

# 第 4 章

# 存储体系

郑宏 副教授  
计算机学院  
北京理工大学

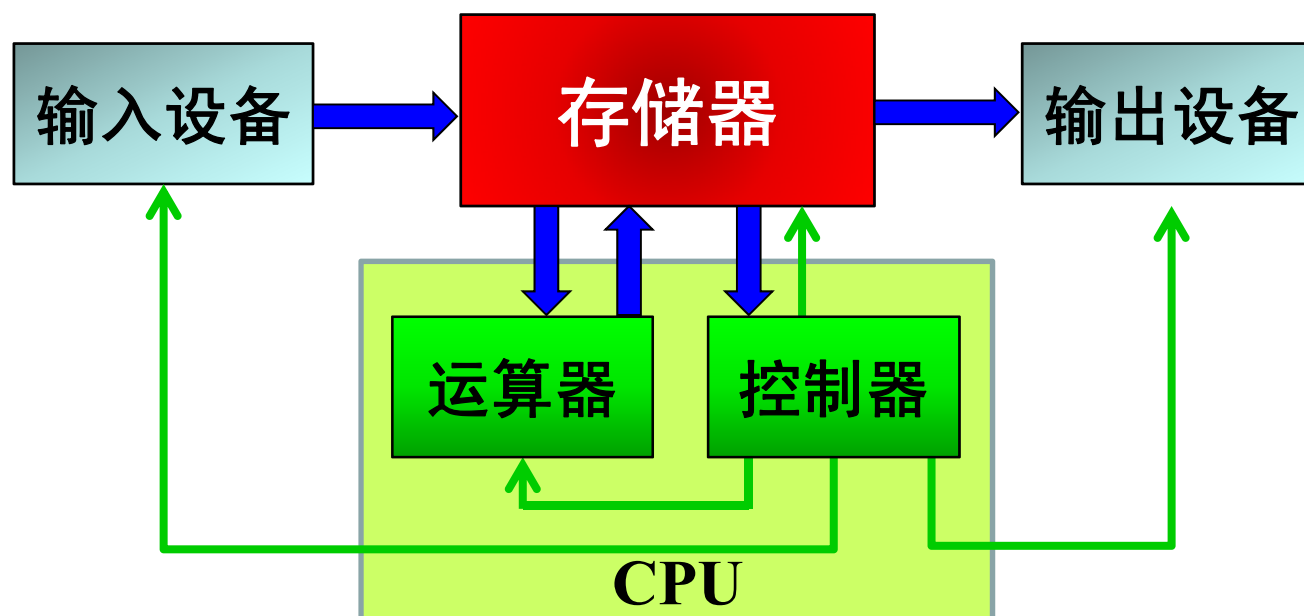
# 第四章 存储体系

学习内容:

- 4.1 存储体系概念和并行存储系统
- 4.2 虚拟存储系统
- 4.3 高速缓冲存储器 (Cache)
- 4.4 Cache - 主存 - 辅存三级层次
- ARM 存储系统

# 4.1 存储体系概念和并行存储系统

## ■ 现代计算机系统都以存储器为中心



存储器是各种信息存储和交换的中心

# 4.1 存储体系概念和并行存储系统

## ■ 存储器

存储系统和存储器是两个完全不同的概念！

- 存储数据的器件。
- 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、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。

# 4.1 存储体系概念和并行存储系统

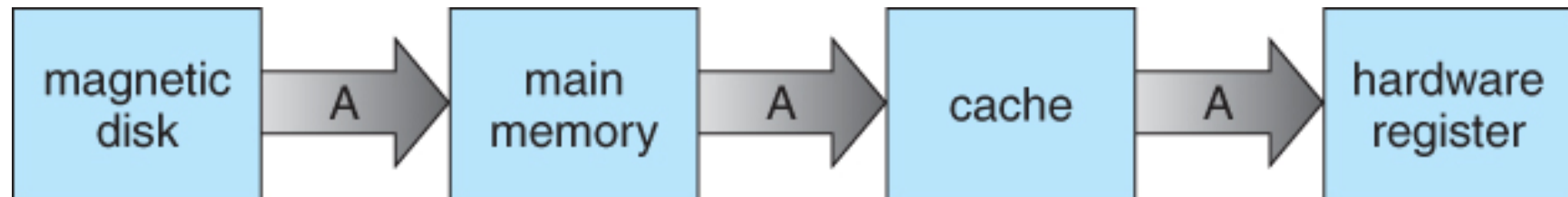
## ■ 存储器（续）

- **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。

# 4.1 存储体系概念和并行存储系统

## ■ 存储系统（存储体系）

- 两个或两个以上的速度、容量、价格不同的存储器采用硬件，软件或软、硬件结合的办法联接成的一个系统。



## 4.1.1 存储体系的引出

- 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贮容量。

**高速度、低价格、大容量！**

# 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

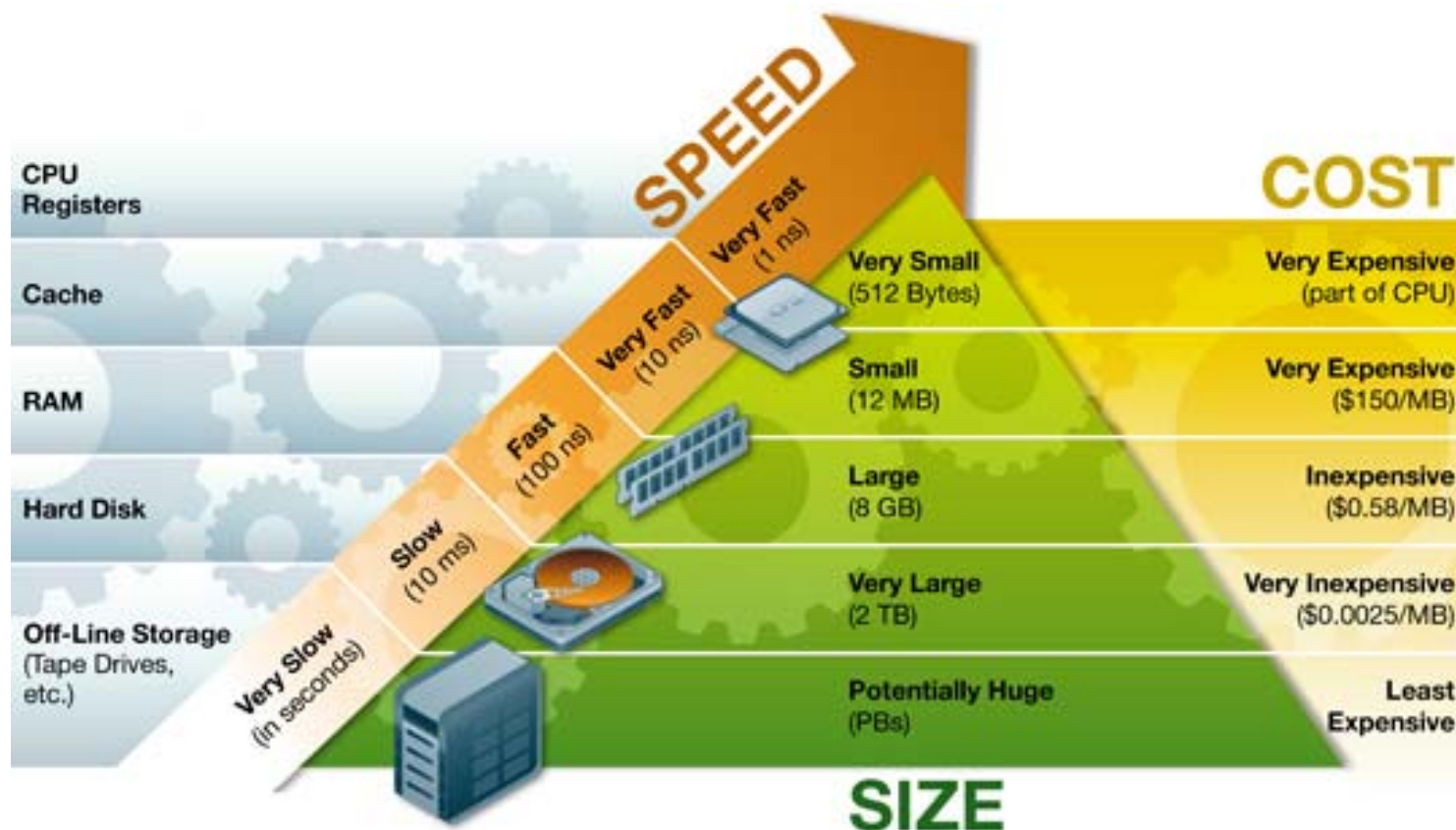
■ 但上述要求是**相互矛盾**的

- 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



# 1. 容量、速度和价格的矛盾

- 不同存储器的速度、价格、容量差异巨大。



- 需要考虑怎样设置才能获得高性能价格比！

## 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）

$m$  —— 并行工作的存储体的个数（设计）

● 可以看出，它即与存储器器件有关，也与设计有关。

## 2. 容量、速度和价格分析

### ■ 速度

- 可以用访问时间 $T_A$ 、存贮周期 $T_M$ 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与存贮器时间关系的一个重要指标。

## 2. 容量、速度和价格分析

### ■ 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。

## 2. 容量、速度和价格分析

### ■ 速度（续）

- 单体存贮器：  $B_M = W / T_M$
- 多体或多字存贮器：  $B_M = m \times W / T_M$
- 可以看出：它即与存储器器件有关，也与设计有关。

## 2. 容量、速度和价格分析

### ■ 价格

- 可以用总价格  $C$  或 位价格  $c$  表示

- 价格  $c = S_M / T_M = W \times L \times m / T_M$

与容量成正比，与速度成反比

与访问时间成反比，与速度成正比

- 可以看出：它即与存储器器件有关，也与设计有关。

## 2. 容量、速度和价格分析

- 综上所述，在三个性能指标中：
  - 字数、 $T_A$ 和 $T_M$ 主要与器件工艺有关
  - 字长和存储体个数则可由系统设计者确定
  - 改进工艺和设计可以解决矛盾

### 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：

- 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。



# Memory 优化

## ■ DDR:

### ● DDR2

- ◆ Lower power (2.5 V  $\rightarrow$  1.8 V)
- ◆ Higher clock rates (266 MHz, 333 MHz, 400 MHz)

### ● DDR3

- ◆ 1.5 V
- ◆ 800 MHz

### ● DDR4

- ◆ 1-1.2 V
- ◆ 1600 MHz

## ■ GDDR5 is graphics memory based on DDR3

# Memory Optimizations

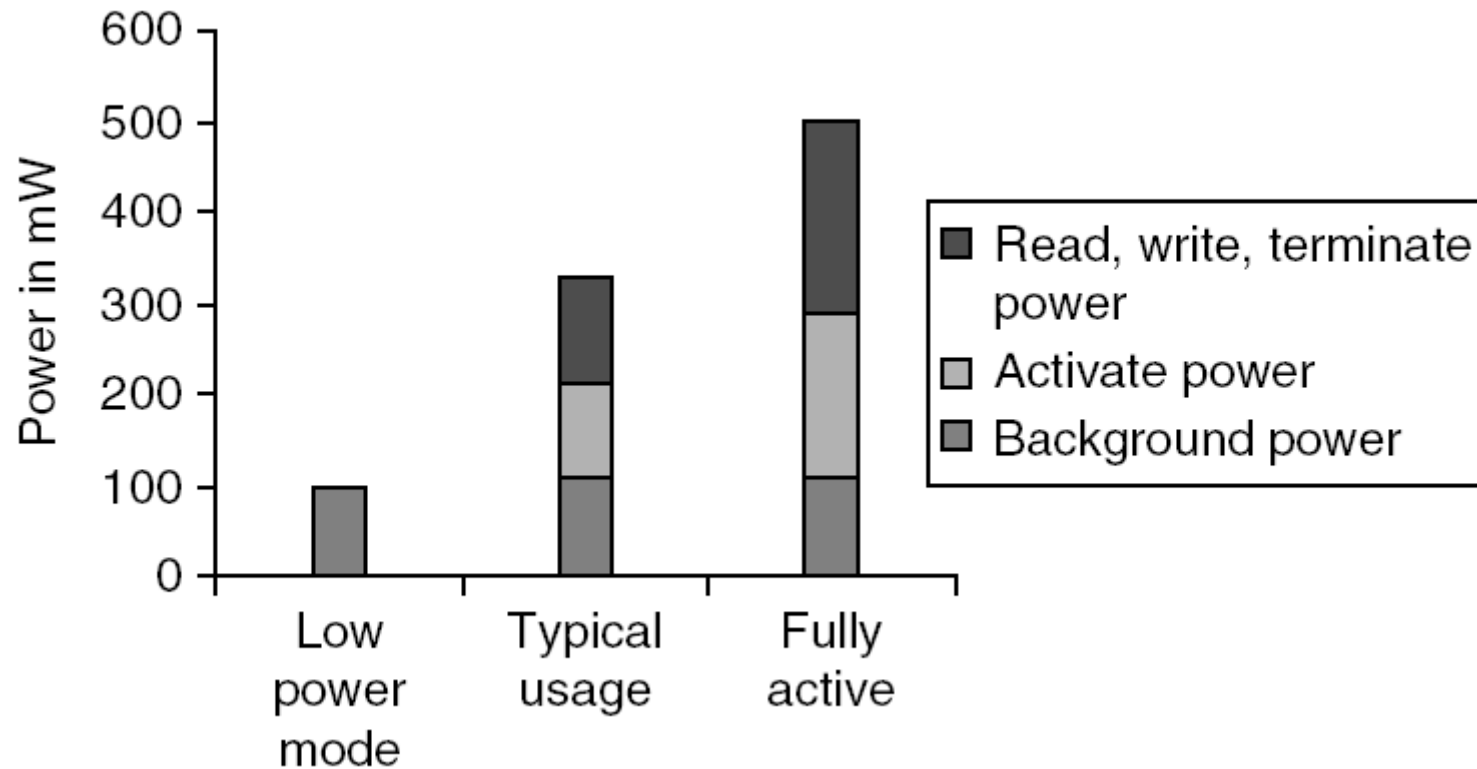
## ■ Graphics memory:

- Achieve 2-5 X bandwidth per DRAM vs. DDR3
  - ◆ Wider interfaces (32 vs. 16 bit)
  - ◆ Higher clock rate
    - Possible because they are attached via soldering instead of socketed DIMM modules

## ■ Reducing power in SDRAMs:

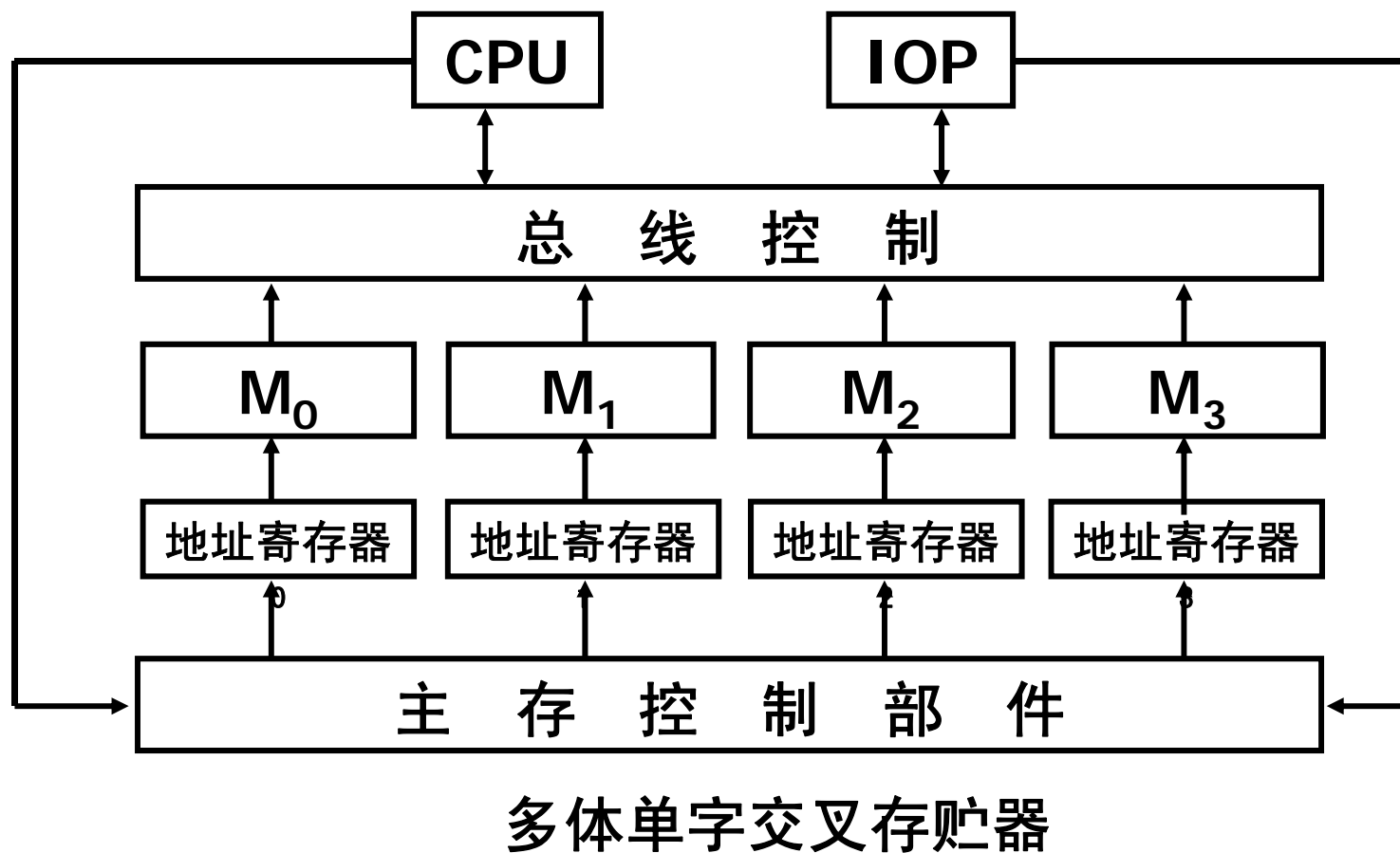
- Lower voltage
- Low power mode (ignores clock, continues to refresh)

# Memory Power Consumption



### 3. 解决矛盾的措施

#### ■ 构成并行主存系统



### 3. 解决矛盾的措施

#### ■ 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，转移指令使存贮系统效率下降。

### 3. 解决矛盾的措施

#### ■ 使用存储器系统

- 用不同工艺的多种存储器组成存贮器系统，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. 解决矛盾的措施

#### ■ 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70年代，合理成本、足够容量的主存的存储周期比CPU拍宽大一个数量级
- 为此，还需要进一步采取措施，采用**存储体系（存储层次）**

## 4.1.2 并行存储系统

### ■ 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:

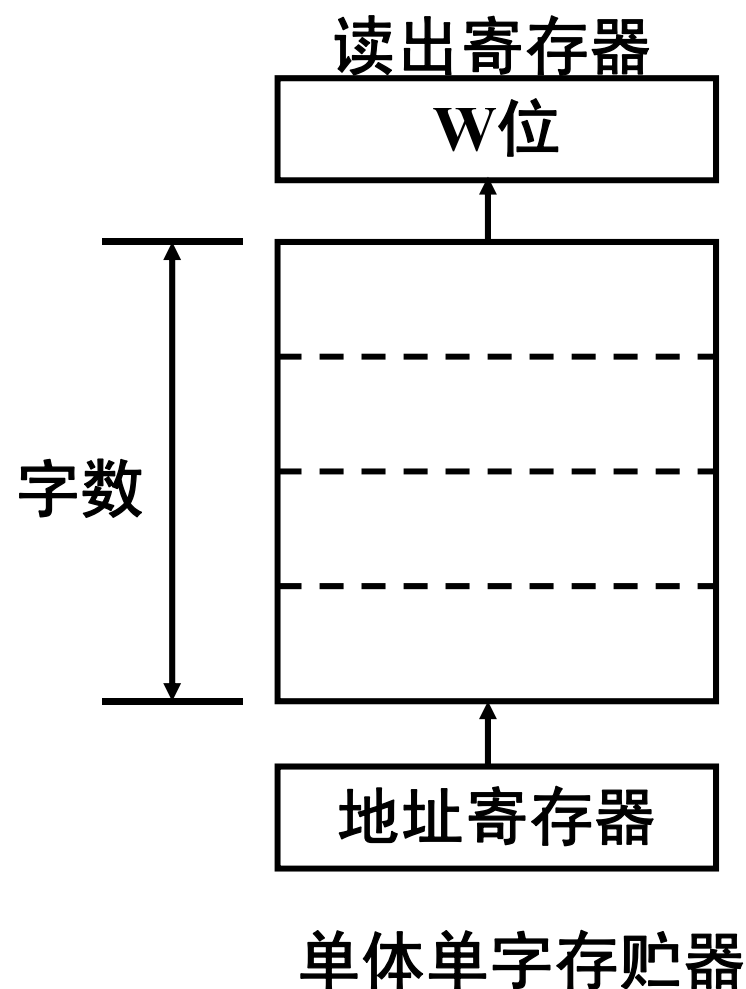
- 单体多字
- 多体单字交叉存储器
- 多体多字交叉存储器



# 单体单字存贮器

- 有一个字长为  $W$  位的存贮器，一次可以访问一个存贮器字。
- 若存贮器字长与 CPU 字长相等，其最大频宽为：

$$B_M = W / T_M$$



# 提高存储系统性能的途径：并行

■ 要提高频宽，只有设法提高存贮器字长  $W$  才行。有三种方案：

- 单体多字存贮器
- 多体单字交叉存贮器
- 多体多字交叉存贮器

# 单体多字存贮器

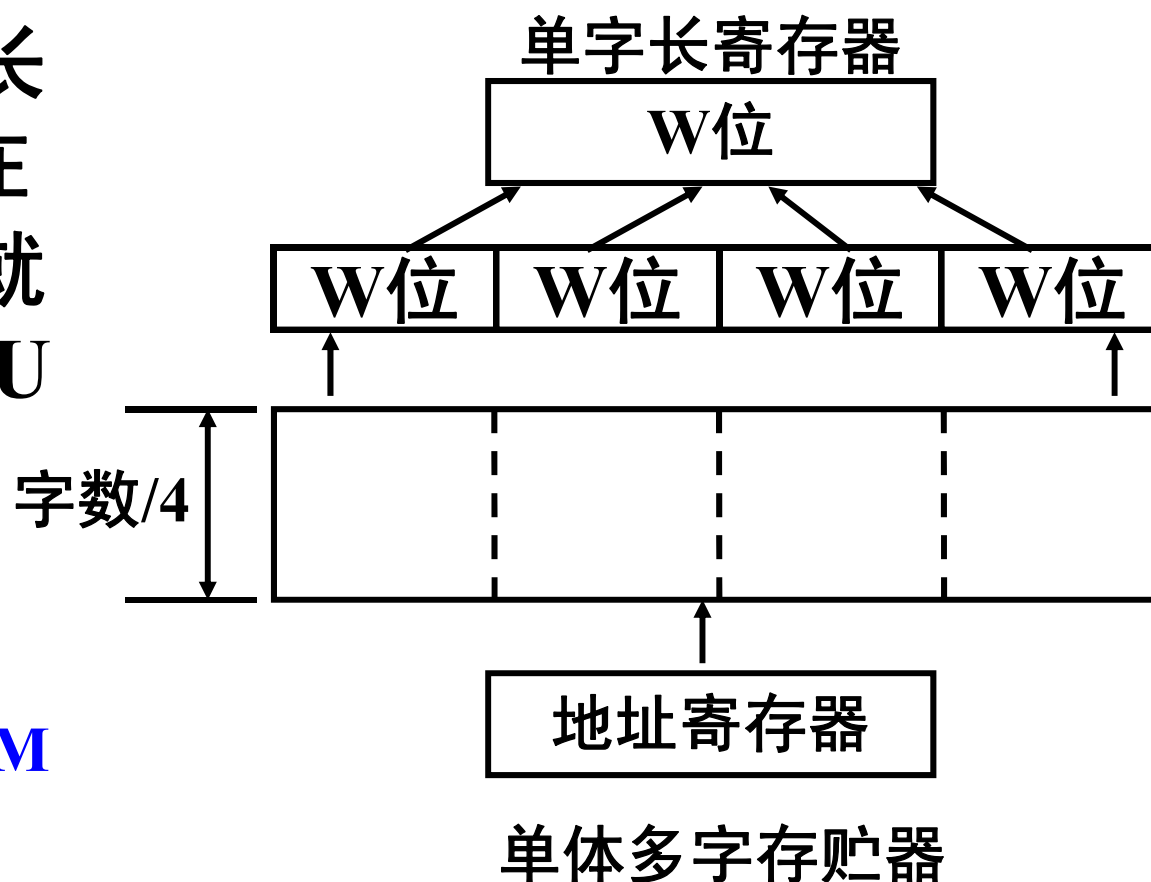
增加存贮器的字长  
( $m$ 倍)，这样在  
一个主存周期内就  
可以读出多个CPU  
字。

