计算机组成与体系结构

第6章 存储系统设计

张 钰
ZJSS@BIT.EDU.CN
北京理工大学计算机学院
SCHOOL OF COMPUTER SCIENCE & TECHNOLOGY
BEIJING INSTITUTE OF TECHNOLOGY

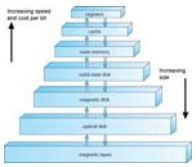
存储器是计算机系统的核心部件之一，其容量、速度和价格是必须要考虑的因素。

主要目标:

在尽可能低的价格下，提供尽可能高的速度及尽可能大的存储容量。

高速度、低价格、大容量!

存储系统是由几个容量、速度和价格各不相同的存储器构成的系统。



6.1.1 存储器分类和存储系统定义

早期冯·诺依曼型计算机硬件系统以运算器为中心，而现代计算机硬件系统都以存储器为中心。在计算机运行过程中，存储器是各种信息存储和交换的中心，存放指令、操作数和运算结果。

存储器分类

在一台计算机中，通常有多种存储器。

- 按用途分：主存储器、Cache、通用寄存器、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、RAM、SRAM、DRAM等。
- 访问方式：直接译码、先进先出、随机访问、相联访问、块传送、文件组等。

存储器
- 主存储器：存放正在运行的程序与数据。
- 辅助存储器：存放等待运行的程序与数据。
- 通用寄存器组：存放最经常用到的数据。

存储系统和存储器是两个不同的概念！

1. 容量、速度和价格的矛盾

大容量	希望能放得下所有软件
高速度	尽量和CPU的速度相匹配
低价格	系统价格中较小而合理的比例

但上述要求是相互矛盾的
- 容量越大，因延迟增加而使速度降低
- 容量越大，存储体总体价格就越高
- 速度越高，价格将越高

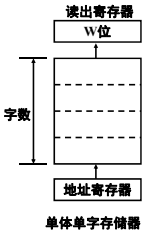
2. 容量、速度和价格分析

容量
- S_M = W * l * m，其中：

W —— 存储体的字长（位 或 字节）（设计）

l —— 每个存储体的字数（工艺）

m —— 并行工作的存储体个数（设计）
- 可以看出，它即与存储器器件有关，也与设计有关。



单体单字存储器

速度
- 可以用访问时间T_a、存储周期T_m和频宽B_m表示。
- T_a：存储器从接到访存读申请，到数据被读到数据总线上所需要的时间，它是启动一个访存读操作后，CPU必须等待的时间，是确定CPU与存储器时间关系的一个重要指标。

速度（续）
- T_m：连续启动一个存储体所需要的时间，即存储器进行一次存/取所需要的时间，一般它总比T_A大。
- B_m：表示存储器可以提供的数据传输率，用每秒钟传送的位数或字节数表示。分为最大(极限)频宽和实际频宽。
- 最大(极限)频宽是存储器连续访问时所能提供的频宽。

速度（续）
- 单体存储器：$B_m = W / T_m$
- 多体存储器：$B_m = m \times W / T_m$
- 可以看出：它即与存储器器件有关，也与设计有关。

价格

- 可以用总价格C 或 位价格 c 表示
- 价格 $c = C/S_M = C/(W \times l \times m)$

可以看出：它即与存储器器件有关，也与设计有关。

3. 解决矛盾的措施

为满足系统对存储器的性能要求，可以采取以下措施：

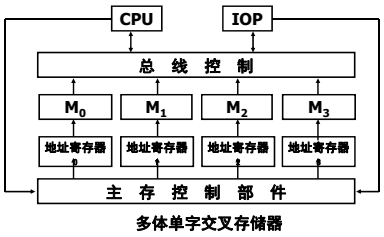
（1）改进工艺和技术，降低成本、提高速度

- 问题：只采用一种工艺的单一存储器无法同时满足上述三个方面的要求。

（2）构成并行主存系统

- 采用单一存储器，但在组成上引入并行和重叠技术，构成并行主存系统，即多体交叉存储器，在位价格基本不变的情况下，使主存的频宽得到较大的提高。
- 问题：提高频宽的能力是有限的
  - m越多，负载变重，延时增加
  - 系统效率不是很高，因为指令的读取不是顺序的，转移指令使存储系统效率下降。

（2）构成并行主存系统（续）



多体单字交叉存储器

（3）使用存储器系统

- 用不同工艺的多种存储器组成存储器系统，使信息以各种方式分布于不同的存储器上。
- 例如：至少有主存和辅存两种存储器
  - 主存：价格高、速度快、容量小，存放程序的活跃部分。
  - 辅存：价格低、速度慢、容量大，存放暂时不用的部分。

（3）使用存储器系统（续）

- 问题：主存速度仍不能满足CPU的要求
- 例如，在上世纪70年代，合理成本、足够容量的主存的存储周期比CPU节拍宽度大一个数量级

（4）存储层次（存储体系）

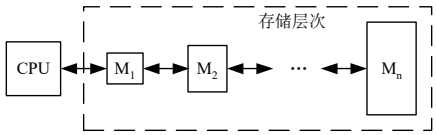
为了解决单一种类存储器的价格、容量及速度之间的矛盾，计算机系统的存储系统总是利用多种不同的存储器构成。

如何组织这些不同的存储器呢?

存储层次!

存储系统定义：

- 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。
- 对应用程序员透明。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。



多级存储层次

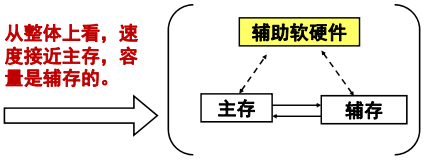
两个典型分支：

- **虚拟存储系统**
- **Cache存储系统**

虚拟存储系统

解决容量问题

- 主存－辅存存储层次（二级存储层次）



主存－辅存存储层次

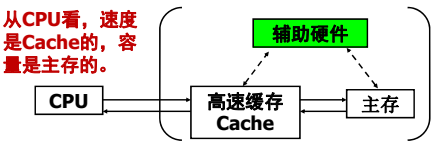
解决容量问题（续）

- 利用了I/O处理可与CPU并行操作的能力；
- 借助操作系统（OS）、硬件等实现地址变换和程序定位，实现主存和辅存之间的数据传送，使主存、辅存构成了一个完整的整体。
- 对应用程序员是透明；
- 主存－辅存存储层次的不断发展和完善，就形成了虚拟存储器。

Cache存储系统

解决速度问题

- Cache－主存存储层次（二级存储层次）



Cache－主存存储层次

解决速度问题（续）

- 在CPU和主存之间增加一级速度快、容量小、位价格较高的高速缓冲存储器（Cache）
- 借助于辅助硬件实现 Cache和主存之间的传送，使Cache和主存构成一个整体
- 从CPU看，速度接近Cache的速度，容量是主存的容量，每位价格接近于主存的价格。
- 对应用程序员和系统程序员都是透明的。

Cache－主存层次，又称Cache存储系统，由Cache和主存储器构成。其主要目的是提高存储器速度，弥补主存速度的不足。

主存－辅存层次，又称虚拟存储系统，由主存储器和联机的辅存（磁盘存储器）构成。其主要目的是扩大存储器容量，弥补主存容量的不足。

多级存储层次

二级存储层次可以扩展到多级存储层次

多级存储层次

多级存储层次（续）

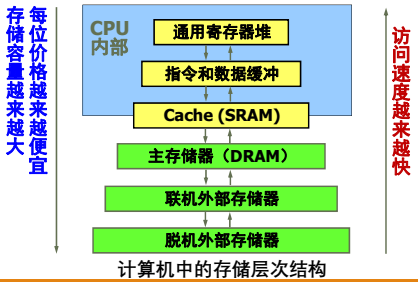
- 从CPU看是一个整体，速度接近于M_1，容量是M_n的，位价格接近于M_n

$T \approx \min(T_1, T_2, \ldots, T_n)$，用存储周期表示
$S \approx \max(S_1, S_2, \ldots, S_n)$，用MB或GB表示
$C \approx \min(C_1, C_2, \ldots, C_n)$，用每位的价格表示

6.1.2 存储系统层次结构

计算机中多个层次的存储器，由

通用寄存器堆→指令或数据缓冲器→Cache→主存储器→联机外部存储器→脱机外部存储器组成。



计算机中的存储层次结构

各级存储器的主要性能特性（1/2）

存储器层次	通用寄存器	缓冲栈	Cache
存储周期	< 10ns	< 10ns	10 - 60ns
存储容量	< 512B	< 512B	8K-2MB
价格$c/KB	1200	80	3.2
访问方式	直接译码	先进先出	相联访问
材料工艺	ECL	ECL	SRAM
分配管理	编译器分配	硬件调度	硬件调度
带宽MB/s	400-8000	400-1200	200-800

各级存储器的主要性能特性（2/2）

存储器层次	主存储器	磁盘存储器	脱机存储器
存储周期	60-300ns	10 - 30ms	2 - 20 min
存储容量	32M-1GB	1G-1TB	5G-10TB
价格$c/KB	0.36	0.01	0.0001
访问方式	随机访问	块访问	文件组
材料工艺	DRAM	磁表面	磁、光等
分配管理	操作系统	系统/用户	系统/用户
带宽MB/s	80-160	10-100	0.2 - 0.6

多级存储层次

CPU与主存储器的速度差距越来越大

1955年，第一台大型机IBM704，CPU和主存储器的工作周期均为12微秒，目前，CPU的工作速度提高了4个数量级以上，主存储器的工作速度仅提高两个数量级。今后，CPU与主存储器的速度差距会更大。

研究存储系统的目的就是要找出解决这一问题的办法。

多级存储层次的两个原则：

原则1：一致性原则

- 同一信息可以处于不同层次的存储器中。同一信息在不同层次存储器中的值应保持相同。

原则2：包含原则

- 高层次存储器中的信息包含在低层次存储器中的信息中，即：

$$信息_{高层} \in 信息_{低层}$$

不同存储层次之间的数据传送

- 把哪些数据从M_n传送到M_1?
- 把数据放在M_1的什么地方?
- 如果能预判出下步要访问的程序块，并提前将它们取到M_1中，就可以使存储系统有效工作。
- 能否预判？程序局部性 = 预判的基础（后面6.3.1定义）

