托业，助您实现人生宏伟大业！

# 复旦托业基础班

# 计算机系统基础讲义

FUDAN-TOEIC E&T CENTER 2003
复旦托业
睿智所托·业则广阔

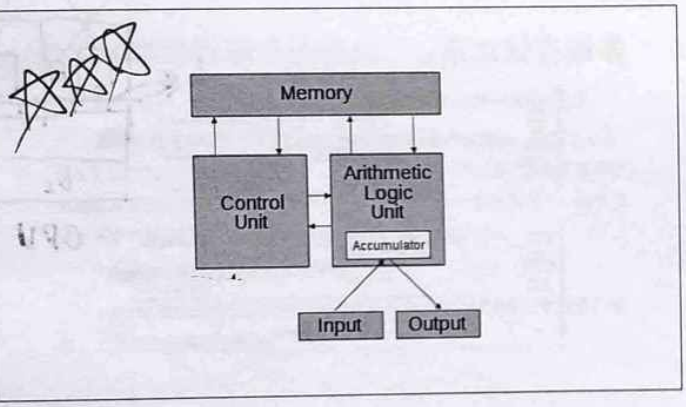
百度大纲上的题目，搜一下，全部搞清楚

对着大纲做书上课后题。

# 计算机体系结构

## 与其他学科的交叉

- 主要包括：计算机组成原理、计算机操作系统、汇编语言、数据结构、编译原理等




- 64位?
- ARM、GPU、CPU
- 缓存？内存?
- 分布式、集群
- 输入、输出
- 中断
- 关机如何实现
- 程序的执行

## 计算机系统的层次结构

**计算机系统：由软件和硬件组成。**

**计算机系统是相当复杂的系统，在分析设计时一般采用层次结构的观点和方法，可以从不同的角度去构建计算机系统的层次结构。**

## 从计算机系统组成角度划分层次结构

**从计算机系统组成的角度来划分的一种层次结构模型如下图所示。**

**自下而上，表明了设计和构建一台计算机时的逐层生成过程，每层都在下一层的基础上增加功能。**

CPU → 分层 → 流水线 { 相关 { 资源 数据 结构 } 办法 }

↓

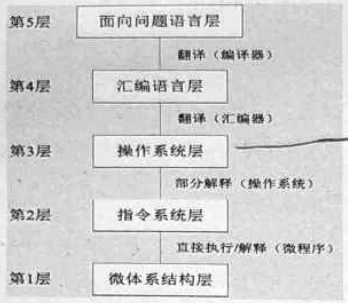
Mem →  → 指标 { 容量 速度 价格 } → 替换算法 { Random FIFO LRU }

↓

硬盘 → 线程 进程 、I/O 多路复用 → 中断

↓

机器之间

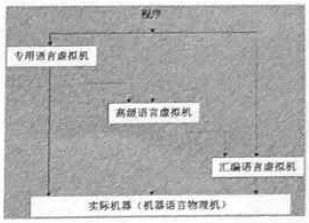



结合硬件与软件的层

从计算机系统组成角度划分的层次结构模型

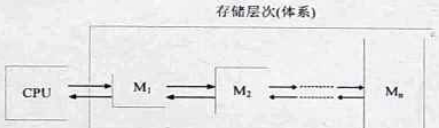
## 从语言功能角度划分层次结构

计算机功能可描述为"能执行用某些程序设计语言编写的程序"，下图所示的是语言功能层次模型。




## 多级存储体系

- 二级存储体系结构可以进一步扩展到多级存储层次，对CPU而言，存储系统是一个整体：越靠近CPU的存储器存取速度越快，存储容量越小，也即下图中M1的存取速度最接近于CPU，而存储系统总体容量和单位存储价格接近于离CPU"最远"的Mn。



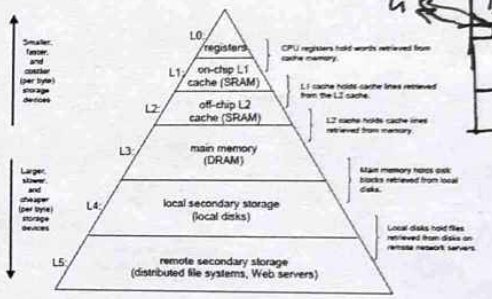

数据调用过程

register ALU

L1 L2

CPU

## 多级存储体系(续1)




## 冯.诺依曼结构

- 几十年来计算机的体系结构尽管不断改进，但冯·诺依曼体制的核心概念仍沿用至今，绝大多数实用的计算机仍属于冯·诺依曼机。
- 冯·诺依曼机由运算器、控制器、存储器和输入/输出设备组成。

## 冯·诺依曼计算机结构的主要特点：

- 采用存储程序方式，程序的指令和数据存放在同一存储器中。
- 指令由操作码和地址码组成。
- 控制流由指令流产生。
- 存放在存储器中的指令和数据，都是以二进制编码表示的，从它们本身是无法区别的。
- 机器以运算器为中心。

指令：MIPS包括：

算术运算指令

数据传输指令

条件转移指令

无条件转移指令

Amdahl 定律：可提升性能倍数 = $\frac{1}{\text{串行执行时间比} + \frac{\text{并行执行时间比}}{\text{CPU数}}}$

并行执行时间比也就是可改进比例。

α：为改进部分所用时占比。

K：该部分性能提升比例

∴ 通过优化：1. 串行执行时间比（减少串行占比，增大并行占比）（可改进比例）

2. 增加CPU数。（改进加速比）

未提升性能。

$T_{old}$.

$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/K$

$\therefore S = \frac{T_{old}}{T_{new}} = \frac{1}{1-\alpha+\frac{\alpha}{K}}$

多条件提升性能，Amdahl定律扩展：$S_n = \frac{1}{(1-\sum F_i) + \sum \frac{F_i}{S_i}}$

## 计算机系统设计的量化原则

## 计算机系统设计的量化原则

- 1. 利用并行性
  - ◆在系统级使用并行，如服务器采用多处理器/机和多磁盘技术，可有效地提高型服务器的吞吐量性能。
  - ◆在单处理器级，指令间的重叠执行和同时执行可有效地加快指令的执行速度，具体方法是采用流水线和超标量结构。
  - ◆在部件级也可以发掘并行性，如主存储器采用多体交叉结构，可以并行访问多个存储模块；算术逻辑运算部件ALU的先行进位加法器，利用超前进位和多位同时相加并行求和，使运算时间大为缩短。

## 2.加快经常性事件的速度

- 在计算机设计中，经常性事件速度的加快能够显著提高整个系统的性能。

例如：CPU中的两个数相加时，相加结果可能产生溢出，出现溢出的情况比较少见，不溢出才是比较常见的情况。因此，可以通过优化不溢出相加的操作来提高机器的性能。而发生溢出的概率很小，即使发生了，处理较慢也不会对系统性能产生很大的影响。

此原则也同样适用于资源的分配。

例如：处理器中的取指和译码单元要比乘法单元使用得更加频繁，因此，应优先优化这两个单元。

## Amdahl定律

- Amdahl定律可以阐述为：系统中某一部件由于采用某种更快的执行方式后所获得系统性能的提高，与这种执行方式的使用频率或占总执行时间的比例有关。

Amdahl定律定义了一台计算机系统采用某种改进措施所取得的加速比。

$$\text{加速比} = \frac{\text{采用改进措施后计算机的性能}}{\text{没有采用改进措施时计算机的性能}} = \frac{\text{没有采用改进措施时某任务的执行时间}}{\text{采用改进措施后某任务的执行时间}}$$