其最大频宽为：

$$B_M = m \times W / T_M$$

缺点：

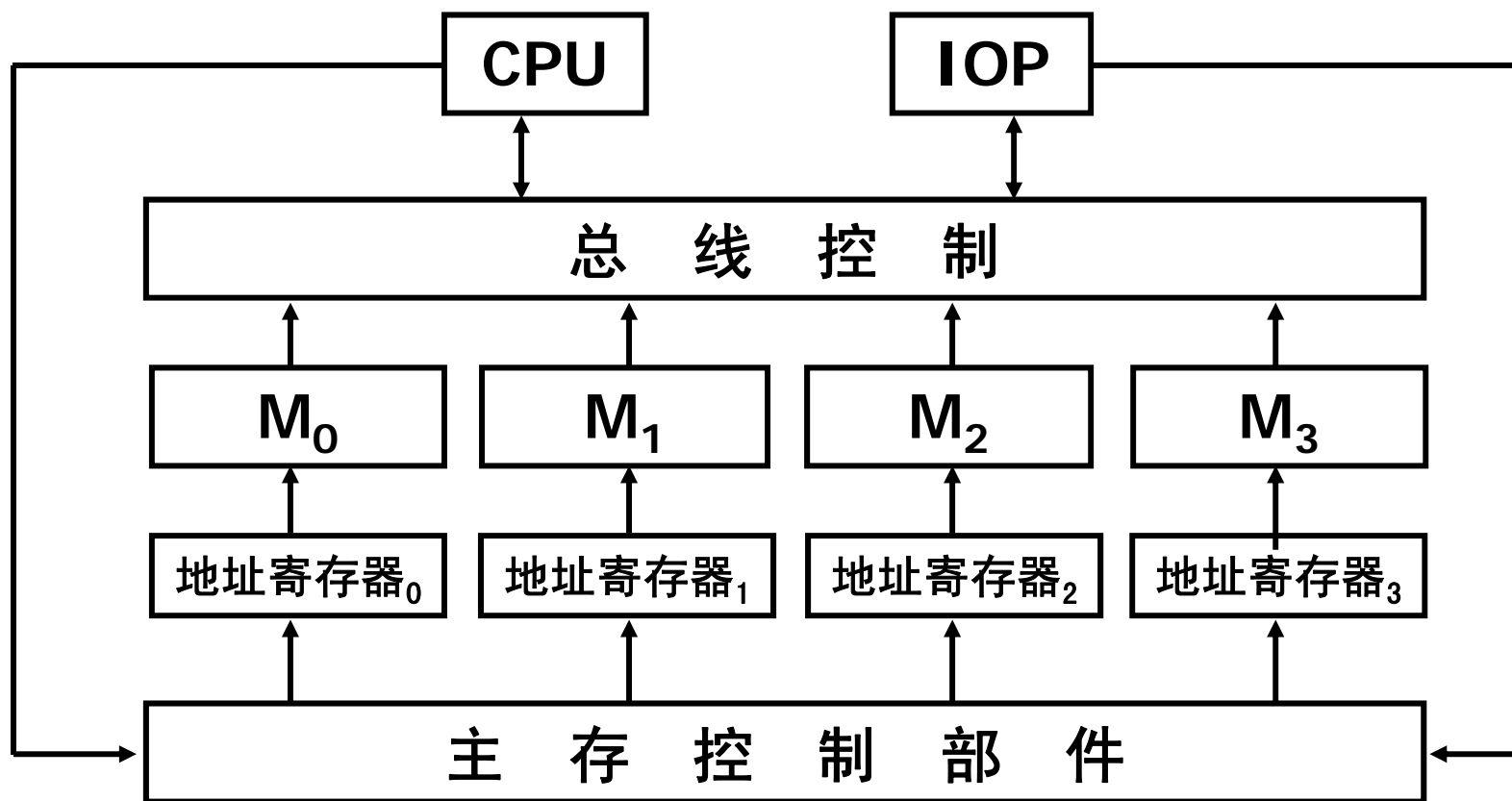
- 需要位数足够多的寄存器；
- 多次访问总线。



# 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： **$B_M = m \times W / T_M$**

# 多体单字存贮器



多体单字交叉存贮器

# 多体单字存贮器

## ■ 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

# 多体单字存储器

■ 为减少分体冲突，CPU字在主存中可按模 $m$ 交叉编址。分为：

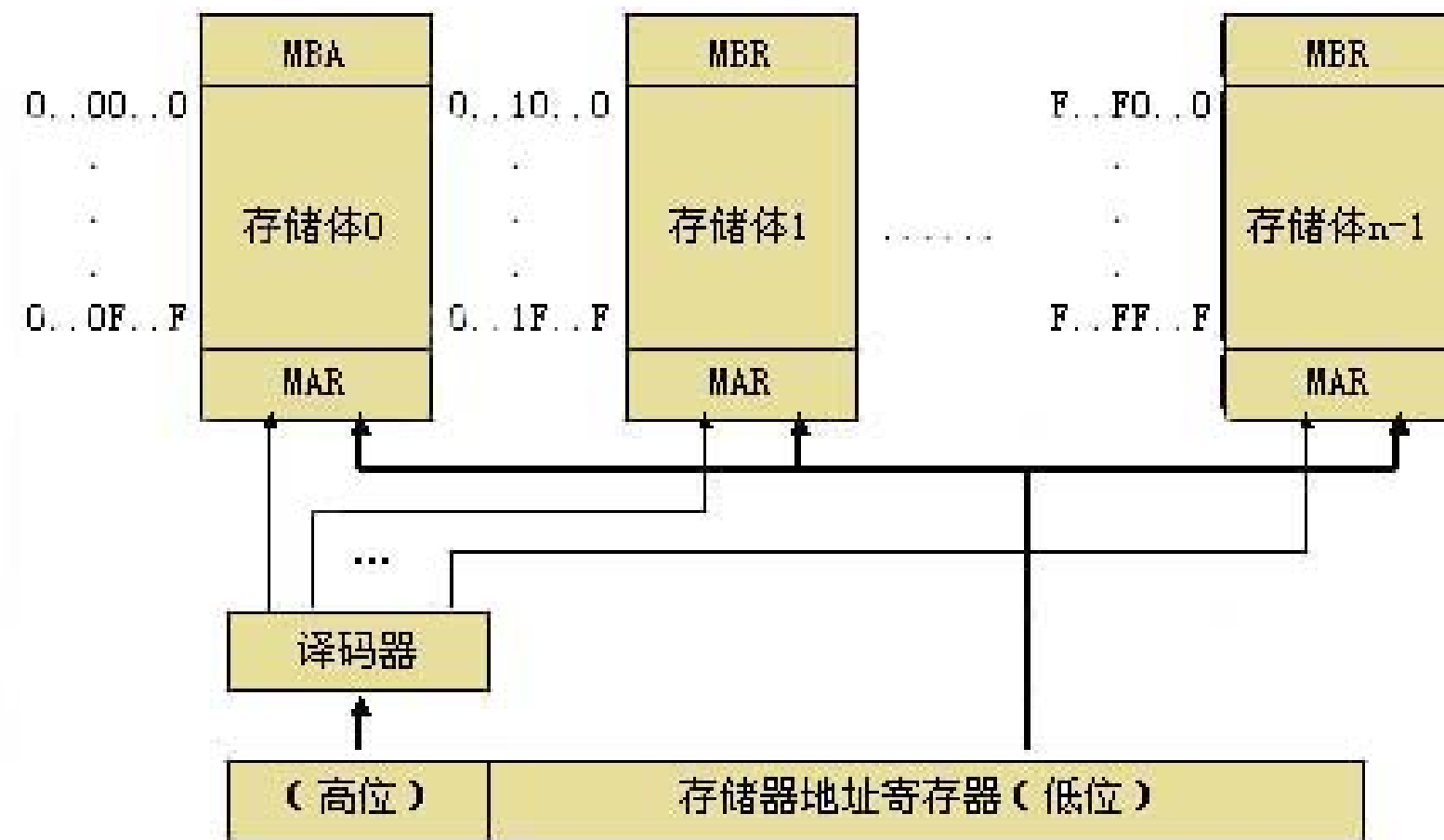
- 高位交叉

- ◆ 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

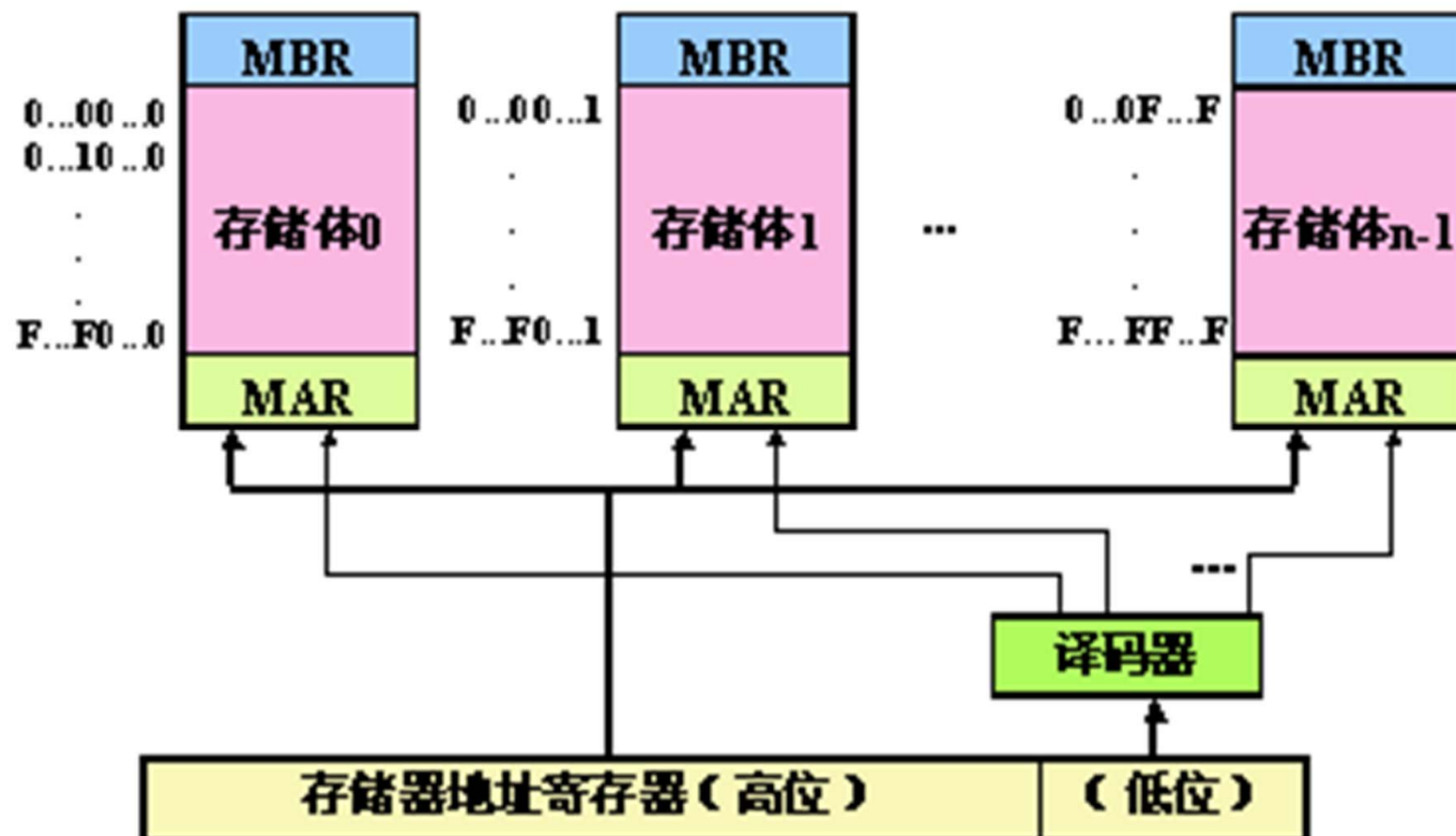
# 多体单字存贮器



高位交叉访问存储器

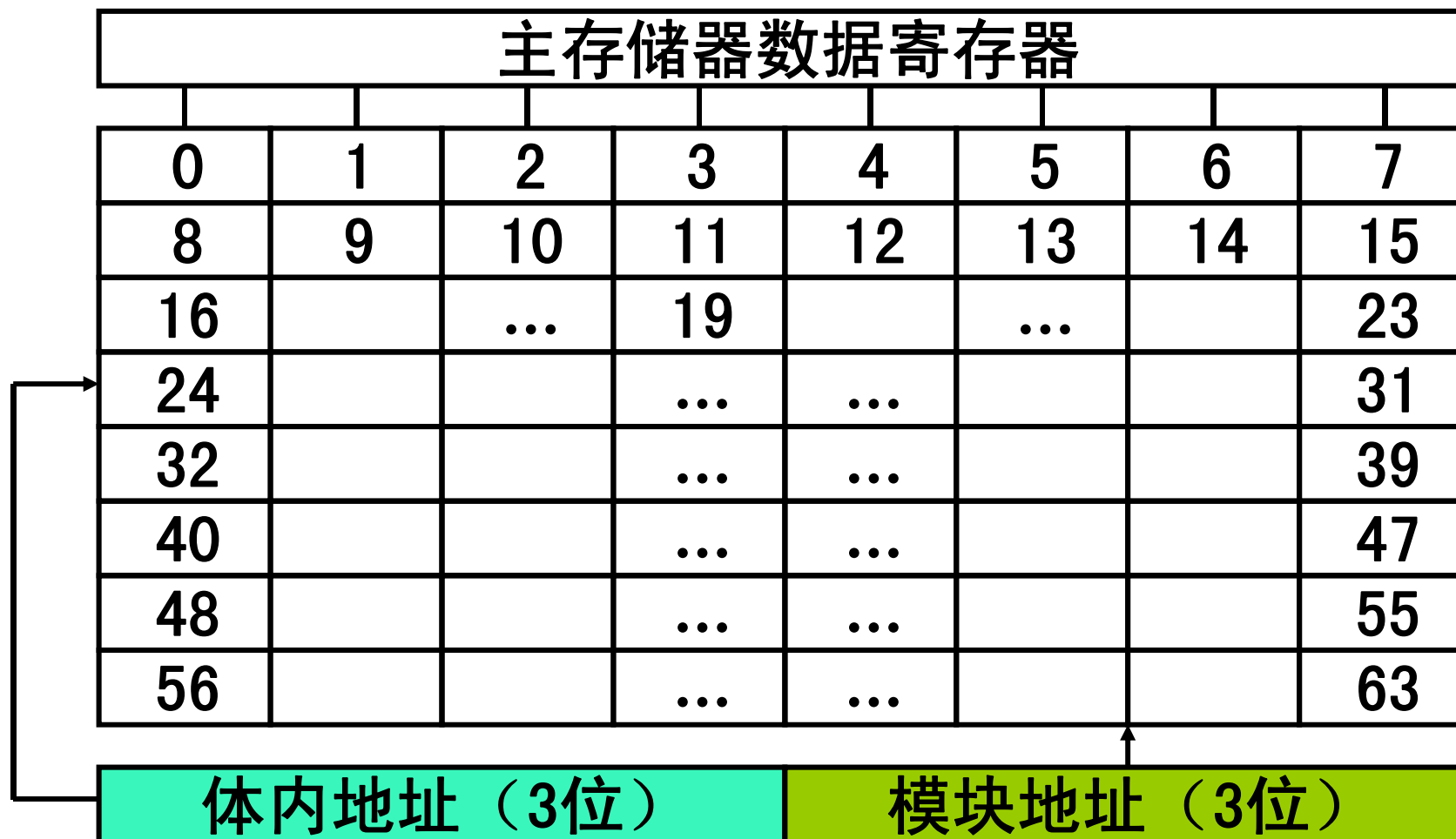


# 多体单字存贮器



低位交叉访问存储器

# 多体单字存储器



低位交叉访问存储器 - 8个存储体

# 多体单字存储器

由于采用交叉存储编址，对于任何CPU读写访问或与外设DMA传送，只要是对主存连续字的成块传送，就可以实现多模块流水式并行存取，亦即使多个模块在任一时刻同时并行工作，大大提高存储器的带宽

由于CPU的速度比主存快，同时从主存取出多条指令，必然会提高机器的运行速度。

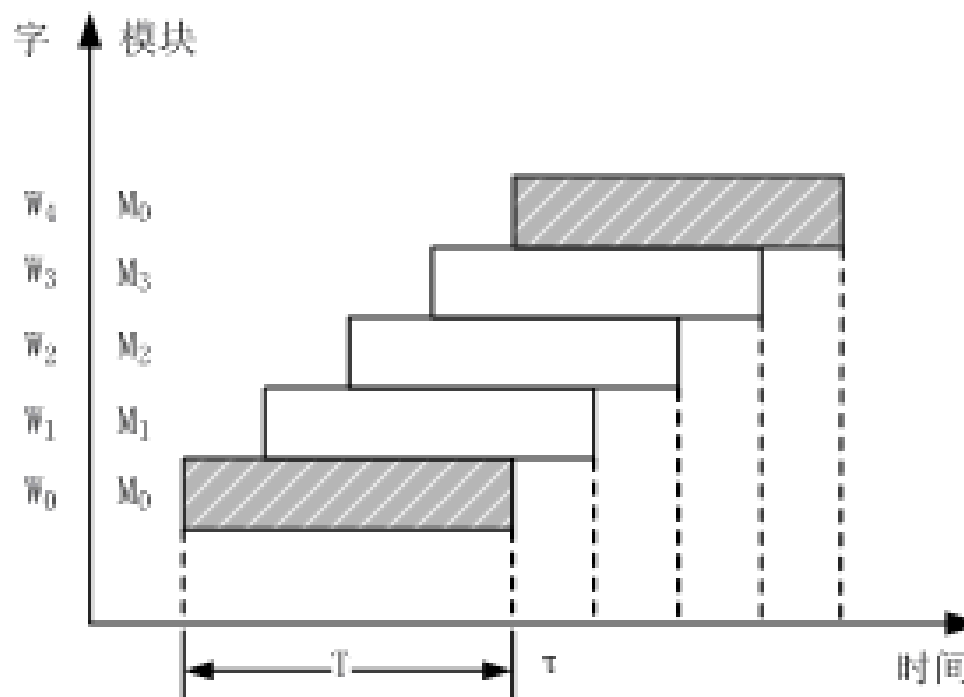


图 3-9 四模块交叉存储器的流水线方式存取示意图

## 低位交叉访问存储器

# 多体单字存储器

模4低位交叉编址( $m=4, j=0,1,2,3$ )

模块 $M_j$	地址编址序列 $m*i+j$	对应二进制地址码最末二位的状态
$M_0$	0, 4, 8, 12, ..., $4i+0, \dots$	00
$M_1$	1, 5, 9, 13, ..., $4i+1, \dots$	01
$M_2$	2, 6, 10, 14, ..., $4i+2, \dots$	10
$M_3$	3, 7, 11, 15, ..., $4i+3, \dots$	11

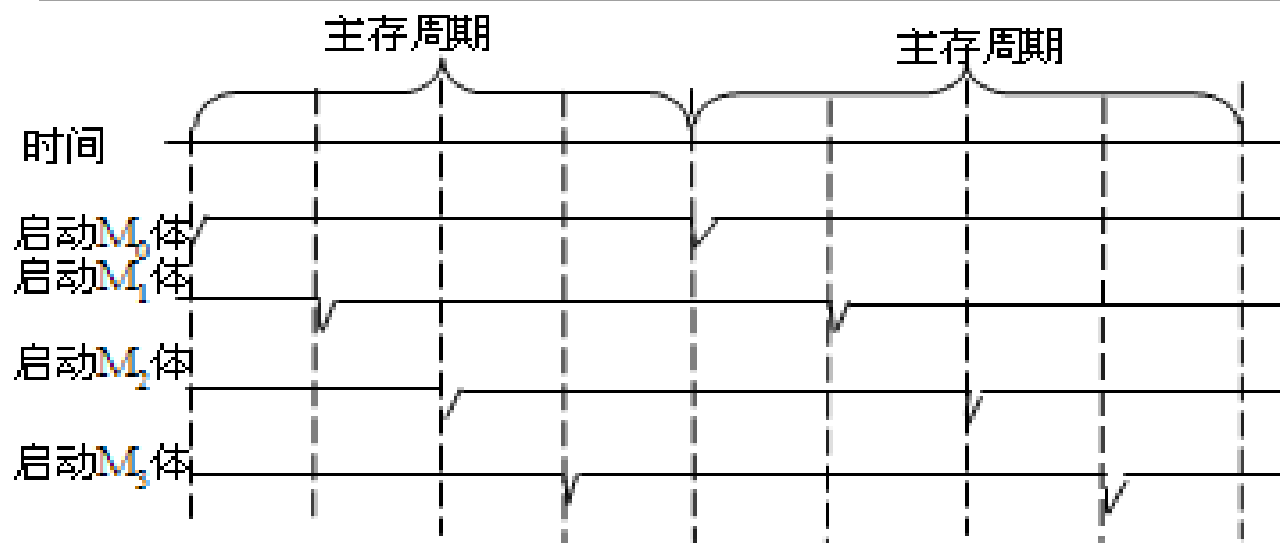


图 4个分体时启动的时间关系

## 低位交叉访问存储器

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。

# 多体多字存贮器

- 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**。

## 4.1.2 并行存储系统

### ■ 存在的问题

- 提高 $m$ 值，可以提高主存系统的最大频宽，但主存的实际频宽并不随着 $m$ 值的增加而线性提高。
- 原因在于：
  - ◆  $m$ 值越大，存贮器数据总线越长，总线上并联的负载越重，门的级数增加，传输延迟增加；
  - ◆ 转移指令使系统的效率下降。而且数据分布的离散性很大，因此实际的频宽可能还要低一些。

## 4.1.2 并行存储系统

- 可见，单纯增大 $m$ 值来提高并行主存系统的频宽十分有限，而且性能价格比会随着 $m$ 的增大而下降。
- 因此必须从系统结构上改进。

## 4.1.3 存储体系的定义与分支

- 为了解决单一种类存贮器的价格、容量及速度之间的矛盾，计算机系统的存贮系统总是利用多种不同的存贮器构成。
- 如何组织这些不同的存贮器呢？

**存储体系！**



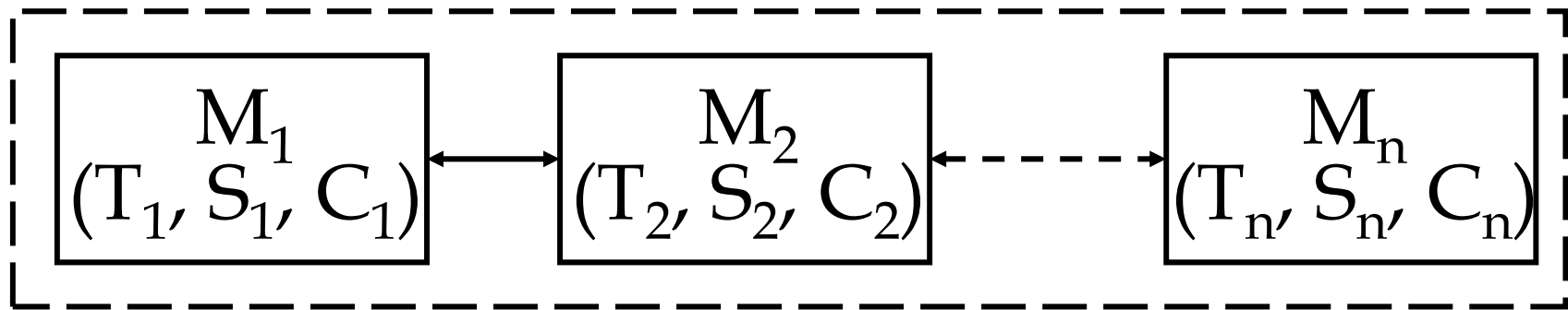
## 4.1.3 存储体系的定义与分支

### ■ 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。

## 4.1.3 存储体系的定义与分支

从外部看:



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

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

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

## 4.1.3 存储体系的定义与分支

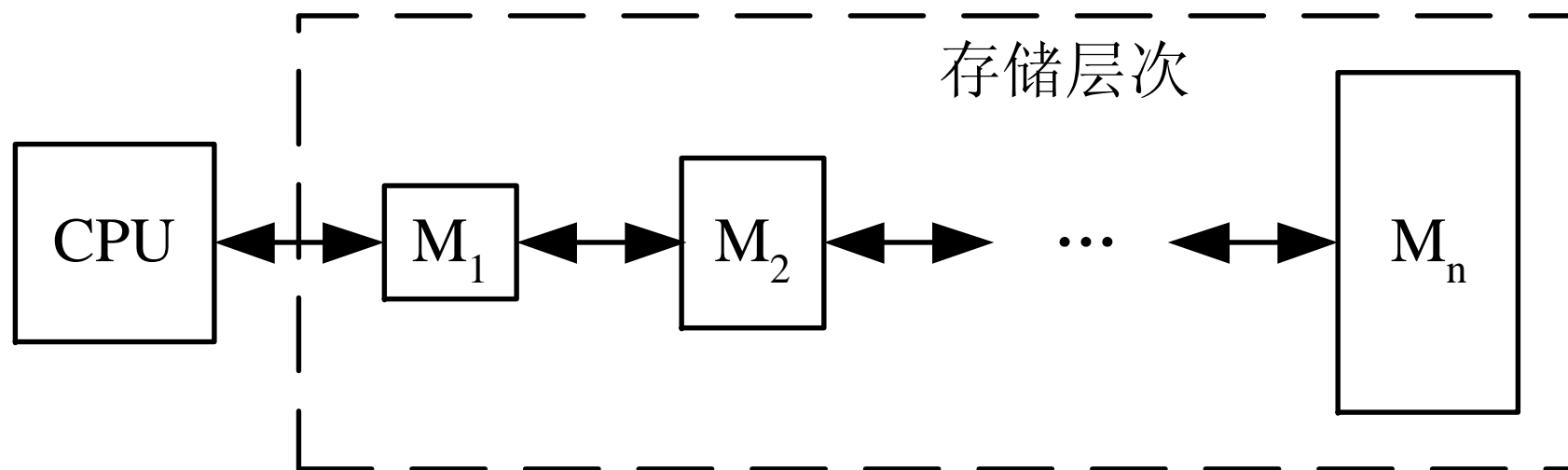


图 多级存储层次

两个典型分支：

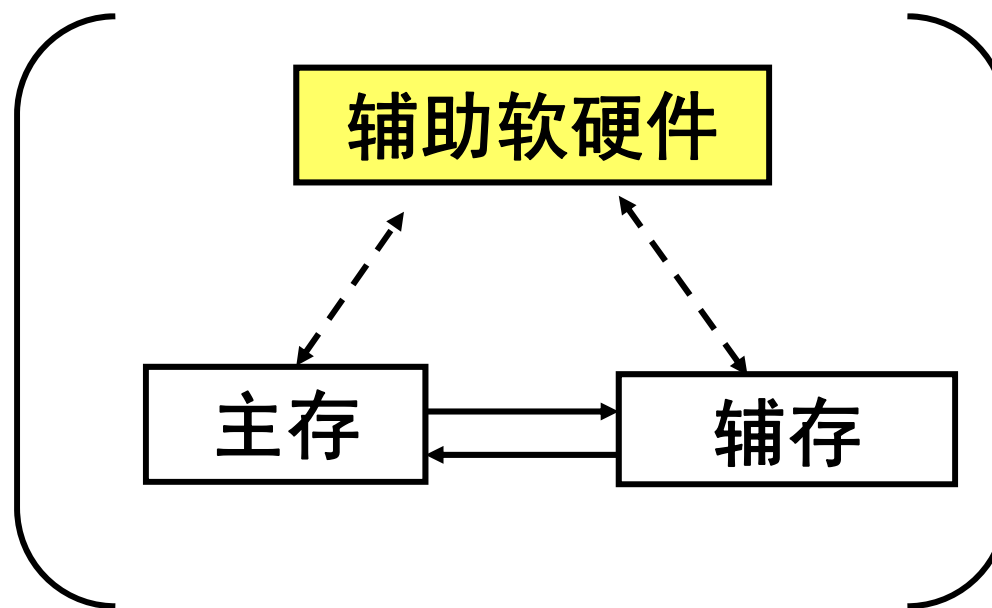
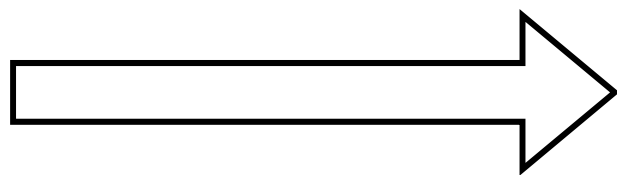
- 虚拟存储系统
- Cache存储系统

# 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）

从整体上看，速度接近主存，容量是辅存的。



主存—辅存存储层次

# 虚拟存储系统

## ■ 解决容量问题（续）

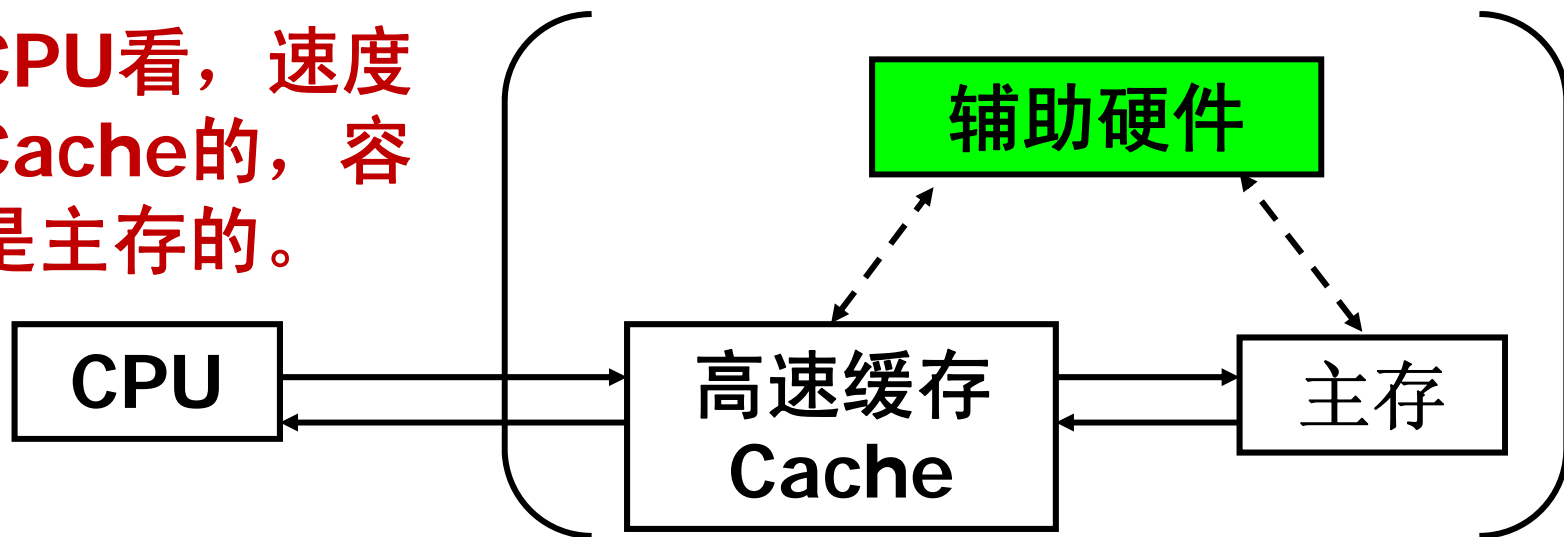
- 利用了I/O处理可与CPU并行操作的能力；
- 借助操作系统、硬件等实现地址变换和程序定位，实现主存和辅存之间的数据传送，使主存、辅存构成了一个完整的整体。
- 对应用程序员是透明；
- 主存—辅存存贮层次的不断发展和完善，就形成了虚拟存贮器。

# Cache存储系统

## ■ 解决速度问题

### ● Cache—主存存贮层次（二级存储层次）

从CPU看，速度是Cache的，容量是主存的。



### Cache—辅存存贮层次

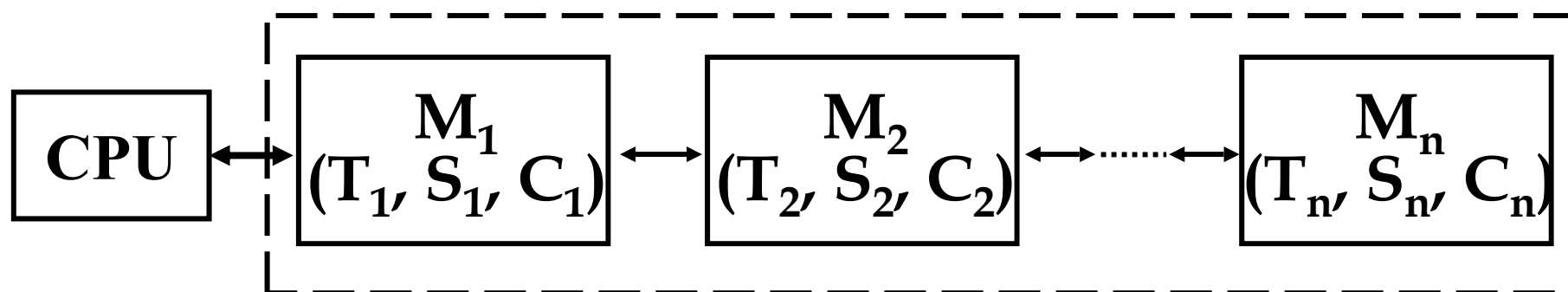
# Cache存储系统

## ■ 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贮器（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
- 对应用程序员和系统程序员都是透明的。

# 多级存贮层次

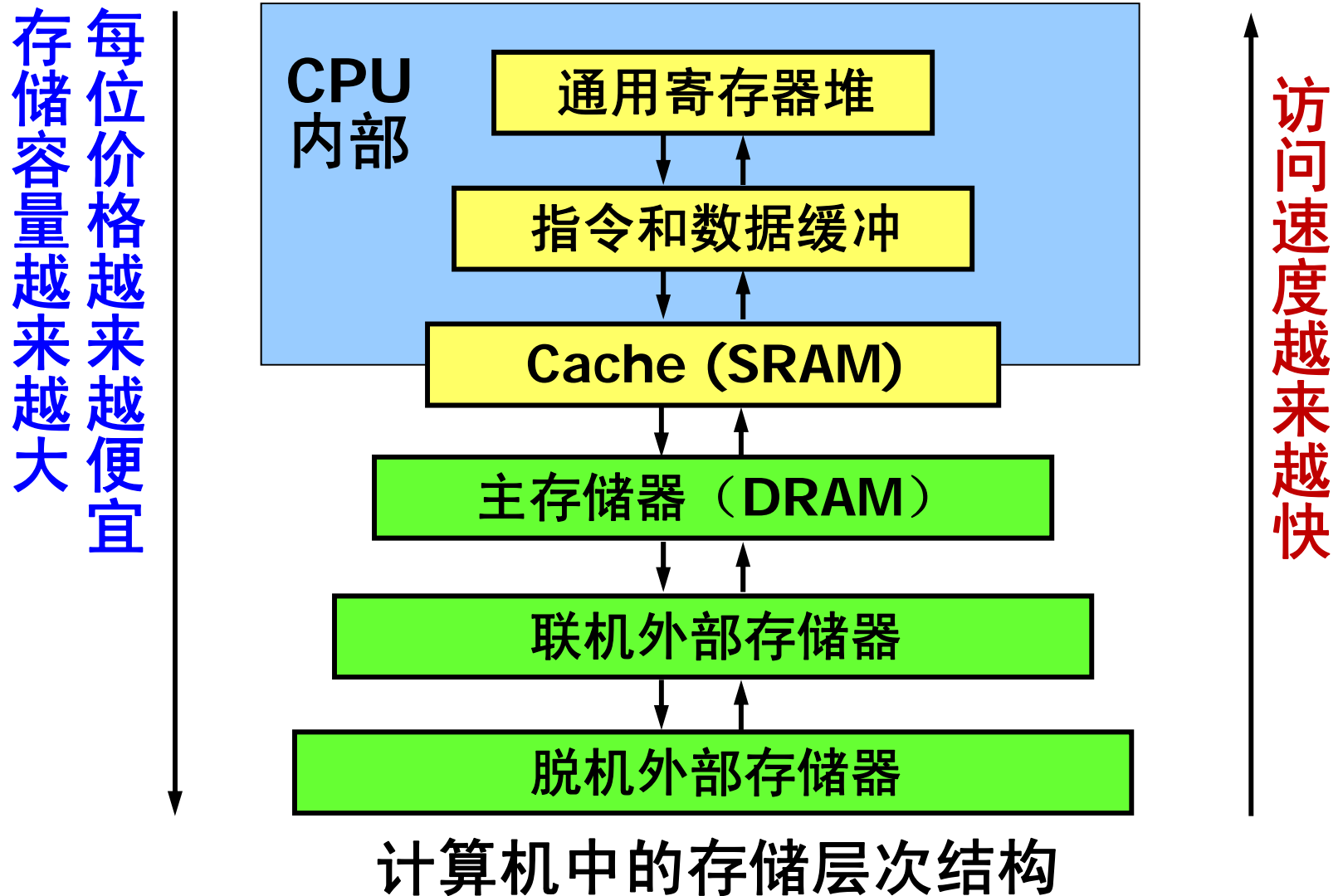
- 二级存贮层次可以扩展到多级存贮层次



多级存贮层次



## 4.1.3 存储体系的定义与分支



# 多级存贮层次

各级存储器的主要性能特性 (1/2)

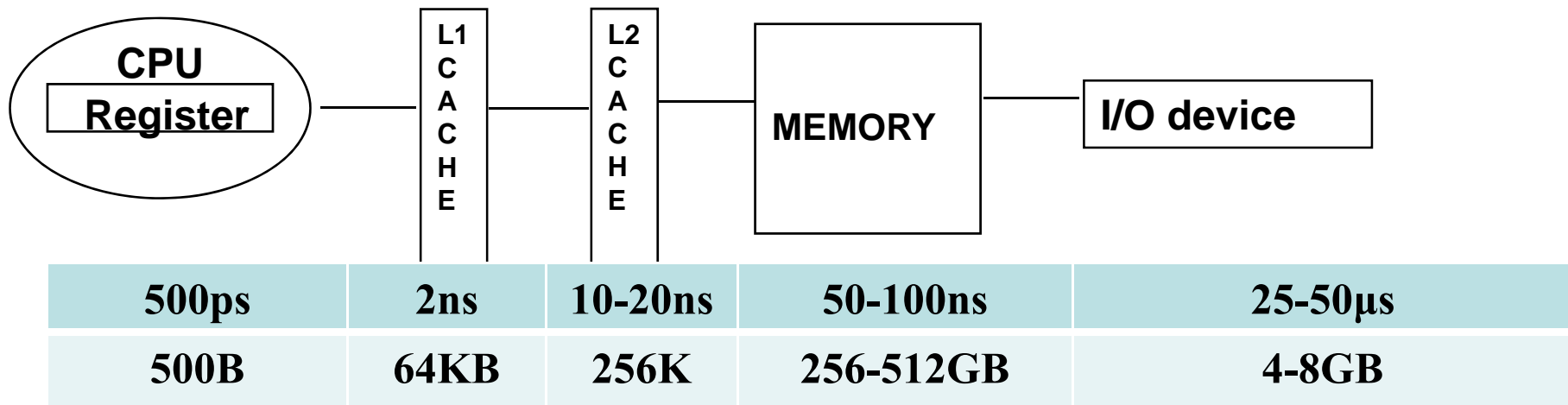
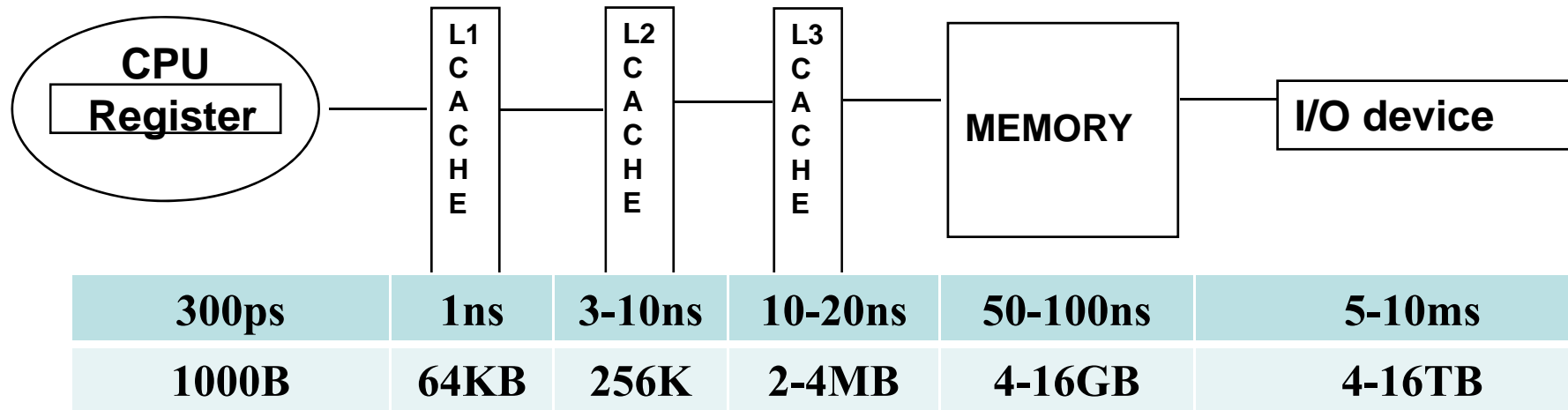
存储器层次	通用寄存器	缓冲栈	Cache
存储周期	< 10ns	< 10ns	10 - 60ns
存储容量	< 512B	< 512B	8K-2MB
价格\$/KB	1200	80	3.2
访问方式	直接译码	先进先出	相联访问
材料工艺	ECL	ECL	SRAM
分配管理	编译器分配	硬件调度	硬件调度
带宽MB/s	400-8000	400-1200	200-800

# 多级存贮层次

## 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\$/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

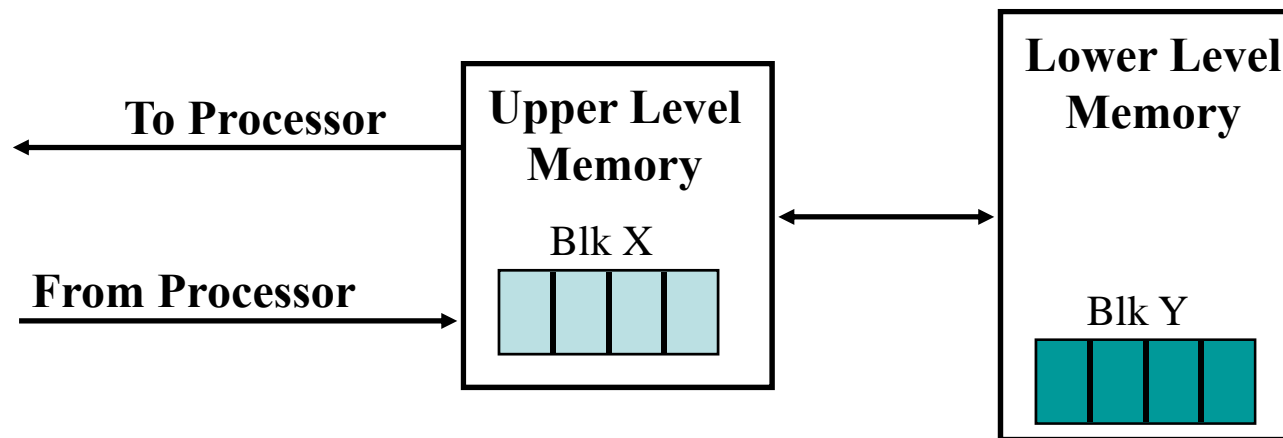
# 现代计算机系统的多级存储层次



## 4.1.3 存储体系的定义与分支

### ■ 不同存贮层次之间的数据传送

- 程序或数据分布在不同层次的存贮器上;
- 为了使存贮体系有效工作，当CPU要用到某个地址的内容时，总是希望被访问的数据已经在 $M_1$ 中;
- 因此，需要在不同层次的存贮器传送数据。



## 4.1.3 存储体系的定义与分支

### ■ 不同存贮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，就可以使存贮体系有效工作。
- 能否预判？ **程序局部性 = 预判的基础**

# 程序局部性

## ■ 定义：

程序在执行时所用到的指令和数据的分布不是随机的，而是相对地簇聚成块或页。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。

# 程序局部性

- 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**是存贮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。



# 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：

$$\text{信息}_{\text{高层}} \in \text{信息}_{\text{低层}}$$

## 4.1.4 存储体系的性能参数

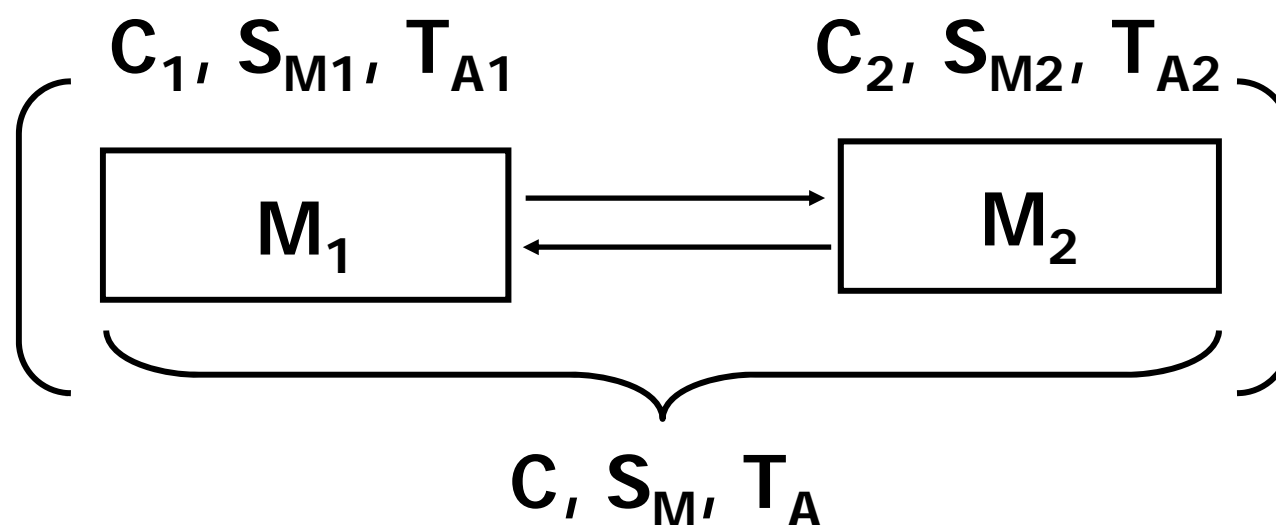
### ■ 有以下三个:

- 每位价格 $c$
- 命中率 $H$
- 等效访问时间 $T_A$

### ■ 下面以二级存贮层次为例来分析

## 4.1.4 存贮体系的性能参数

### ■ 二级存贮层次



## 4.1.4 存贮体系的性能参数

### ■ 每位价格c

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

- 为使c接近于 $c_2$ ，应使 $S_{M_1} \ll S_{M_2}$

## 4.1.4 存贮体系的性能参数

### ■ 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） $= 1 - H$

## 4.1.4 存贮体系的性能参数

### ■ 等效访问时间 $T_A$

假设 $M_1$ 访问和 $M_2$ 访问是同时启动

$$T_A = HT_{A1} + (1 - H)T_{A2}$$

假设 $M_1$ 访问和 $M_2$ 访问不是同时启动

$$\begin{aligned} T_A &= HT_{A1} + (1 - H)(T_{A1} + T_{A2}) \\ &= T_{A1} + (1 - H)T_{A2} \end{aligned}$$

希望 $T_A$ 接近 $T_{A1}$ 为好。

访问效率  $e = T_{A1}/T_A$ ，越接近 1 越好。

## 4.1.4 存贮体系的性能参数

### ■ 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

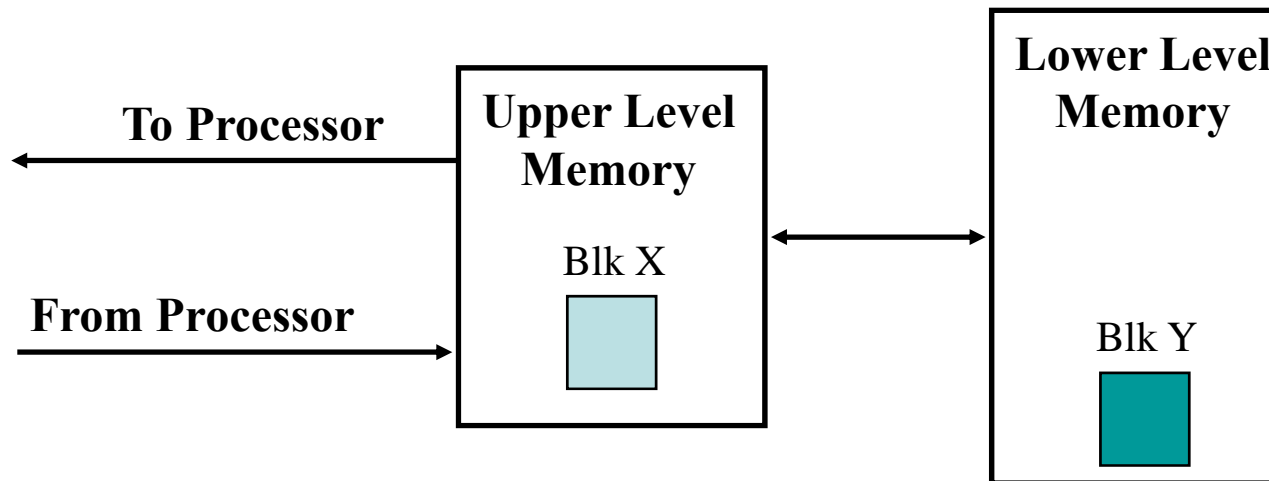
# 存储层次的性能参数(1/2)

## ■ 假设采用二级存储：M1 和 M2

- M1和M2的容量、价格、访问时间分别为：

$S_1$ 、 $C_1$ 、 $T_{A1}$

$S_2$ 、 $C_2$ 、 $T_{A2}$





# 存储层次的性能参数 (2/2)

## ■ 存储层次的平均每位价格 $C$

- $C = (C_1 * S_1 + C_2 * S_2) / (S_1 + S_2)$

## ■ 命中(Hit): 访问的块在存储系统的较高层次上

- 若一组程序对存储器的访问，其中  $N_1$  次在 M1 中找到所需数据， $N_2$  次在 M2 中找到数据 则
- Hit Rate (命中率): 存储器访问在较高层命中的比例  $H = N_1 / (N_1 + N_2)$
- Hit Time (命中时间): 访问较高层的时间,  $T_{A1}$

## ■ 失效(Miss): 访问的块不在存储系统的较高层次上

- Miss Rate (失效) =  $1 - (\text{Hit Rate}) = 1 - H = N_2 / (N_1 + N_2)$
- 当在 M1 中没有命中时: 一般必须从 M2 中将所访问的数据所在块搬到 M1 中, 然后 CPU 才能在 M1 中访问。
- 设传送一个块的时间为  $T_B$ , 即不命中时的访问时间为:  $T_{A2} + T_B + T_{A1} = T_{A1} + T_M$   
 $T_M$  通常称为**失效开销**。