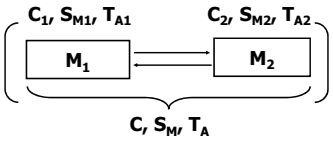
6.1.3 存储体系的性能参数

有以下三个：

- 每位价格c
- 命中率H
- 等效访问时间T_A

下面以二级存储层次为例来分析

二级存储层次



每位价格c

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

- 为使c接近于c_2，应使$S_{M1} << S_{M2}$

存储系统的容量

要求：

存储系统的容量等于M2存储器的容量。
提供尽可能大、能随机访问的地址空间。

方法有两种：

只对M2存储器进行编址，M1存储器只在内部编址。
另外设计一个容量很大的逻辑地址空间。

命中率H

- CPU产生的逻辑地址能在M_1中访问到的概率。
- 命中率H可由实验或模拟方法获得。
- 若逻辑地址流在M_1中访问到的次数为R_1，在M_2中的访问次数为R_2，则：

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

- 显然，不命中率（失效率）＝1－H

等效访问时间T_A

假设M_1访问和M_2访问是同时启动

$$TA = HT1 + (1-H)T2$$

假设M_1访问和M_2访问不是同时启动

$$TA = HT1 + (1-H)(T1+T2) = T1 + (1-H)T2$$

希望T_A接近T_1为好。
访问效率 e = T_1/T_A，越接近 1 越好。

二级存储层次的访问效率

$$e = \frac{T_{A1}}{T_A} = \frac{T_{A1}}{HT_{A1} + (1-H)T_{A2}} = \frac{1}{H + (1-H) \times \frac{T_{A2}}{T_{A1}}}$$

- 存储系统的访问效率主要与命中率和两级存储器的速度之比有关。

- 速度差不要太大
- 命中率越高越好

影响命中率的因素很多，例如：
- 程序的地址流
- 地址预判算法
- M_1的容量等

■例6-1：假设某计算机的存储系统由Cache和主存组成。某程序执行过程中访存1000次，其中访问Cache失效（未命中）50次，则Cache的命中率是多少？

解：程序访存次数N1+N2=1000次，其中访问Cache的次数N1为访存次数减去失效次数。

H= $\frac{1000-50}{1000}$ =95%

■例6-2：CPU执行一段程序时，Cache完成存取的次数为5000次，主存完成存取的次数为200次。已知Cache存储周期TC为40ns，主存存取周期TM为160ns。分别求：

（1）Cache的命中率H；

（2）等效访问时间TA；

（3）Cache－主存系统的访问效率e。

解：（1）H= $\frac{5000}{5000+200}\approx 96\%$

（2）TA= H×T1+(1-H)×T2

=0.96×40ns+(1-0.96)×160ns=44.8ns

（3）e= $\frac{T_1}{T_A}$ = 40/44.8 =89.3%

■例6-3：假设$T_2=5T_1$，在命中率H为0.9和0.99两种情况下，分别计算存储系统的访问效率。

解：

当H=0.9时，$e_1=1/(0.9+5(1-0.9))=0.72$

当H=0.99时，$e_2=1/(0.99+5(1-0.99))=0.96$

提高存储系统速度的两条途径：

一是提高命中率H。

二是两个存储器的速度不要相差太大。

其中第二条有时做不到(如虚拟存储系统)，主要依靠提高命中率。

■例6-4：在虚拟存储系统中，两级存储器的速度相差特别悬殊$T_2=10^5\ T_1$。如果要使访问效率e=0.9，问需要有多高的命中率？

解：根据公式可得

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

H = 0.999999

能获得如此高的命中率吗？

提高命中率的方法：程序局部性原理

◦不命中时，将M_2中相邻的几个单元中的数据一起取出送入M_1。

预取后的命中率：$H'=\frac{H+n-1}{n}$

其中：

H－原来的命中率

n－数据块大小×数据重复使用次数

不命中率降低 n 倍

■证明：

采用预取技术之后，不命中率降低n倍：

$$H' = 1 - \frac{1-H}{n} = \frac{H+n-1}{n}$$

也可以采用另外一种证明方法：在原有命中率计算公式中，把访问次数扩大到n倍，这时，由于采用了预取技术，命中次数为：$nN_1+(n-1)N_2$,不命中次数仍为N_2，因此新的命中率为：

$$H' = \frac{nN_1+(n-1)N_2}{nN_1+nN_2} = \frac{N_1+(nN_1+nN_2)-(N_1+N_2)}{nN_1+nN_2} = \frac{H+n-1}{n}$$

■例6-5：在一个Cache存储系统中，当Cache的块大小为一个字时，命中率H=0.8；假设数据的重复利用率为5，计算块大小为 4 个字时，Cache存储系统的命中率是多少？假设$T_2=5T_1$，分别计算访问效率。

■例6-5：解

n=4×5=20, 采用预取技术后, 命中率提高到：

$$H' = \frac{H+n-1}{n} = \frac{0.8+20-1}{20} = 0.99$$

Cache块为1个字大时, H=0.8, 访问效率为：

$$e_1 = 1/(0.8+5(1-0.8)) = 0.55$$

Cache块为4个字大时, H=0.99, 访问效率为：

$$e_2 = 1/(0.99+5(1-0.99)) = 0.96$$

■例6-6：在一个虚拟存储系统中，$T2 = 10^5 T1$，原来的命中率只有0.8，如果访问磁盘存储器的数据块大小为4K字，并要求访问效率不低于0.9，计算数据在主存储器中的重复利用率至少为多少？

■解：假设数据在主存储器中的重复利用率为m，则：

$$0.9 = \frac{1}{H'+(1-H')\cdot 10^5}, \quad H' = \frac{0.8+4096m-1}{4096m}$$

解方程组得m=44，即数据在主存储器中的重复利用率至少为44次。

常规的主存是单体单字存储器，只包含一个存储体。在高速的计算机中，普遍采用并行存储系统，即在一个存取周期内可以并行读出多个字，依靠整体信息吞吐率的提高，以解决CPU与主存之间的速度匹配问题。

6.2.1 交叉访问存储器

特点：
- 在一个存储周期内可以访问到多个数据，从而提高主存频宽。

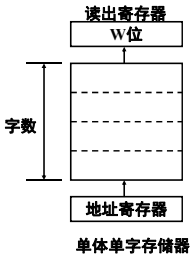
类型：
- 单体多字
- 多体单字交叉存储器
- 多体多字交叉存储器

单体单字存储器

有一个字长为W位的存储器，一次可以访问一个存储器字。

若存储器字长与CPU字长相等，其最大频宽为：

$B_M = W / T_M$



单体单字存储器

提高存储系统性能的途径：并行

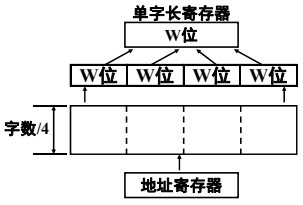
要提高频宽，只有设法提高存储器字长 W 才行。有三种方案：
- 单体多字存储器
- 多体单字交叉存储器
- 多体多字交叉存储器

单体多字存储器

增加存储器的字长（m倍），这样在一个主存周期内就可以读出多个CPU字。

其最大频宽为：

$B_M = m \times W / T_M$



单体多字存储器

缺点：
- **需要位数足够多的寄存器；**
- **多次访问总线。**

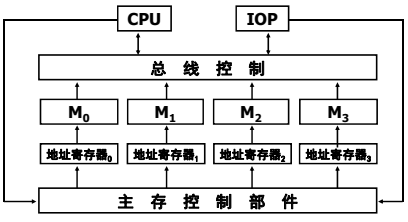
多体单字存储器

由多个容量较小、字长较短的相同存储器芯片组成。

每个芯片都有自己的地址译码、读/写驱动等外围电路。

每个存储体字长都是一个CPU字的宽度，让多个（m个）字长为W位的存储体并行工作，一次可以访问多个存储器字。

其最大频宽为：$B_M = m \times W / T_M$



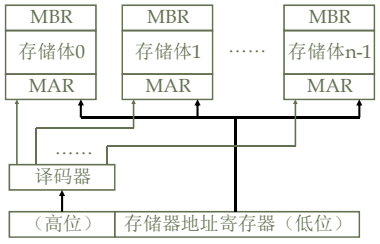
多体单字交叉存储器

优点：
- 实际频宽比单体多字方式高，但总价格和器件的数量相差不多。
- 并行访问不同存储体。

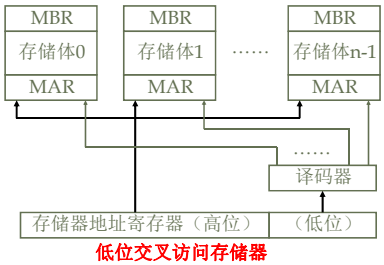
缺点：
- 访问冲突大

1. 取指冲突
2. 读操作数冲突
3. 写数据冲突
4. 读写冲突

为减少分体冲突，CPU字在主存中可按模m交叉编址。分为：
- 高位交叉
  - 实现方法：用地址码的高位部分区分存储体号
  - 主要目的：扩大存储器容量
- 低位交叉
  - 实现方法：用地址码的低位部分区分存储体号
  - 主要目的：提高存储器访问速度



高位交叉访问存储器



低位交叉访问存储器

低位交叉访问存储器 - 8个存储体

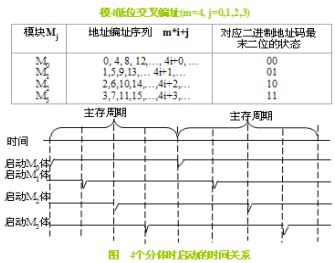


图 4个分体时启动的时间关系

同时给出多个地址，同时访问不同存储体；分时使用总线。适合流水线处理。

低位交叉访问存储器

多体多字存储器

将多体单字存取与单体多字存取结合，进一步提高了频宽。

将上述能并行读出多个CPU字的主存系统称为并行存储系统。

6.2.2 双端口存储器

■双口RAM是指同一个存储器具有两组相互独立的读写控制电路，是一种高速工作的存储器。它有两个独立的端口，分别具有各自的地址线、数据线和控制线，可以对存储器中任何位置上的数据进行独立的存取操作。