加速比反映了使用改进措施后完成一个任务比不使用改进措施完成同一任务加快的比率。

$S = \frac{1}{(1-Fe)+\frac{Fe}{Se}}$

- Amdahl定律中，加速比与两个因素有关：

a. 在原有的计算机上，能被改进的部分在总执行时间中所占的比例，即 [redacted]

这个值称为改进比例，记为Fe，它总小于等于1，例如，如果一个任务在原来机器上的执行时间为60s，其中20s的执行时间可以使用改进措施，那么Fe就是20 / 60。

Fe：可改进比例

b. 改进部分采用改进措施后，相比没有采用改进措施前性能提高的倍数，即 [redacted]

这个值称为改进加速比，记为Se，它总大于1。例如，在原来的条件下程序的某一部分执行时间为5s，而改进后这一部分只需要2s，那么Se就是5 / 2。

Se：改进加速比

## CPU性能公式

- 大多数计算机都有一个产生周期性定时信号的时钟。

1秒能执行多少次

- 时钟的长度可以用时钟周期（如2ns）或其频率（如500MHz）来度量。一个程序执行时所花费的CPU时间可以表示为：

CPU时间 = 一个程序的CPU时钟周期数×时钟周期

或

CPU时间 = 一个程序的CPU时钟周期数 / 频率

- 此外，还可以用一个程序的指令条数（用IC表示）和执行所需的时钟周期数，来计算出执行一条指令所需的平均时钟周期数CPI（Clock cycles Per Instruction）：

CPI = 一个程序的CPU时钟周期数 / IC

- CPI是衡量不同指令和不同实现方法的一个处理器性能指标。

脉冲信号之间的时间间隔称为周期：

| $10^{-9}$ | $10^{-6}$ | $10^{-3}$ | $10^{3}$ | $10^{6}$ | $10^{9}$ |
|---|---|---|---|---|---|
| n | μ | m | k | M | G |
| 纳 | 微 | 毫 | | | |

$2^{10}$

CPU

ALU

状态/中断

地址 register

PC

数据 register

Mem

程序 指令

exe. 数据

一个程序的CPU时钟周期数也可以表示为：

IC×CPI

因此有：

CPU时间=IC×CPI×时钟周期

CPU时间=（IC×CPI）/频率

上式表明，CPU的性能取决于三个要素：

① 时钟周期；✓

② 每条指令所需的平均时钟周期数CPI；✓

③ 指令条数IC。✓

时钟周期取决于硬件技术和组织；CPI取决于计算机组成和指令系统的结构；指令数目取决于系统结构的指令系统和编译技术。

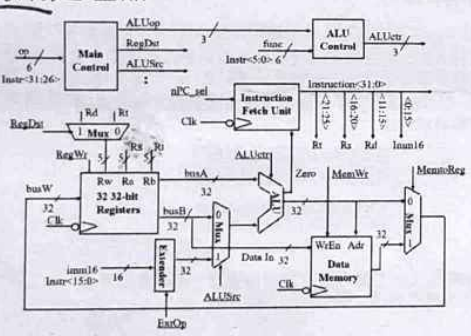
## What is the average CPI? 平均CPI

| Type | $CPI_i$ for type | Frequency | $CPI_i$ x $freq_i$ |
|---|---|---|---|
| Arith/Logic | 4 | 40% | 1.6 |
| Load | 5 | 30% | 1.5 |
| Store | 4 | 10% | 0.4 |
| branch | 3 | 20% | 0.6 |
| | | Average CPI: | 4.1 |

而且时钟处理非常慢

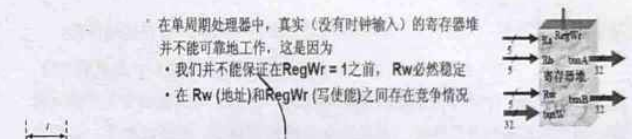
时钟跳一次 一个周期内所有环节都要做掉，但容易产生竞争

## 单周期处理器




## 竞争条件

- 在单周期处理器中，真实（没有时钟输入）的寄存器堆并不能可靠地工作，这是因为
  - 我们并不能保证在RegWr = 1之前，Rw必然稳定
  - 在 Rw (地址)和RegWr (写使能)之间存在竞争情况



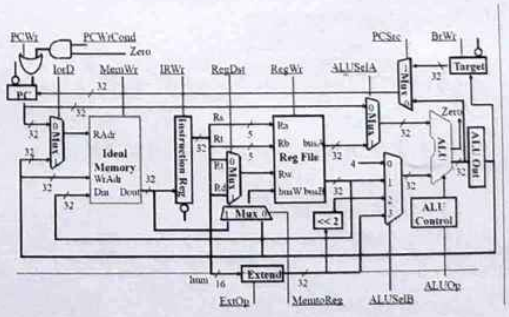


- 在单周期处理器中，真实（没有时钟输入）的存储器并不能可靠地工作，这是因为：
  - 我们并不能保证在WrEn = 1 之前，地址信号(Adr)一定稳定
  - 在 Adr 和 WrEn 之间有竞争


## 如何避免竞争

- 对多周期处理器的解决方案：
  - 在周期 N结束时，确认 地址(Adr)是稳定的
  - 在之后的 一个 周期[第（N+1）周期]发出 Write Enable信号
  - 在撤销 Write Enable信号之前， 地址信号(Adr)不能改变

## 多周期处理器



流水线处理器性能分析

原理：1. 流水线中的各个处理部件可并行工作，从而可使整个程序的执行时间缩短

2. 流水线并不会缩短单条指令的执行时间，（甚至会增加时间）而是提高了指令的吞吐率。

↓

增加流水线寄存器的时间开销

还有通用寄存器

## 一条指令的几个过程段

- 取指令：根据PC（指令计数器）从M（存储器）取出指令送到IR（指令寄存器）
- 译码分析：译出指令的操作性质，准备好所需数据
- 执行：将准备好的数按译出性质进行处理，主要涉及ALU（算术逻辑运算部件）

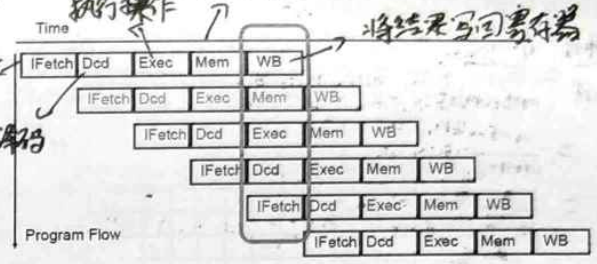

会和加速比一起考

## 流水线工作模式

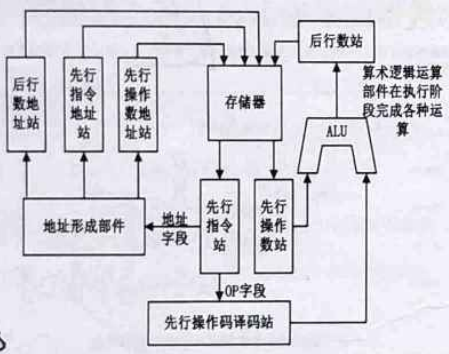
数据内存中读取操作数（访存）

执行操作

将结果写回寄存器




## 流水线结构




会考，算公式

## 吞吐率

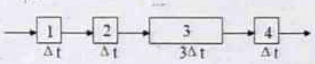
单位时间内能处理的指令条数或输出结果的数量。

(1)最大吞吐率

$$T_{P\max}=\frac{1}{\max\{\Delta t_i\}}=\frac{1}{\max\{\Delta t_i+\Delta t_i\}}$$

