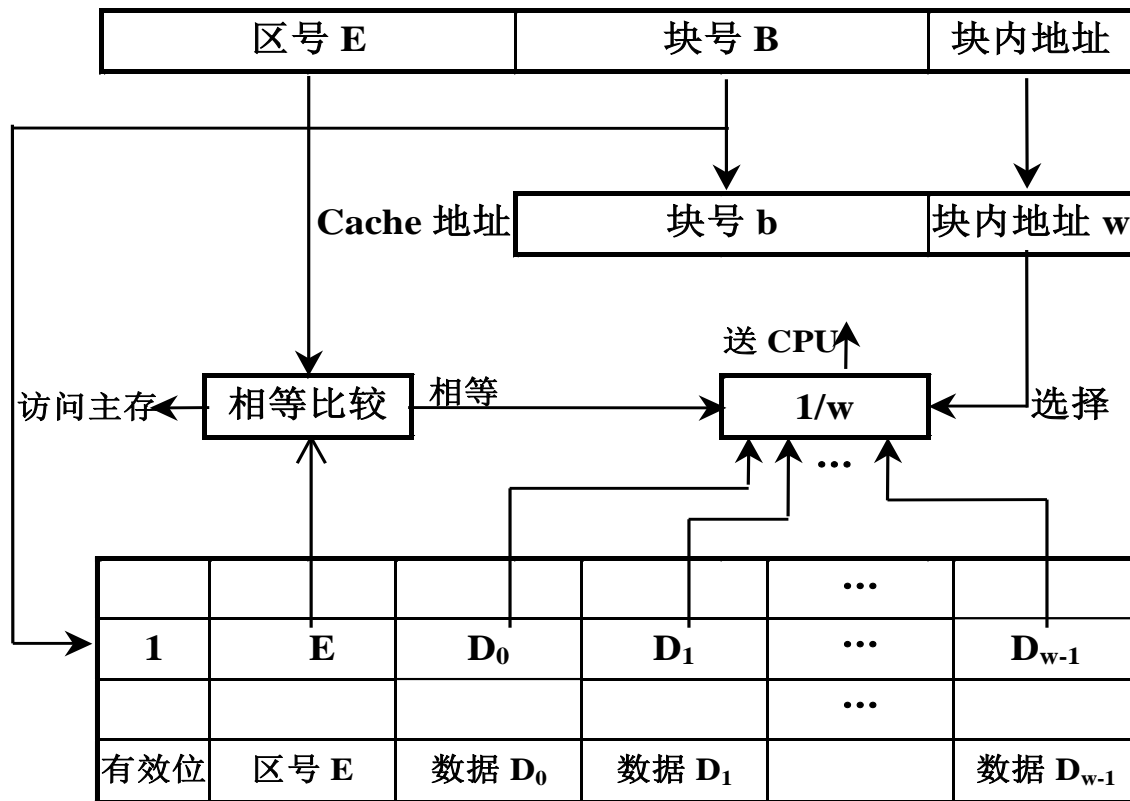


直接映象方式的地址变换规则

➤ 用主存地址中的块号B去访问区号存储器，把读出来的区号与主存地址中的区号E进行比较：

- 比较结果相等，有效位为1，则Cache命中，否则该块已经作废。
- 比较结果不相等，有效位为1，Cache中的该块是有用的，否则该块是空的。

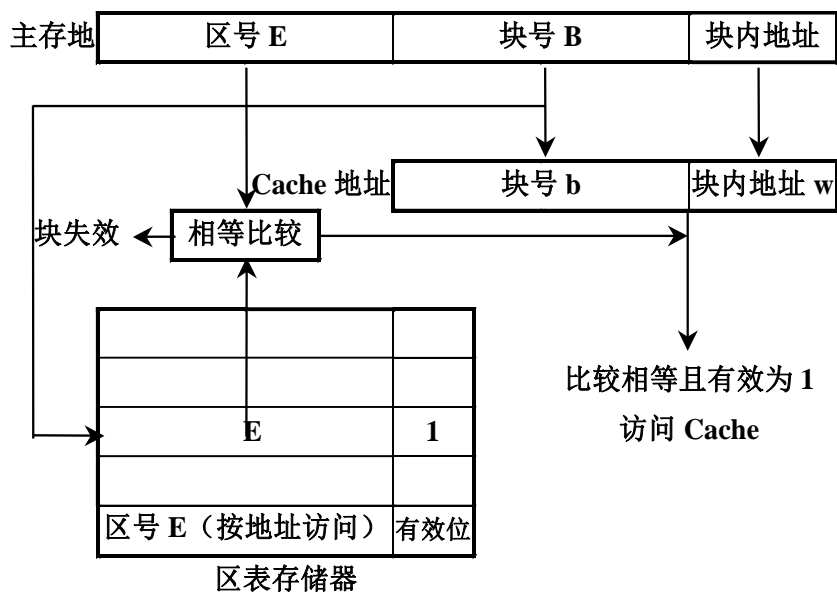
提高Cache速度的一种方法：把区号存储器与Cache合并成一个存储器



按地址访问的 Cache

3.3.2 地址映象与变换方法

2. 直接映象及其变换



直接映象方式的地址变换规则

- 主要优点：硬件实现很简单，不需要相联访问存储器；访问速度也比较快，实际上不需要进行地址变换。
- 主要缺点：块的冲突率比较高。

3.3.2 地址映象与变换方法