## ■ 平均访存时间:

- 平均访存时间  $T_A = HT_{A1} + (1-H)(T_{A1} + T_M) = T_{A1} + (1-H)T_M$

## 4.1.4 存贮体系的性能参数

- 例4-3：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,  $e_2=1/(0.99+5(1-0.99))=0.96$

## 4.1.4 存贮体系的性能参数

- 例4-4：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

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

$$H = 0.99999$$

- 能获得如此高的命中率吗？

## 4.1.4 存贮体系的性能参数

### ■ 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  $\times$  数据重复使用次数

不命中率降低  $n$  倍

## 4.1.4 存贮体系的性能参数

- 例4-5：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。

## 4.1.4 存贮体系的性能参数

### ■ 例4-5：解

$n=4 \times 5=20$ , 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:

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

## 4.1.4 存贮体系的性能参数

■ 例4-6：在一个虚拟存储系统中， $T_2 = 10^5 T_1$ ，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。

# 第四章 存储体系

学习内容:

- 4.1 存储体系概念和并行存储系统
- 4.2 虚拟存储系统
- 4.3 高速缓冲存储器（Cache）
- 4.4 Cache - 主存 - 辅存三级层次
- ARM存储系统

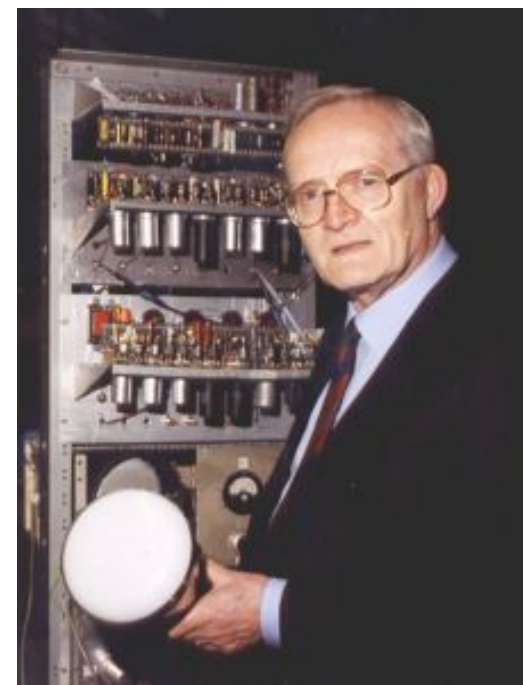


# 主要内容

- 虚拟存储器管理方式
- 页式虚拟存储器
  - 地址映像规则与地址变换
  - 页面替换
- 实现中的问题
  - 页面失效
  - 快表

# 虚拟存储器定义及特点

- 1961年，英国曼彻斯特大学 Tom Kilburn 领导的小组开发；
- 1970年，美国 RCA 公司研究成功虚拟存储器系统；
- 1972年，IBM 公司于在 IBM 370 系统上全面采用了虚拟存储技术。
- 虚拟存储器已成为计算机系统中非常重要的部分。



Tom Kilburn

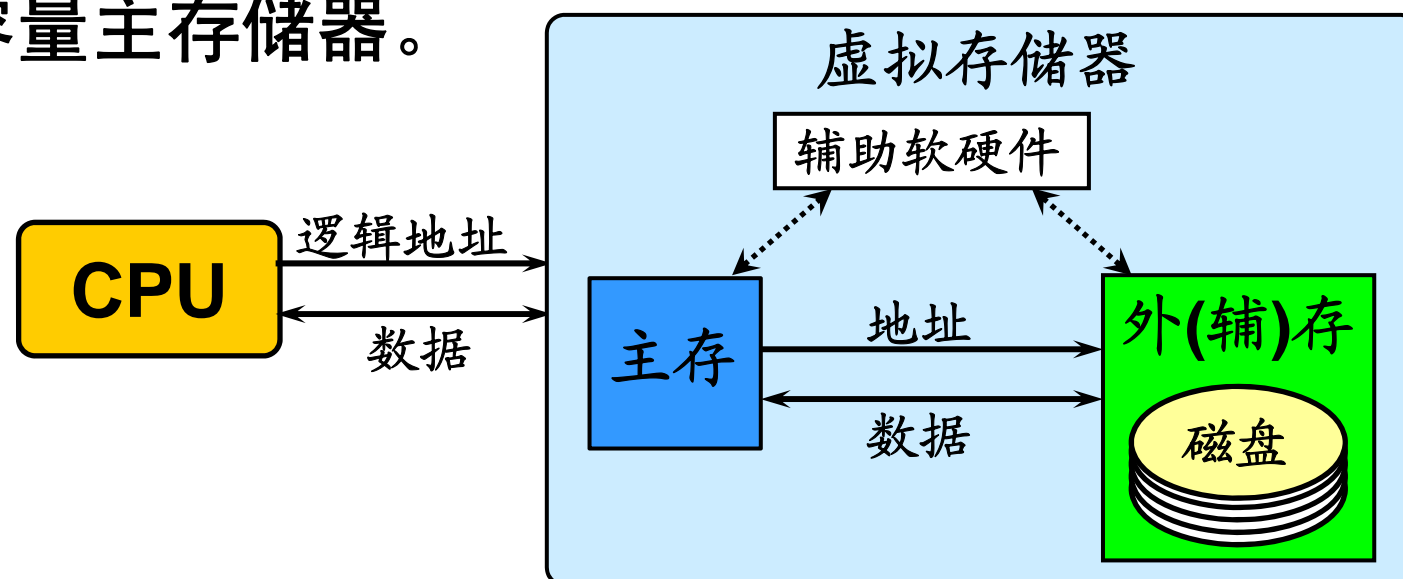
# 虚拟存储器定义及特点

## ■ 定义:

- 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。

# 虚拟存储器定义及特点

- **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。

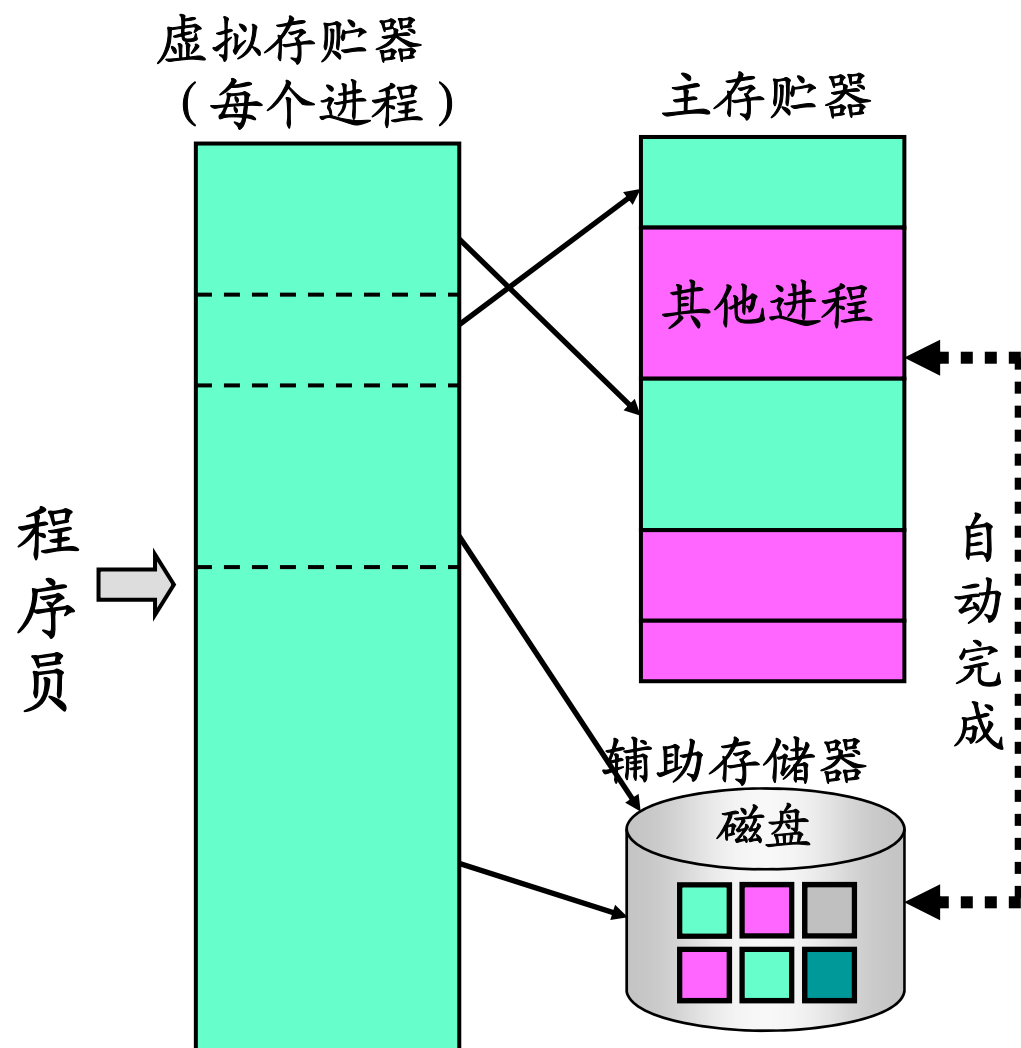


# 虚拟存储器定义及特点

## ■ 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贮器分配工作，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贮器的容量和配置；
- 多用户可以共享存贮器，使更多的程序能够进入主存运行；
- 提供了主存保护机制。

# 虚拟存储器定义及特点



- 程序员可以用机器指令的地址码对整个程序统一编址，就如同应用程序员具有对应于这个地址码宽度的存储空间(称为程序空间)一样，而不必考虑实际主存空间的大小。
- 每个用户程序都运行在不同的地址空间上。

# 地址空间

## ■ 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。

# 程序重定位

- CPU只能执行已经装入主存中的程序。

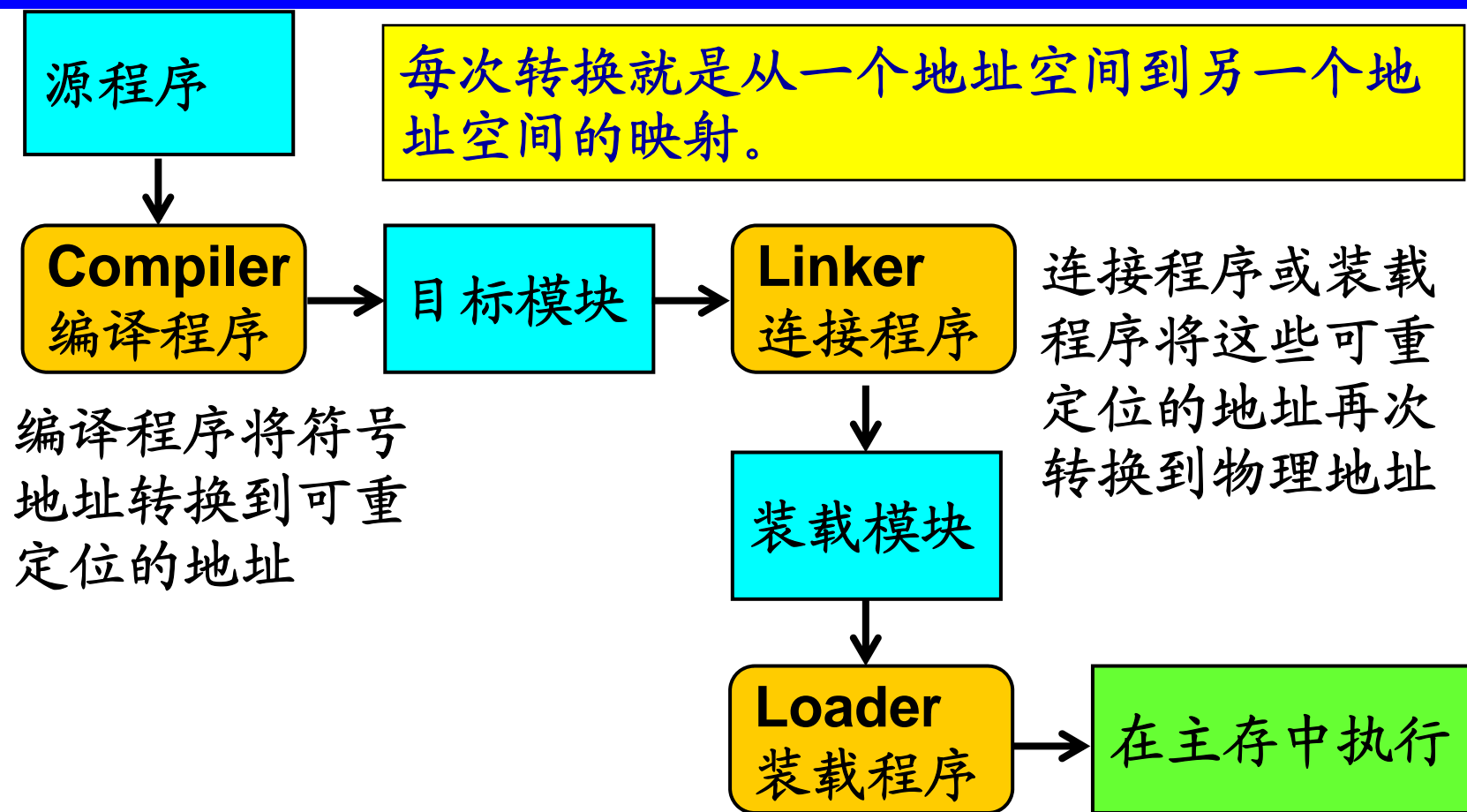
- 程序重定位:

指将程序放在主存的什么位置，即把程序地址空间中的程序地址转换为主存空间中的物理地址的地址转换。也称为地址映射或地址映像。



# 程序重定位

用户程序在执行前需要经过多个地址转换步骤，用户程序地址可能有不同的表示形式。



# 程序重定位

根据地址转换的时间及采用技术的不同，把重定位分为静态重定位和动态重定位。

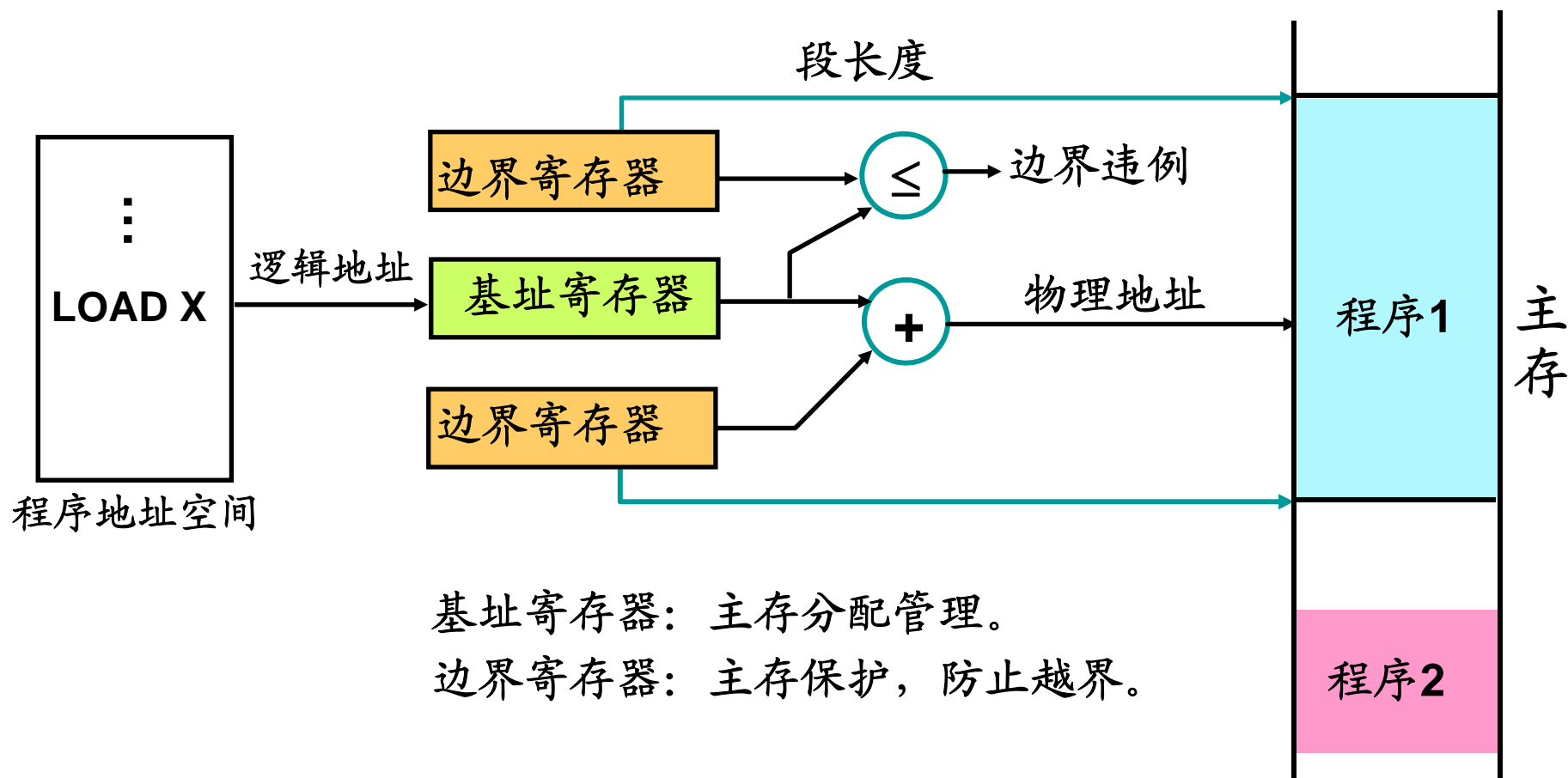
## ■ 静态重定位：

在将目标程序装入到主存时（即执行之前），由装入程序将逻辑地址变换成物理地址。

## ■ 动态重定位：

在程序执行过程中，在CPU访问主存之前，由操作系统、辅助软硬件等完成逻辑地址变换成物理地址。

# 基于基址寄存器的动态再定位



基址寄存器：主存分配管理。

边界寄存器：主存保护，防止越界。

## 基于基址寄存器的动态再定位

## 4.2.1 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。

虚拟存贮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贮器管理方式，主要有三种：

- 段式管理
- 页式管理
- 段页式管理

# 1. 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。

# 1. 段式管理

段表的基本结构

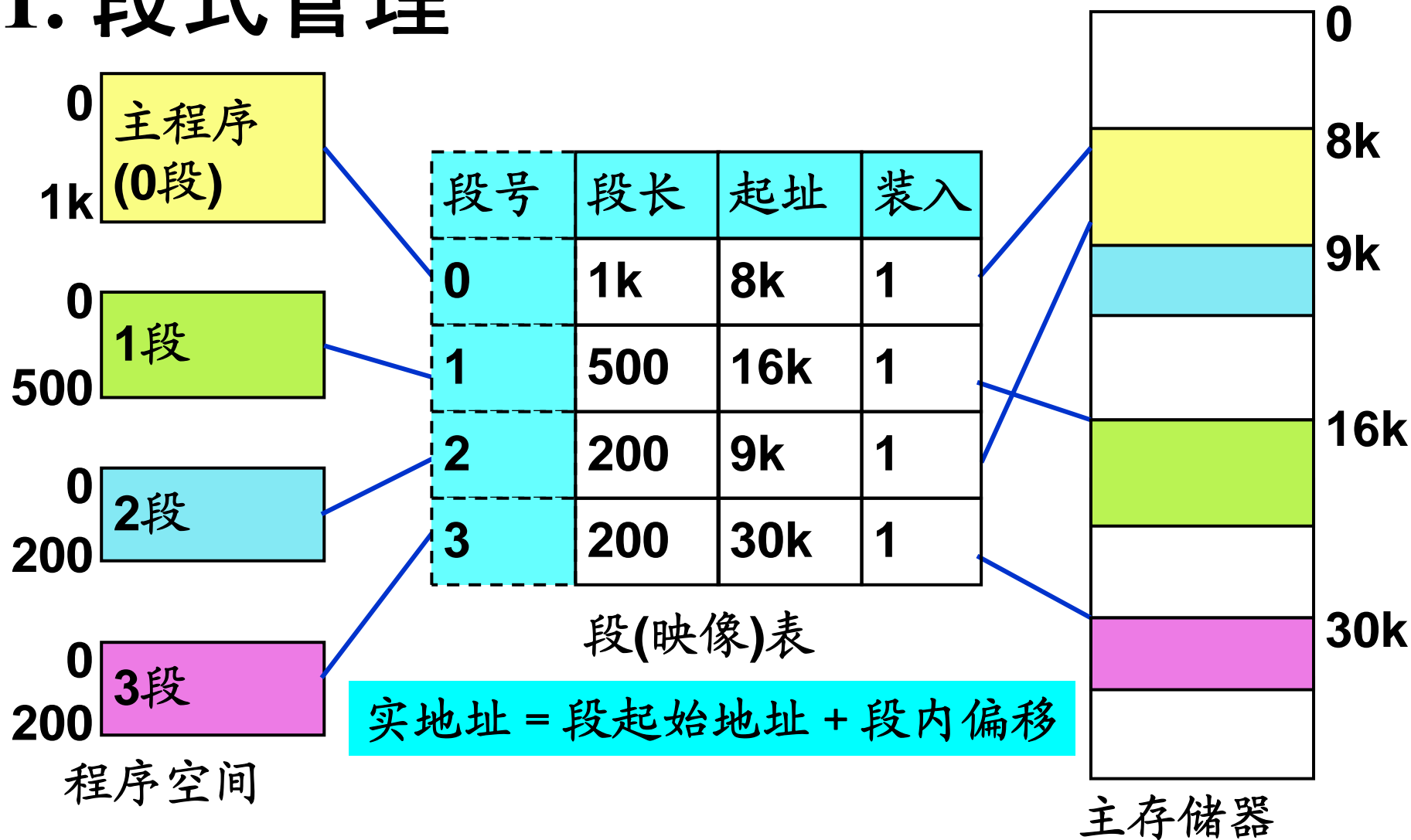
段号	地址	段长	装入位	访问方式
	⋮			

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。

# 1. 段式管理



段式管理中的某道程序的地址变换

# 1. 段式管理

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	段表基址
------	------

段表长度：指明段表的行数，即程序的段数；

段表基址：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



# 1. 段式管理

当有多道程序时（续）：

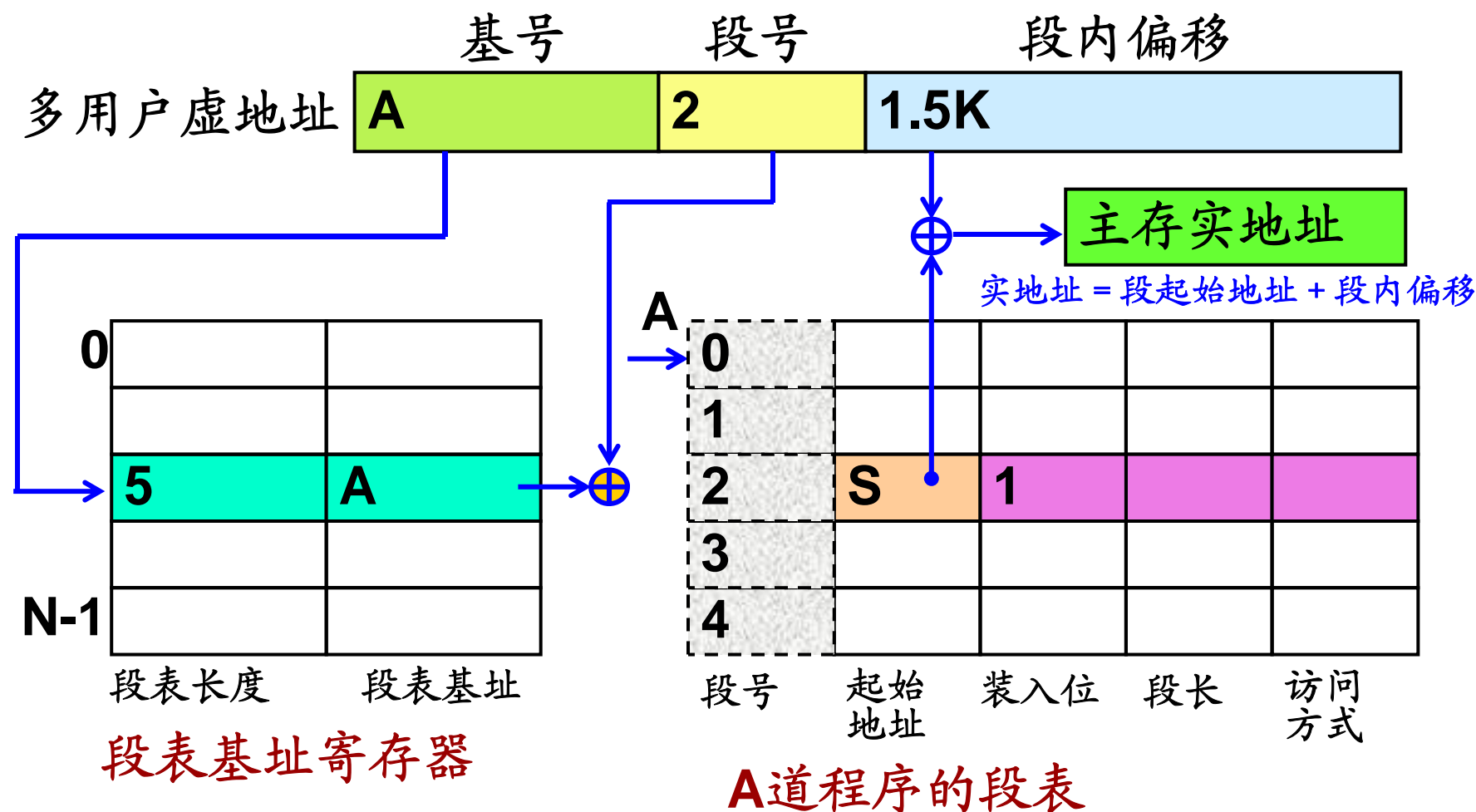
- 系统赋给每道程序一个基号，指明其使用哪一个段表基址寄存器；
- 用户虚地址（指令中的地址码）与该基号结合形成系统的多用户虚地址：