当各$\Delta t_i$相等时，$T_{P\max}=\frac{1}{\Delta t}$

流水线（$\triangle t_i$不等）：




瓶颈：$\Delta t_i$最大的那一个段。$T_{pmax}=1/(3\triangle t)$。

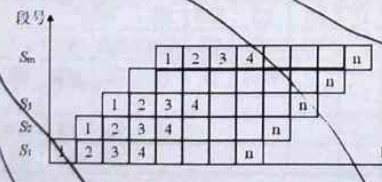
∴实际吞吐率 = 指令条数 / 流水线时间

指令条数

流水线时间

对于线性流水线，完成$n$个任务所需时间为 $T=m\Delta t+(n-1)\Delta t$，实际吞吐率为：

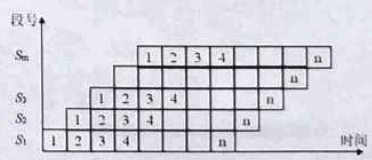
$$T_P=\frac{n}{m\Delta t+(n-1)\Delta t}=\frac{1}{\Delta t(1+\frac{m-1}{n})}=\frac{T_{P\max}}{1+\frac{m-1}{n}}$$




## 流水线吞吐率

对于线性流水线，完成$n$个任务所需时间为 $T=m\Delta t+(n-1)\Delta t$，实际吞吐率为：

$$T_P=\frac{n}{m\Delta t+(n-1)\Delta t}=\frac{1}{\Delta t(1+\frac{m-1}{n})}=\frac{T_{P\max}}{1+\frac{m-1}{n}}$$




流水线时间 = 一条指令所需时间 + (指令数 - 1) × 时间最长的一段指令所需时间

吞吐率 = 指令条数 / 流水线时间

流水线的局限性：

1. 不一致的划分，也就是不同阶段所需要的时间不一致。
   且时钟周期是由最慢阶段的延迟限制的。

2. 流水线加深，过深，收益反而下降。原因是将各个阶段切分成更小的块时，
   由流水线寄存器更新引起的延迟就成为了一个限制因素。

流水线加速 ——流水方式的工作速度与等效的顺序工作方式时间的比值。

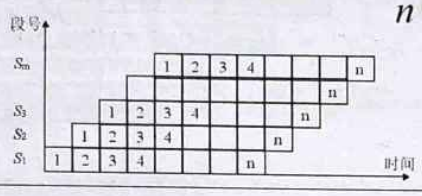
对于线性流水线：

非流水线时间对 / 流水线时间对 $= \frac{n \cdot mat}{mat + (n-1)\Delta t}$

$$S_p = \frac{T_1}{T_k} = \frac{nm}{m+n-1} = \frac{m}{1+\frac{m-1}{n}}$$

$= \frac{nm}{m+n-1}$

$= \frac{m}{1+\frac{m-1}{n}}$




## 流水线中的相关

如果要执行算式S=a/b+c，要通过下列四条指令来执行。

```
LD   R , A
DIV  R , B
ADD  R , C; 要等DIV结果
ST     R , S; 存结果
```

第3条指令ADD R , C执行的前提是第2条指令执行完毕、有了结果以后才能执行。换句话说，只有第2条指令没有执行完毕，结果没有出来，第3条指令就无法执行下去，这就出现了指令因等待前面结果，使后面没指令无法继续下去的现象，即相关。

相关 → 竞争、干扰

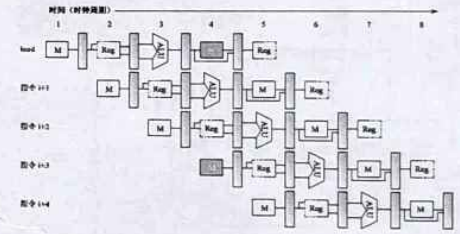
## 流水线中的相关主要分为以下3种类型

- 结构相关
- 数据相关
- 控制相关

同一时钟周期内 / 多用同一功能部件

## 流水线中的结构相关（资源相关）

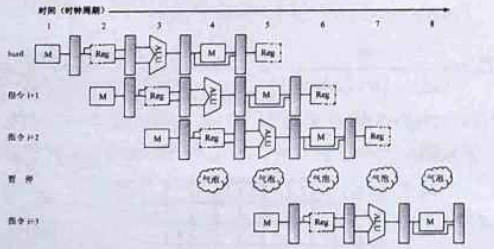
如果某些指令组合在流水线中重叠执行时产生了资源冲突，那么我们称该流水线有结构相关。




由于访问同一个存储器而引起的结构冲突

## 对于这种冲突，通常有以下两种解决方法

解决办法(1)：插入暂停周期，即让流水线在完成前一条指令对数据的存储器访问时，暂停取后一条指令的操作




为消除结构冲突而插入的流水线气泡

引入暂停后的时空图

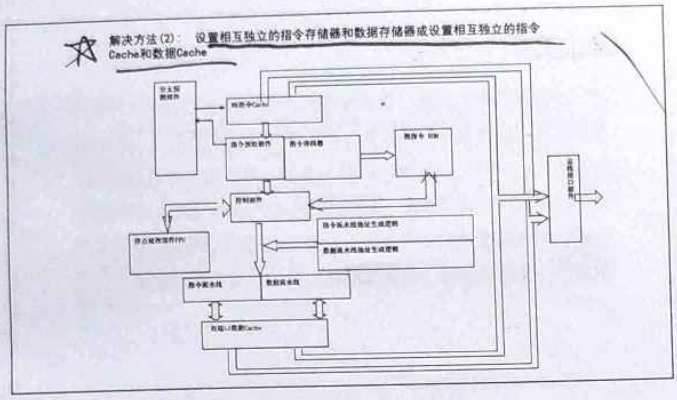
| 指令编号 | 时钟周期 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 指令i | IF | ID | EX | MEM | WB | | | | | |
| 指令i+1 | | IF | ID | EX | MEM | WB | | | | |
| 指令i+2 | | | IF | ID | EX | MEM | WB | WB | | |
| 指令i+3 | | | | stall | IF | ID | EX | MEM | WB | |
| 指令i+4 | | | | | | IF | ID | EX | MEM | WB |
| 指令i+5 | | | | | | | IF | ID | EX | MEM |

IF:

数据冒险：由于指令执行所需要的数据还未准备好所引起的冒险情况，
当即将执行的指令依赖于还未处理完成的数据时，会导致指令无法立刻开始执行，引发数据冒险。

解决办法：① 采用暂停技术 stalling，让当前一条指令停在译码阶段，直到产生它的源操作数的指令通过了写回阶段。

② 数据转发 forwarding：运算结果写回是在WB阶段，通常在前面的指令是在这个阶段完成后取出寄存器中的值作为源操作数。但当需要的数据其实在EX执行阶段就已得到了。而forwarding就是将在EX阶段完成计算后，直接将数据传给下一个指令。这样，就不用等到WB阶段完成后才能取好了。




## 流水线中的数据相关

多条(2条)指令使用了相同的数据地址而发生的冲突

如果下面的条件之一成立，则指令j与指令i数据相关：
（1）指令j使用指令i产生的结果
（2）指令j与指令k数据相关，指令k与指令i数据相关，
则指令j与指令i数据相关。

第2个条件指出，如果两条指令之间存在类似上述的相关链，则它们之间也是相关的。这条相关链甚至可以贯穿整个程序。

## 例如：下面这一段代码存在数据相关。

```
F0      // F0为数组元素
F4 F0   // 加上F2中的值
F4      // 保存结果
R1      // 数组指针递减8个字节
R1      // 如果R1≠R2，则分支
```