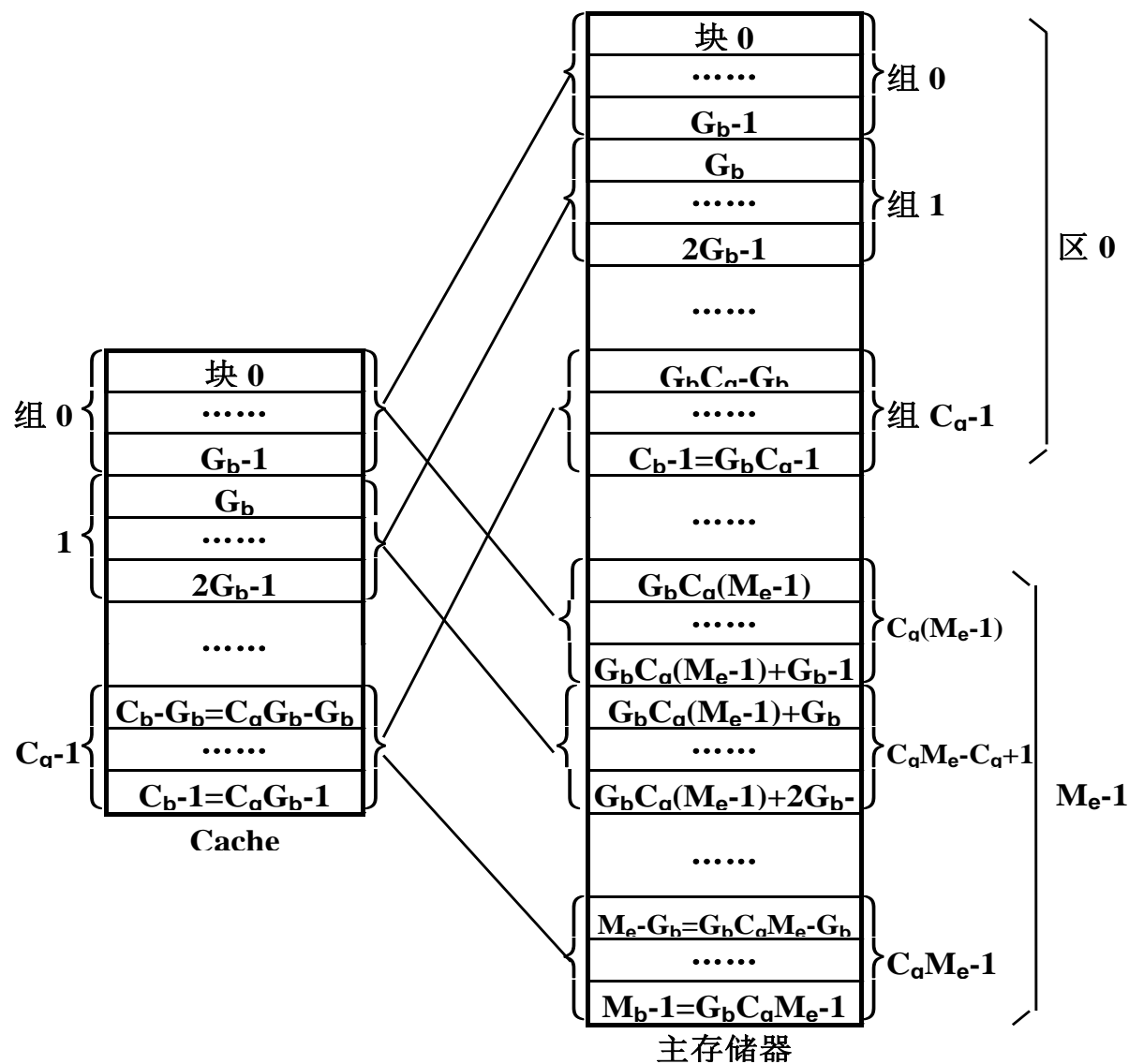
3. 组相联映象及其变换

➤ 映象规则：

- 主存和Cache按同样大小划分成块和组。
- 主存和Cache的组之间采用直接映象方式。
- 在两个对应的组内部采用全相联映象方式。

➤ 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。



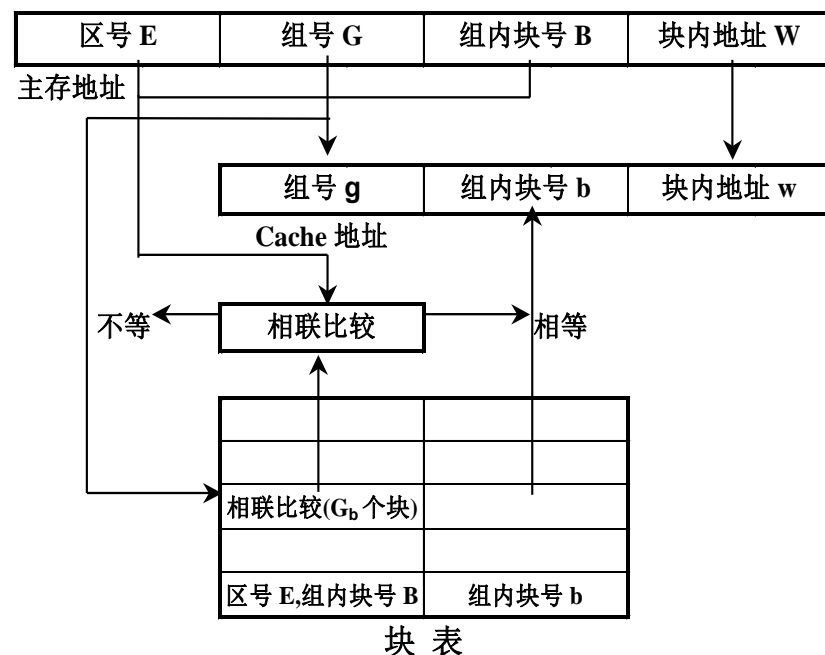
组相联映象方式

3.3.2 地址映象与变换方法

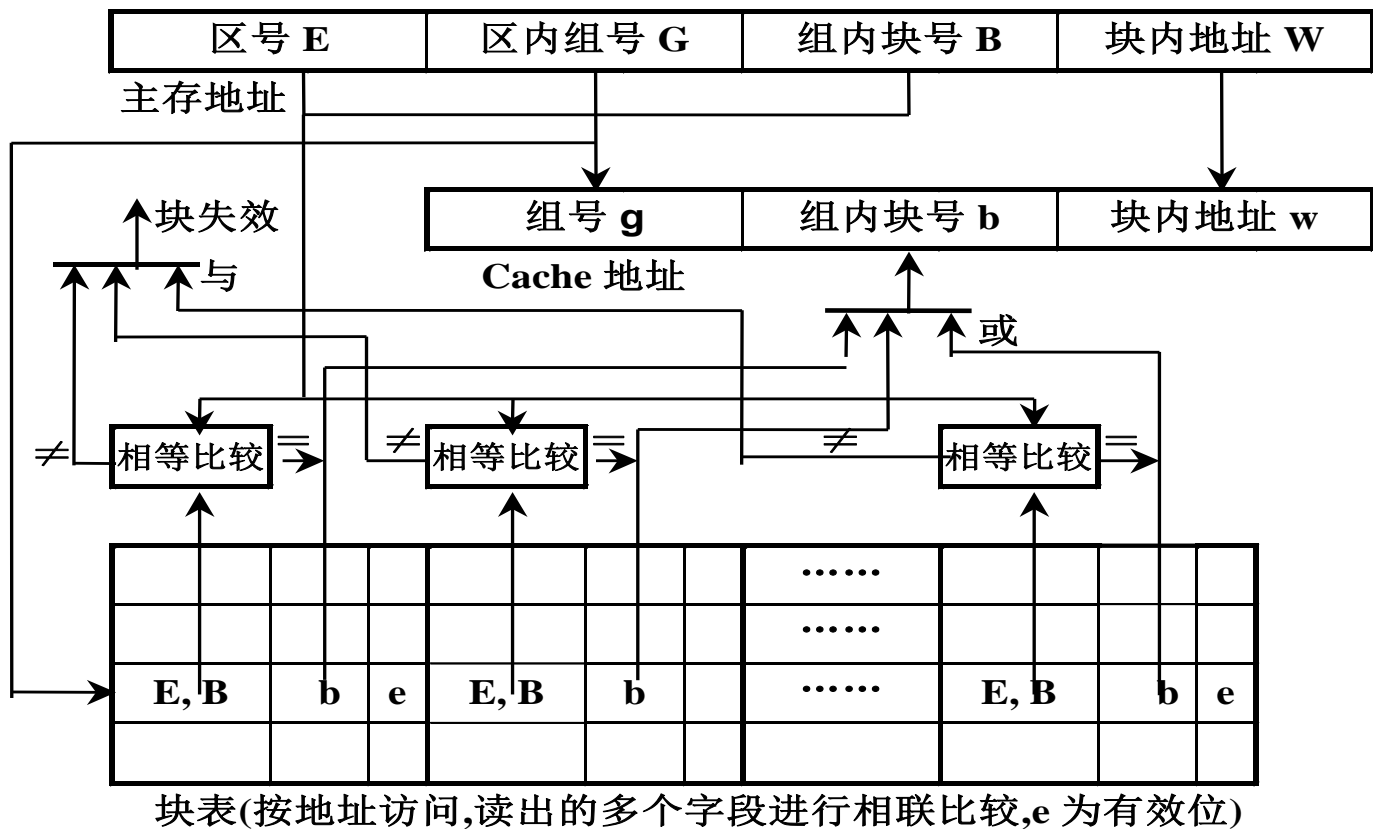
3. 组相联映象及其变换

➤ 组相联映象的地址变换过程

- 用主存地址中的组号G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；
- 如果全部不相等，表示Cache没有命中。