■双口RAM的核心部分是用于数据存储的存储器阵列，可为左、右两个端口所共用。当两个端口的地址不相同时，在两个端口上进行读写操作，一定不会发生冲突。当任一端口被选中驱动时，就可对整个存储器进行存取，每一个端口都有自己的片选控制和输出驱动控制。

■当两个端口同时存取存储器的同一存储单元时，就会因数据冲突造成数据存储或读取错误。两个端口对同一主存操作有4种情况：

① 两个端口不同时对同一地址单元存取数据；

② 两个端口同时对同一地址单元读出数据；

③ 两个端口同时对同一地址单元写入数据；

④ 两个端口同时对同一地址单元，一个写入数据，另一个读出数据。

■在第①、第②种情况时，两个端口的存取不会出现错误，第③种情况会出现写入错误，第④种情况会出现读出错误。为避免第③、④种错误情况的出现，双口RAM设计有硬件“$\overline{BUSY}$”功能输出，其工作原理如下：当左、右端口不对同一地址单元存取时，$\overline{BUSY}_R$ =H，$\overline{BUSY}_L$ =H，可正常存储。

■当左、右端口对同一地址单元存取时，有一个端口的$\overline{BUSY}$ =L，禁止数据的存取。此时，两个端口中，哪个存取请求信号出现在前，则其对应的$\overline{BUSY}$ =H，允许存取；哪个存取请求信号出现在后，则其对应的 $\overline{BUSY}$ =L，禁止其写入数据。

■需要注意的是，两端口间的存取请求信号出现时间要相差在5ns以上，否则仲裁逻辑无法判定哪一个端口的存取请求信号在前；在无法判定哪个端口先出现存取请求信号时，两根控制线不会同时为低电平。这样，就能保证对应于 $\overline{BUSY}$ =H的端口能进行正常存取，对应于 $\overline{BUSY}$ =L的端口不存取，从而避免双端口存取出现错误。

高速缓冲存储器（Cache）与主存一起构成Cache存储系统。

在Cache存储系统中，Cache块的大小一般只有十几到几十字节，Cache存储器一旦发生块失效时，程序是不能切换的，CPU此时只能等待着从主存中将所需的块调入Cache。所以，Cache存储系统的地址映像和变换、替换算法都全部采用硬件来实现。

6.3.1程序局部性

定义：程序在执行时所用到的指令和数据的地址分布不是随机的，而是相对簇聚的。它包括时间局部性和空间局部性。

时间局部性

- 最近的未来要用到的信息可能就是当前正在使用的信息——这是由程序的循环造成的。

空间局部性

- 最近的未来要用到的信息可能就是当前信息的相邻信息——这是由程序的顺序执行造成的。

基于局部性，可以得出如下结论：

- M1 不必存放整个程序，只需存放近期使用过的块或页即可（时间局部性）。
- 调入时，一并把数据所在的块或页一起调入（空间局部性）。

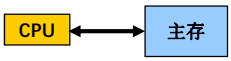
预判的准确性是存储层次设计好坏的主要标志，很大程度上，取决于所使用的算法和地址映像与变换方式。

6.3.2 高速缓冲存储器

高速缓冲存储器（Cache）是一种小容量存储器，由快速的SRAM组成，直接制作在CPU芯片内，访问速度可以与CPU的速度相匹配。Cache用来存放当前最急需处理的程序和数据。

CPU-主存瓶颈

高速计算机的性能通常受到主存带宽和响应时间的限制。



●响应时间(Latency)：一次访存所需要的时间。

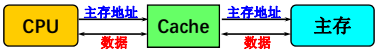
主存访存时间 >> 处理器机器周期

●带宽(bandwidth)：单位时间内的访存次数。

假设每条指令需要一个机器周期，一条指令需要访问主存*m*次，意味着每个机器周期需要访存*m*次。

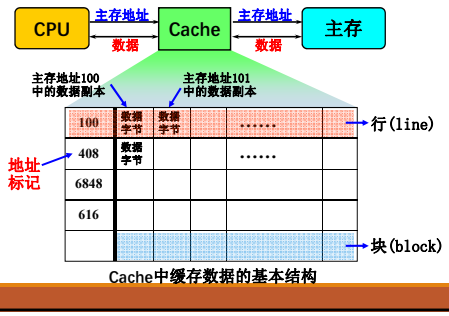
解决速度问题方法：

. 在CPU和主存之间设置速度快、容量小的高速缓冲存储器(Cache)。依据程序局部性原理，将未来要用到的指令或数据从低速主存预取到高速cache中，从而减少平均响应时间，提高平均访问速度。

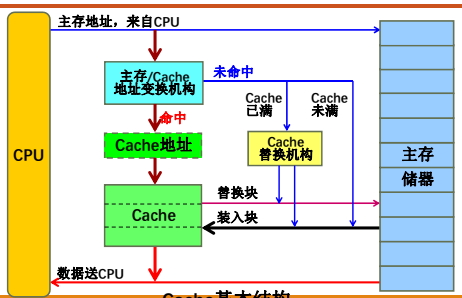


6.3.3 Cache存储系统基本结构

在Cache存储系统中，Cache和主存都被分成若干个大小相等的块，每块由若干字节组成。由于Cache的容量远小于主存的容量，所以Cache中的块数要远少于主存中的块数，它保存的信息只是主存中最急需执行的若干块的副本。如果Cache命中，把主存地址变换成Cache地址，直接访问Cache。如果Cache失效（不命中），用主存地址访问主存，从主存中读出一个字送往CPU，同时，把包括该字在内的一整块都装入Cache。



Cache中缓存数据的基本结构



Cache基本结构

Cache工作原理：写入

将CPU给出的主存地址
变换为Cache地址，搜索Cache

在Cache中找到
（命中）

写Cache，写主存
（存在一致性问题）

在Cache中未找到
（未命中）

写主存
（与Cache无关）

Cache特点

■Cache与CPU采用相同工艺；

■地址映象、变换、替换、调度等由专门的硬件实现；

■ Cache靠近CPU或就放在CPU中，以减少与CPU之间的传输延迟；

■ Cache—主存之间的信息交换对所有程序员都透明；

Cache访问主存的优先级高于其他系统访问主存的优先级；

■除了Cache和CPU有直接的通路外，主存和CPU也有直接的通路，可以实现读直达和写直达；

6.3.4 地址映像和变换

Cache系统须解决三个问题：

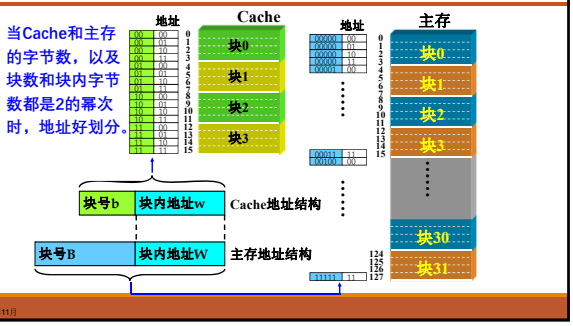
1. 定位问题
   - 将主存中的数据装入cache的哪个位置；
   - 如何知道主存中的数据已经装入cache（即是否命中）；
   - 如果命中，如何形成Cache地址并访问主存。
2. 替换问题
   - 若未命中或失效，需将数据从主存调入Cache；
   - 若Cache满，则按何种算法将Cache中的数据替换出去。
3. 数据一致性问题
   - 如何保证Cache内容与主存内容的一致。

地址映像是把存放在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。

地址变换是当程序已经装入到Cache之后，在实际运行过程中，把主存地址变换成Cache地址。

为便于进行地址的映象和变换，也便于替换和管理，把Cache和主存等分成相同大小的块，这样，Cache—主存地址映像就演变为主存中的块如何与Cache中的块相对应。

当Cache和主存的字节数，以及块数和块内字节数都是2的幂次时，地址好划分。



可以采用的地址映像方法有很多。选择依据：
- 地址变换的硬件要容易实现；
- 地址变换的速度要快；
- 主存空间利用率要高；
- 发生块冲突的概率要小。

块冲突：

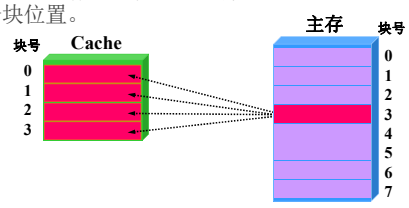
主存中的块要调入Cache中的某个位置，但该位置已经被其他主存块所占用。

在Cache—主存存储层次，典型的地址映像与变换方法主要有：

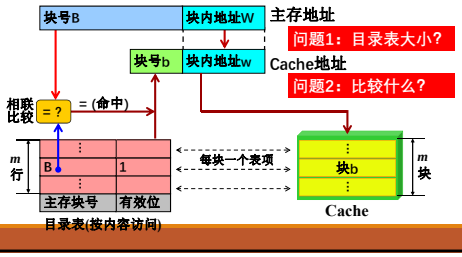
1. 全相联映像和变换
2. 直接映像和变换
3. 组相联映像和变换

（1）全相联映像和变换

映像规则：主存中的任意一块都可以装入到Cache中的任意一个块位置。



地址变换：采用相联存储器构成的目录表，以硬件方式实现。



优点：

- 块冲突概率最低；
- Cache空间利用率最高。

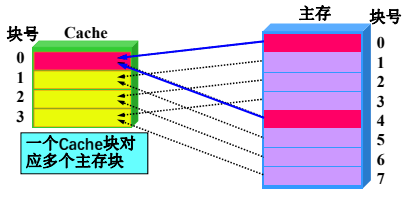
缺点：

- 需要一个相联存储器，其代价很高。
- 相联比较所花费的时间将影响Cache的访问速度。

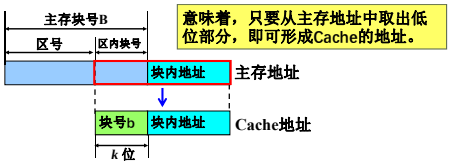
（2）直接映像及其变换

映像规则：主存中的每一块只能装入到Cache内唯一一个指定的块位置。

为便于地址变换，设：
Cache块号b＝(主存块号B) mod (Cache块数)
则：0=0|4 mod 4, 1=1|5 mod 4, 2=2|6 mod 4, 3=3|7 mod 4