## 流水线的控制相关：

控制相关是指因为程序的执行方向可能被改变而引起的相关。

典型的程序结构是"if-then"结构。
请看一个示例.

```
if p1 {
        S1;
    };
S;
if p2 {
        S2;
    };
```

## 续

控制相关带来了以下两个限制：

（1）控制相关于一个分支的指令不能被移到分支之前执行。
if…then程序中，then后面的语句不能移至if之前执行。

（2）没有控制相关于一个分支的指令不能移至该分支指令之后从而受这个分支控制，如if…then程序中，if前的指令不能移至then部分中执行。

## 指令

控制相关：当处理器无法根据处于取指阶段的当前指令来确定下一条指令的地址时，就会出现控制冒险。
由转移指令引起，有可能使预取的指令作废了。

解决办法：预测BJ的转移分支并通过暂停技术来动态调整流水线的流程。

## 编址方式

• 如某台机器，按字节编址，数据有字节（8位）、半字（双字节）、单字（4字节）和双字（8字节）不同宽度。主存数据宽度64位，即一个存储周期可访问8个字节。采用按字节编址，大于字节宽度的数据是用该数据的首字节地址来寻址的。一种存放数据的方法是，在主存中允许数据从任意字节地址单元存放，如图2-8（a）所示，这种方法很容易出现一个数据跨主存宽度边界存储的情况；对于跨界存放的数据，即使数据宽度小于或等于主存宽度，也需要两个存储周期才能访问到，导致访问速度显著下降。

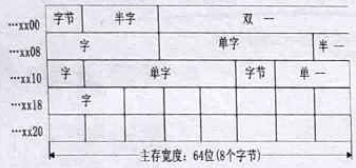
## 编址方式

• 另一种数据存放方法是，要求数据在主存中存放的地址必须是该数据宽度（字节数）的整数倍，即双字地址的最低3个二进制位必须为000，单字地址最低2位必须为00，半字地址最低1位必须为0，如图2-8（b）所示。这种存放方法也称为按整数边界存储方式，它可以使访问任意宽度的数据都只用一个存储周期。虽然浪费了一些存储空间，但是速度比上一种方法有显著提高。

半字、全字怎么去存？

## 数据在主存中的存放方式

分段




（a）数据按任意字节地址存放

## 数据在主存中的存放方式

分块


（b）数据按整数边界存放

## 降低重复的访问

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

```
16*x   -->   x << 4
```

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

## 存储系统介绍

内存是为了提升硬盘的读取速度

虚拟内存是为了通过硬盘扩大内存的大小

## 存储系统介绍

- 存储系统

指计算机中由存放程序和数据的各种存储设备、控制部件及管理信息调度的设备（硬件）和算法（软件）所组成的系统。

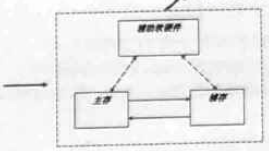
计算机系统中，一般使用具有层次结构的存储系统，主要可分为三个存储层面：高速缓冲存储器、主存储器和辅助存储器。

高速缓冲存储器主要用于改善主存储器与中央处理器（CPU）的速度匹配问题，而辅助存储器则主要用于扩大计算机系统的存储空间。

## 存储系统的层次结构

- 层次存储系统是指把各种不同存储容量、存取速度、访问方式和单位存储价格的存储器，按照一定的层次结构组成多层存储器，并通过管理软件和辅助硬件有机组合成统一的存储体系，使计算机系统中使用到的各种程序和数据层次的分布到各个存储器中。

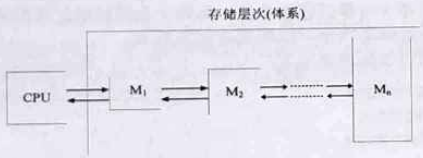
实现存储上的算法



主/辅存结构

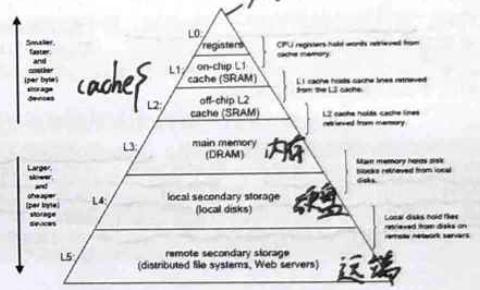
## 多级存储体系

- 二级存储体系结构可以进一步扩展到多级存储层次，对CPU而言，存储系统是一个整体：越靠近CPU的存储器存取速度越快，存储容量越小，也即下图中M1的存取速度最接近于CPU，而存储系统总体容量和单位存储价格接近于离CPU“最远”的Mn。




## 多级存储体系(续1)

寄存器




## 存储系统的性能参数

- 存储器有三个主要的性能指标：存储容量、存取速度、存储单位价格。
- 用户期望：期望存储器价格尽可能低，提供尽可能高的存取速度和尽量大的存储容量。系统的观点，计算机系统性能的发挥要求存储器存取速度与CPU相匹配，而容量上又应尽可能装入所有系统和用户软件；应用的观点，要求存储器的价格只能占整个计算机系统硬件价格的一小部分。
- 矛盾的现实：存取速度越快，存储器的单位存储价格就越高；在一定的单位存储价格下，存储容量越大，存储器的总价就越高。

## 存储系统的性能参数(续1)

- **存储容量**

引入虚拟存储系统，为计算机系统使用者另设额外的虚拟地址空间，它既不是主存储器的地址空间，也不是磁盘存储器的地址空间，是将主存和辅存的地址空间统一编址，形成的一个庞大的存储空间。

- **单位存储价格**

$$P=\frac{P_1 \cdot C_1 + P_2 \cdot C_2}{C_1 + C_2}$$ 为使得C1 ≈ C2，则需要M2 » M1

- 存取速度和访问命中率

访问命中率，指CPU访问存储系统时在主存储器中一次访问得到所需信息的概率。

$$H=\frac{N_1}{N_1+N_2} \qquad T=H \cdot T_1+(1-H) \cdot T_2 \qquad e=\frac{T_1}{T}$$

## 5.2 高速缓冲存储器Cache

- 计算机系统为改善CPU与主存储器之间的速度匹配问题，在CPU和主存储器之间加入一个高速、小容量的缓冲存储器Cache，构成Cache—主存储器的存储系统，使得存储系统对CPU而言，速度接近于高速缓冲存储器Cache，存储容量接近于主存储器。

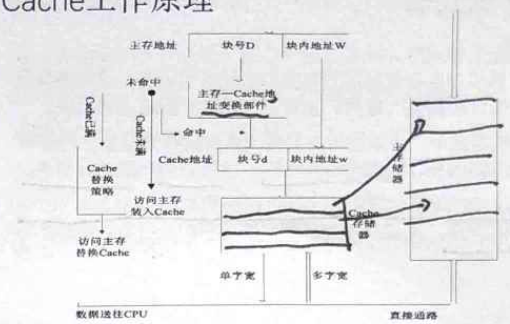
- Cache存储器主要由三个部分组成：

(1)Cache存储器，用于存放由主存储器调入的指令与数据块；

(2)地址转换部件，用于实现主存储器地址到Cache存储器地址的转换；

(3)替换部件，当缓存满时根据指定策略进行数据块替换，并对地址转换部件做对应修改。

## Cache工作原理




## Cache工作原理(续1)

- 系统工作时，地址转换部件维护一个映射表，用于确定Cache存储器中是否有所要访问的块，以及确定其位置。该映射表中的每一项对应于Cache存储器的一个分块，用于指出当前该块中存放的信息对应于主存储器的哪个分块。