多用户虚地址

- 通过查表即可自动转换成主存的物理地址；

# 1. 段式管理



段式管理中的多道程序时的地址变换

# 1. 段式管理

## ■ 优点:

- 便于程序和数据的共享;
- 易于以段为单位实现存贮保护。

## ■ 缺点:

- 段长变化大, 程序段在主存中的起点随意, 给主存的分配带来困难;
- 主存利用率可能下降, 因为可能存在不可用的段内零头和段间零头;
- 地址变换所花费的时间比较长, 需做两次加法运算;
- 对辅存(磁盘存储器)的管理比较困难, 因为象磁盘是按固定长度的块访问的。

## 2. 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

- ☑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**



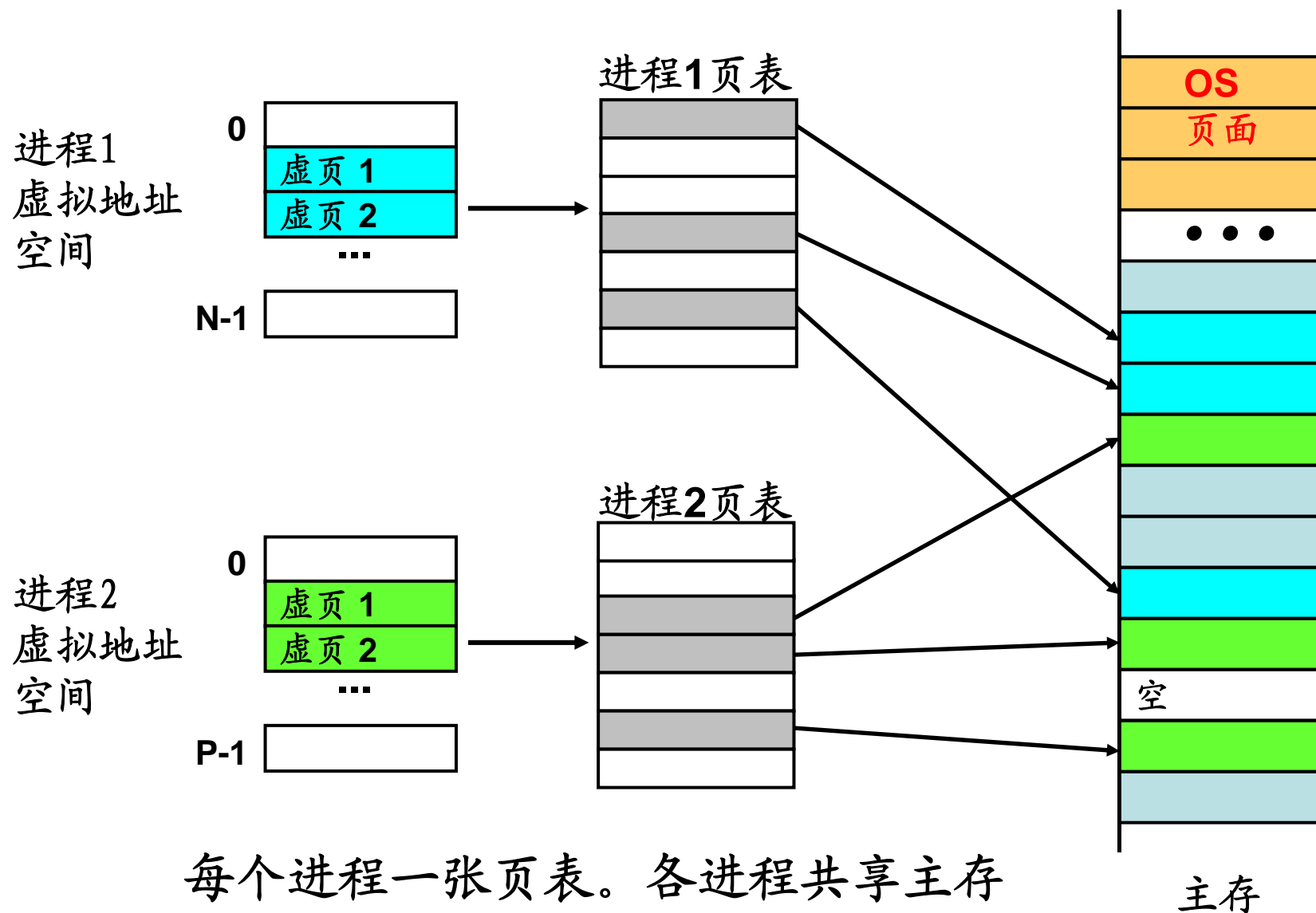
程序空间的虚地址结构

## 2. 页式管理

### ■ 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。

## 2. 页式管理



## 2. 页式管理

页表的基本结构

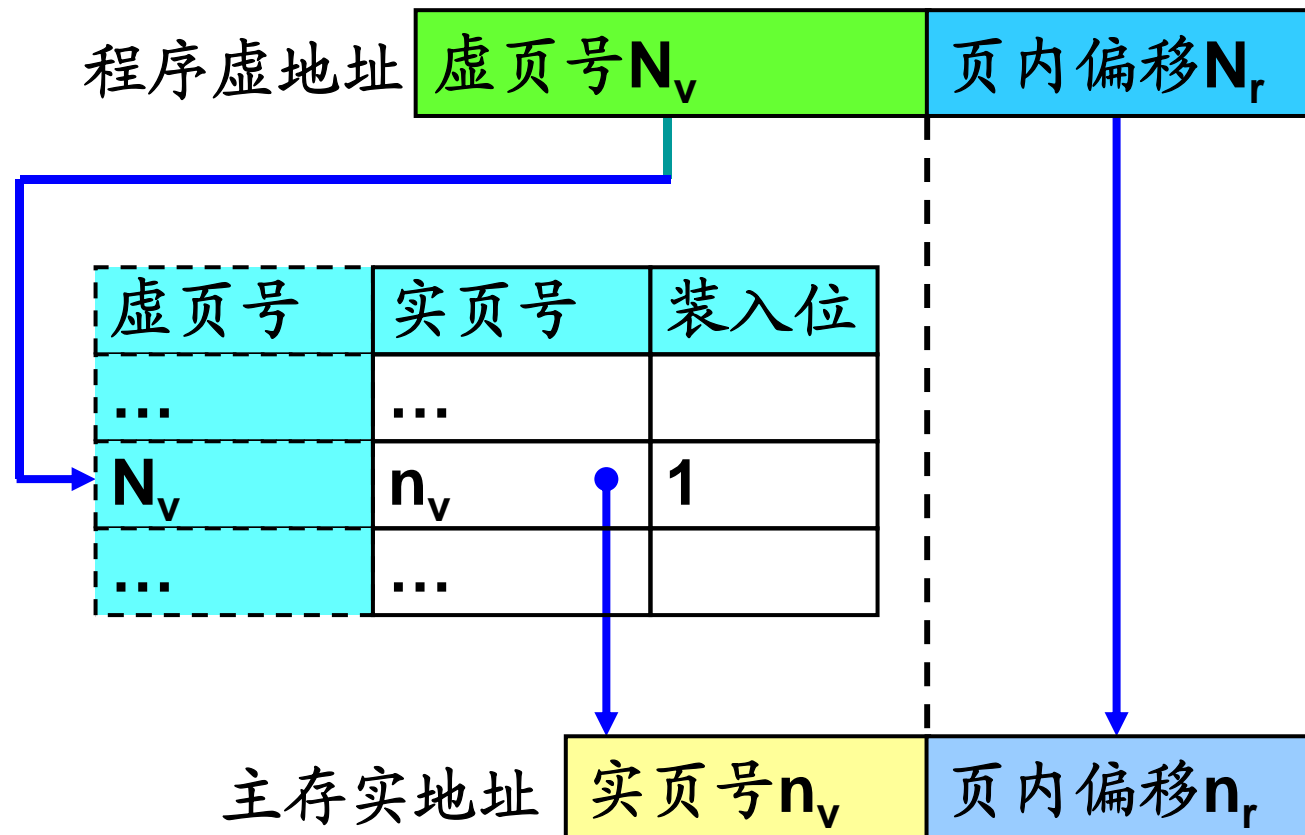
虚页号	实页号	装入位	修改位	专用位
	⋮			

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。

## 2. 页式管理



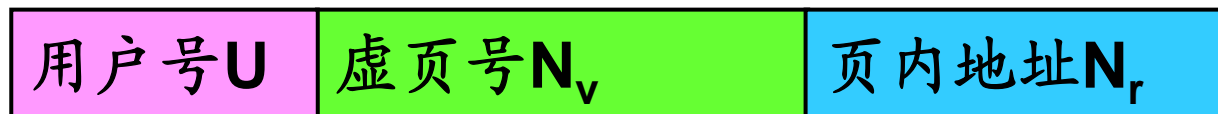
利用页表的地址变换



## 2. 页式管理

当有多道程序时：

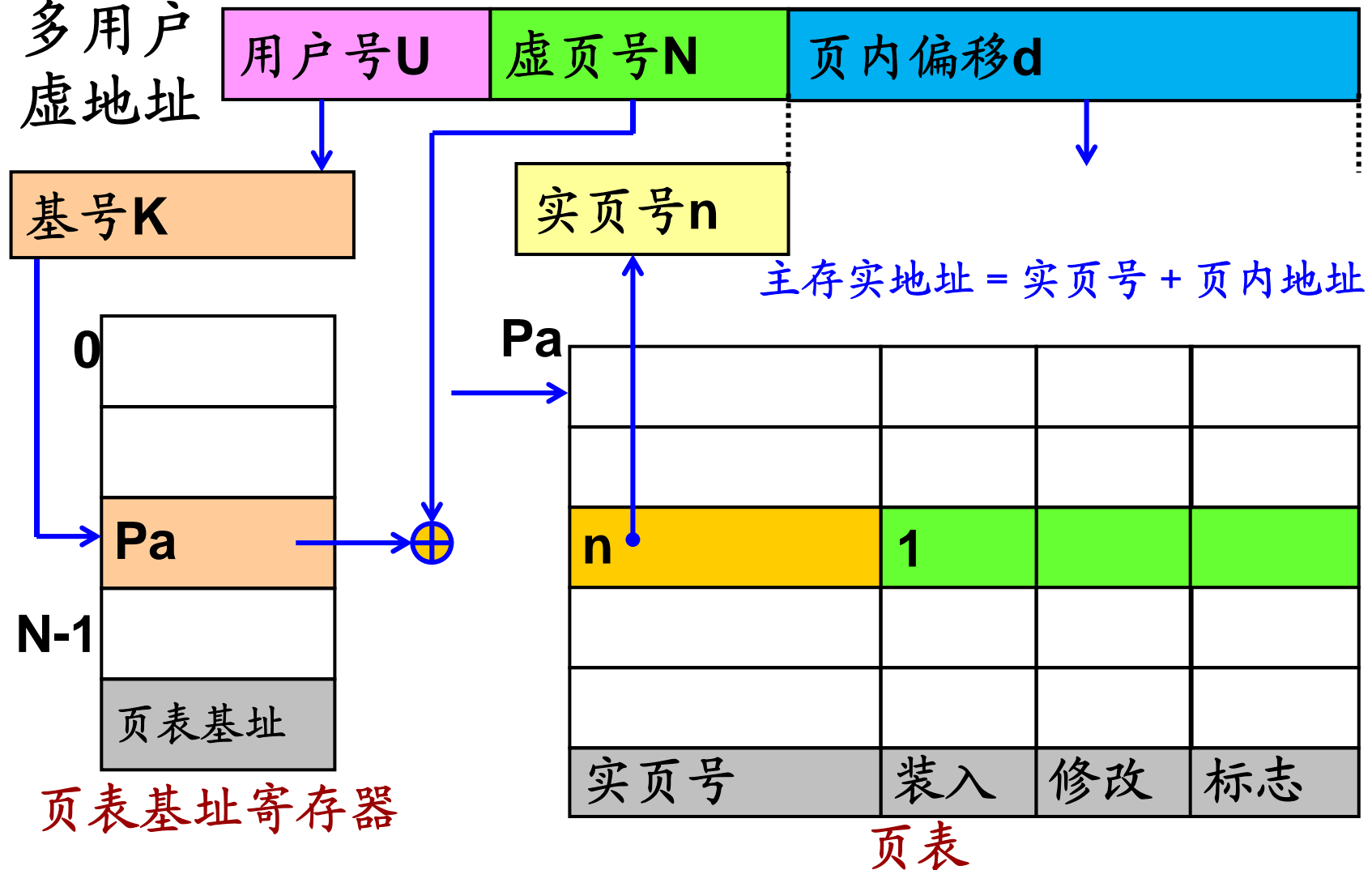
- 假设系统中最多可以有 $N$ 道程序，因此有 $N$ 个页表。
- 可以设置 $N$ 个页段表基址寄存器指示每道程序所对应的页表的基址（起始地址）。
- 系统赋给每道程序一个用户号，转换为基号后，指明其使用哪一个页表基址寄存器；
- 用户虚地址（指令中的地址码）与该用户号结合形成系统的多用户虚地址：



多用户虚地址结构

## 2. 页式管理

多用户  
虚地址



页式管理中的多道程序时的地址变换

## 2. 页式管理

### ■ 优点:

- 对程序员完全透明;
- 所需映象表硬件少, 查表和地址变换速度快;
- 零头浪费减少, 主存利用率优于段式管理;
- 磁盘管理容易。

### ■ 缺点:

- 页不能表示一个完整的程序功能, 程序或模块的独立性、保护、共享等的实现较为困难;
- 页表占用空间多。

## 2. 页式管理

### ■ 注意:

- 页表本身也按页管理;

若页表大小超过了一个页面的大小，上述查表方法就不正确了。  
需要采取措施。例如采用多级页表。

# 页式管理和段式管理

Aspect	Page	Segment
Words/Address	One - contains page and offset	Two - possible large max-size hence need Seg and offset address words
Programmer visible	No	Sometimes yes
Replacement	Trivial - due to fixed size	Hard - need to find contiguous space ==> GC necessary or wasted memory
Memory Inefficiency	Internal fragmentation - wasted part of a page	External fragmentation - due to variable size blocks
Disk Efficiency	Yes - adjust page size to balance access and transfer time	Not always - segment size varies

# 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：

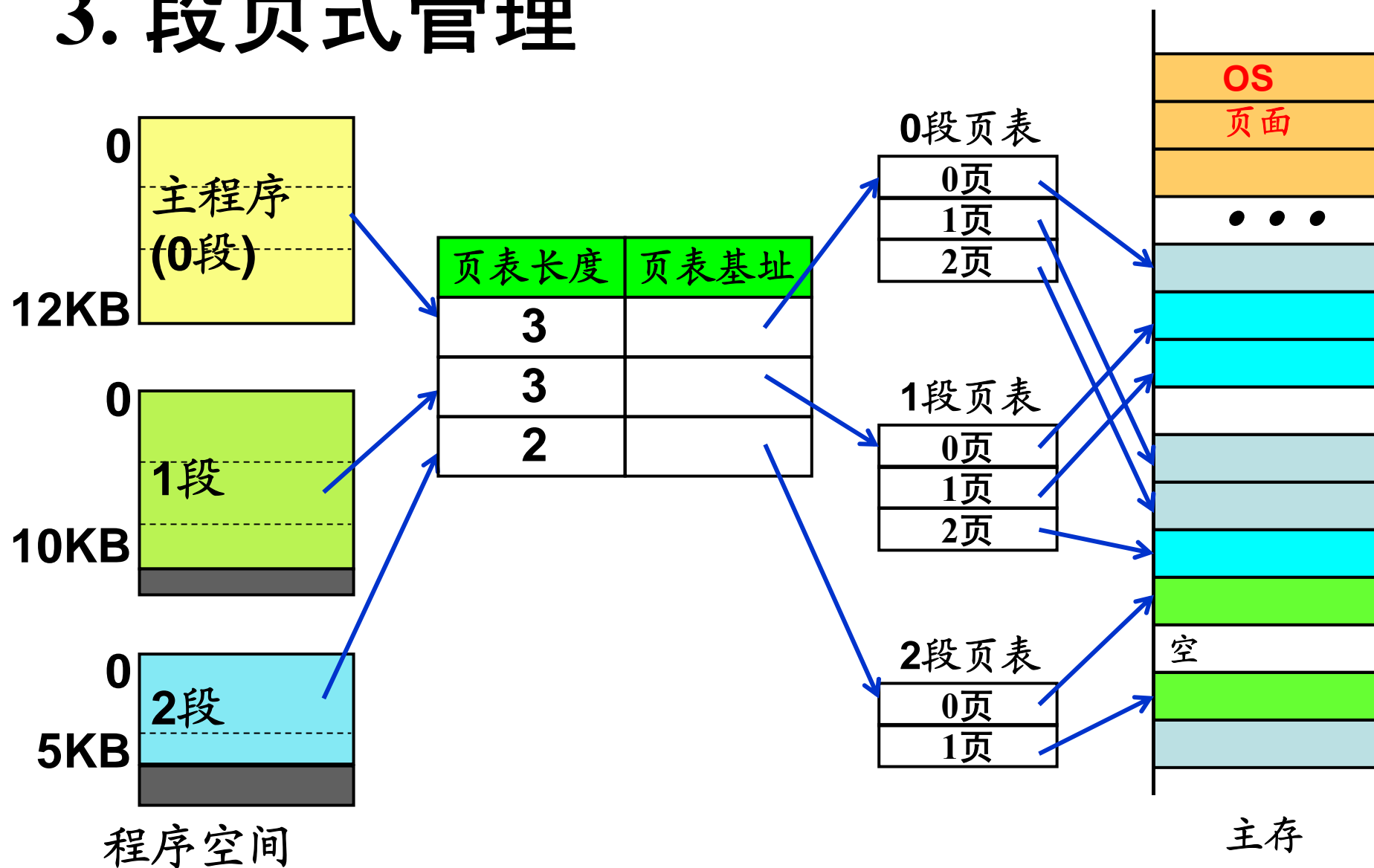
- 将主存空间等分成固定大小的页（称为**实页**），并按顺序编号；
- 将程序先按模块**分段**，每段再等分成固定大小（与实页大小相同）的页（称为**虚页**），并按顺序编号（**段的长度必须是页长度的整数倍**）；
- 每道程序通过**一个段表和一组页表**进行定位；

# 3. 段页式管理

## 一般原理（续）

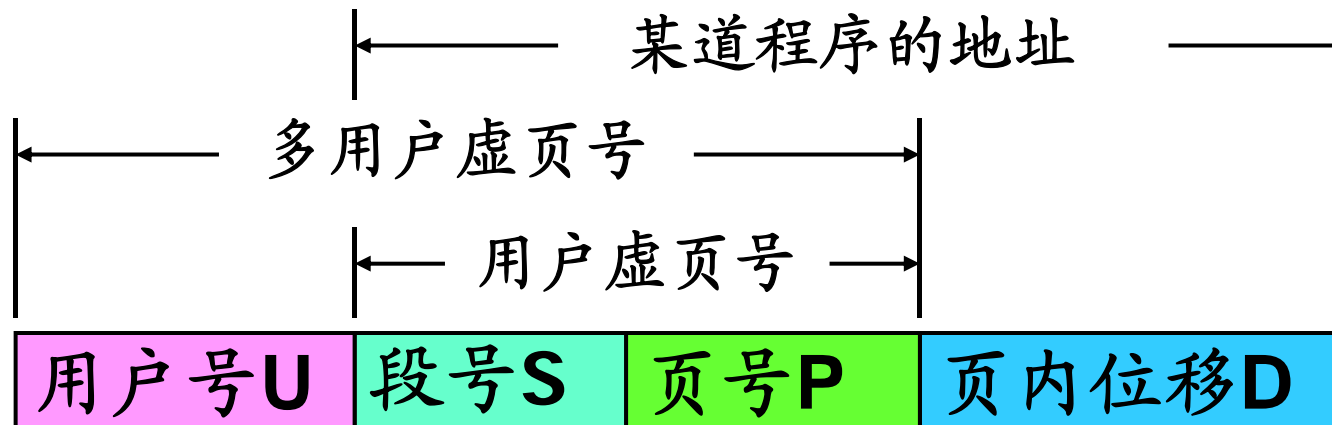
- 段表中的每一行对应一个段。地址字段指示该段的页表在主存中的起始位置，段长字段指示该段的页表的行数（虚页数）；
- 每个段都有一个页表。页表的地址字段指示当该页已经装入主存时，该页在主存中的实页号；
- 对于多道程序系统，每道程序都需要有一个用户号（需转换为基号），以指明该道程序的段表起点在哪一个段表基址寄存器中。

# 3. 段页式管理





### 3. 段页式管理

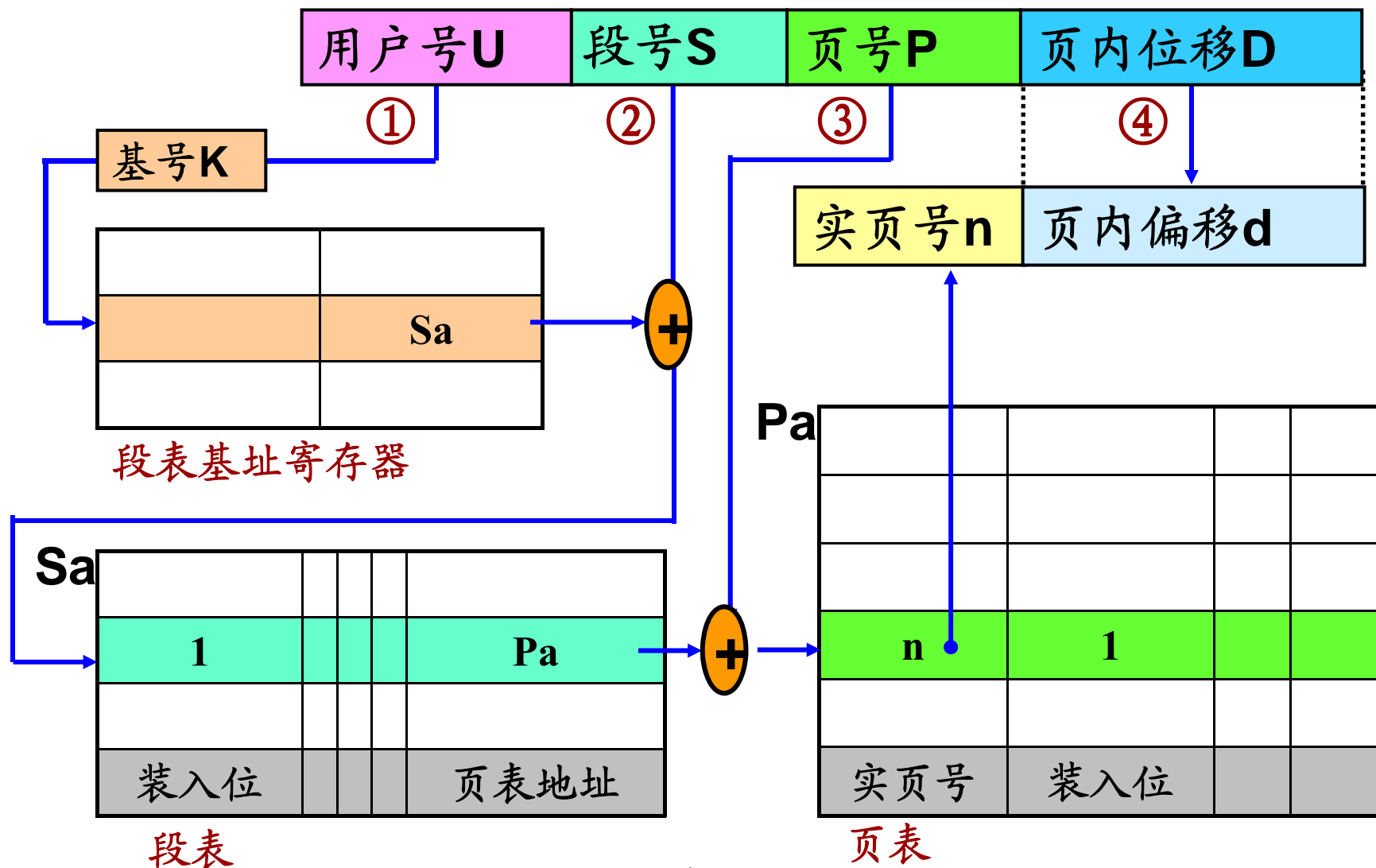


多用户虚地址结构



主存单元的实地址结构

# 3. 段页式管理



段页式管理中的多道程序时的地址变换

### 3. 段页式管理

#### ■ 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:

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

其中:

$N_p$ 为页面的大小

$N_v$ 为虚拟存储空间大小

$N_d$ 为一个页表存储字的大小

### 3. 段页式管理

- 例4-7：虚拟存储空间大小 $N_v=4\text{GB}$ ，页的大小 $N_p=1\text{KB}$ ，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：

$$g = \left\lceil \frac{\log 24G - \log 21K}{\log 21K - \log 24} \right\rceil = \left\lceil \frac{32 - 10}{10 - 2} \right\rceil = 3$$