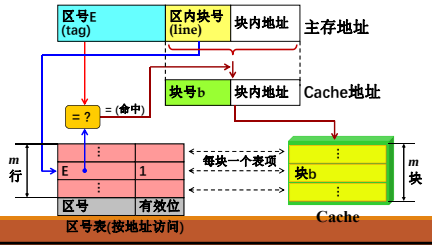


地址变换：
若Cache块号b＝(主存块号B) mod (Cache块数)，
设Cache块数＝2^k，当表示为二进制数时，Cache块号b与主存块号B的低 k 位完全相同。



意味着，只要从主存地址中取出低位部分，即可形成Cache的地址。

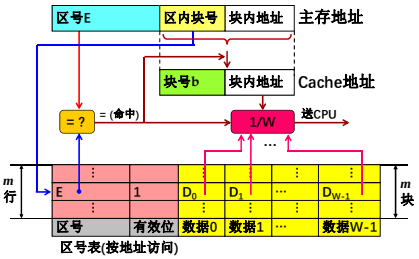
地址变换：设置一个按地址访问的区表存储器（称之为区号表），存放Cache中每一块目前被主存中哪个区的对应块所占用。



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，实际上不做地址变换。

缺点：
- 块冲突概率很高。

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。



（3）组相联映像及其变换

组相联映像将主存和Cache按同样大小划分成块，Cache空间等分成大小相同的组，组里有若干个块。让主存中的任何一块只能被放置到Cache中唯一的一个指定组，然后用全相联映像装入Cache中对应组的任何一块位置上，即组间采取直接映像，而组内采取全相联映像。

组相联实际上是全相联映像和直接映像的折衷方案。

当组内块数 S=Cache块数时，组相联就变成了全相联；

当组内块数 S=1 时，组相联变成了直接映像；

S越大，冲突越少，地址变换就越复杂；

S越小，冲突越多，地址变换就越容易。

假设Cache空间分成C_g组（$C_g=2^g$），每组为G_b块（$G_b=2^b$）。

主存地址分为三部分：标记、组号、块内地址；

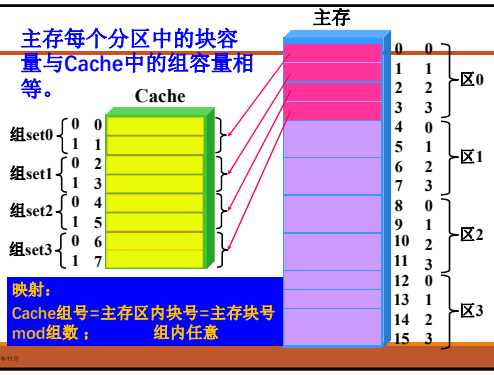
Cache地址分为三部分：组号、组内块号、块内地址。

主存地址的组号由G来表示，它的宽度Cache地址的组号g是一致的。

组相联映象方式Cache分组（set），而主存不再分组。

主存除了分块之外，还按照Cache的组（set）容量（个数）分区。主存每个分区中的块容量与Cache中的组容量相等。

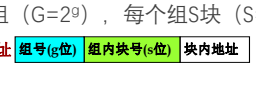
主存每个分区中的块容量与Cache中的组容量相等。



映像规则：

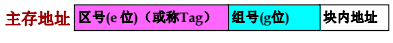
- 主存块到Cache组set之间采用直接映像方式（即1个主存块只能映像到1个Cache组）；
- 在对应的组内部采用全相联映像方式（即1个主存块可以放入指定Cache组内任何1个块位置——组内随便放）。

将Cache分成G组（$G=2^g$），每个组S块（$S=2^s$）：

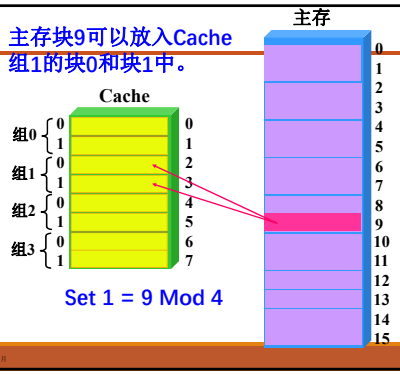


组内块数S称为相联度。
每组有S块的组相联称为S路组相联。

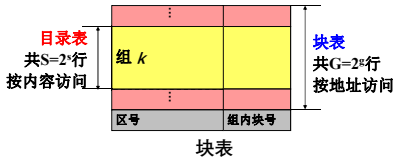
将主存按同样大小划分成块，并分为E个分区（$E=2^e$）：

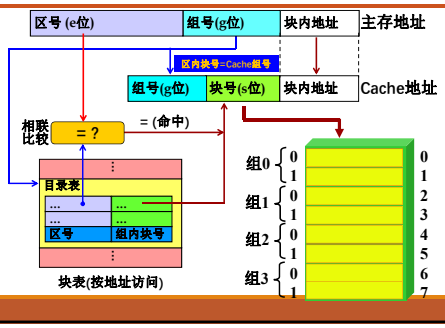


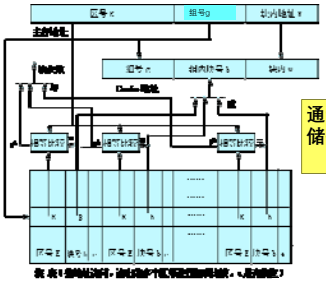
主存块9可以放入Cache组1的块0和块1中。



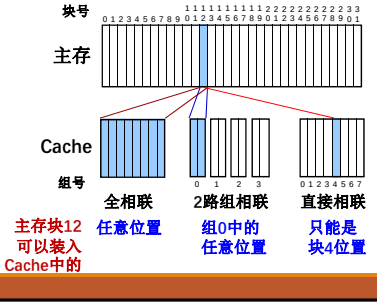
地址变换：根据区内块号按地址访问块表，取得目录表地址，按内容（区号）访问目录表。







通过“单体多字存储器”来并行查询。



■例6-7：某计算机的Cache共有16块，采用2路组相联映像方式（即每组2块）。每个主存块大小为32字节，按字节编址。主存129号单元所在主存块应装入到的Cache组号是多少？

解：由于每个主存块大小为32字节，按字节编址。根据计算主存块号的公式，主存块号=

$\lfloor \text{主存地址} / \text{块大小} \rfloor = \lfloor \frac{129}{32} \rfloor = 4$，所以主存129号单元所在的主存块应为第4块。若Cache共有16块，采用2路组相联映像方式，可分为8组。根据组相联映像的映像关系，主存第4块转入Cache第4组。

优点：
- 块冲突概率比直接映像低得多；
- Cache空间利用率也比直接映像提高；
- 块失效率明显降低。

缺点：
- 实现难度和造价要比直接映像方式高。

6.3.5 Cache替换算法

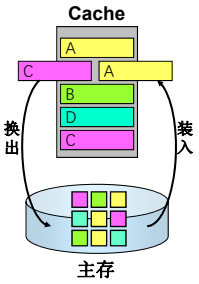
块失效是该块未装入Cache，需要从主存中调块。块冲突（块争用）是两个以上的主存块想要进入Cache中同一个块位置的现象。块失效时不一定发生块冲突，但块冲突一定是由块失效引起的。

当块失效后，需要从主存将一个主存块调入Cache，而此时Cache已满，就会发生块冲突。只有腾出Cache中某个块后才能接纳由主存调入的新块，选择Cache中哪个块作为被替换的块，就是替换算法要解决的问题。

在Cache存储器中，通常容量远小于主存容量。

问题：如果已满，同时发生Cache失效，此时，只有强制腾出Cache中的某个块，以存放由主存调入的新块。

根据何种算法、应将哪个Cache块替换出去？



确定替换算法的原则：
- 是否有高命中率
- 是否易于实现，硬件成本是否低

典型替换算法：

随机（RAND）算法
- 用随机数产生器来形成被替换页的页号。

先进先出（FIFO）算法
- 选最早装入Cache的块作为被替换的块。

近期最少使用（LFU）算法
- 选择近期最少访问的块作为被替换的块。

近期最久未使用（LRU）算法（最常用）
- LFU的变形；
- 选择近期最久未被访问过的块作为被替换的块。

在Cache存储系统中，实际上有可能采用的只有FIFO和LRU两种算法。

为保持较高的命中率，绝大多数替换算法是根据程序过去的行为预测其未来的行为，从而确定被替换的块。

1.随机（RAND）算法
- 用随机数产生器来形成被替换块的块号。
- 优点：简单、易于实现
- 缺点：命中率很低
  - 因为没有利用Cache使用的“历史”，反映不了程序的局部性

2.先进先出（FIFO）算法

- 选最早装入Cache的块作为被替换的块。
- 优点：简单、易于实现
  - 在Cache块表中为每个实块设置一个计数器字段。调入新块时，新装入块的计数器为0，其他块则加1。替换计数器值最大的那个块
- 缺点：不能正确反映程序局部性。最先调入的可能是最经常使用的

3.近期最少使用（LFU）算法

- Least Frequently Used
- 选择近期最少访问的块作为被替换的块。
- 优点：比较正确地反映了程序的局部性
- 缺点：实现比较困难
  - 需要为每个块配置一个很长的计数字段

4.近期最久未使用（LRU）算法

- Least Recently Used
- LFU的变形
- 选择近期最久未被访问过的块作为被替换的块。
- 该算法将LFU中的“多”和“少”简化成“有”和“无”，实现方便。

5.优化替换（OPT）算法

- LRU和FIFO都是根据块使用的“历史”情况来估计未来的块使用情况。
- 如果能根据未来的块使用情况，将未来的近期内不用的块替换出去，则命中率一定很高。
- OPT就是这样一种算法。