- 为提高CPU对Cache存储器的访问命中率，Cache存储器的工作原理是基于**程序访问局部性原理**的，它不断地将与当前指令集相关联的一部分后继指令集从主存储器读取到Cache存储器，以供CPU访问，从而达到存储系统与CPU速度匹配的目的。

## 地址映象与变换方法

- 地址映象是将主存储器中的数据分块按某种规则装入Cache存储器中，并建立主存储器地址与Cache存储器地址之间的对应关系。
- 地址变换是指当主存储器中的分块按照地址映象方法装入Cache存储器后，在实际运行过程中，主存储器地址如何转换成为相应的Cache存储器地址。
- 地址的映象和变换是紧密相关的，采用什么样的地址映象方法，就有与这种映象方法相对应的地址变换方法。
- 一般可分为以下几种类型：

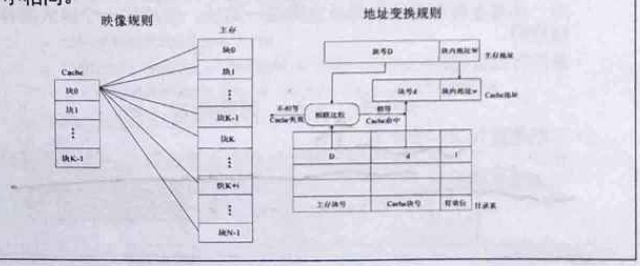
(1)全相联映象及其变换方法

(2)直接映象及其变换方法

(3)组相联映象及其变换方法

## (1)全相联映象及其变换方法

- 全相联映象是指主存储器中的任意分块可以被放置到Cache存储器中的任意一个位置。其中，主存储器与Cache存储器的分块大小相同。



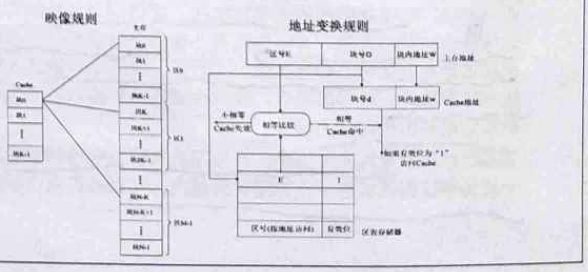

特点：随意映射，没有冲突缺失，命中高。

缺点：命中时间长，

硬件实现复杂，开销大，空间增加（比较器位数多）

## (2)直接映象及其变换方法

- 直接映象是指将主存储器中的某一分块在Cache存储器中都有唯一对应的位置。主存储器按Cache大小分成若干区，在区内进行分块，分块大小与Cache存储器中分块大小相等，主存储器中每个区包含分块的个数与Cache存储器中分块的个数相等。




直接映象：

特点：容易实现，命中时间短

无需考虑替换问题

但不够灵活，命中率低，Cache存储空间得不到充分利用。

Cache 的访问时间:

$\alpha$: hit ratio

$1-\alpha$: miss ratio

$t_{ave} = \alpha t_c + (1-\alpha)(t_c + t_m) = t_c + (1-\alpha)t_m$

$t_c$: 命中时间　$t_m$: 缺失损失　缺失率

程序决定. 要写出具有很好的局部性的程序.

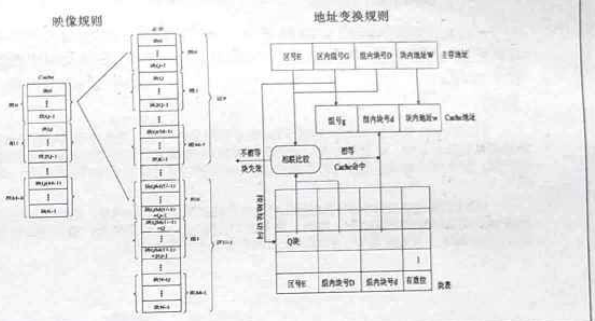
## (3)组相联映象及其变换方法

- 组相联映象把主存储器和Cache按同样大小划分成块，再将主存储器和Cache按同样大小划分成组，每一组由相同的块数组成，然后将主存储器按Cache大小分成区，主存储器每个区的组数与Cache的组数相同。

组结合直接映射和全相联映射的优点.

- 组相联映象在各组之间是直接映象，但组内各块之间是全相联映象。

## (3)组相联映象及其变换方法(续1)




## Cache替换算法及实现

- 当CPU读Cache时，有两种可能：

(一) 需要的数据已在Cache中，那么只需直接访问Cache；

(二)是需要的数据尚未装入Cache，则CPU从主存储器中读取信息的同时，需按所需的映象规则将该地址所在的那块存储内容从主存储器拷贝到Cache中。

- 对于第二种情况，若该块所映象的Cache块位置已全部被占满，则必须选择将Cache中的某一块替换出去，需要Cache替换算法解决如何选择被换出块的问题。

## Cache替换算法

- **随机替换算法**

随机法是Cache替换算法中最简单的一种。这种方法是随机地选择可以被替换的一块进行替换。有些系统设置一个随机数产生器，依据所产生的随机数选择替换块，进行替换。

- **先进先出替换算法(FIFO)**

这种策略总是把最先调入的Cache块作为被替换的块替换出去。

- **最近最少使用替换算法(LRU)**

LRU法是依据各块的使用情况，总是选择最近最久没被使用的块作为被替换的块进行替换。因为目前为止最久没有被访问的块，很可能也是将来最少访问的块。

## Cache替换算法(续1)

- **堆栈替换算法**

堆栈替换算法使用栈顶到栈底各项的先后次序来记录Cache中或Cache中同一组内各个块被访问的先后顺序。栈顶栏存放近期最近被访问过的块的块号，栈底恒存放近期最久没有被访问过的块的块号，即准备被替换掉的块的块号（LRU堆栈实现）。

- **比较对替换算法**

LRU算法用一组硬件的逻辑电路记录同一组中各个块使用的时间和次数，然后按照各个块被访问过的时间顺序排序，从中找出最久没有被访问过的块。用一个两态的触发器的状态来表示两个块之间的先后顺序，再经过门电路就可以找到LRU块。

## Cache替换算法(续2)

- **各种Cache替换算法的优缺点**

(1)随机替换算法：

优点是简单，容易实现；缺点是没有考虑到Cache块的使用历史情况，没有利用程序的局部性特点，从而命中率较低，失效率较高。

(2)FIFO替换算法：

优点是由于它不需记录各个块的使用情况。实现起来也较容易；缺点是虽然考虑到了各块进入Cache的先后顺序这一使用“历史”，但还不能正确地反映程序的局部性特点。

(3) LRU替换算法：

LRU算法较好地反映了程序的局部性特点，失效率较低，但LRU算法比较复杂，硬件实现较困难，特别是当组的大小增加时，LRU的实现代价将越来越高。

## cache一致性问题

- 由于Cache中保存的是主存储器的一部分副本，则有可能在一段时间内，主存储器中某单元的内容与Cache中对应单元的内容出现不一致。

例如：

（1）CPU在写入Cache时，修改了Cache中某单元的内容，而主存储器中对于单元的内容却可能没有改变，还是原来的。此时，如果I/O处理器或其他处理器要用到主存储器中的数据，则可能会出现主存储器内容跟不上Cache对应内容的变化而造成的数据不一致性错误。