- 提高Cache访问速度的一种方法：用多个相等比较器来代替相联访问



组相联映象方式典型机器的 Cache 分组情况

机器型号	Cache 块数 C_b	每组的块数 G_b	Cache 组数 C_g
DEC VAX-11/780	1024	2	512
Amdahl 470/V6	512	2	256
Intel i860 D-Cache	256	2	128
Honeywell 66/60	512	4	128
Amdahl 470/V7	2048	4	512
IBM 370/168	1024	8	128
IBM3033	1024	16	64
Motolola 88110	256	2	128

3.3.3 Cache替换算法及其实现

➤ 使用的场合：

- 直接映象方式实际上不需要替换算法
- 全相联映象方式的替换算法最复杂
- 主要用于组相联、段相联等映象方式中

➤ 要解决的问题：

- 记录每次访问Cache的块号
- 在访问过程中，对记录的块号进行管理
- 根据记录和管理结果，找出替换的块号

➤ 主要特点：全部用硬件实现

3.3.3 Cache替换算法及其实现

1. 轮换法及其实现

用于组相联映象方式中，有两种实现方法。

方法一：每块一个计数器

- 在块表内增加一个替换计数器字段，计数器的长度与Cache地址中的组内块号字段的长度相同。
- 替换方法及计数器的管理规则：新装入或替换的块，它的计数器清0，同组其它块的计数器都加“1”。
- 在同组中选择计数器的值最大的块作为被替换的块。

Solar-16/65小型机的Cache采用位选择组相联映象方式。Cache每组的块数为4，因此，每块用一个2位的计数器。

序列	初始值	装入块00	装入块01	装入块10	装入块11	替换块00
块 00	00	00	01	10	11	00
块 01	00	01	00	01	10	11
块 10	00	01	10	00	01	10
块 11	00	01	10	11	00	01

3.3.3 Cache替换算法及其实现

1. 轮换法及其实现

用于组相联映象方式中，有两种实现方法。

方法二：每组一个计数器







- 替换规则和计数器的管理：本组有替换时，计数器加“1”，计数器的值就是要被替换出去的块号。
- 例：NOVA3机的Cache采用组相联映象方式，Cache每组的块数为8，每组设置一个3位计数器。在需要替换时，计数器的值加“1”，用计数器的值直接作为被替换块的块号。
- 轮换法的优点：实现比较简单，能够利用历史上的块地址流情况
- 轮换法的缺点：没有利用程序的局部性特点

3.3.3 Cache替换算法及其实现

2. LRU算法及其实现

- 为每一块设置一个计数器，计数器的长度与块号字段的长度相同
- 计数器的使用及管理规则：新装入或替换的块，计数器清0，同组中其它块的计数器加1。命中块的计数器清0，同组的其它计数器中，凡计数器的值小于命中块计数器原来值的加1，其余计数器不变。需要替换时，在同组的所有计数器中选择计数值最大的计数器，它所对应的块被替换。

IBM 370/165机的Cache采用组相联映象方式。每组有4块，为了实现LRU替换算法，在块表中为每一块设置一个 2 位的计数器。在访问Cache的过程中，块的装入、替换及命中时，计数器的工作情况如表：

块地址流	主存块 1		主存块 2		主存块 3		主存块 4		主存块 5		主存块 4	
	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器	块号	计数器
Cache块0	1	00	1	01	1	10	1	11	5	00	5	01
Cache块1		01	2	00	2	01	2	10	2	11	2	11
Cache块2		01		10	3	00	3	01	3	10	3	10
Cache块3		01		10		11	4	00	4	01	4	00
	装入		装入		装入		装入		替换		命中	

3.3.3 Cache替换算法及其实现

2. LRU算法及其实现

➤ 主要优点：

- 1) 命中率比较高，
- 2) 能够比较正确地利用程序的局部性特点，
- 3) 充分地利用历史上块地址流的分布情况，
- 4) 是一种堆栈型算法，随着组内块数增加，命中率单调上升。

➤ 主要缺点：控制逻辑复杂，因为增加了判断和处理是否命中的情况。

3.3.3 Cache替换算法及其实现

3. 比较对法

- 以三个块为例，分别称为块A、块B、块C
- 表示方法：用 $T_{AB} = 1$ 表示 B块比 A 块更久没有被访问过（A比B更近被访问过）。如果表示块 C 最久没有被访问过：

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

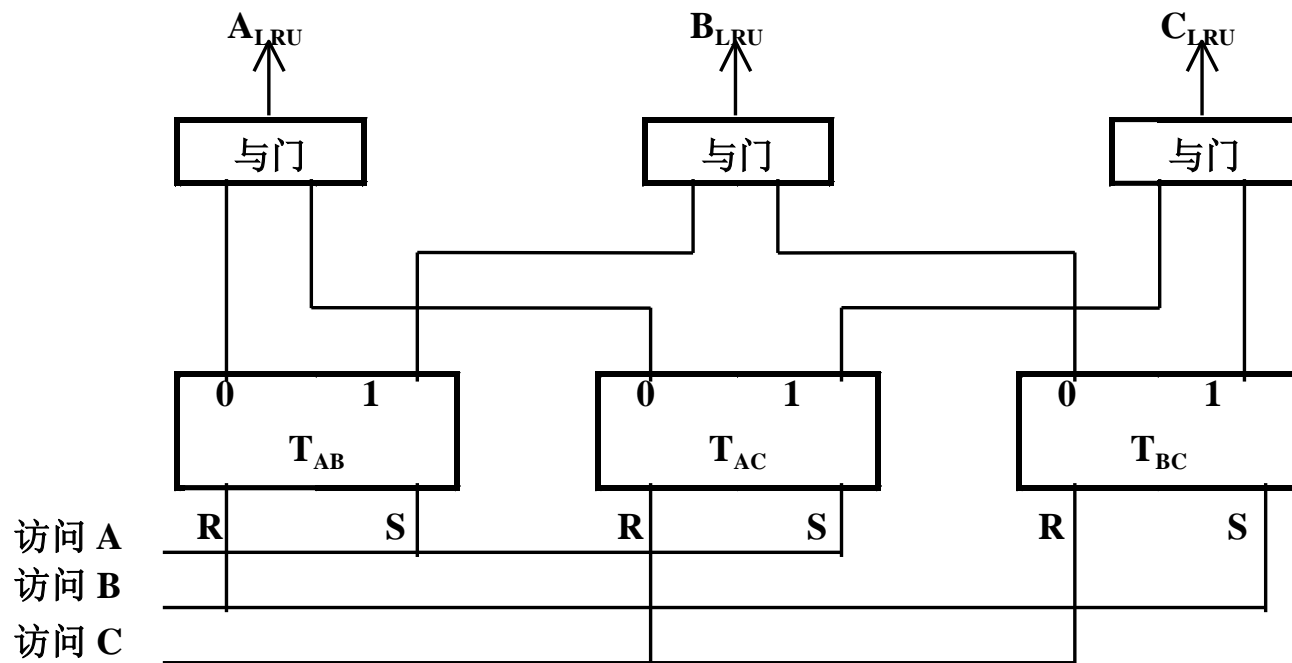
$$A_{LRU} = \overline{T_{AB}} \cdot \overline{T_{AC}}$$