5.优化替换（OPT）算法（续）

- 在时刻 t 找出主存中每个块将要用到的时刻 t_i。
- 选择t_i - t 最大的块进行替换。
- 但其实现是不现实的
  - 需要让程序运行两次。第一次是得到块地址流，获得使用信息；第二次是正常执行。
- 所以，该替换算法只是一个理想化的算法，可以作为评价其他替换算法的标准。

替换算法一般是通过用典型的块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。当然影响命中率的因素除了替换算法外，还因地址流、块大小、Cache容量等不同而不同。

■例6-8：**设有一个程序，有1至5共5块，执行时的块地址流(即执行时依次用到的程序块的块号)为：**

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

假设分配给这个程序的Cache存储器共有3个块。分析FIFO、LRU、OPT块替换算法对这3块Cache的使用情况，包括调入、替换和命中率等。

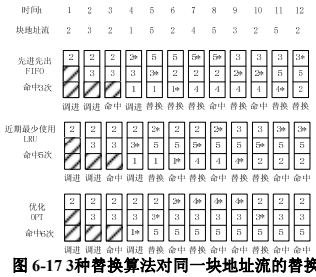
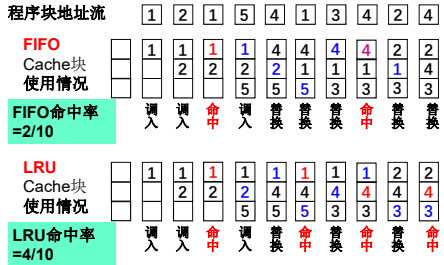


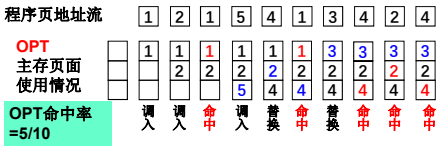
图 6-17 3种替换算法对同一块地址流的替换过程

■例6-8补充：一个程序共有5个块组成，程序执行过程中的块地址流如下：

1，2，1，5，4，1，3，4，2，4

假设分配给这个程序的Cache存储器共有3个块。分析FIFO、LRU、OPT块替换算法对这3块Cache的使用情况，包括调入、替换和命中率等。(注：与书中块地址流不同，带动画）





由上图可见，FIFO的命中率最低，而LRU的命中率非常接近于OPT。

命中率也与页地址流有关。例如一个循环程序，当所需页数大于分配给它的页数时，无论FIFO还是LRU的命中率都明显低于OPT。

■加例：一个循环程序，依次使用1、2、3、4四个块，分配给这个程序的Cache块数为3个。请分析FIFO、LRU、OPT块替换算法对Cache块的使用调度情况。

程序块地址流：1 2 3 4 1 2 3 4 1 2

FIFO
Cache块使用情况

1	1	1	4	4	4	3	3	3	2
	2	2	2	1	1	1	4	4	4
		3	3	3	2	2	2	1	1
调入	调入	调入	替换	替换	替换	替换	替换	替换	替换

FIFO命中率=0/10

LRU
Cache块使用情况

1	1	1	4	4	4	3	3	3	2
	2	2	2	1	1	1	4	4	4
		3	3	3	2	2	2	1	1
调入	调入	调入	替换	替换	替换	替换	替换	替换	替换

LRU命中率=0/10

程序块地址流：1 2 3 4 1 2 3 4 1 2

OPT
Cache块使用情况

1	1	1	1	1	1	1	1	1	2
	2	2	2	2	2	3	3	3	3
		3	4	4	4	4	4	4	4
调入	调入	调入	替换	命中	命中	替换	命中	命中	替换

FIFO命中率=4/10

从上述两个例子中可以看到：

- FIFO命中率最低。
- 对某种块地址流，LRU算法也可能和FIFO算法一样糟，例如，对于循环程序，会发生所不希望的连续块失效——颠簸现象。
- 颠簸：

  下次就要使用的块本次被替换出去而发生的连续块失效的现象。

命中率与块地址流有关，也与分配给该道程序的Cache块数有关。

通常，随着Cache块数的增加，主存块进入Cache的机会就越多，命中率就可能越高。

问题：会提高吗？

对于LRU算法，命中率随着Cache块数的增加而增加

对于FIFO算法，命中率可能随着Cache块数的增加，反而下降

堆栈型替换算法

定义：

对任意一个程序的块地址流作两次Cache块数分配，分别分配 m 个Cache块和 n 个Cache块，并且有 $m \leq n$。如果在任何时刻t，Cache块集合 Bt 都满足关系：

$$Bt(m) \subseteq Bt(n)$$

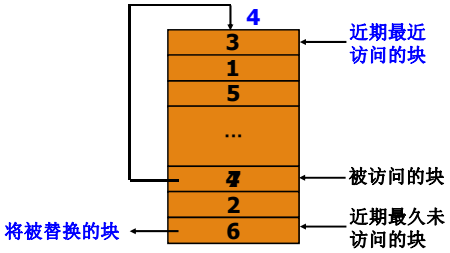
则这类算法称为堆栈型替换算法

堆栈型替换算法的命中率随分配的Cache块数的增加而单调上升，至少不会下降。

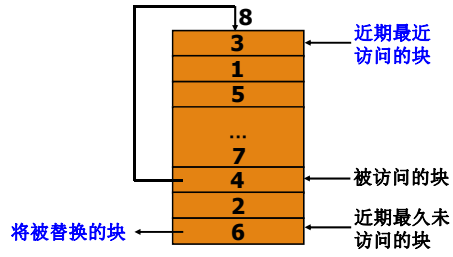
可以证明：

- LRU是堆栈型替换算法
- OPT是堆栈型替换算法
- FIFO不是堆栈型替换算法

可以利用堆栈实现堆栈型替换算法
- 堆栈容量：2^b
- 过程：检查要访问的Cache块号是否在堆栈中？
- 是：将该块调置栈顶，并把该项上面的项下推一行，该项下面的不动
- 否：新块压入堆栈，弹出栈底



要访问的块4已在Cache时的情况



要访问的块8不在主存时的情况

结果：
栈顶恒为近期最近访问过的页号。
栈底恒为近期最久没有访问过的页号。
堆栈要求：
- 有相联比较功能
- 全下推或部分下推功能
- 从中间抽走一项的功能

利用堆栈处理技术的分析模型可以对堆栈型替换算法命中率及应该分配的Cache块数进行评估。

■例6-9：一个程序共有5个块组成，程序执行过程中的块地址流如下：

1，2，1，5，4，1，3，4，2，4

假设分配给这个程序的Cache存储器共有3个块。采用堆栈法的LRU块替换算法对这3块Cache的使用情况，包括调入、替换和命中率等。

程序块地址流	1	2	1	5	4	1	3	4	2	4
LRU Cache块使用情况	1	2	1	5	4	1	3	4	2	4
		1	2	1	5	4	1	3	4	2
				2	1	5	4	1	3	3
	调入	调入	命中	调入	替换	命中	替换	命中	替换	命中

LRU命中率=4/10

Cache替换算法与实现

Cache通常使用组相联或直接映像，而不采用全相联映像。

当所要访问的块不在Cache中时，则发生块失效。

当所能装入的Cache块都已被装满时，则出现块冲突，必须进行块替换。

替换算法：确定被替换的主存块。

实现：全部用硬件

两种方法：

- 堆栈法：使用硬堆栈
- 比较对法：使用逻辑电路、触发器等

比较对法

用触发器（硬联逻辑）实现。

基本思路：

- 让各个块成对组合，用一个触发器的状态来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。

例如：有A、B、C 三块，之间的组合共有 $C_3^2 = 3$ 组合：AB、AC、BC

各对内块的访问顺序分别用两态“触发器”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。

如果C为最久未被访问过的块，三个块的排列顺序有两种可能：

块A、块B、块C

块B、块A、块C

根据逻辑关系，很容易写出块C最久没有被访问过表达式：

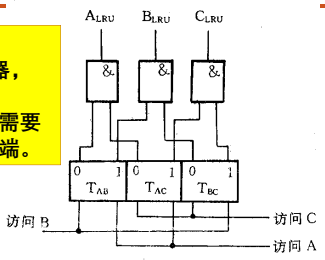
$$C_{LRU} = T_{AB} \cdot T_{AC} \cdot T_{BC} + \overline{T_{AB}} \cdot T_{AC} \cdot T_{BC} = T_{AC} \cdot T_{BC}$$

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

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

3个块时：

- **3个触发器，**
- **3个与门，**
- **每个与门需要两个输入端。**



用比较对法实现LRU算法

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	…	$p(p-1)/2$
门数	3	4	8	16	64	256	…	p
门输入端数	2	3	7	15	63	255	…	$p-1$

当每组的块容量为8块或8块以上时，所要的触发器个数及与门输入端个数很多，硬件实现的成本很高。

随着每组中的块容量增加，所需要的触发器的个数及与门的个数成平方关系增加。

6.3.6 Cache更新策略

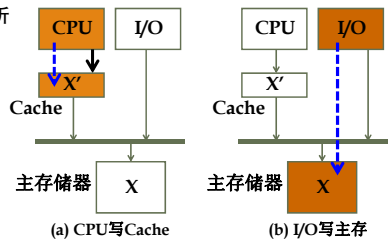
Cache－主存存储层次对所有程序员透明

Cache内容是主存内容的一小部分副本

但Cache内容有可能与主存内容不一致：

- CPU更新（写）了Cache而未更新主存；
- I/O更新了主存而未更新Cache；

1. Cache透明性分析



Cache与主存不一致的两种情况

必须解决Cache的一致性问题

解决问题的关键：写Cache时如何更新主存的内容。

“写”操作所占的比例

- Load指令：26%
- Store指令：9%

“写”在访问Cache操作中所占的比例：

- 9%/(26%＋9%)≈25%

大概率事件优先原则：优化Cache读操作

Amdahl定律：不可忽视“写”速度

“写”问题

- 读出标识，确认命中后，对Cache写 （串行操作）
- Cache与主存内容的一致性问题

写策略就是要解决：何时更新主存问题

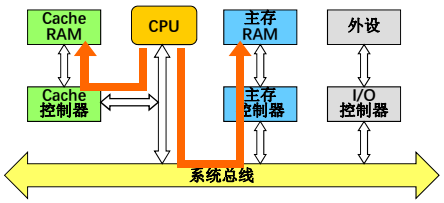
Cache一致性算法：2种写策略

写直达法（Write-through）

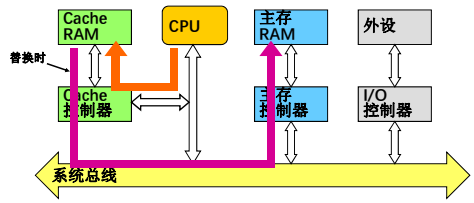
- CPU在执行写操作时，利用直接通路，把数据同时写入Cache和主存

写回法 （Write-Back）

- 也称为抵触修改法
- CPU数据只写入Cache，不写入主存
- 为每个Cache块设置“修改位”
- 仅当替换时，才把修改过的Cache块写回到主存



写直达法：数据同时写入Cache和主存



写回法：数据只写入Cache，替换时才写入主存

两种“写”策略的比较

	写直达法	写回法
可靠性	好于写回法	块替换前仍存在一致性问题
与主存的通信量		少于写直达法达10多倍
控制复杂性	比写回法简单	
硬件实现代价		比写直达法低得多