（2）I/O处理机或其他处理机已修改了主存储器某个单元的内容，而Cache中对于单元的副本内容却可能没有改变。这时，如果CPU访问Cache并读取数据，也可能会出现Cache内容跟不上主存储器对于内容的变化而产生的不一致性错误。

## cache一致性问题(续1)

- 对于Cache中的副本与主存储器中的内容能否保持一致，是Cache能否可靠工作的一个关键问题。要解决这个问题，首先要选择合适的Cache更新算法。
- 对Cache不一致性问题的解决，主要是需要更新主存储器内容，一般有两种常用更新算法：

(1)写回法：写回法也称拷回法。CPU执行“写”操作时，只把信息写入Cache，而不写入主存储器，仅当该块需被替换时，才相应Cache块写回到主存储器。

(2)写直达法：写直达法也称存直达法。CPU在执行“写”操作时，不仅把信息写入到Cache，而且也写回主存储器，当某一块需要被替换时，不必再写回到主存储器，只需要直接调入新块并覆盖该块即可。

## cache一致性问题(续2)

- 写回法与写直达法比较：

(1)写回法较写直达法速度要快；
(2)在可靠性方面，写回法不如写达法；
(3)写直达法的控制较写回法简单；
(4)写直达法易于实现，但硬件实现代价相对较大。

- 进行“写”操作时，也可能出现写不命中。由于“写”操作并不需要访问该单元中的所有的数据，在出现Cache写不命中时，无论写回法还是写直达法，都需要考虑在写操作的同时是否将其调入Cache，一般有两种选择：

(1)按写分配法：写失效时，除了完成写入主存储器外，还把该写地址单元所在的块由主存储器调入Cache。
(2)不按写分配法：写失效时，只完成写入主存储器，而不把该写地址单元所在的块从主存储器调入Cache。

## Cache性能分析

- 评价Cache存储器，主要是看Cache命中率的高低，命中率主要与下面几个因素有关：

(1)程序在执行过程中的地址流分布情况，其中地址流的分布情况是由程序本身决定的；可通过编写适应Cache的代码；

(2)当发生Cache块失效时，所采用的替换算法；

(3)Cache的容量，块的大小、块的总数；

(4)采用组相联时组的大小等。

## Cache友好代码

```
1 int sumarrayrows(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (i = 0; i < M; i++)
6         for (j = 0; j < N; j++)
7             sum += a[i][j];
8     return sum;
9 }
```

| a[i][j] | j=0 | j=1 | j=2 | j=3 | j=4 | j=5 | j=6 | j=7 |
|---|---|---|---|---|---|---|---|---|
| i=0 | 1 [m] | 2 [h] | 3 [h] | 4 [h] | 5 [m] | 6 [h] | 7 [h] | 8 [h] |
| i=1 | 9 [m] | 10 [h] | 11 [h] | 12 [h] | 13 [m] | 14 [h] | 15 [h] | 16 [h] |
| i=2 | 17 [m] | 18 [h] | 19 [h] | 20 [h] | 21 [m] | 22 [h] | 23 [h] | 24 [h] |
| i=3 | 25 [m] | 26 [h] | 27 [h] | 28 [h] | 29 [m] | 30 [h] | 31 [h] | 32 [h] |

```
1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

| a[i][j] | j=0 | j=1 | j=2 | j=3 | j=4 | j=5 | j=6 | j=7 |
|---|---|---|---|---|---|---|---|---|
| i=0 | 1 [m] | 5 [m] | 9 [m] | 13 [m] | 17 [m] | 21 [m] | 25 [m] | 29 [m] |
| i=1 | 2 [m] | 6 [m] | 10 [m] | 14 [m] | 18 [m] | 22 [m] | 26 [m] | 30 [m] |
| i=2 | 3 [m] | 7 [m] | 11 [m] | 15 [m] | 19 [m] | 23 [m] | 27 [m] | 31 [m] |
| i=3 | 4 [m] | 8 [m] | 12 [m] | 16 [m] | 20 [m] | 24 [m] | 28 [m] | 32 [m] |

## Cache友好代码(续1)

*适当字节填充，减少缓冲不命中数量*