**1KB**页面，1页只有**256**个存储字；**3级**页表。故： **$256 \times 256 \times 64 = 4\text{M}$** 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

- 三种不同的虚拟存贮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。

## 4.2.2 页式虚拟存储系统

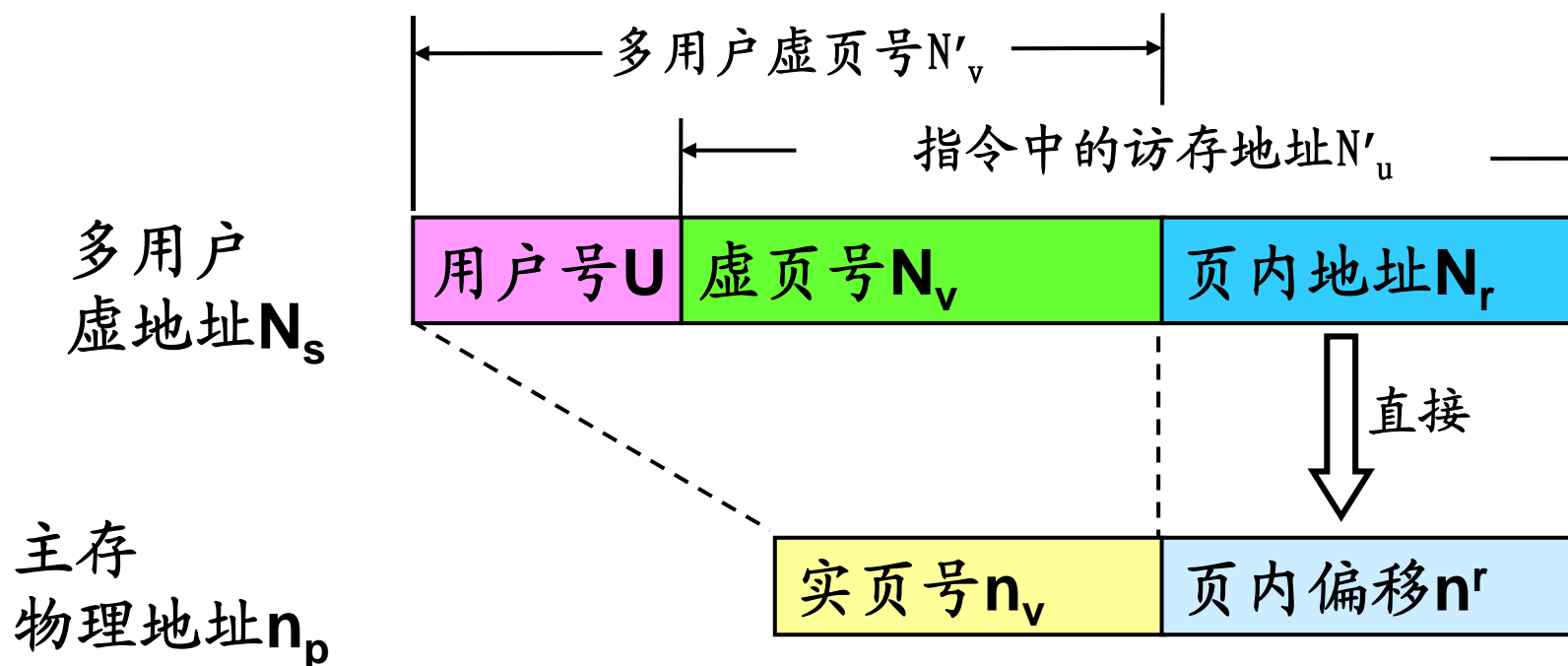
- 页式虚拟存储系统是采用页式存贮管理的主-辅存存贮层次。
- 一般来说，虚拟存储系统中的用户程序空间比实际主存空间大得多。反映在页号上，就是：

虚页号 $N_v$  >> 实页号 $n_v$

- 此外，主存空间是被多个用户共享的。

## 4.2.2 页式虚拟存储系统

- 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

## 4.2.2 页式虚拟存储系统

### ■ 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 $N_s$ 与实主存地址 $n_p$ 之间的对应关系。

# 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 $2^{nv}$ （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。

# 全相联地址映象与变换

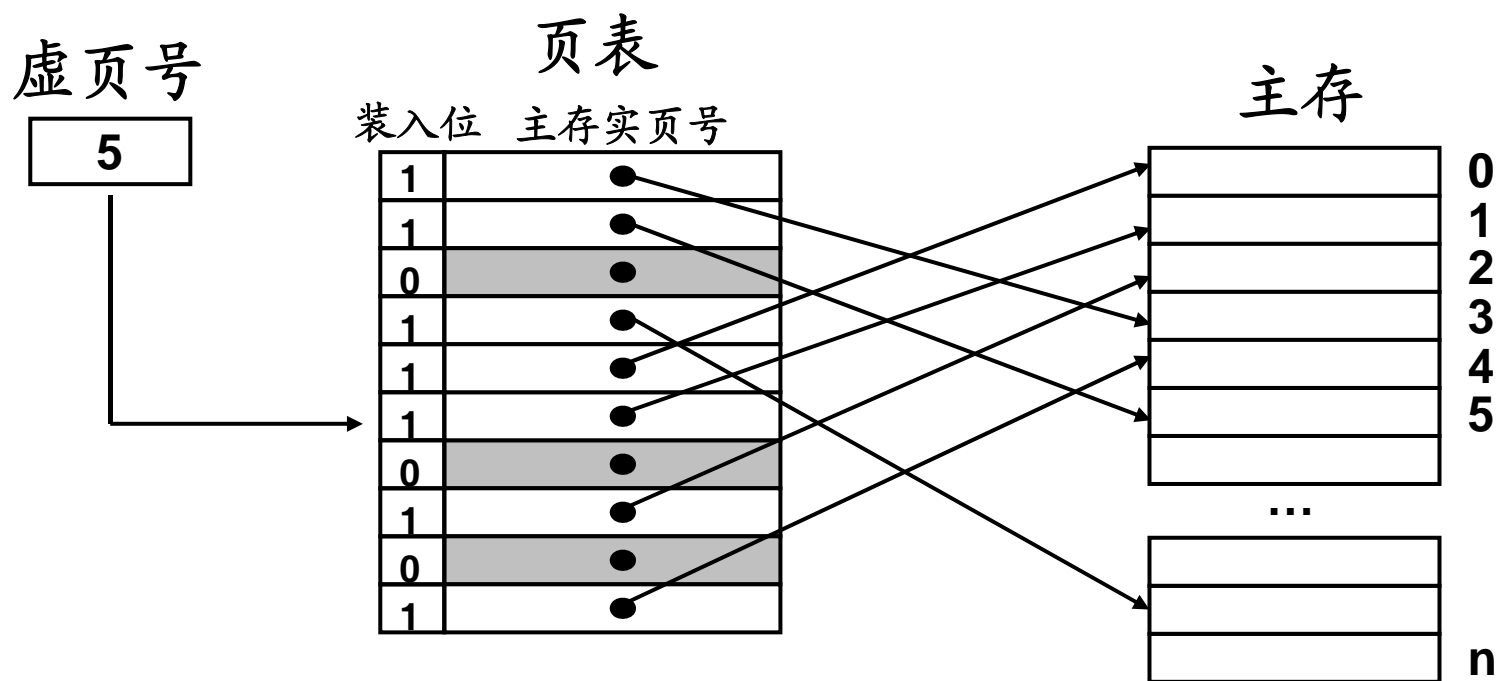
## 地址变换的两种方法:

- 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。

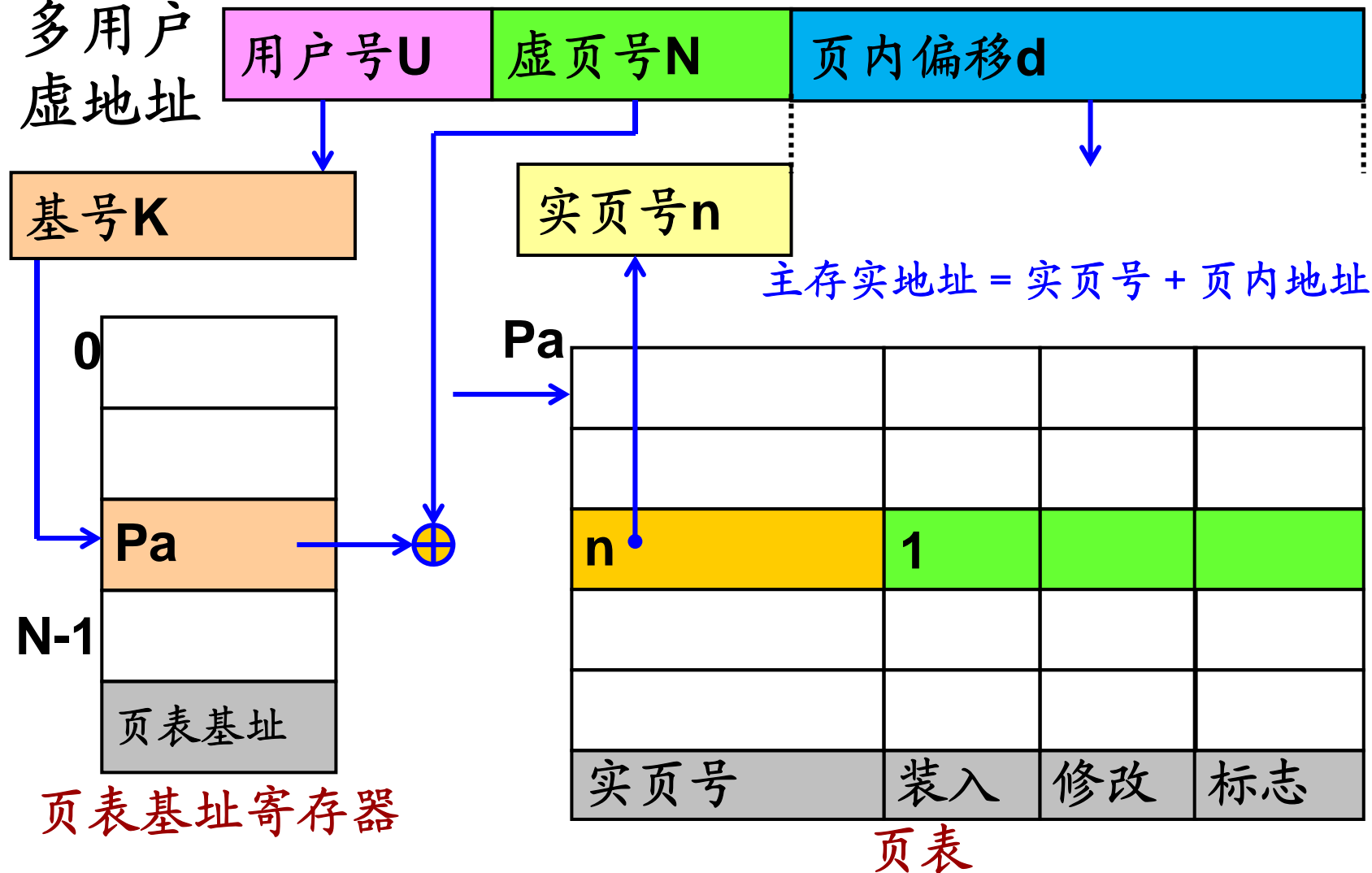
# 页表法

- 采用**页表**作为地址映像表，通过查表实现地址变换。



# 页表法

多用户  
虚地址




页式管理中的多道程序时的地址变换

# 页表法

## ■ 页表法的不足:

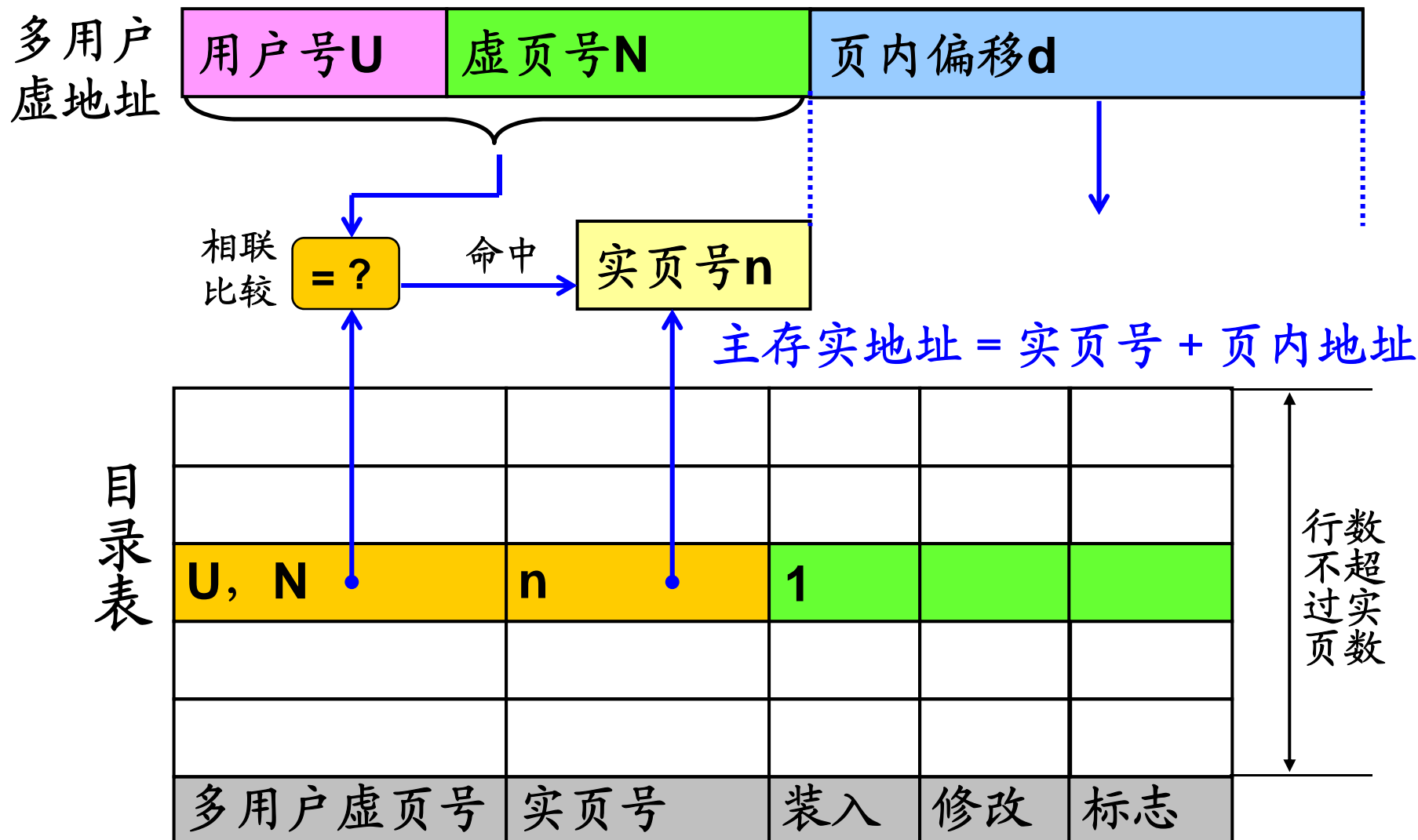
- 每道程序页表的表长=虚页数( $2^{N_v}$ );
- $N$ 道程序的页表行数将达到  $N * 2^{N_v}$ , 远大于实页数 $2^{n_v}$ ;
- 而装入位为 1 的行数最多只能有实页数 ( $2^{n_v}$ ) 个, 其他行将成为无用的行, 大大降低了页表的空间利用率。

压缩页表空间  相联目录法

# 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  $2^{n_v}$  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 $N_v$ 标识虚页号。地址变换时按内容（即基号B和用户虚页号 $N_v$ ）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。**

# 相联目录法



页式虚拟存储器利用目录表的虚实地址变换



# 页表法与相联目录表法

## ■ 价格

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

## ■ 速度

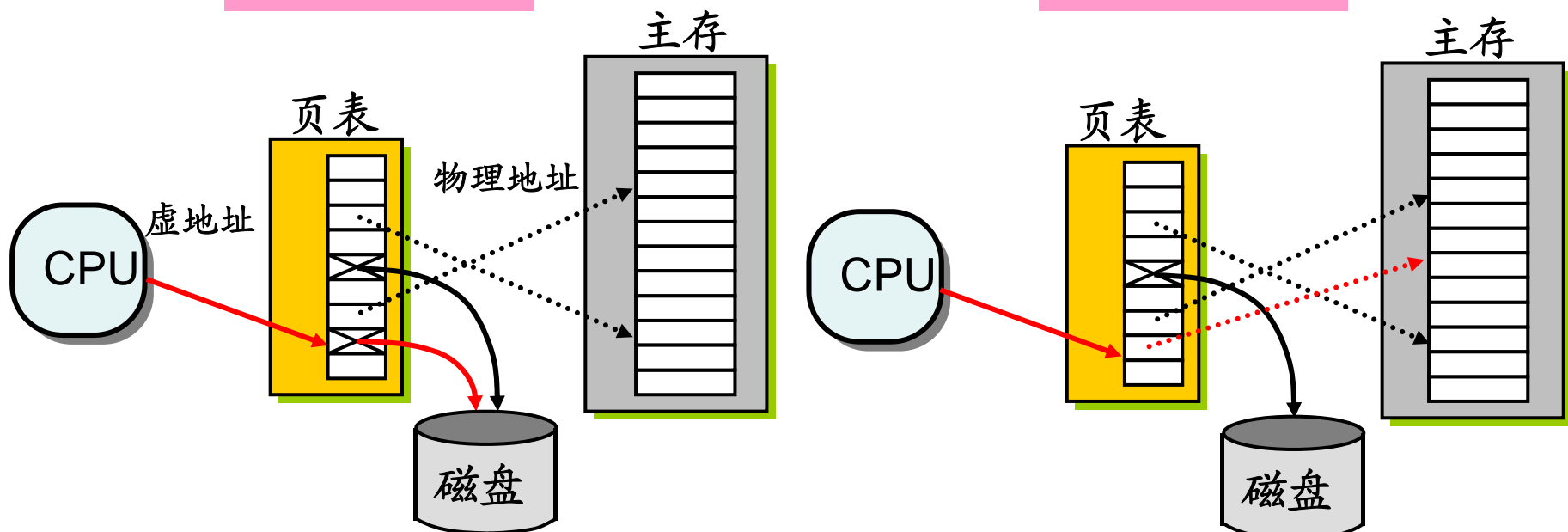
- 页表法稍快，目录表法慢。

一般，在虚拟存贮器中不直接采用目录表法来存贮全部虚页号与实页号的对应关系。

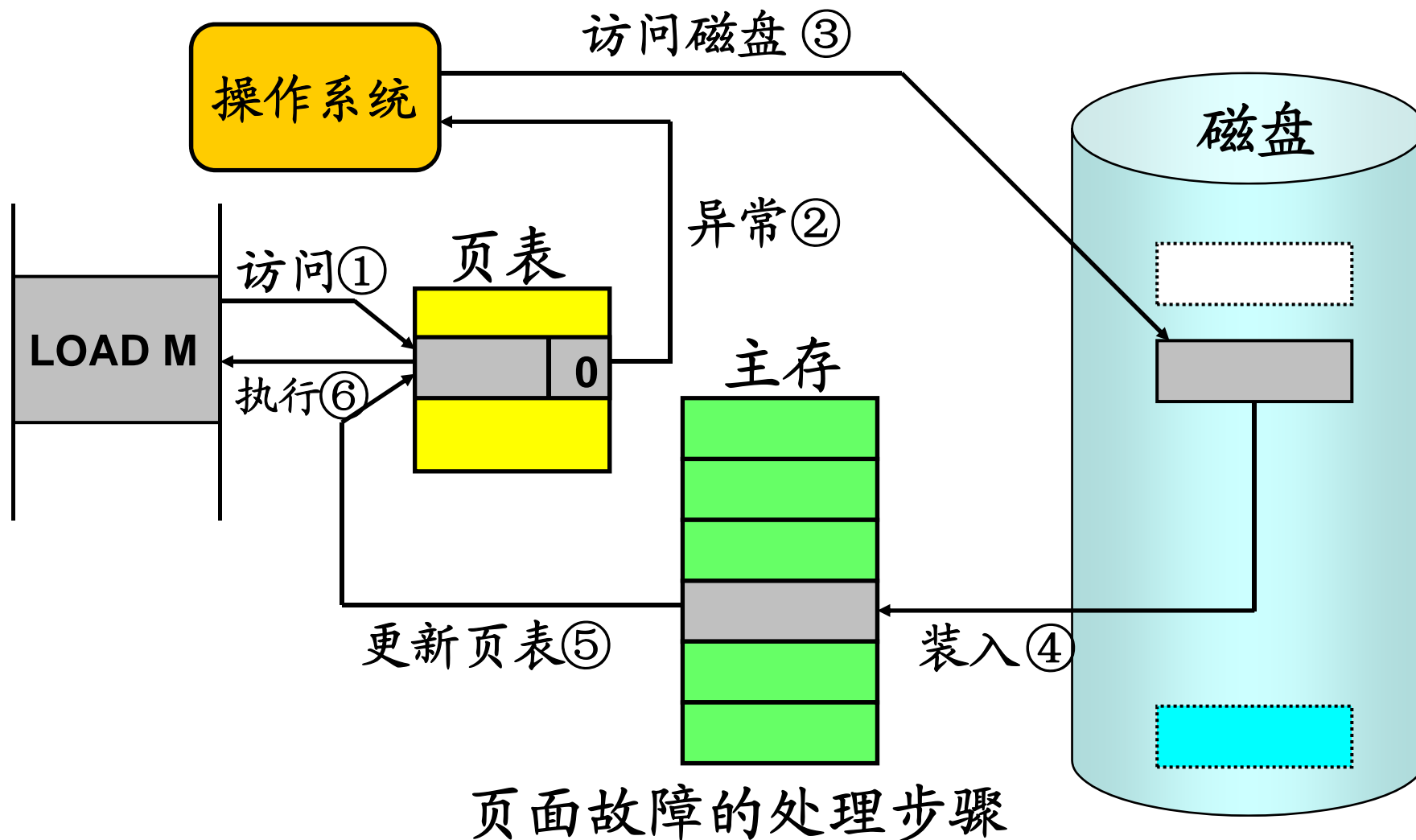
但这种思想可以被用来提高地址变换速度。例如，设置一个高速小容量硬件目录表，存放近期最常用的虚—实页对应关系——**快表**。

## 2. 页面故障（页面失效）

- 若页表指示多用户虚地址 $N_s$ 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。

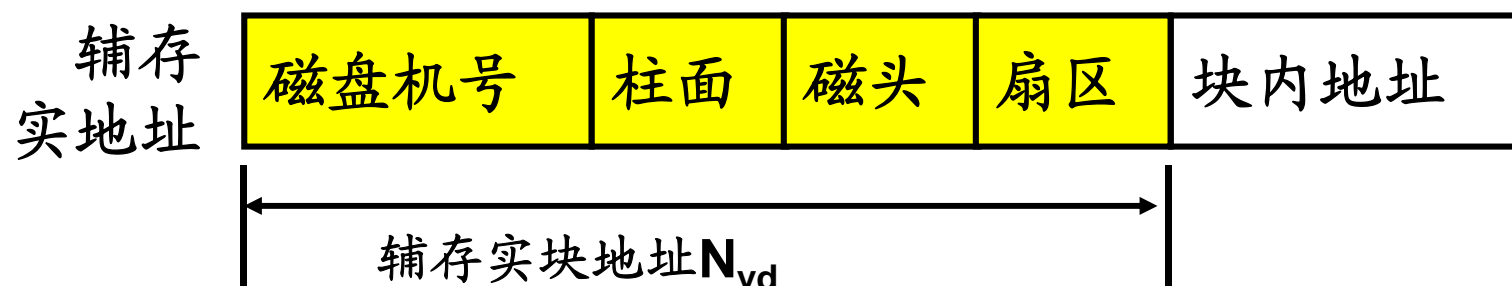


## 2. 页面故障



# 虚地址—辅存实地址变换

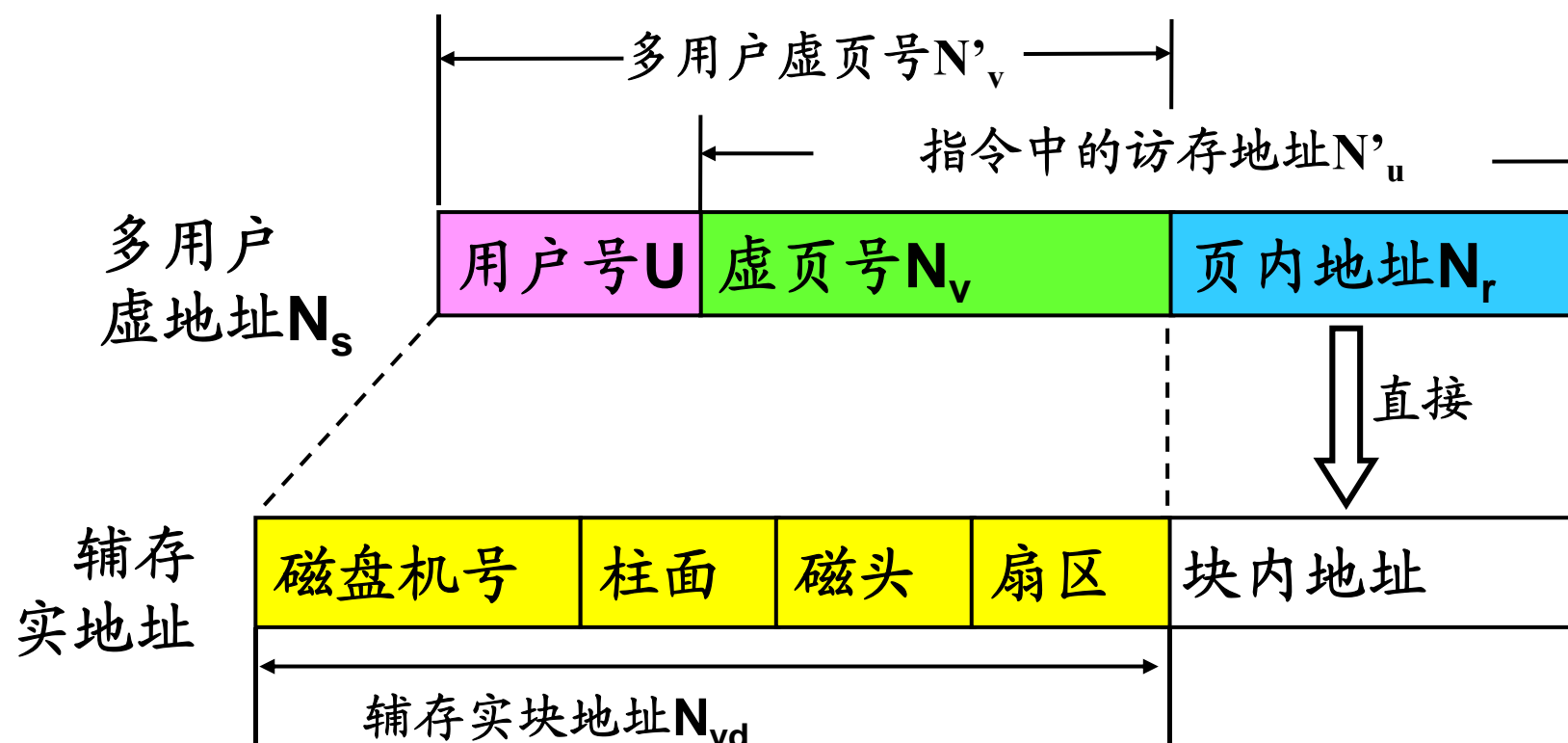
- 若要把该道程序的虚页调入主存，必须给出该虚页在辅存中的实际地址，即**实现虚地址到辅存实地址的变换**。
- 由于是从外部辅存中调页，因此将虚地址到外部辅存实地址的变换称之为**外部地址变换**。
- 为提高调页效率，辅存一般按信息块编址，而且让信息块的大小等于页面的大小。