与主存的通信量

例如：写操作占总访存次数的20%，Cache命中率为99%，每块4个字。当Cache发生块替换时，有30%块需要写回主存，其余的因未被修改过而不必写回主存。则对于WT法，写主存次数占总访存次数的20%。而WB法为(1–99%)×30%×4=1.2%。因此，WB法与主存的通信量要比WT法少10多倍。

“写”调块

“写”操作必须在确认是命中后才可进行

当出现写不命中时，是否需要将主存块调入Cache?

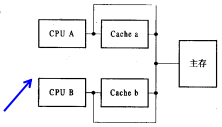
两种方法：

- 不按写分配法：在写Cache不命中时，把所要写的字直接写入主存，不调块；
- 按写分配法：在写Cache不命中时，写入主存，并把单元所在块调入Cache；

写策略与调块

一般：

- 写直达法 —— 不按写分配
- 写回法 —— 按写分配



单处理机系统

- 大多采用写回法

共享主存的多处理机

- 为保证各处理机经主存交换信息时不出错，较多采用写直达法
- 多Cache一致性算法

“写”缓冲器

写回法和写直达法都需要少量缓冲器

写回法中存放将要写回的块，不必等待写回主存后才进行Cache取

写直达法中存放要写回主存的内容，减少CPU等待写主存的时间

缓冲器对Cache－主存透明

两种写策略

写直达法（Write through）

- 优点：易于实现，容易保持不同层次间的一致性
- 缺点：速度较慢

写回法

- 优点：速度快，减少访存次数
- 缺点：一致性问题

2. Cache的预取算法

Cache的命中率对机器的性能影响很大

如何预取提高Cache的命中率?

命中率与很多因素有关:

- 预取算法
- 块大小
- 预取开销等

预取算法有如下几种:

- 按需取：在出现Cache不命中时，把一个块取到Cache中来
- 恒预取：无论Cache是否命中，都把下一块取到Cache中
- 不命中预取：当Cache不命中，把本块和下一块一起取到Cache中

主要考虑因素：

- **命中率的提高；**
- **Cache与主存之间通信量的增加。**

虚拟存储系统由主存储器（DRAM）和联机工作的辅助存储器（磁盘存储器）共同组成。主存使用DRAM，容量小，速度快；磁盘容量大，价格低。应用程序员将其看成是一个存储器，可使用很大的虚拟空间。与前述的Cache存储系统相似，虚拟存储系统也必须考虑交换块的大小、地址映像、替换问题、写一致性问题等。

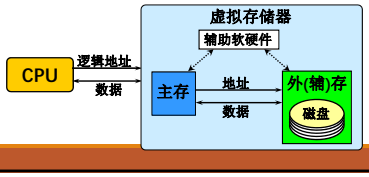
Cache与虚拟存储系统的区别

比较项目 \ 存储层次	Cache存储系统	虚拟存储系统
目　的	弥补主存速度的	弥补主存容量的不足
存储管理实现	由专用硬件实现	软件、硬件实现
访问速度的比值（第一级和第二级）	几比一	几百比一
典型的块(页)大小	几十个字节	几百到几千个字节
CPU对第二级的访问方式	可直接访问	均通过第一级
失效时CPU是否切换	不切换	切换到其他进程
透明性	对所有程序员透明	仅对应用程序员透明

- 虚拟存储器是一个大容量存储器的逻辑模型，它指的是主存－辅存层次；
- 它借助于磁盘等外部存储器扩大主存容量，以透明的方式给用户提供了一个比实际主存空间大得多的程序空间，允许应用程序员使用比实际主存空间大得多的程序空间（虚拟存储空间）来访问主存。

主要解决容量问题：容量和价格是外(辅)存的，速度是主存的。

允许应用程序员使用比实际主存空间大得多的程序空间（虚存空间）来访问主存。在系统软件和辅助硬件的管理下，就象拥有一个单一的、 可直接访问的大容量主存储器。



1.虚拟存储器特点

- 依赖于程序的局部性原理，提高了主存利用率；
- 对应用程序员透明（对系统程序员基本不透明）。应用程序员不必再做存储器分配工作，由系统提供的调入功能和替换功能等定位机制自动完成；
- 程序不再受现有物理内存空间的限制，每道程序都有独立的程序空间（虚拟存储空间），编程变得更容易。程序的执行独立于存储器的容量和配置；
- 多用户可以共享存储器，使更多的程序能够进入主存运行；
- 提供了主存保护机制。

2.地址空间（3种）

程序空间与程序地址：
- 程序空间：程序员用来编写程序的地址空间。也称为虚空间、虚拟存储空间、逻辑空间。
- 程序地址：程序员编写程序时所使用的地址，也称为虚地址、逻辑地址。

主存空间与主存地址：
- 主存空间：主存储器的地址空间。也称为主存地址空间、主存物理空间或实存地址空间。
- 主存地址：主存储器的地址。也称为物理地址、实地址 。

辅存空间与辅存地址：
- 辅存空间：辅助存储器的地址空间。也称为辅助地址空间、辅助物理空间或实存地址空间。
- 辅存地址：辅助存储器的地址。也称为辅存物理地址、辅存实地址。

6.4.1 虚拟存储器的管理方式

在虚拟存储器中，虚存空间比实存空间大得多，因此虚地址无法与实地址一一对应。

虚拟存储器采用地址映像表机构实现程序的动态定位。根据地址映像算法的不同，形成了不同的虚拟存储器管理方式，主要有三种：

- ■页式管理
- ■段式管理
- ■段页式管理

1. 页式管理

基本思想： 将主存空间和程序空间分成页，按页进行调入、调出和管理。

一般原理：

☑将主存空间等分成固定大小的页（称为实页），并按顺序编号。这样，主存实地址 = 实页号 + 页内地址

主存单元的实地址结构

实页号p	页内偏移d

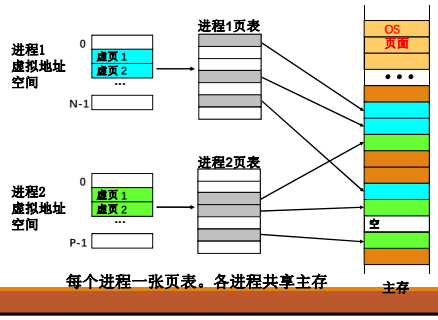
☑将程序空间也等分成固定大小（与实页大小相同）的页（称为虚页），并按顺序编号。这样，程序虚地址 = 虚页号 + 页内地址

程序空间的虚地址结构

虚页号P	页内偏移D

一般原理（续）：

- 程序的起点必须处于一个页面的起点；
- 每个虚页都可以装入主存中的任一实页位置；
- 由于虚存和实存的页面大小相同，所以记录虚实地址的映像关系，只需记录虚页号与实页号的对应关系即可；
- 为每道程序设立一张页表，记录哪一个虚页装入到哪一个实页位置。



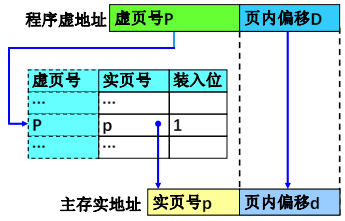
每个进程一张页表。各进程共享主存

页表的基本结构

虚页号	实页号	装入位	修改位	专用位
	⋮			

该字段可无　　每个虚页一行

- ✓**虚页号：** 指明为哪个虚页。当虚页号从0开始顺序编号时，与页表中的行号相对应，则页表中可以不设虚页号。
- ✓**实页号：** 指明该虚页装入到了哪个实页。
- ✓**装入位：** 指明该虚页是否已装入主存。装入位有效时，实页号才有效。
- ✓**专用位：** 指明其他信息，例如是否共享。



利用页表的地址变换

当有多道程序时：

假设系统中最多可以有N道程序，因此有N个页表。

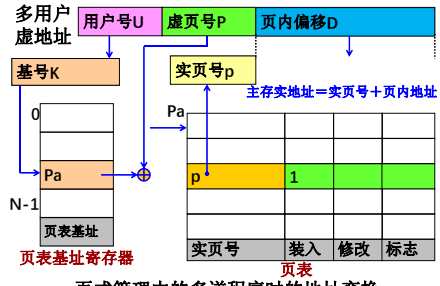
可以设置N个页表基址寄存器指示每道程序所对应的页表的基址（起始地址）。

系统赋给每道程序一个用户号，转换为基号后，指明其使用哪一个页表基址寄存器；

用户虚地址（指令中的地址码）与该用户号结合形成系统的多用户虚地址：

用户号U	虚页号P	页内地址D

多用户虚地址结构



页式管理中的多道程序时的地址变换

优点：
- 主存利用率比较高，只有不到一页的浪费；
- 页表相对比较简单，保存的字段比较少；
- 地址变换速度快，只要查页号之间的对应关系；
- 磁盘管理容易。

缺点：
- 程序的模块化性能不好。页不能表示一个完整的程序功能；
- 页表占用空间多。

2. 段式管理

基本思想：将程序按逻辑意义分成段，按段进行调入、调出和管理。

一般原理：

- 每个程序段都从地址0开始编址；

逻辑地址 | 段号 | 段内偏移

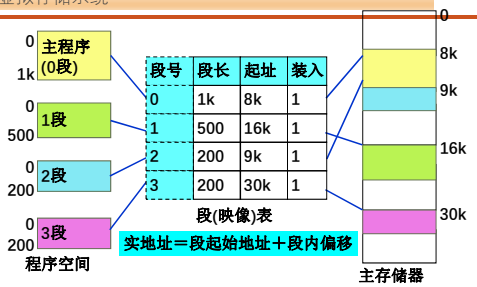
- 每个程序段可以映像到主存的任意位置；
- 每道程序都有一张段（映像）表，以存放各程序段装入主存的起始地址等情况。