```
1 float dotprod(float x[8], float y[8])
2 {
3     float sum = 0.0;
4     int i;
5
6     for (i = 0; i < 8; i++)
7         sum += x[i] * y[i];
8     return sum;
9 }
```

| Element | Address | Set index | Element | Address | Set index |
|---|---|---|---|---|---|
| x[0] | 0 | 0 | y[0] | 32 | 0 |
| x[1] | 4 | 0 | y[1] | 36 | 0 |
| x[2] | 8 | 0 | y[2] | 40 | 0 |
| x[3] | 12 | 0 | y[3] | 44 | 0 |
| x[4] | 16 | 1 | y[4] | 48 | 1 |
| x[5] | 20 | 1 | y[5] | 52 | 1 |
| x[6] | 24 | 1 | y[6] | 56 | 1 |
| x[7] | 28 | 1 | y[7] | 60 | 1 |

Float x[8]

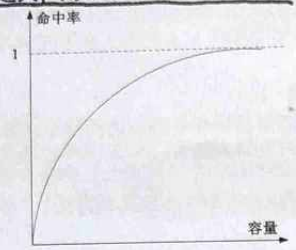
| Element | Address | Set index | Element | Address | Set index |
|---|---|---|---|---|---|
| x[0] | 0 | 0 | y[0] | 48 | 1 |
| x[1] | 4 | 0 | y[1] | 52 | 1 |
| x[2] | 8 | 0 | y[2] | 56 | 1 |
| x[3] | 12 | 0 | y[3] | 60 | 1 |
| x[4] | 16 | 1 | y[4] | 64 | 0 |
| x[5] | 20 | 1 | y[5] | 68 | 0 |
| x[6] | 24 | 1 | y[6] | 72 | 0 |
| x[7] | 28 | 1 | y[7] | 76 | 0 |

Float x[12]

## Cache性能分析(续1)

- Cache命中率与容量的关系

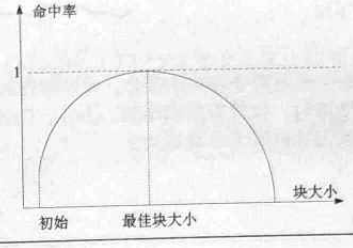
  Cache的容量越大，则Cache的命中率将越高。




## Cache性能分析(续2)

- Cache命中率与块大小的关系

  在采用组相联映象方式的Cache中，当Cache的容量一定时，块的大小也会影响命中率。




## Cache性能分析(续3)

- Cache命中率与组数的关系

  当Cache的容量一定时，分组的数目对于Cache命中率的影响也是很明显的：组数分得越多，命中率会下降，命中率会随着组数的增加而下降；当组数不太大时，命中率降低得相当少；当组数超过一定数量时，命中率下降非常快。

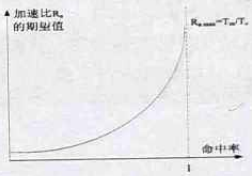
- Cache系统加速定义：

$$R_a = \frac{T_m}{T} = \frac{T_m}{H \bullet T_C + (1-H) \bullet T_m} = \frac{1}{H \bullet \frac{T_c}{T_m} + (1-H)}$$

$$T = H \bullet T_c + (1-H) \bullet T_m$$

## Cache性能分析(续4)

- 从Cache系统加速比定义可以看出，Cache系统的加速比与Cache的命中率H和主存储器访问周期Tm及Cache访问周期Tc有关，而Cache系统中，主存储器的访问周期和Cache的访问周期一般是一定的，所以只要提高Cache的命中率，就可以获得较高的Cache系统加速比，提高存储系统的速度。




## Cache性能的优化

- 除加速比定义衡量Cache存储系统性能外，Cache存储器的平均访问时间是测评存储系统性能的一种更好的指标：

  平均访问时间 = 命中时间 + 失效率*失效开销

- 从平均访问时间这一指标来看，还可以从3个方面对Cache的性能进行优化：

  （1）降低Cache失效率

  （2）减少失效开销

  （3）减少命中时间

## 降低Cache失效率的方法

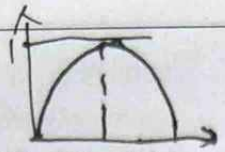
- Cache失效的原因分析：

  (1)强制性失效：对块第一次访问，该块不在Cache中，需从主存储器中将该块调入Cache中。

  (2)容量失效：如果程序执行时，Cache容纳不了所需的所有块，则会发生容量失效。当某些块被替换后，可能随后重新访问又被调入。

  (3)冲突失效：在组相联映象或直接相联映象中，如果太多的冲突块映象到同一组中，产生冲突，则可能会出现某个块刚被替换出去，随后又重新访问而被再次调入。

## 降低Cache失效率的方法(续1)



- 增加Cache块大小

Cache命中率和块大小的关系：在Cache容量一定时，当块大小开始增加时，命中率也开始增加，但当增加到一定程度后，命中率反而开始下降。

失效率和命中率是两个相关的概念，命中率增加时，失效率下降；命中率下降时，失效率反而增加。另外，Cache容量越大，则使失效率达到最低的块大小就越大。

## 降低Cache失效率的方法(续2)

- 增加Cache容量

Cache容量越大，命中率越高，相关，失效率则越低。但增加Cache容量不仅会增加成本，而且也可能会因为复杂的电路结构等而增加Cache的访问时间。

- 指令和数据硬件预取

指令和数据硬件预取是指在处理器访问指令和数据之前，就把它们预取到Cache中或预取到可以比存储器访问速度更快的外部缓冲区中。

指令预取一般有恒预取和不命中预取两种方法。

## 降低Cache失效率的方法(续3)

- 编译器控制预取

硬件预取的一种替代方法是在编译时加入预取指令，在数据被使用之前发出预取请求。有以下两种方式：

(1)寄存器预取：将数据预取到寄存器中。

(2)Cache预取：只将数据预取到Cache中，并不放入寄存器。

- 编译器优化以降低Cache失效率

这种方法是采用软件方法来优化Cache性能，试图通过优化编译时间来改善Cache性能：

(1)程序代码和数据重组

(2)循环交换

(3)分块

## 减少Cache失效开销

- 与降低失效率一样，减少Cache失效开销同样可以缩短Cache存储器的平均访问时间并提高Cache的性能。

(1)采用两级Cache：在原Cache和存储器之间增加一级Cache，构成两级Cache。其中第一级Cache可以让它小到足以与快速的处理器运行时钟周期相匹配，而第二级Cache则让它大到足以捕获到对内存进行的大多数访问，从而有效地降低了失效开销。

(2)让读失效优先于写：提高写直达Cache性能最重要的方法是设置一个容量适中的写缓存。然而写缓存中可能包含读失效时所需单元的最新值，这个值尚未写入存储器，导致了存储器访问的复杂化。解决方法是让读失效等待，直至写缓存为空。

全写完再去读

## 减少Cache失效开销(续1)

- 合并写缓冲区

采用写直达法的Cache要有一个写缓冲区，如果写缓冲区为空，就把被替换的数据和相应地址写入缓冲区。

- 请求字处理技术

处理器在同一时刻只需要调入块中的一个字(即请求字)，不必等到全部的块调入Cache，就可以将该字送往处理器并重新启动处理器进行访问，一般有以下两种策略：

(1)请求字优先：调块时，先向存储器请求处理器所要的请求字。一旦该请求字到达即送往处理器，让处理器继续执行，同时可以从存储器中调入该块的其他字。

(2)提前重启动：在请求字没到达处理器时，处理器处于等待状态。

## 减少命中时间

- 除了通过降低失效率和减少失效开销来优化Cache性能的方法以外，还可通过减少命中时间来优化Cache的性能。命中时间也是平均访问时间的一个组成部分，它的重要性在于它会影响处理器的时钟频率。

(1)小而简单的Cache减少命中时间

采用容量小、结构简单的Cache，这样快表较小，查表的速度较快，从而有效地提高了Cache的访问速度。