磁盘的辅存实地址的格式

# 虚地址-辅存实地址变换

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 $N_{vd}$ 。

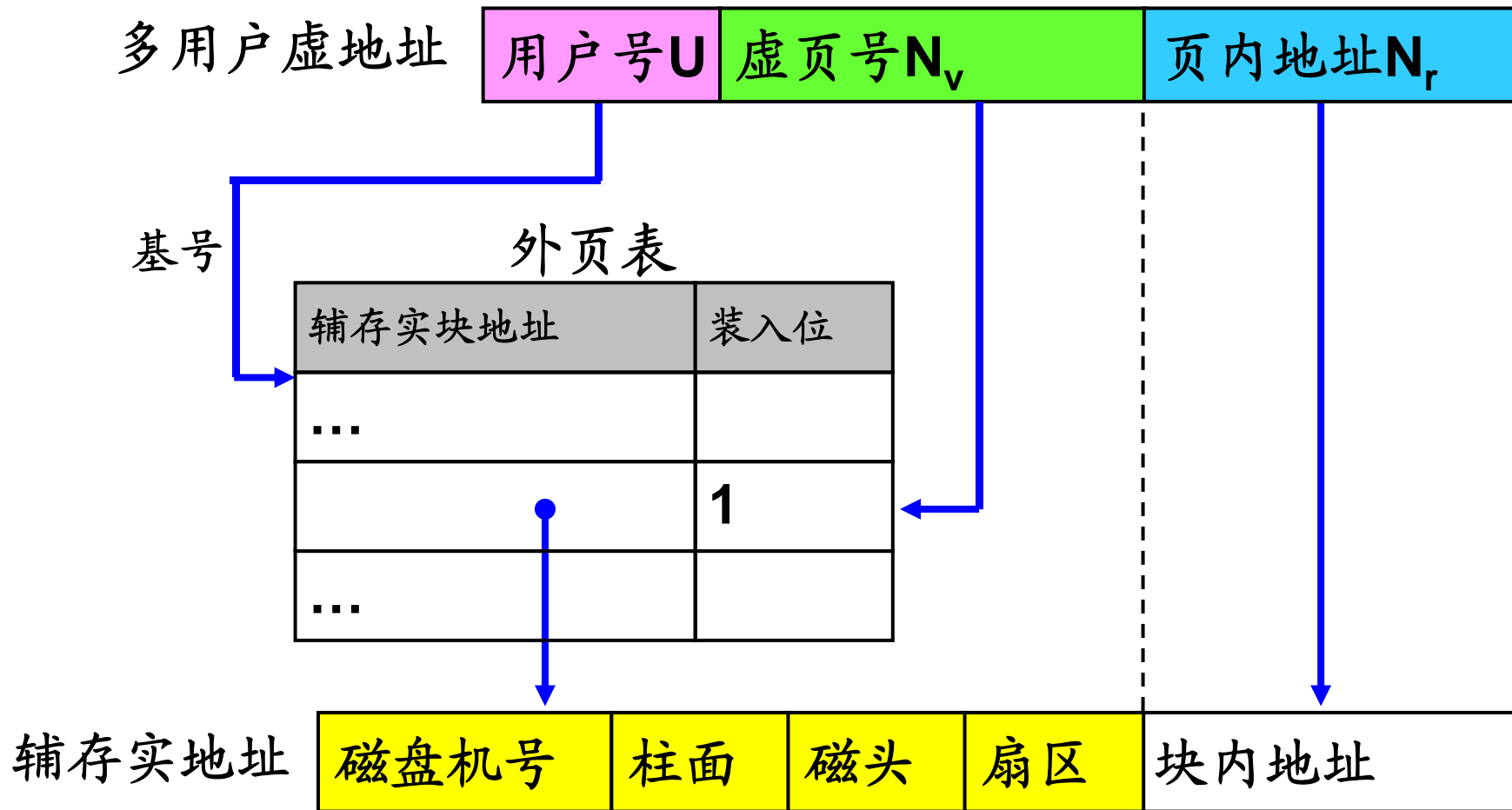


多用户虚地址到辅存物理地址的变换

# 虚地址-辅存实地址变换

- 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  $N_{vd}$  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

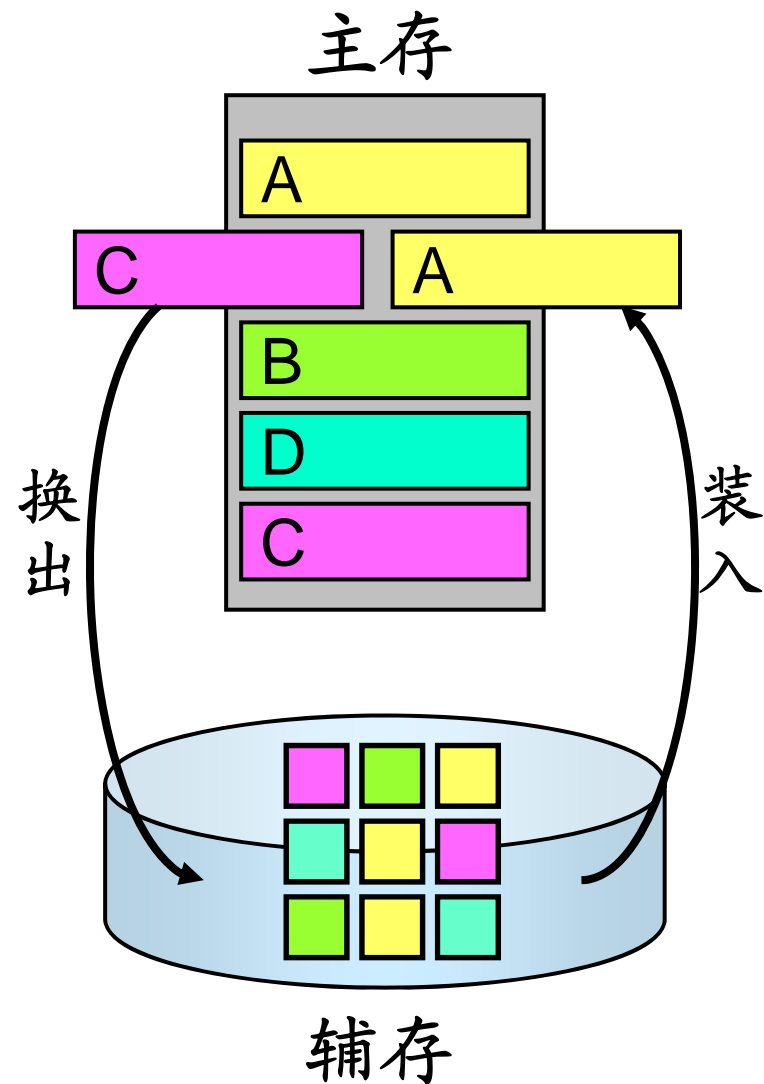
# 虚地址-辅存实地址变换



利用外页表的虚地址-辅存实地址的地址变换

### 3. 页面替换

- 在虚拟存贮器中，通常主存容量远小于辅存容量。
- **问题：**如果主存已满，同时发生页面故障，此时，只有强制腾出主存中的某个页面，以存放由辅存调入的新页。
- **根据何种算法、应将哪个虚页替换出去？**





# 3. 页面替换

## ■ 确定替换算法的原则:

- 是否有高命中率
- 是否易于实现
- 辅助软、硬件成本是否低廉

# 3. 页面替换算法

在虚拟存储系统中，实际上有可能采用的只有**FIFO**和**LRU**两种算法。

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

- 选最早装入主存的页作为被替换的页。

## ■ 近期最少使用（LFU）算法

- 选择近期最少访问的页作为被替换的页。

## ■ 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页作为被替换的页。

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，从而确定被替换的页面。

# 3. 页面替换算法

## ■ 随机（RAND）算法

- 用随机数产生器来形成被替换页的页号。
- 优点：简单、易于实现
- 缺点：命中率很低
  - ◆ 因为没有利用主存使用的“历史”，反映不了程序的局部性

# 3. 页面替换

## ■ 先进先出（FIFO）算法

- 选最早装入主存的页作为被替换的页。
- 优点：简单、易于实现
  - ◆ 由OS在主存页面表中为每个实页设置一个计数器字段。调入新页时，新装入页的计数器为0，其他页则加1。替换计数器值最大的那个页
- 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. 页面替换

## ■ 近期最少使用（LFU）算法

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

# 3. 页面替换

## ■ 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页作为被替换的页。
- 该算法将LRU中的“多”和“少”简化成“有”和“无”，实现方便。

# 页面替换

- 近期最久未使用（LRU）算法（续2）

- ✓ 在主存页面表中为每个实页设置一个使用位字段
- ✓ 开始时，所有的使用位字段=0。以后若某页被访问过，则置使用位字段=1
- ✓ 当占用位都为1，同时又发生页面失效时，选择使用位字段=0的页面进行替换
- ✓ 显然，使用位字段不能都为1，否则无法确定该替换哪个页
- ✓ 使使用位字段不同时为1的方法有两种：
  - ◆ 随机期法
  - ◆ 定期法

# 3. 页面替换

## ■ 优化替换（OPT）算法

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



# 3. 页面替换

## ■ 优化替换（OPT）算法（续）

- 在时刻  $t$  找出主存中每个页将要用到的时刻  $t_i$ 。
- 选择  $t_i - t$  最大的页面进行替换。
- 但其实现是不现实的
  - ◆ 需要让程序运行两次。第一次是得到页地址流，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。

### 3. 页面替换

■ 例1：一个程序共有5个页面组成，程序执行过程中的页地址流如下：

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

假设分配给这个程序的主存储器共有3个页面。分析FIFO、LRU、OPT页面替换算法对这3页主存的使用情况，包括调入、替换和命中率等。

# 3. 页面替换

程序页地址流

1	2	1	5	4	1	3	4	2	4
---	---	---	---	---	---	---	---	---	---

**FIFO**

主存页面  
使用情况

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

**FIFO命中率**  
**=2/10**

调入 调入 命中 调入 替换 替换 替换 命中 替换 替换

**LRU**

主存页面  
使用情况

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

**LRU命中率**  
**=4/10**

调入 调入 命中 调入 替换 命中 替换 命中 替换 命中

# 3. 页面替换

程序页地址流

1	2	1	5	4	1	3	4	2	4
---	---	---	---	---	---	---	---	---	---

**OPT**  
主存页面  
使用情况

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

**OPT命中率  
=5/10**

调入	调入	命中	调入	替换	命中	替换	命中	命中	命中
----	----	----	----	----	----	----	----	----	----

### 3. 页面替换

- 例2: 一个循环程序, 依次使用1、2、3、4四个页面, 分配给这个程序的主存页面数为3个。请分析FIFO、LRU、OPT页面替换算法对主存页面的使用调度情况。

# 3. 页面替换

程序页地址流

1	2	3	4	1	2	3	4	1	2
---	---	---	---	---	---	---	---	---	---

**FIFO**

主存页面  
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

**FIFO命中率**  
**=0/10**

调入 调入 调入 替换 替换 替换 替换 替换 替换 替换

**LRU**

主存页面  
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

**LRU命中率**  
**=0/10**

调入 调入 调入 替换 替换 替换 替换 替换 替换 替换

# 3. 页面替换

程序页地址流

1	2	3	4	1	2	3	4	1	2
---	---	---	---	---	---	---	---	---	---

**OPT**

主存页面  
使用情况

	1	1	1	1	1	1	1	1	2
		2	2	2	2	2	3	3	3
			3	4	4	4	4	4	4

**FIFO命中率**  
**=4/10**

调入

调入

调入

替换

命中

命中

替换

命中

命中

替换

### 3. 页面替换

从上述两个例子中可以看到：

- FIFO命中率最低。
- 对某种页地址流，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页面故障——**颠簸**现象。
- **颠簸**：

下次就要使用的页面本次被替换出去而发生的连续页面故障的现象。



### 3. 页面替换

- 命中率与页地址流有关，也与分配给该道程序的实页数有关。
- 通常，随着实页数的增加，虚页进入主存的机会就越多，命中率就可能越高。
- 问题：会提高吗？（书图4-33，4-34）
- 对于LRU算法，命中率随着实页数的增加而增加
- 对于FIFO算法，命中率可能随着实页数的增加，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  $t$ ，主存页面数集合  $B_t$  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

# 堆栈型替换算法

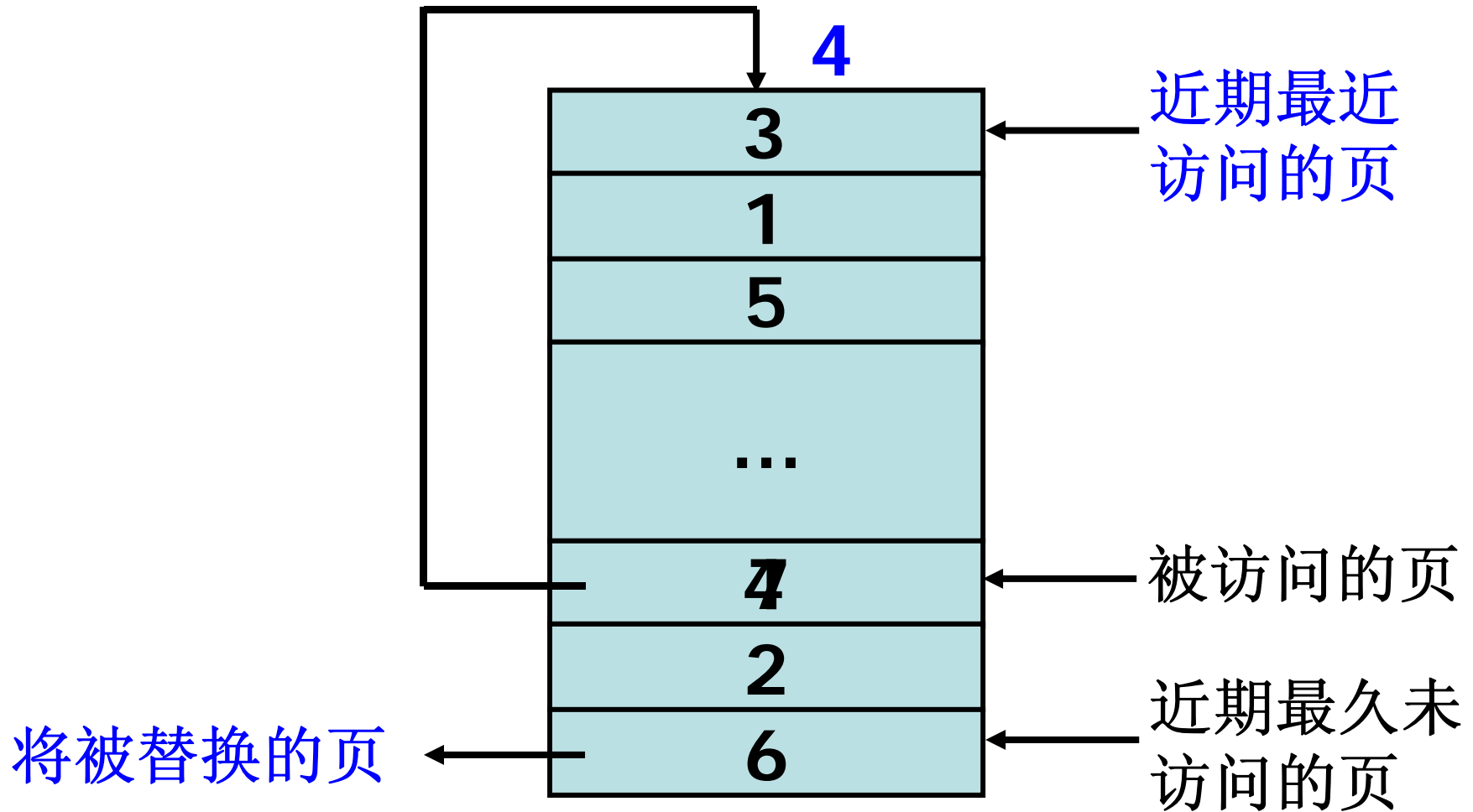
- 堆栈型替换算法的命中率随分配的实页数的增加而单调上升，至少不会下降。
- 可以证明：
  - LRU是堆栈型替换算法
  - OPT是堆栈型替换算法
  - FIFO不是堆栈型替换算法

# 堆栈型替换算法

## ■ 可以利用堆栈实现堆栈型替换算法

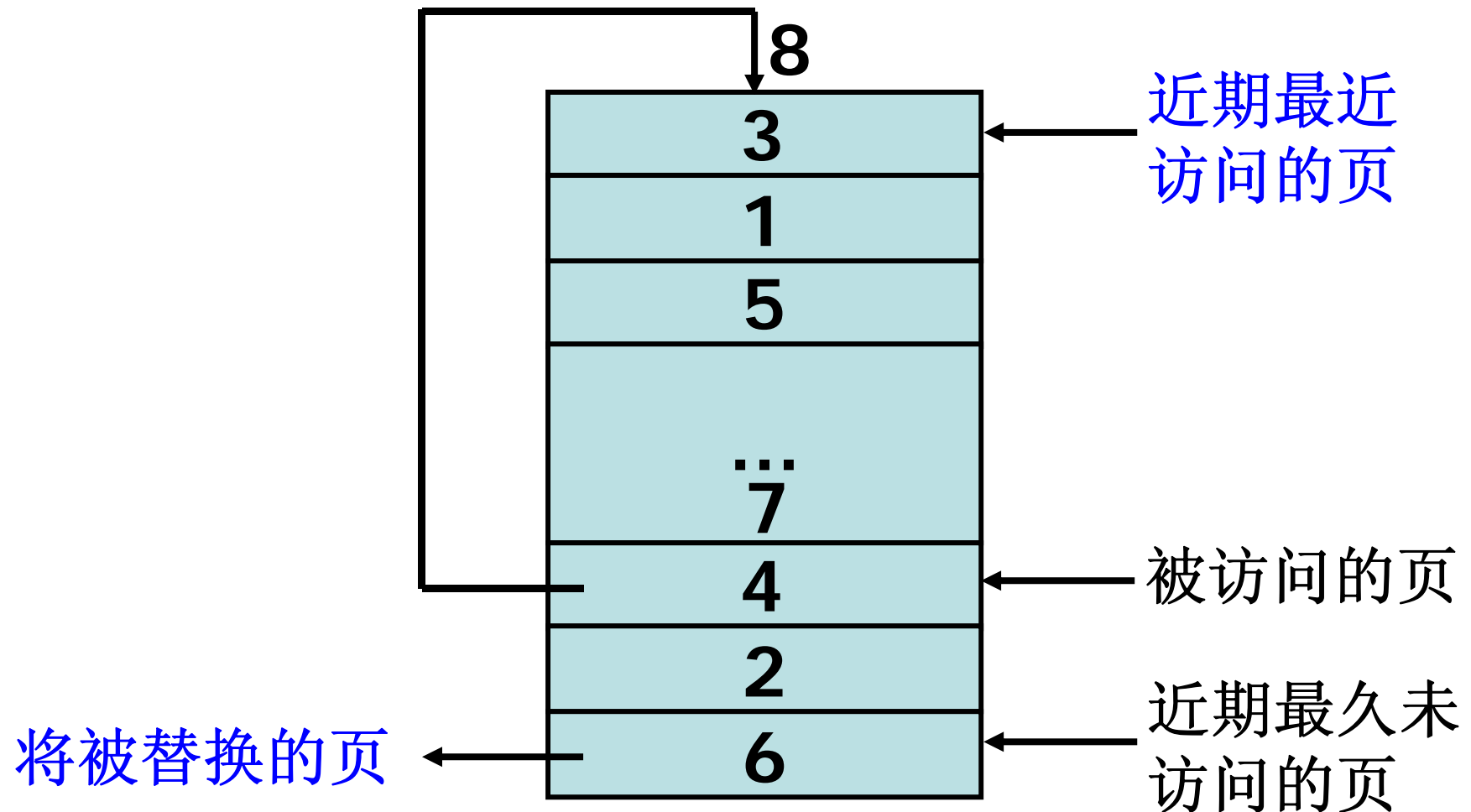
- 堆栈容量：  $2^{nv}$
- 过程： 检查要访问的虚页号是否在堆栈中？
- 是： 将该页面调置栈顶，并把该项上面的项下推一行，该项下面的不动
- 否： 新虚页号压入堆栈，弹出栈底

# 堆栈型替换算法



要访问的虚页4已在主存时的情况

# 堆栈型替换算法



要访问的虚页**8**不在主存时的情况

# 堆栈型替换算法

- 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实页数进行评估。

# 堆栈型替换算法

■ 例3：设某虚拟存储器上运行的程序含5个虚页，其页地址流依次为

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

采用LRU替换。

①用堆栈对该页地址流模拟一次，画出此模拟过程，并标出实页数为3, 4, 5时的命中情况。

②为获得最高的命中率，应分配给该程序几个实页即可？其可能的最高命中率是多少？



# 堆栈型替换算法

页地址流		4	5	3	2	5	1	3	2	5	1	3
堆栈内容	S(1)	4	5	3	2	5	1	3	2	5	1	3
	S(2)		4	5	3	2	5	1	3	2	5	1
	S(3)			4	5	3	2	5	1	3	2	5
	S(4)				4	4	3	2	5	1	3	2
	S(5)						4	4	4	4	4	4
	S(6)											
实页数	n=3					H						
	n=4					H		H	H	H	H	H
	n=5					H		H	H	H	H	H

# 堆栈型替换算法

- 解②：为获得最高的命中率，应分配给该程序 4 个实页。最高页命中率为  $H=6/11$

## 4. 页式虚拟存储系统工作全过程

- 教科书 P108 描述了页式虚拟存储系统工作的全过程。请自学。

## 4.2.3 页式虚拟存储系统实现中的问题

- 页面故障的处理
- 提高等效访问速度
- 提高命中率和CPU利用率

# 页面故障的处理

- 页面失效如何处理是页式虚拟存储系统设计的关键之一。
- 一般应如下处理：
  - 页面失效不能按一般的中断处理，应当将其看作是一种故障，由CPU立即响应处理。
  - 发生页面失效后，应解决如何保护和恢复故障点现场的问题。
  - 替换算法的选择很重要，不允许出现指令或操作数跨页存贮的那些页被轮流从主存中替换出去的“颠簸”现象，因此给一道程序分配的实页数应有一个下限
  - 页面大小不能过大，否则主存中的页数将减少，从而出现大量的页面失效

# 提高等效访问速度

- 从等效访问速度公式可以看出，影响虚拟存贮器等效访问速度的因素有两个：
  - 尽可能短的访问主存时间；
  - 主存命中率；

假设 $M_1$ 访问和 $M_2$ 访问是同时启动

$$T_A = HT_{A1} + (1-H)T_{A2}$$

假设 $M_1$ 访问和 $M_2$ 访问不是同时启动

$$T_A = T_{A1} + (1-H)T_{A2}$$

# 提高等效访问速度

- 关键是提高内部地址变换的速度，即加快多用户虚地址 $N_s$ 到主存实地址 $n_p$ 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，正是系统结构设计任务。