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

- 每次访问之后要改变触发器的状态
 - 在访问块A之后： $T_{AB} = 1$, $T_{AC} = 1$
 - 在访问块B之后： $T_{AB} = 0$, $T_{BC} = 1$
 - 在访问块C之后： $T_{AC} = 0$, $T_{BC} = 0$

3.3.3 Cache替换算法及其实现

3. 比较对法



每组 3 个块的比较对法

3.3.3 Cache替换算法及其实现

3. 比较对法

- 需要的触发器个数为 $G_b(G_b - 1)/2$
- 与门个数为 G_b ,
- 每个门的输入端个数为 $G_b - 1$
- 当每组的块数比较多时, 采用分级办法实现, 实质上是用降低速度来换取节省器件。

例: IBM 3033机的Cache, 每组的块数为16, 分3级。从第1级到第3级分别为4、2、2。共需要触发器个数为: $6+4+8=18$ 。如果不分级则需要触发器120个。

$[(0, 1) (2, 3)] [(4, 5) (6, 7)] [(8, 9) (A, B)] [(C, D) (E, F)]$

3.3.3 Cache替换算法及其实现

3. 比较对法

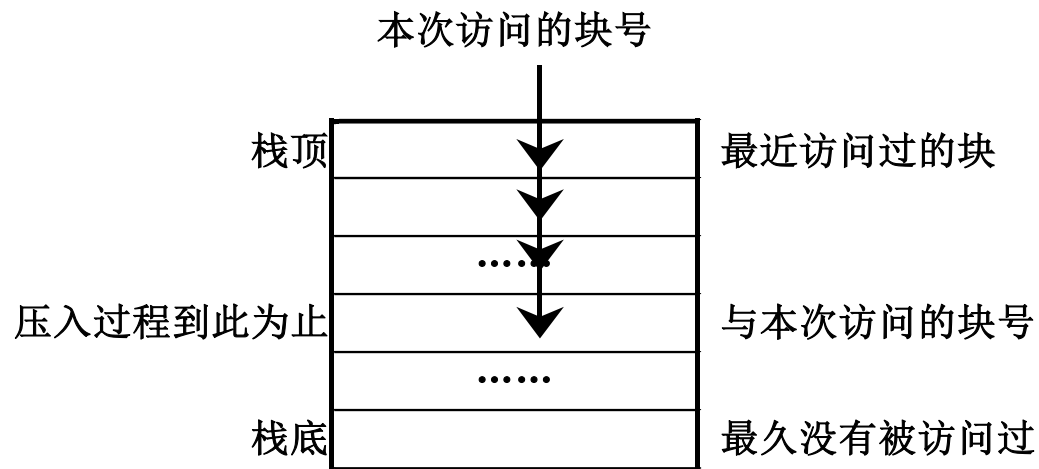
每组块数	3	4	6	8	16	64	256
触发器个数	3	6	15	28	120	2016	32640
与门个数	3	4	6	8	16	64	256
与门输入端数	2	3	5	7	15	63	255

比较对法中每组块数与所需硬件的关系

3.3.3 Cache替换算法及其实现

4. 堆栈法

- 把本次访问的块号与堆栈中保存的所有块号进行相联比较。
 - 如果有相等的，则Cache命中。把本次访问块号从栈顶压入，堆栈内各单元中的块号依次往下移，直至与本次访问的块号相等的那个单元为止，再往下的单元直至栈底都不变。
 - 如果没有相等的，则Cache块失效。本次访问的块号从栈顶压入，堆栈内各单元的块号依次往下移，直至栈底，栈底单元中的块号被移出堆栈，它就是要被替换的块号。



例如： 每组为4块， 则堆栈有4个存储单元， 每个单元2位。

$$NA: I_0 I_1 \neq A_0 A_1$$

$$NB: I_0 I_1 \neq B_0 B_1$$

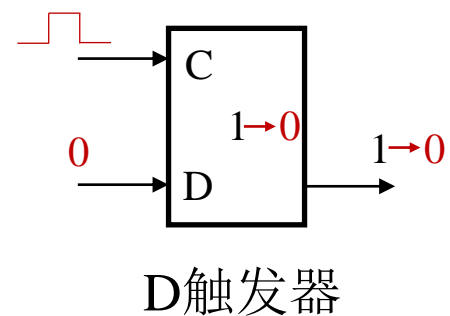
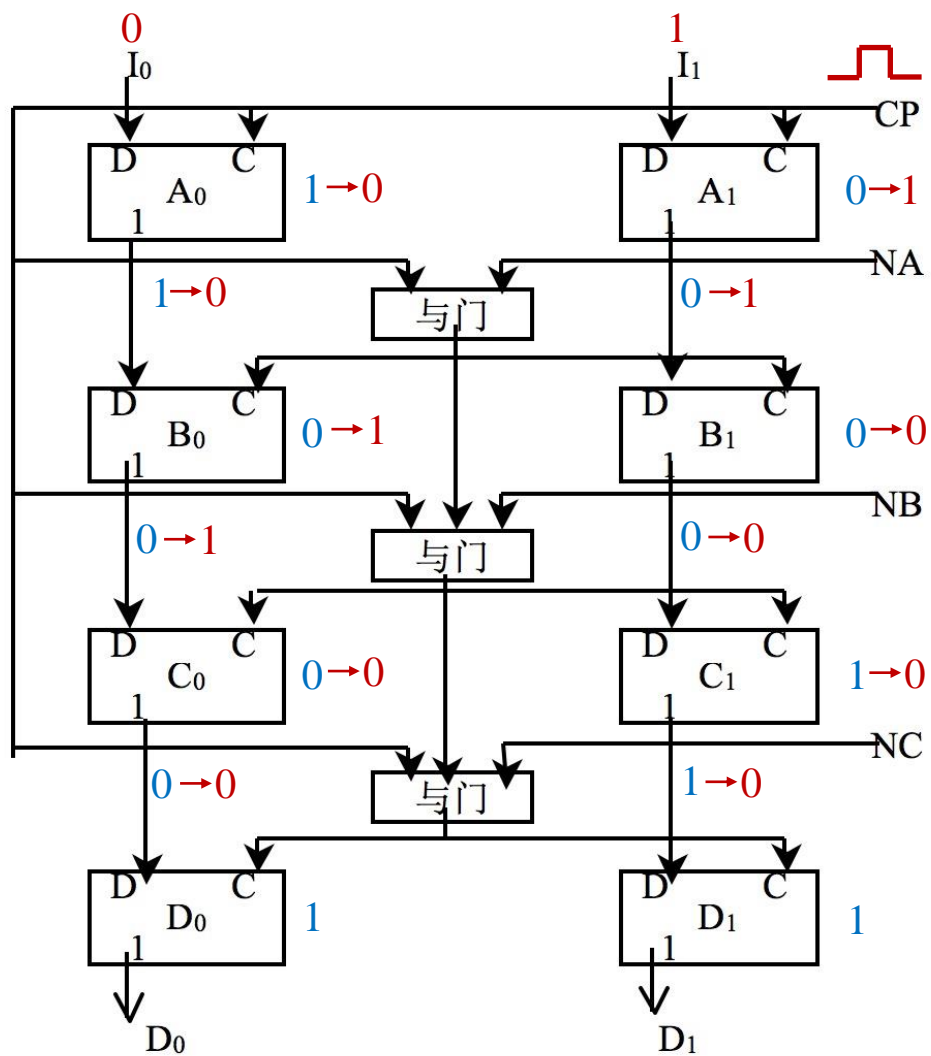
$$NC: I_0 I_1 \neq C_0 C_1$$