段表的基本结构

段号	地址	段长	装入位	访问方式
	⋮			

该字段可无　　每个程序段一行

- ✓段号：指示程序段的编号或段名。当段号从0开始顺序编号时，与段表中的行号相对应，则段表中可以不设段号。
- ✓地址：若该段已经装入主存，指示其在主存的起始地址。
- ✓装入位：指示该段是否已经装入主存。
- ✓段长：指明该段的大小。
- ✓访问方式：指明该段允许的访问方式，如只读、可执行等。



段式管理中的某道程序的地址变换

当有多道程序时：

- 假设系统中最多可以有N道程序，因此有N个段表；
- 可以设置N个段表基址寄存器指示每道程序所对应的段表的基址（起始地址）和段表大小。

段表长度	段表基地址

段表长度：指明段表的行数，即程序的段数；
段表基地址：该道程序的段表在主存中的起始地址。

段表基址寄存器的结构

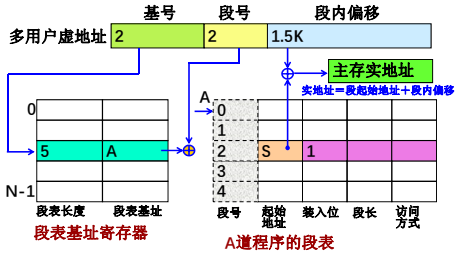
当有多道程序时（续）：

- 系统赋给每道程序一个基号，指明其使用哪一个段表基址寄存器；
- 用户虚地址（指令中的地址码）与该基号结合形成系统的多用户虚地址：

基号	段号	段内偏移

多用户虚地址

- 通过查表即可自动转换成主存的物理地址；



段式管理中的多道程序时的地址变换

优点：

- 程序的模块化性能好。程序分成多个段，可并行编程。
- 便于程序和数据的共享。段按功能划分，主存只装一份。
- 程序的动态链接和调度比较容易
- 易于以段为单位实现存储保护。

缺点：

- 地址变换所花费的时间比较长，做两次加法，查两次表；
- 主存利用率可能下降，因为存在段内零头和段间零头；
- 对辅存（磁盘存储器）的管理比较困难，磁盘是固定长度的块访问的。

3. 段页式管理

段式管理和页式管理各有优缺点。段页式管理将段式管理和页式管理结合起来。

一般原理：

将主存空间等分成固定大小的页（称为实页），并按顺序编号；

将程序先按模块分段，每段再等分成固定大小（与实页大小相同）的页（称为虚页），并按顺序编号（段的长度必须是页长度的整数倍）；

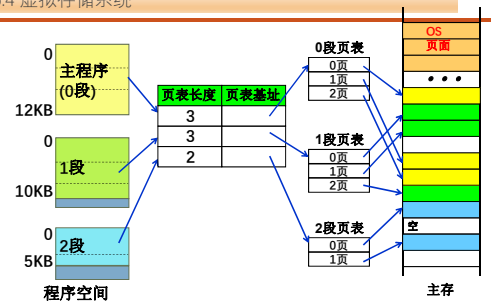
- 每道程序通过一个段表和一组页表进行定位；

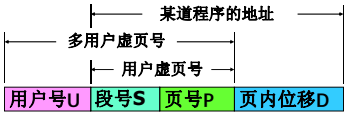
一般原理（续）

段表中的每一行对应一个段。地址字段指示该段的页表在主存中的起始位置，段长字段指示该段的页表的行数（虚页数）；

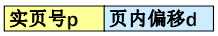
每个段都有一个页表。页表的地址字段指示当该页已经装入主存时，该页在主存中的实页号；

对于多道程序系统，每道程序都需要有一个用户号（需转换为基号），以指明该道程序的段表起点在哪一个段表基址寄存器中。

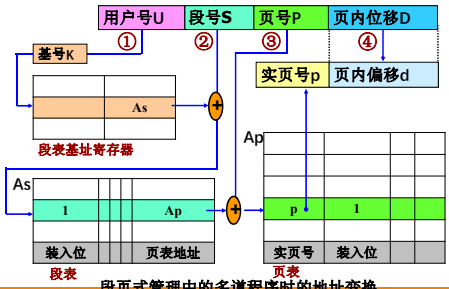




多用户虚地址结构



主存单元的实地址结构



段页式管理中的多道程序时的地址变换

缺点：

- 每一次地址变换至少需要两次查表，即查段表和页表，所以速度慢；
- 表空间较大，可能需要多级页表。

页表级数的计算公式：$g=\left\lceil \frac{\log_2 Nv-\log_2 Np}{\log_2 Np-\log_2 Nd} \right\rceil$

其中：
Np为页面的大小
Nv为虚拟存储空间大小
Nd为一个页表存储字的大小

■例6-10：虚拟存储空间大小Nv = 4GB，页的大小Np = 1KB，每个页表存储字占用4个字节。整个页表共有4M个表项，远大于一个页面，所以需要建立多级页表。计算得到页表的级数：

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

1KB页面，1页只有256个存储字；3级页表。故：256×256×64=4M字。通常把1级页表驻留在主存储器中，2、3级页表只驻留一小部分在主存。

虚拟存储器的管理方式：

三种不同的虚拟存储器管理方式各有优缺点；

三种方式都采用映像表机制实现虚实地址变换；

页式管理目前使用最为普遍。

6.4.2 页式虚拟存储器构成

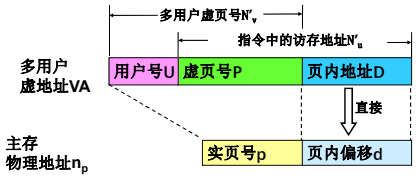
页式虚拟存储系统是采用页式存储管理的主-辅存存储层次。

一般来说，虚拟存储系统中的用户程序空间比实际主存空间大得多。反映在页号上，就是：

虚页号P >> 实页号p

此外，主存空间是被多个用户共享的。

页式虚拟存储系统必须解决：如何把大的多用户虚拟空间压缩到小的主存空间中（即虚实地址变换）



多用户虚地址到主存物理地址的变换

需解决的主要问题：

1. **定位问题**
- **虚存单元与主存单元的对应关系；**
- **如何知道虚存单元中的数据已经装入主存（即是否命中）；**
- **如果命中，如何形成主存物理地址并访问主存。**

2. **替换问题**
- **若未命中或失效，需将数据从辅存调入主存。如何将虚存逻辑地址变换为辅存物理地址；**
- **若主存中无可用位置，则按何种算法将主存中的数据替换出去；**

3. **性能问题**
- **如何提高地址变换速度。**

1. 地址映像与变换

定义：

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P与实主存地址p之间的对应关系。

一个主存地址A由两部分组成，实页号p和页内偏移d。

实页号p	页内偏移d

主存地址A的组成

一个虚地址Av由三部分组成，用户号U、虚页号P和页内偏移D。

用户号U	虚页号P	页内偏移D

多用户虚拟地址Av的组成

■全相联地址映象与变换

地址映象规则：

每道程序的任何一个虚页都可以装入到主存的任何一个实页位置。

仅当同时调入主存的虚页数超过2^p（实页数）时，才会出现实页冲突。

全相联地址映象的实页冲突概率最低。

实页冲突或页面争用：两个或两个以上的虚页想进入主存中的同一实页位置的现象。
页面冲突会使执行效率降低。

地址变换的两种方法：

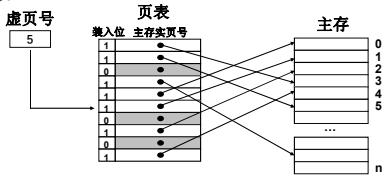
■页表法

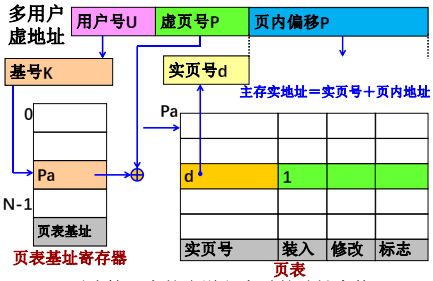
■相联目录法或目录表法

由于是将虚页调入主存，因此将虚地址到主存实地址的变换称为内部地址变换。

（1）页表法

采用页表作为地址映像表，通过查表实现地址变换。





页式管理中的多道程序时的地址变换

页表法的不足：

- 每道程序页表的表长 = 虚页数(2^P)；
- N道程序的页表行数将达到 $N*2^P$，远大于实页数2^P；
- 而装入位为 1 的行数最多只能有实页数（2^P）个，其他行将成为无用的行，大大降低了页表的空间利用率。

压缩页表空间 ➡ 相联目录法

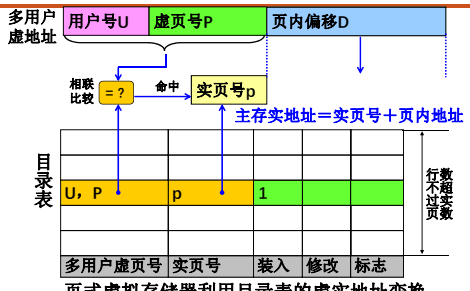
（2）相联目录法

将页表压缩，只存放已装入主存的那些虚页与实页位置的对应关系。该压缩了的页表最多有2^P行，即：表长 = 实页数。

已装入主存的那些虚页的虚页号是分散的，因此无法用虚页号访问该压缩了的页表；

用基号B和用户虚页号P标识虚页号。地址变换时按内容（即基号B和用户虚页号P）访问该表，得到该虚页所装入的实页号；

采用按内容访问的相联存储器构造目录表。



页式虚拟存储器利用目录表的虚实地址变换

（3）页表法与相联目录表法比较

价格

- 页表采用便宜的随机存储器，容量很大；
- 目录表法采用价格较高的相联存储器，容量不会很大；

速度

- 页表法稍快，目录表法慢。

一般，在虚拟存储器中不直接采用目录表法来存储全部虚页号与实页号的对应关系。
但这种思想可以被用来提高地址变换速度。例如，设置一个高速小容量硬件目录表，存放近期最常用的虚－实页对应关系—快表。