# 提高等效访问速度

## ■ 方法:

- 目录表（相联目录法）（前面已讲过）
- 快慢表
- 散列函数

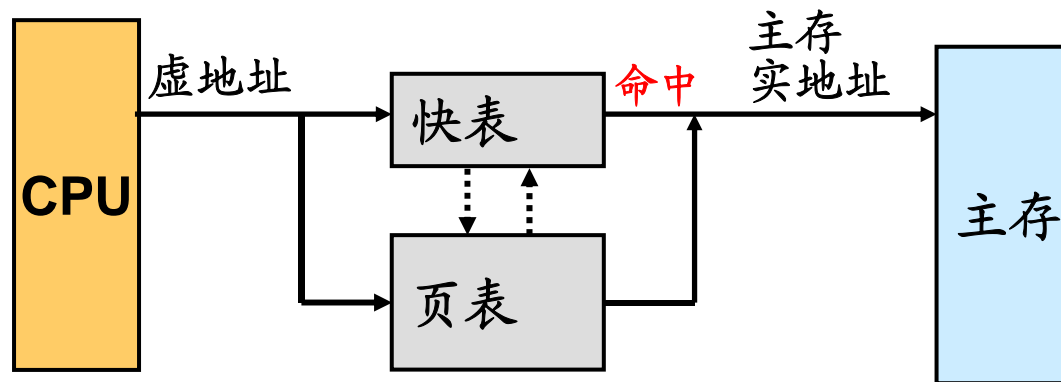


# 快慢表

- 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。**

# 快慢表

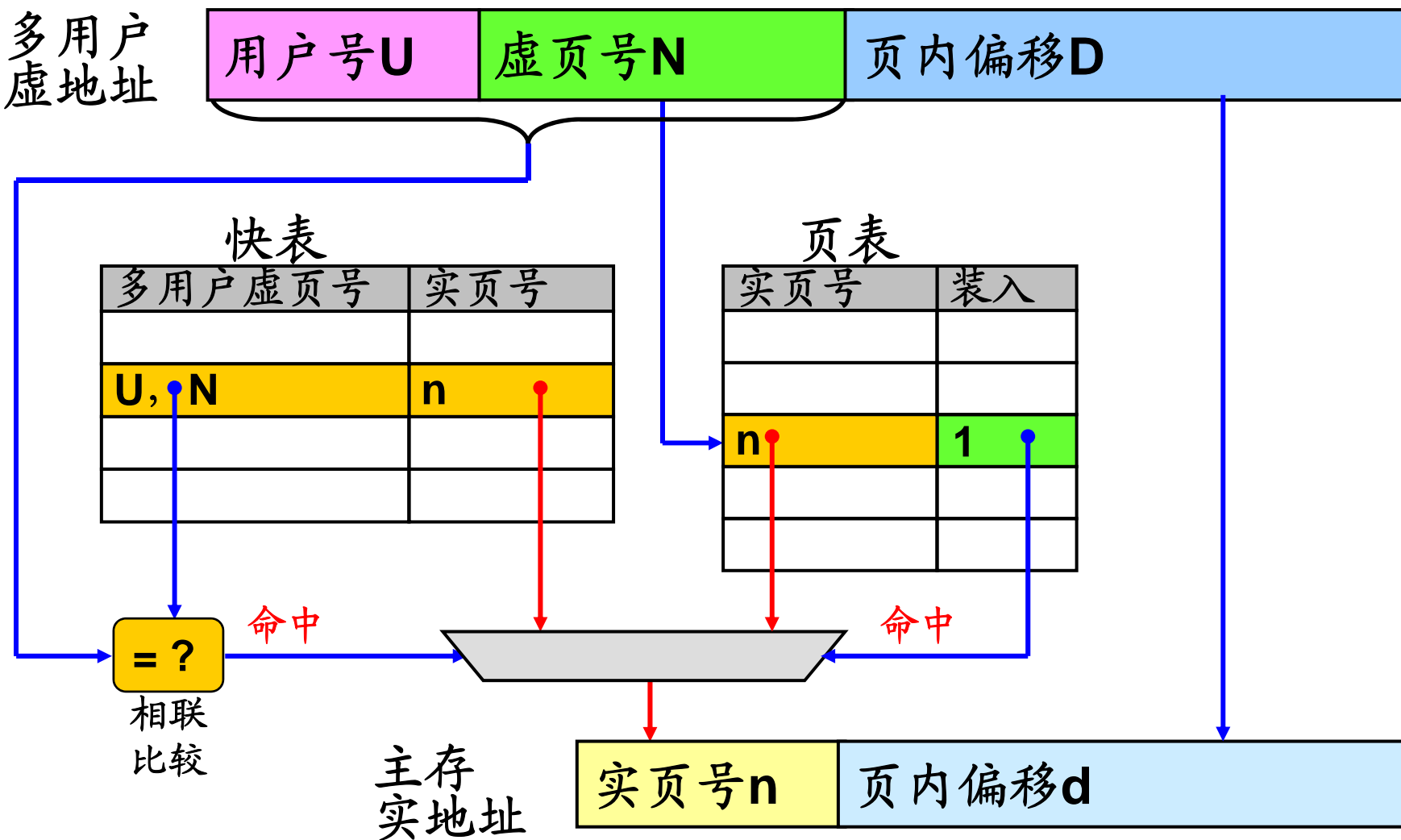
- 查表时，**同时查快表和慢表**，并将快表中不存在的内容从慢表中调入快表。
  - 显然，这里也要用到替换算法，一般也采用LRU
- 进一步减少相联比较的位数，也能加快速表的查找速度。



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。

# 快慢表

多用户  
虚地址



同时查快表和页表进行地址变换

# 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**。

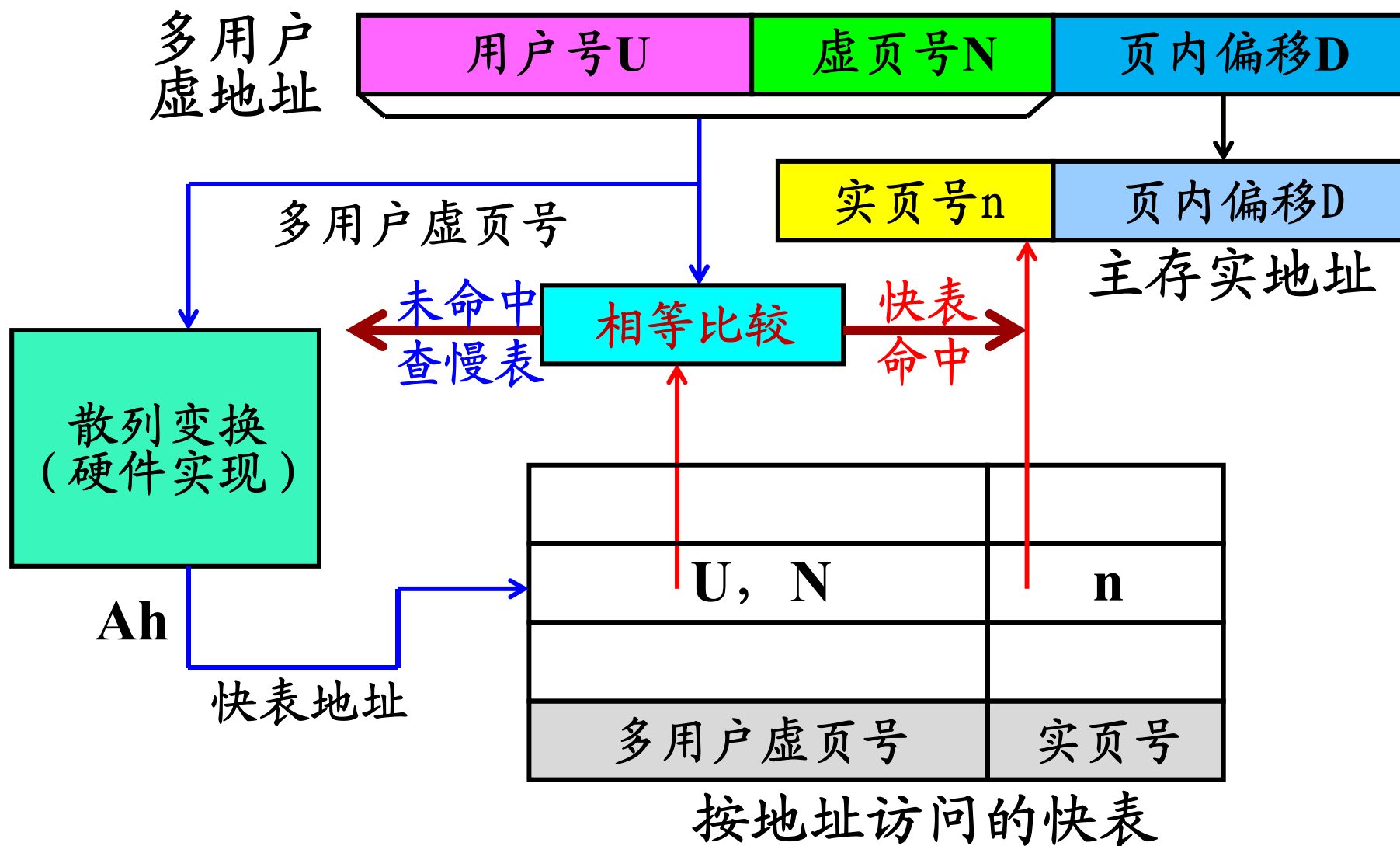
# 散列函数

- **目的：**把相联访问变成按地址访问，从而加大快表容量。

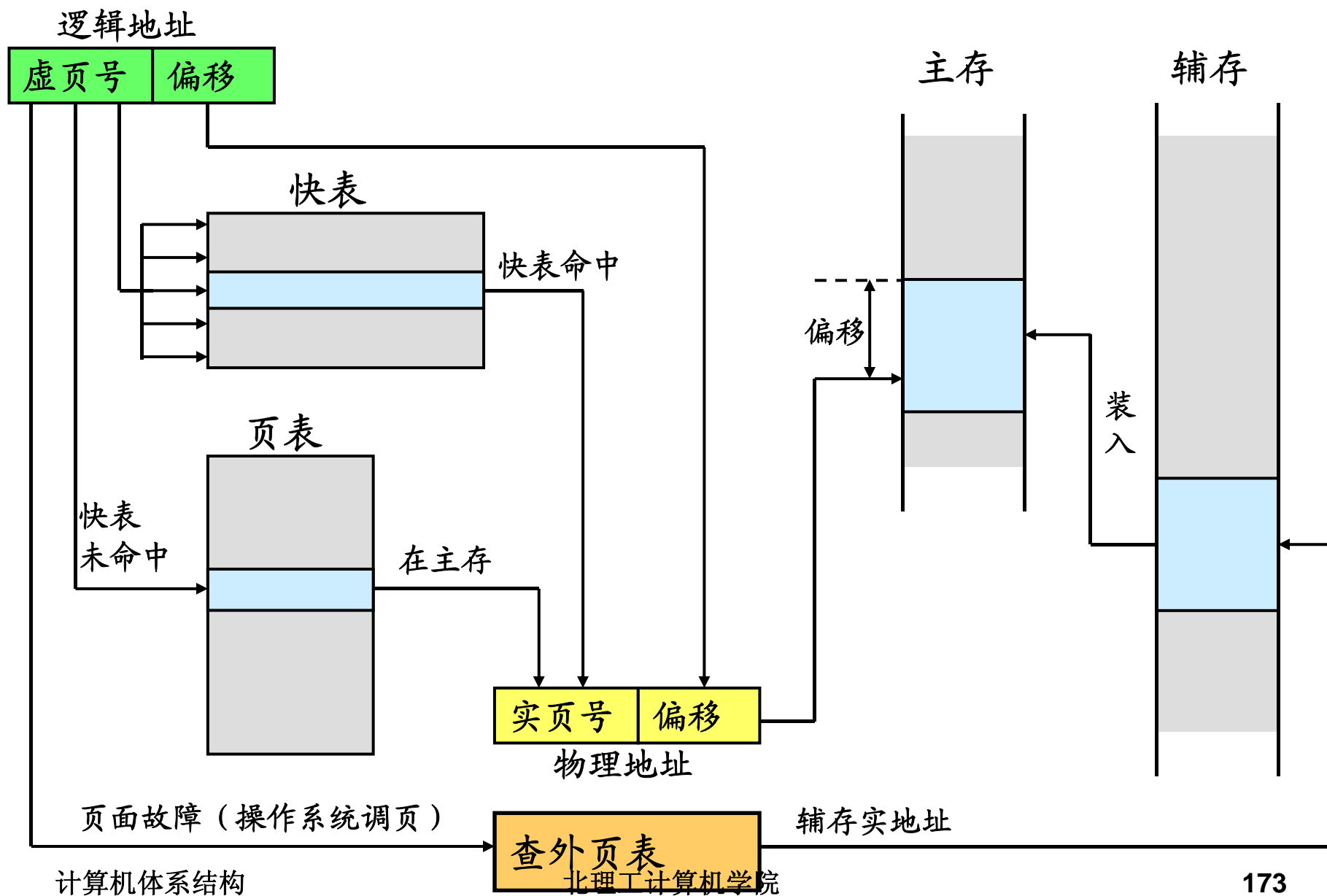
散列（Hashing）函数： $Ah = H(Pv)$ ,  
20位左右  $\Rightarrow$  5~8位

- 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。

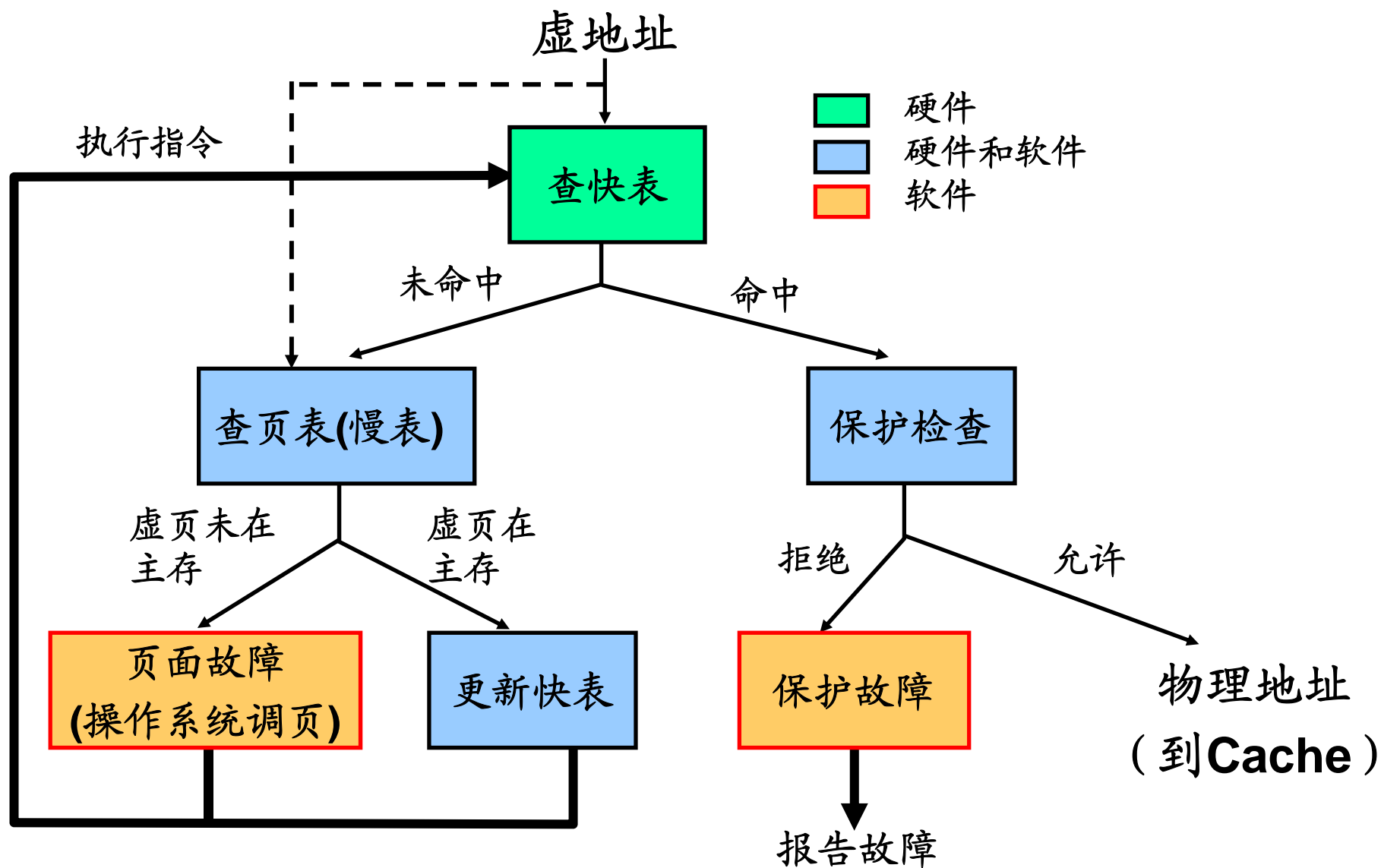
# 散列函数



# 页式虚拟存储系统工作过程



# 页式虚拟存储器工作过程



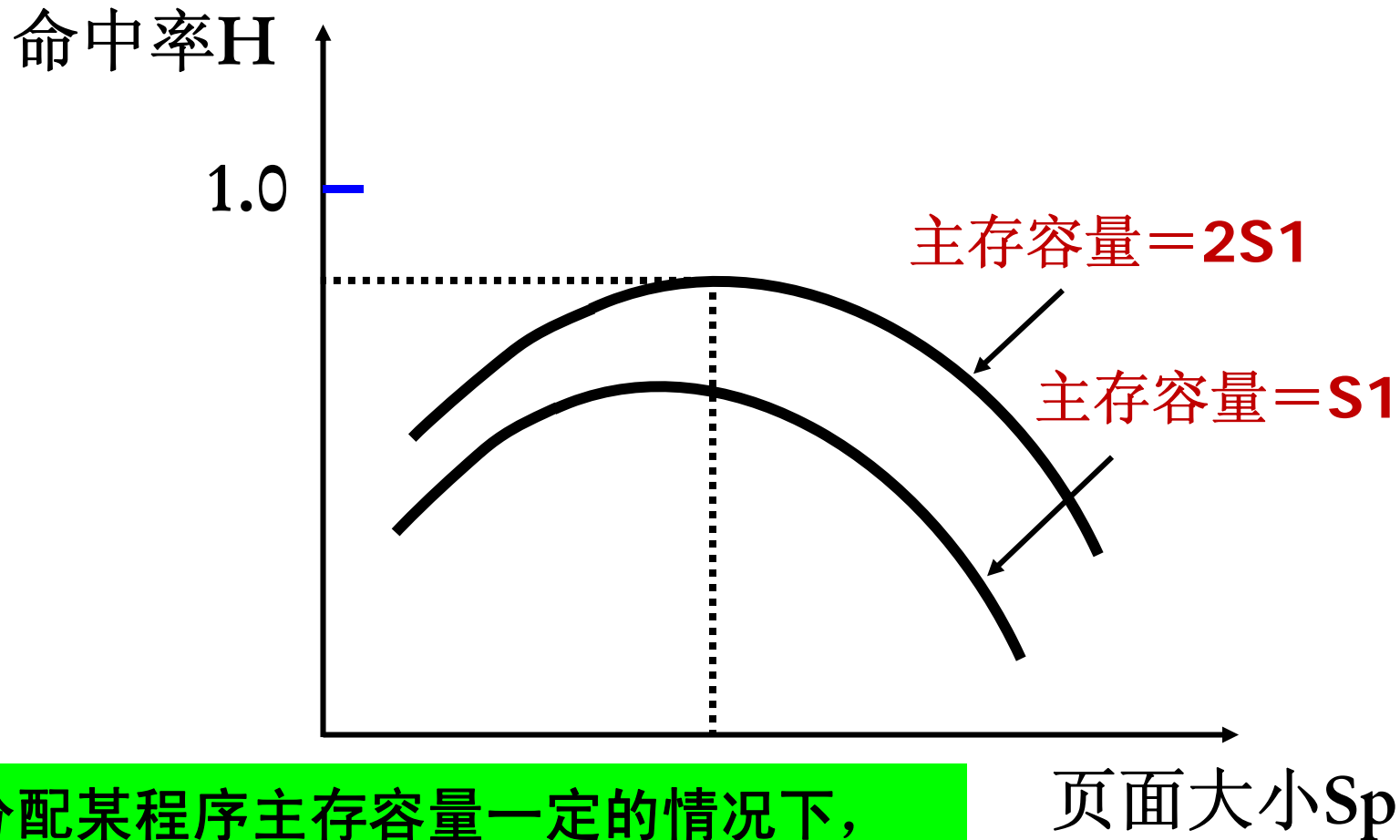


# 提高命中率和CPU利用率

## ■ 影响主存命中率的主要因素：

- 程序执行过程中的页地址流分布情况；
- 所采用的页面替换算法；
- 页面大小；
- 主存储器的容量；
- 所采用的页面调度算法
- 等

# (1) 页面大小与命中率



在分配某程序主存容量一定的情况下，当页面大小  $S_p$  为某一个值时，命中率  $H$  到达最大。

# (1) 页面大小与命中率

- 假设  $A(t)$  和  $A(t+1)$  是相邻两次访问主存储器的逻辑地址，让  $d = |A(t) - A(t+1)|$
- 如果  $d < S_p$ ，随着  $S_p$  的增大， $A(t)$  和  $A(t+1)$  在同一页面的可能性增加，即  $H$  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  $S_p$  的增大，主存的页面数减少，页面的替换将更加频繁。 $H$  随着  $S_p$  的增大而降低；

# (1) 页面大小与命中率

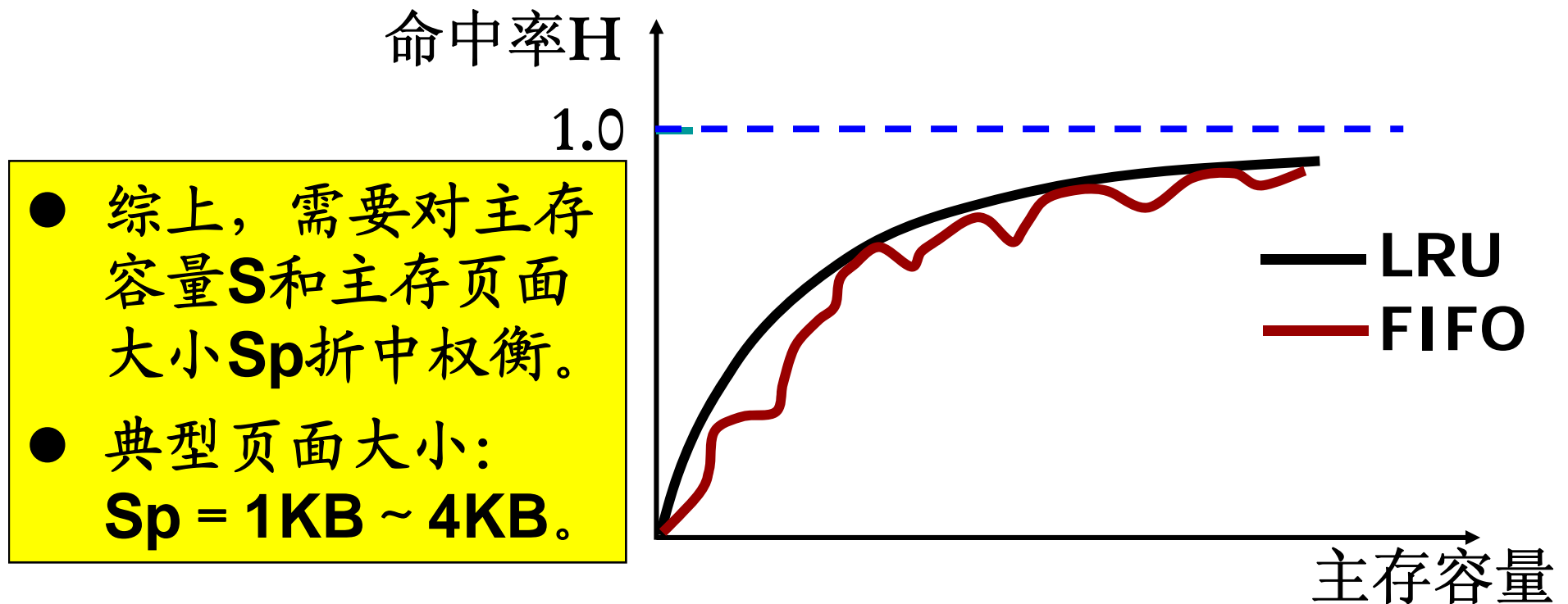
- 当 $S_p$ 比较小的时候，前一种情况是主要的， $H$ 随着 $S_p$ 的增大而提高；
- 当 $S_p$ 达到某一个最大值之后，后一种情况成为主要的， $H$ 随着 $S_p$ 的增大而降低；
- 当然，页面大小相同情况下，增大主存容量，相应的页面数就越多，命中率就越高。

## (2) 主存容量与命中率

- 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, 主存利用率下降,  $H$  几乎不再提高。**

## (2) 主存容量与命中率

- 从细节看，主存容量对堆栈型替换算法和其他算法的影响是不同。



### (3) 页面调度策略与命中率

主存命中率也与页面调度策略有关。

#### ■ 请求式页面调度策略

- 当使用到的时候，再调入主存。
- 优点：主存利用率高。
- 缺点：经常发生页面失效，尤其在程序开始的一段时间内

#### ■ 预取式页面调度策略

- 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。
- 优点：可以避免在程序开始运行时，频繁发生页面失效的情况。
- 缺点：如预取了用不到的页面，则浪费调入时间，占用主存空间。