$$NA = (I_0 \oplus A_0) \vee (I_1 \oplus A_1)$$

$$NB = (I_0 \oplus B_0) \vee (I_1 \oplus B_1)$$

$$NC = (I_0 \oplus C_0) \vee (I_1 \oplus C_1)$$

$I_0 \oplus A_0$	$I_1 \oplus A_1$	NA
0 ($I_0 = A_0$)	0 ($I_1 = A_1$)	0
0 ($I_0 = A_0$)	1 ($I_1 \neq A_1$)	1
1 ($I_0 \neq A_0$)	0 ($I_1 = A_1$)	1
1 ($I_0 \neq A_0$)	1 ($I_1 \neq A_1$)	1



$$NA: I_0I_1 \neq A_0A_1$$

$$NB: I_0I_1 \neq B_0B_1$$

$$NC: I_0I_1 \neq C_0C_1$$

每组4块的堆栈法逻辑图

3.3.3 Cache替换算法及其实现

4. 堆栈法

- 堆栈法的主要优点：块失效率比较低，因为它采用了LRU算法；硬件实现相对比较简单。
- 堆栈法的主要缺点：速度比较低，因为它需要进行相联比较。
- 堆栈法与比较对法所用触发器的比例：

$$\frac{G_b(G_b - 1)}{2} : G_b \cdot \log_2 G_b$$

- 其中， G_b 是Cache每一组的块数。
- 当 G_b 大于8时，堆栈法所用的器件明显少于比较对法。

3.3.4 Cache存储系统性能分析

1. 加速比与命中率的关系

➤ Cache存储系统的加速比 S_p 为：

$$S_p = \frac{T_m}{T} = \frac{T_m}{H \cdot T_c + (1 - H) \cdot T_m} = \frac{1}{(1 - H) + H \cdot \frac{T_c}{T_m}}$$

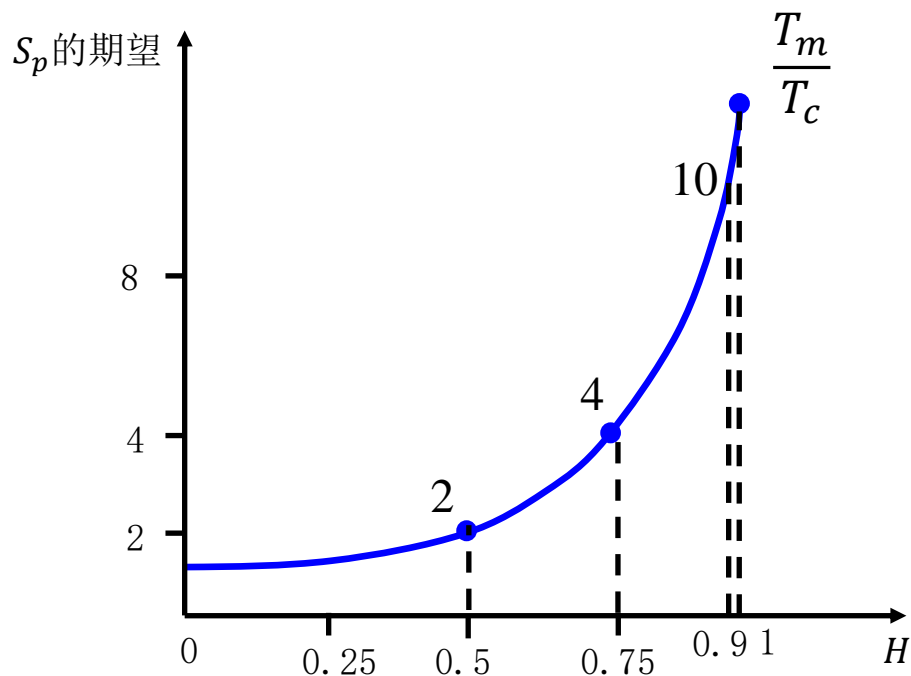
- T_m 为主存储器的访问周期，
- T_c 为Cache的访问周期，
- T 为Cache存储系统的等效访问周期，
- H 为命中率。

➤ 提高加速比的最好途径是提高命中率

3.3.4 Cache存储系统性能分析

1. 加速比与命中率的关系

➤ Cache存储系统的加速比 $S_p = \frac{T_m}{T} = \frac{T_m}{H \cdot T_c + (1-H) \cdot T_m} = \frac{1}{(1-H) + H \cdot \frac{T_c}{T_m}}$



3.3.4 Cache存储系统性能分析

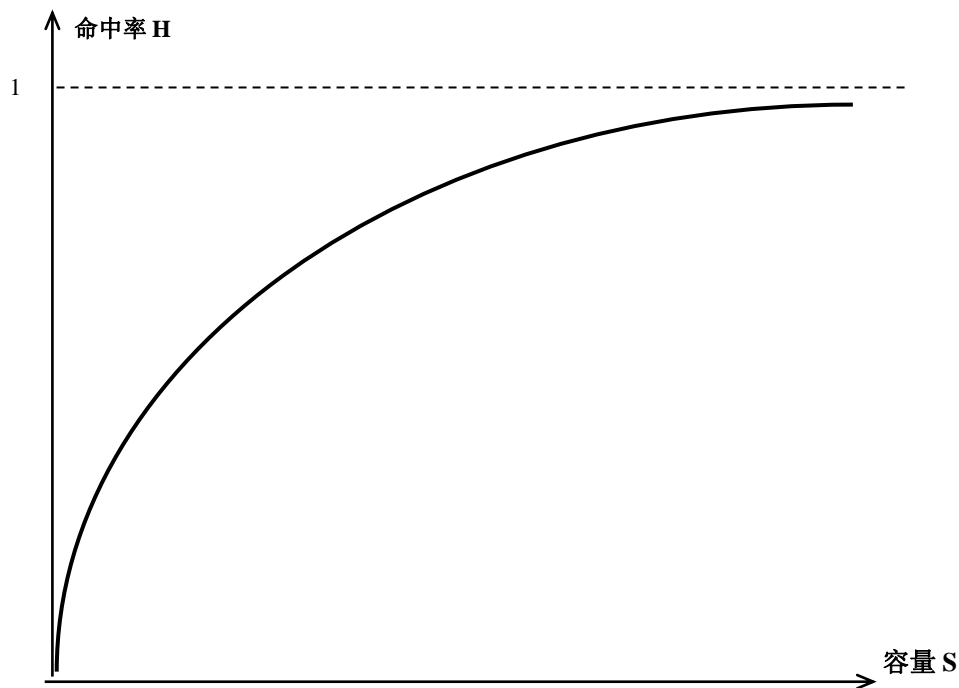
2. 命中率的相关因素

- 程序执行过程中的地址流分布情况
- 当发生Cache块失效时，所采用的替换算法
- Cache的容量
- 在组相联映像方式中，块的大小和分组的数目
- Cache预取算法