■**虚地址—辅存实地址变换**

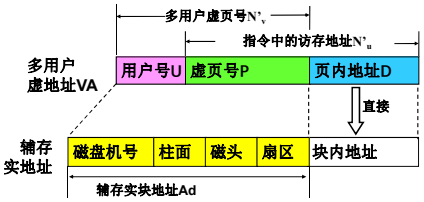
若要把该道程序的虚页调入主存，必须给出该虚页在辅存中的实际地址，即实现虚地址到辅存实地址的变换。

由于是从外部辅存中调页，因此将虚地址到外部辅存实地址的变换称之为外部地址变换。

为提高调页效率，辅存一般按信息块编址，而且让信息块的大小等于页面的大小。

磁盘的辅存实地址的格式

虚地址到辅存实地址的变换就是将多用户虚页号N'_v变换成辅存实块地址Ad。



多用户虚地址到辅存物理地址的变换

外部地址变换方法：页表法。

为每道程序设置一个存放多用户虚页号N'_v与辅存实块地址Ad对应关系的页表。

将用于外部地址变换的页表称为外页表。

将用于内部地址变换的页表称为内页表。

由于虚地址到辅存实地址的变换机会少，变换速度慢，因此外页表通常存在辅存中

内部地址变换：

虚页号变换成主存实页号，进而多用户虚拟地址Av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。

外部地址变换：

虚页号变换成磁盘实地址，主要由软件实现。

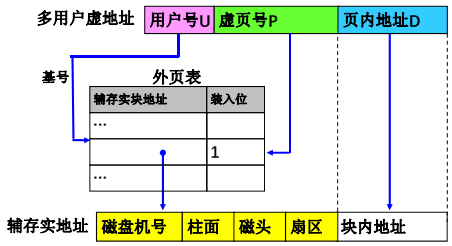
首先通过查外页表得到磁盘实地址，然后再查主存实页表，看主存是否有空页。若有空页，把磁盘存储器实地址和主存储器实页号送入输入输出处理机。在输入输出处理机的控制下，把要访问数据所在的一整页都从磁盘存储器调入到主存储器。

内部地址变换失败（未命中－页面失效），要进行外部地址变换，其目的是要找到磁盘的实地址，并把需要的那一页调入主存。

在操作系统中，把页面失效当作一种异常故障来处理。

每个用户程序都有一张外页表，虚拟地址空间中的每一页，在外页表中都有对应的一个存储字。

每一个存储字除了磁盘存储器的地址之外，至少还包括一个装入位



利用外页表的虚地址-辅存实地址的地址变换

2．页面替换算法

当要用到的指令或数据不在主存中时，会产生页面失效，也就是出现缺页，此时需要去辅存中将包含该指令或数据的一页调入主存。

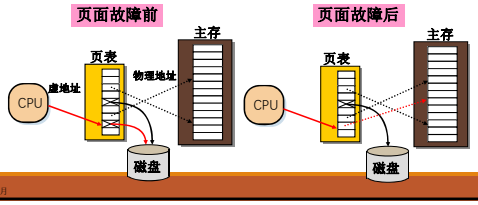
页面冲突（页面争用）是指两个以上的虚页想要进入主存中同一个页面位置的现象。页面失效时不一定发生页面冲突，但页面冲突一定是由页面失效引起的。

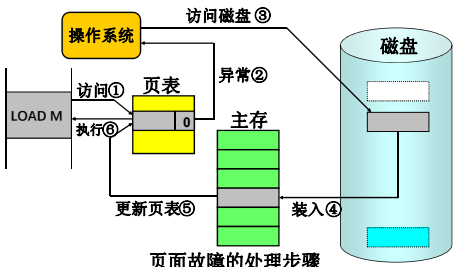
通常辅存空间比实存空间大得多，必然会出现主存的已满又发生页面失效的情况，这时将辅存的一页调入主存会发生页面冲突。只有强制腾出主存中某个页才能接纳由辅存中调来的新页。选择主存中哪个页作为被替换的页，就是页面替换算法要解决的问题。

与前述的Cache替换算法类似，页面替换算法也有随机算法、先进先出算法（FIFO）、近期最久没有访问算法（LRU）等。在虚拟存储系统中，实际上采用的是FIFO和LRU两种替换算法，页面替换算法一般由软件实现。

3. 页面故障（页面失效）

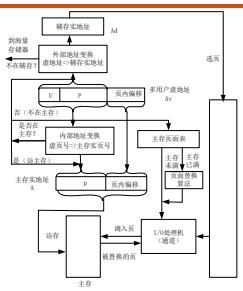
若页表指示多用户虚地址VA所在的虚页未装入主存（未命中），将发生页面故障或页面失效，需要到辅存中调页。
由操作系统负责将虚页从磁盘装入主存，并更新页表。





页面故障的处理步骤

4. 页式虚拟存储系统工作全过程



5.提高虚拟存储系统等效访问速度的措施

关键是提高内部地址变换的速度，即加快多用户虚地址AV到主存实地址PA的变换，因为每次访问主存，都必须先进行内部地址变换。

如何从逻辑结构上提高内部地址变换的速度，正是系统结构设计的任务。

方法：

- 目录表（相联目录法）（前面已讲过）
- 快慢表
- 散列函数

（1）快慢表

根据程序的局部性，在大多数规模较大的机器上，采用增设“快表”的途径来解决。

快表：用快速硬件构成小容量的“相联目录表”，存放当前正在使用的虚实地址映象关系。

- 又称为TLB (Translation Lookaside Buffer)，小容量(几～几十个字)。

慢表：将原先存放全部。虚实地址映象关系的页表称为慢表。

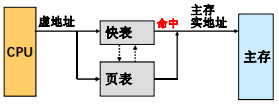
快表是慢表中很小的一部分副本。

快表和慢表构成了表层次。

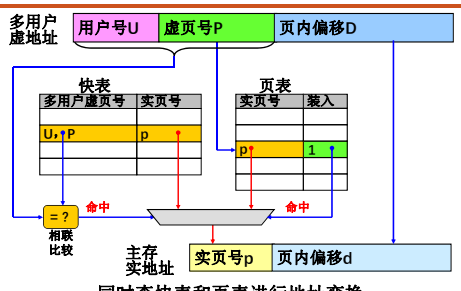
查表时，同时查快表和慢表，并将快表中不存在的内容从慢表中调入快表。

- 显然，这里也要用到替换算法，一般也采用LRU

进一步减少相联比较的位数，也能加快快表的查找速度。



为加快变换速度，查表时，同时查快表和慢表，并将快表中不存在的内容从页表中调入快表。



同时查快表和页表进行地址变换

（2）散列函数

增大快表容量，会提高命中率，但速度会降低，成本会增加。

可以采用高速按地址访问的存储器构成大容量快表，用散列（Hashing）方法实现按内容访问，并用硬件实现散列函数。

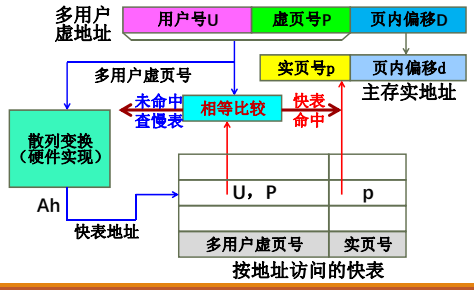
目的：把相联访问变成按地址访问，从而加大快表容量。

散列（Hashing）函数：Ah = H(Pv),
20位左右 ⇒ 5~8位

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。



6.4.3 TLB

TLB （Translation look-aside Buffer）

经常访问的页面地址存放在一个小容量的高速缓冲存储器中的快速查找页表称为转换后备缓冲器（TLB），简称快表

页表一般很大，存放在主存中。

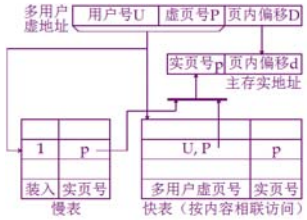
- 导致每次访存可能要两次访问主存，一次读取页表项，一次读写数据
- 解决办法：采用 TLB

- 存放近期经常使用的页表项，是整个页表的部分内容的副本。
- 基本信息：
  VPN##PPN##Protection Field##use bit ## dirty bit
- OS修改页表项时，需要刷新TLB，或保证TLB中没有该页表项的副本
- TLB必须在片内
  - 速度至关重要
  - TLB过小，意义不大
  - TLB过大，代价较高
  - 相联度较高（容量小）

与快表相对应，存放在主存中的页表称为慢表，快表只是慢表中一部分内容的副本，只存放了慢表中很少的一部分。快表与慢表也构成了一个两级存储系统，其速度近似于快表的，其容量近似于慢表的，原理同Cache-主存层次。

快表容量很小（几十个字），由高速硬件实现，采用相联方式访问。

如下图所示，同时查快表和慢表，快表查到就立即中止慢表查找。当快表中查不到时，再从存放在主存中的慢表中查找实页号，并将慢表查到的实页号送入主存，并送入快表。若快表已满，就要采用替换算法。



6.4.4 CPU的一次访存操作

在一个具有Cache和虚拟存储器的系统中，CPU的一次访存操作可能会涉及到TLB、页表、Cache、主存和磁盘的访问。

CPU访存过程中存在以下三种失效（未命中）情况：

① TLB失效：要访问的页面对应的页表项不在TLB中。

② 缺页：要访问的页面不在主存中。

③ Cache失效：要访问的主存块不在Cache中。