3.3.4 Cache存储系统性能分析

2. 命中率的相关因素：Cache的容量

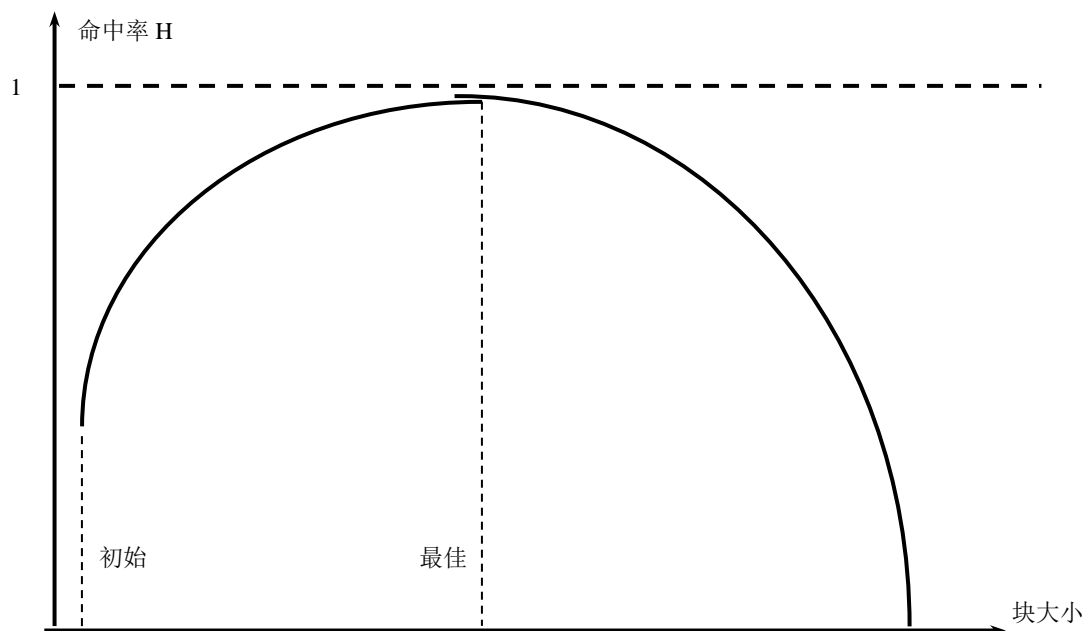
- Cache的命中率随它的容量的增加而提高
- Cache容量到达一定程度时，命中率提高很慢



3.3.4 Cache存储系统性能分析

2. 命中率的相关因素：Cache块的大小

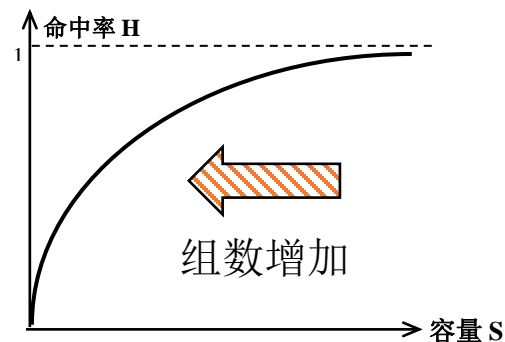
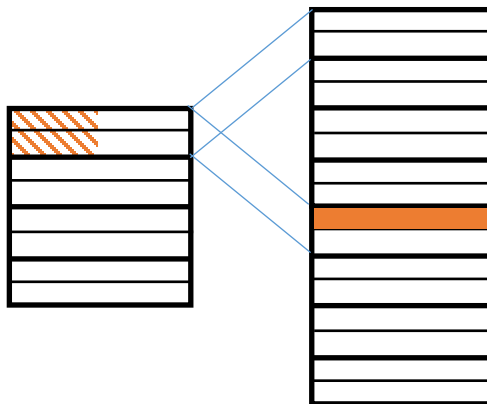
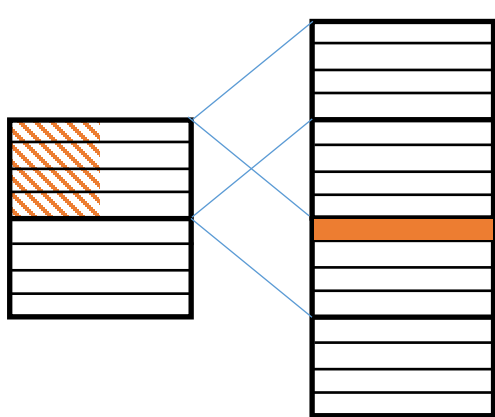
- 在组相联方式中，块大小对命中率非常敏感
- 块很小时，命中率很低；随着块大小增加命中率也增加，有一个极大
- 当块非常大时，进入Cache中的数据可能无用
- 当块大小等于Cache容量时，命中率将趋近零



3.3.4 Cache存储系统性能分析

2. 命中率的相关因素：Cache组数

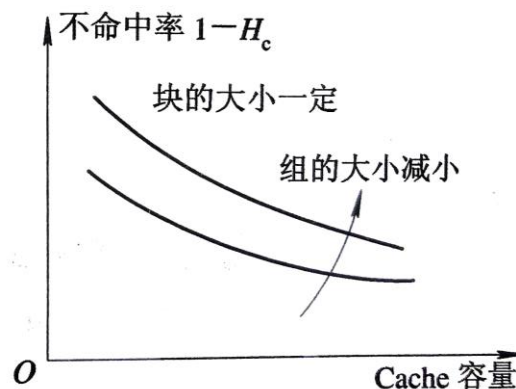
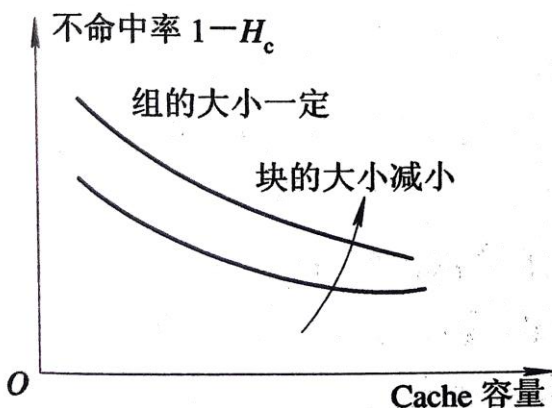
- 在组相联方式中，组数对命中率的影响很明显
- 随着组数的增加，Cache的命中率要降低。
- 当组数不太大时（小于512），命中率的降低很少
- 当组数超过一定数量时，命中率的下降非常快



3.3.4 Cache存储系统性能分析

2. 命中率的相关因素

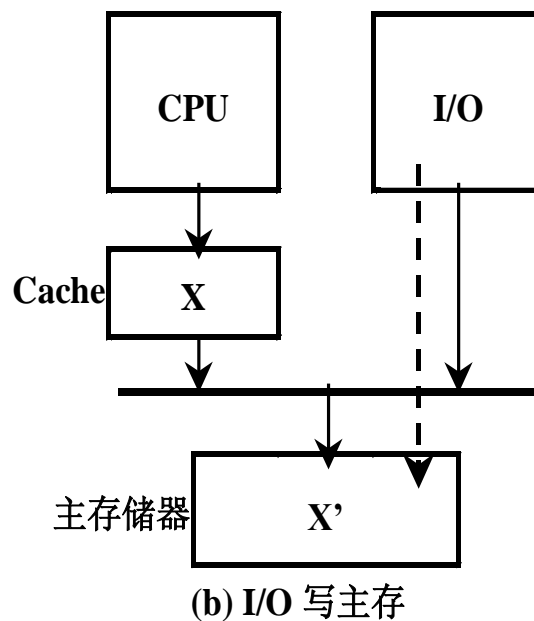
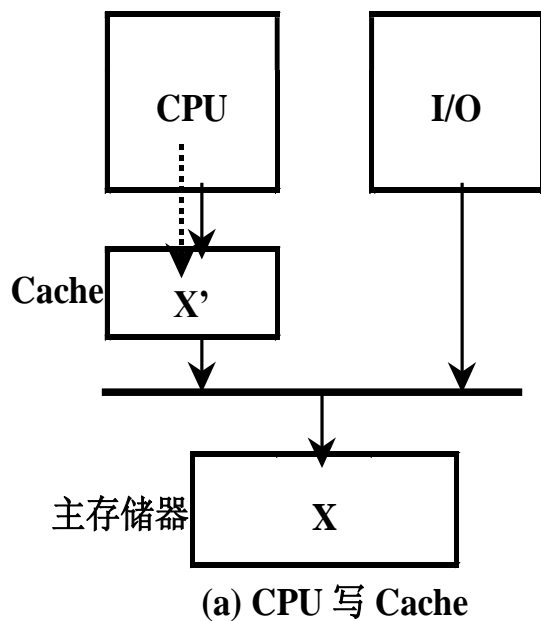
- 程序执行过程中的地址流分布情况
- 当发生Cache块失效时，所采用的替换算法
- Cache的容量
- 在组相联映像方式中，块的大小和分组的数目
- Cache预取算法



3.3.5 Cache的一致性

➤ 造成Cache与主存的不一致的原因：

- 由于CPU写Cache，没有立即写主存
- 由于I/O处理机或I/O设备写主存



3.3.5 Cache的一致性

➤ Cache的更新算法

- 写直达法（Write-through，WT）：CPU的数据写入Cache时，同时也写入主存。
- 写回法（Write-Back，WB）：CPU的数据只写入Cache，不写入主存，仅当替换时，才把修改过的Cache块写回主存。

3.3.5 Cache的一致性

➤ Cache的更新算法（写直达法、写回法）比较

- 1) 可靠性: 写直达法优于写回法。写直达法能够始终保证Cache是主存的副本。如果Cache发生错误，可以从主存得到纠正。
- 2) 与主存的通信量: 写回法少于写直达法。
 - 对于写回法，大多数操作只需要写Cache，不需要写主存；当发生块失效时，可能要写一个块到主存；即使是读操作，也可能要写一个块到主存。
 - 对于写直达法，每次写操作，必须写、且只写一个字到主存。
 - 实际上，写直达法的写次数很多、每次只写一个字；写回法是的写次数很少、每次要写一个块。

例：写操作占总访存次数的20%，Cache的命中率为99%，每块为4个字。当Cache发生块替换时，有30%块需要写回到主存，其余的块因为没有被修改过而不必写回主存。试比较写回法与写直达法的访存通信量（百分比）。

解：

对于写直达法：写主存次数占总访存次数的20%，

对于写回法： $(1-99\%) \times 30\% \times 4 = 1.2\%$ 。

因此，与主存的通信量，写回法要比写直达法少10多倍。

3.3.5 Cache的一致性

➤ Cache的更新算法（写直达法、写回法）比较

3) 控制的复杂性：写直达法比写回法简单。

- 对于写回法，要为每块设置一个修改位，而且要对修改位进行管；为了保证Cache的正确性，通常要采用比较复杂的校验方式或校正方式。
- 对于写直达法，不需要设置修改位；只需要采用简单的奇偶校验即可。由于Cache始终是主存的副本，Cache一旦有错误可以从主存得到纠正。

3.3.5 Cache的一致性

➤ Cache的更新算法（写直达法、写回法）比较

4) 硬件实现的代价：写回法要比写直达法好。

- 对于写直达法：为了缩短写Cache流水段的时间，通常要设置一个小容量的高速寄存器堆（后行写数缓冲站），每个存储单元要有数据、地址和控制状态等3部分组成。每次写主存时，首先把写主存的数据和地址写到高速寄存器堆中。每次读主存时，要首先判断所读数据是否在这个高速寄存器堆中。
- 写回法不需要设置高速缓冲寄存器堆。

3.3.5 Cache的一致性

➤ 写Cache的两种方法

1) 不按写分配法：在写Cache不命中时，只把所要写的字写入主存。

2) 按写分配法：在写Cache不命中时，还把一个块从主存读入Cache。

➤ 目前，在写回法中采用按写分配法，在写直达法中采用不按写分配法。

3.3.6 Cache的预取算法

➤ 预取算法有如下几种：

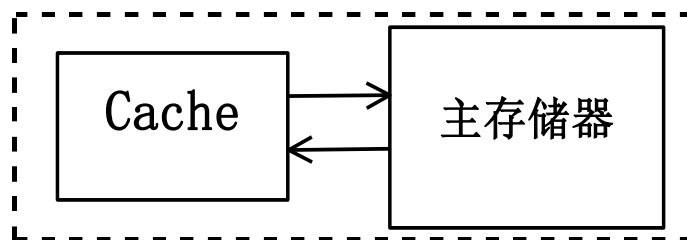
- 1) 按需取。当出现Cache不命中时，才把需要的一个块取到Cache中。
- 2) 恒预取。无论Cache是否命中，都把下一块取到Cache中。
- 3) 不命中预取。当出现Cache不命中，把本块和下一块都取到Cache中。

➤ 主要考虑因素：命中率是否提高，Cache与主存间通信量。

- 恒预取能使Cache不命中率降低75~85%
- 不命中预取能使Cache不命中率降低30~40%

3.4 三级存储系统

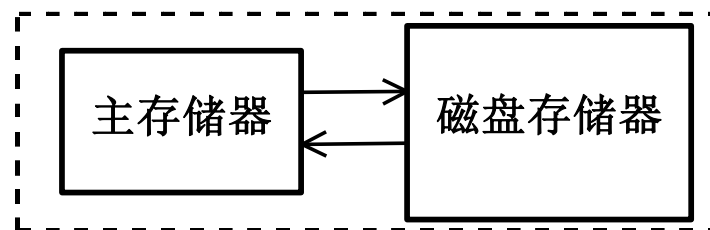
- 虚拟存储系统和Cache存储系统可同时存在
- 存储系统可以有多种构成方法
- 不同的构成只是实现技术不同



↑ 系统程序员看:

速度接近 Cache 的速度,
存储容量是主存的容量,
每位价格接近主存储器。

Cache 存储系统



↑ 应用程序员看:

速度接近主存储器的速度,
存储容量是虚拟地址空间,
每位价格接近磁盘存储器。

虚拟存储系统

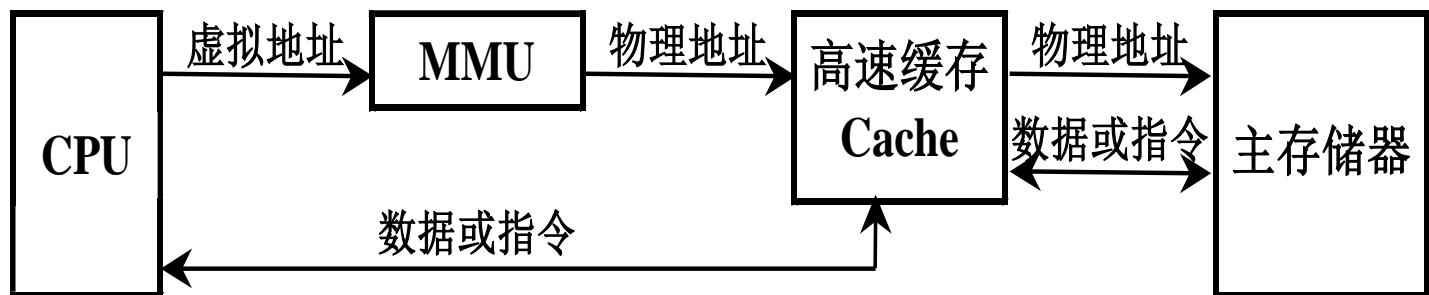
3.4.1 存储系统的组织方式

- 两个存储系统的组织方式：
 - 又称为：物理地址Cache存储系统
 - 目前的大部分处理机采用这种两级存储系统
- 一个存储系统组织方式：
 - 又称为：虚拟地址Cache存储系统
 - 如Intel公司的i860等处理机采用这种组织方式
- 全Cache系统：
 - 没有主存储器，由Cache和磁盘组成存储系统。

3.4 三级存储系统

1. 两个存储系统的组织方式（物理地址Cache）

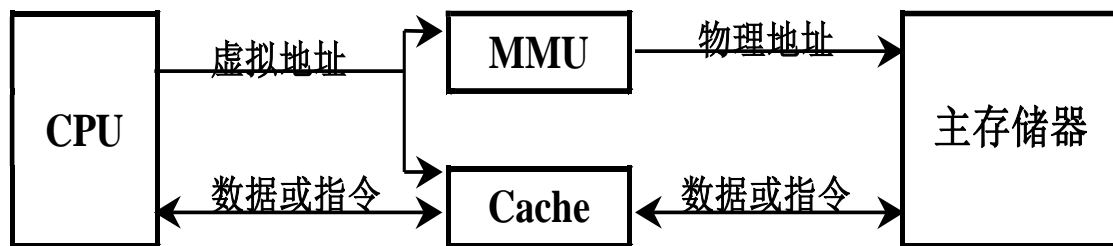
- CPU用程序虚地址访存，经过存储管理部件（Memory Management Unit, MMU）中的地址变换部件变换成主存物理地址访Cache。
- 如果命中，则访问Cache；若不命中，则将该主存物理地址的字和该字的一个主存块与Cache的某相应块交换，同时将CPU所需要的指令或数据送往CPU
- 目前的大部分处理机采用这种两级存储系统



3.4 三级存储系统

2. 一个存储系统组织方式（虚地址Cache）

- CPU访存时，将虚拟地址直接送往MMU和Cache。
- 若Cache命中，数据、指令就直接送到CPU；若Cache不命中，则由MMU将虚地址转换成物理地址访主存，将含该地址的数据块或指令块与Cache交换的同时，将所需要的指令和数据送往CPU。



3.4 三级存储系统

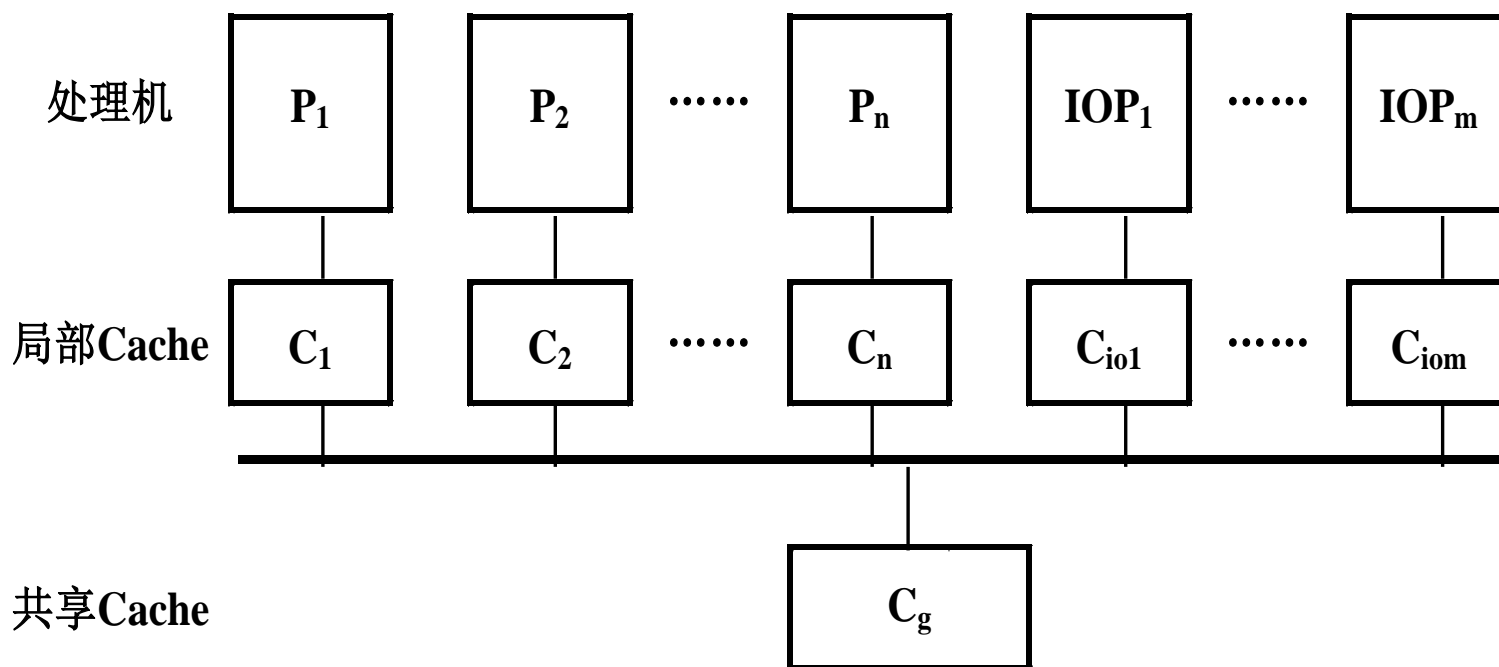
3. 全Cache存储系统

- 建立存储系统的目的：获得一个速度接近Cache，容量等于虚拟地址空间的存储器。
- 这个存储器如何构成，具体分成几级来实现，只是具体的实现技术而已。
- 随着计算机硬件和软件技术的发展，存储系统的实现技术也在不断改变。
- 最直接最简单的方法：用一个速度很高，存储容量很大的存储器来实现。
- 全Cache(all-Cache)是一种理想的存储系统。

3.4 三级存储系统

3. 全Cache存储系统

➤ 一种多处理机系统中的全Cache存储系统



本章重点：

1. 存储系统的定义及主要性能计算。
2. 并行存储器的工作原理。
3. 虚拟存储系统的工作原理。
4. 虚拟存储器中加快地址变换的方法。
5. 虚拟存储系统的页面替换算法。
6. Cache存储系统的地址映象及变换方法。
7. Cache存储系统的替换算法。
8. Cache存储系统的加速比。