(2)路预测减少命中时间

路预测要求Cache中预留特殊的比较位，用来预测下一次访问Cache时可能会用到的路或块。

## 减少命中时间(续1)

(3)踪迹Cache(Trace Cache)减少命中时间

踪迹Cache中存储的是处理器所执行的动态指令序列，而不是用于存储主存储器中给出的静态指令序列。L例如，在Pentium4处理器的踪迹Cache中由于使用了译码微操作，从而节省了译码时间。

(4)流水线Cache访问

流水线Cache访问方法是将流水线、Cache访问以及使一级Cache命中时的有效时延分散到几个时钟周期。它实际上并不能真正减少Cache命中时间，但可以提供较短的周期时间和高宽带。

## 虚拟存储器VIRTUAL EMORY

"页"好 程序.地址寻码.

段式.页式

## 1虚拟存储器

- 虚拟存储器是由主存和联机的外存共同组成的，在主存的容量不能满足要求时，数据可存放在外存中，但在程序中仍按地址进行访问外存空间。
- 在虚拟存储器中，应用程序员是直接用机器指令的地址码对整个程序统一编址的，虚拟存储器的空间大小取决于它能产生的地址位数。
- 从程序员的角度看，存储空间扩大了，并可以放得下整个程序，程序不必作任何修改就可以以接近于实际主存的速度在这个虚拟存储器上运行。

## 1虚拟存储器工作原理

- 根据采用的存储映象算法，可以将虚拟存储器的管理方式分为段式、页式和段页式3种。
- 段式管理

将主存按段分配的存储管理方式称为段式管理。对于一个复杂的程序可以分解为多个逻辑上相当独立的模块。这些模块可以是主程序，各种子程序，也可以是表格、向量、数组等。每个模块都是一个单独的程序段，都从0地址开始相对编址，段的长度可长可短，甚至可以事先无法确定。在段式管理的系统中，当某程序段由辅存调入主存时，系统在主存中给该段分配一块空间，并给出基址(即该段在主存中的起始地址)，由基址和每个单元在段内的相对位移量就可以形成这些单元在主存中各自的实际地址，进行访问。

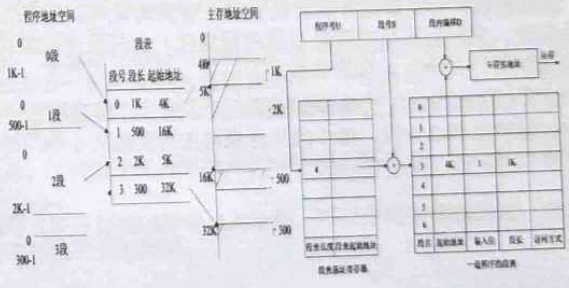
## 1虚拟存储器工作原理(续1)

- 在段式管理中，每一道程序都有一个段表，用以指明各段在主存中的状态信息。
- 段表中包括段号(或段名)、段长和起始位置等信息。断号字段用于存放程序段的断号或名称，如果断号是连续的，则断号这一字段信息可以去掉，直接根据起始位置和段长来实现程序段到主存储器的映象。
- 段长是该段的长度，可用于访问地址的越界检查。起始地址用于存放该程序段装入主存中的起始(绝对)地址。此外，还可能会设置一个装入位来指示该段是否已装入主存，如装入位为“1”，表示已装入；为“0”，表示尚未装入。

## 1虚拟存储器工作原理(续2)

- 段式管理地址映像方法和地址变换方法

对程序进行切割



## 1 虚拟存储器工作原理(续3)

- 段式虚拟存储器的主要优点:
  (1)程序的模块性能较好。
  (2)分段还便于程序和数据的共享。
  (3)容易以段为单位实现存储保护。
  (4)程序的动态链接和调度比较容易。

- 段式虚拟存储器的主要缺点:
  (1)地址变换费时。
  (2)主存储器利用率低。
  (3)对辅存管理较困难。

## 1 虚拟存储器工作原理(续4)

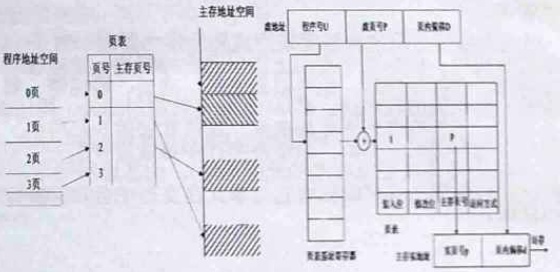
- 页式管理

页式存储是将虚拟存储空间和实际的存储空间都等分成固定大小的页，各虚拟页可以装入到主存中不同的页面位置。页是一种逻辑上的划分，页面的大小随机器而异。目前，一般计算机系统中，页的大小通常指定为磁盘存储器物理块大小的整数倍。

在虚拟存储器中，虚拟地址空间中划分的页称为虚页，主存地址空间中划分的页称为实页。则页式管理的地址映象就是完成虚拟地址空间中的虚页到主存地址空间中的实页的变换。页式管理用一个页表，其中包括页号、主存页号等。页号一般用于存放该页在页表中的页号(或行号)，因此可以省略；页的长度是固定的，因此也省略了。页表中也可以设置一些标志位，如装入位、修改位等。

## 1 虚拟存储器工作原理(续5)

- 页式管理地址映像方法和地址转换方法




## 1 虚拟存储器工作原理(续6)

- 页式虚拟存储的主要优点:
  (1)主存的利用率高。
  (2)页表相对简单。
  (3)地址映象与变换速度较快。
  (4)对辅存的管理比较容易。

- 页式虚拟存储的主要缺点:
  (1)程序的模块化性能不好。由于用户程序被强制按固定大小的页来划分，因此一页可能是程序段的某一部分，也可能包含了两个或两个以上的程序段。
  (2)页表很长，从而要占用很大的存储空间。

## 1 虚拟存储器工作原理(续7)

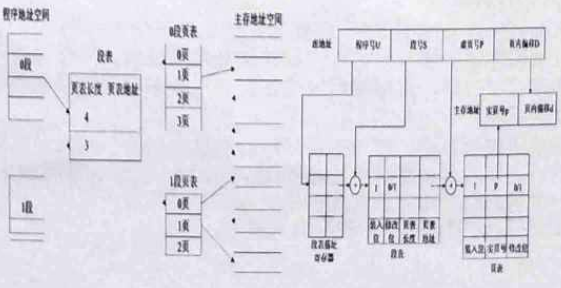
- 段页式管理

段式管理和页式管理各有其优缺点，段页式管理则是将两者结合起来，同时利用段式管理在程序模块化方面的优点和页式管理在管理主存和辅存方面的优点。

将主存储器的物理空间等分成固定大小的页，将虚拟存储空间中的程序按模块分段，每个段又分成与主存页面大小相同的页。

## 1 虚拟存储器工作原理(续8)

- 段页式管理地址映像方法和地址转换方法



## 2 地址映像与转换

- 当把大的虚存空间中的内容压缩到小的主存空间中去，则主存中的每一个页或段必须与多个虚拟存储空间中的页或段相对应。
- 主存中的一个实页要对应多少个实页与采用的映象方式有关。此时，就不可避免地发生两个以上的虚页想要进入主存中同一个页面位置的现象，这种现象称为页面争用或实页冲突。一旦发生实页冲突，只能首先装入其中的一个虚页，等到不再需要这个虚页时才可装入其它的，从而导致了执行效率的下降。因此，映象方式的选择主要应考虑能否尽量减小实页冲突概率。
- 操作系统一般都允许将每道程序的任何虚页可以映象到任何实页位置，即全相联映象。仅当一个任务要求同时调入主存的页面超出主存页数时，两个虚页才会争用同一个实页位置。因此，全相联映象的实页冲突率最低。

## 3 内部地址变换优化

- 在虚拟存储系统中，如果不采取有效的措施，则访问主存储器的速度要降低很多。造成这个速度降低的主要原因是：每次进行访存时，都必须进行内部地址变换，其概率为100%。所以，如何加快用户虚地址到主存实地址的变换，是缩短访存时间的关键。只要内部地址的变换速度高到使整个系统的访存速度非常接近于不采用虚拟存储器时的访存速度，虚拟存储器的性能才能真正实现。
- 在虚拟存储器进行地址变换时，首先必须查段表或页表，或既查段表也查页表，来完成虚地址到实地址相应的转换。由于页表容量较大且存放在主存中，因此每访存一次，还需因查表而加一次访存；如果采用的是段页式虚拟存储器，则需因两次查表而加两次访存。
- 结果是主存储器的访问速度比不采用虚拟存储器的访存速度要慢2到3倍。
- 如何提高页表的访问速度？

## 3 内部地址变换优化(续1)

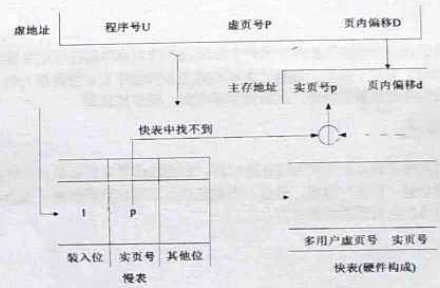
- 快表 地址的cache

根据程序的局部性特点，对页表内各项的访问并不完全是随机的。在一段时间内，实际可能只用到表中很少的几项。因此，应该重点提高使用概率较高的这部分页表的访问速度。可以使用快速硬件来构成比全表小得多的表，表中存放的是近期经常要使用的页表项，这个表称为“快表”。

- 相应地，原先存放全部虚地址和实地址之间映象关系的表还是存放在主存中，将其称为“慢表”。

## 3 内部地址变换优化(续2)

- 具有快表的地址转换方法




## 3 内部地址变换优化(续3)

- 快表的存在对所有的程序员都是透明的。
- 实际上，快表与慢表也构成了一个两级存储层次，其访问速度接近于快表的速度，存储容量却是慢表的容量。当然，快表的命中率如果不高，则系统的效率就会大大降低。
- 要提高快表的命中率，最直接的办法是增加快表的容量。快表的容量越大，其命中率就越高；但容量越大时，其相联查找的速度就越慢。所以，快表的命中率和查表速度是矛盾的。

## 3 内部地址变换优化(续4)

- 为了提高查找速度，可以减少相联比较的位数。
- 在同样容量的情况下，相联比较的位数越少，则相联查找所花费的时间就会越少。
- 例如，将虚地址中参与相联比较位中的用户字段u去掉，这是因为快表在一段时间内总是对应于同一个任务或用户，它们的u值是不变的。这样，相联比较的位数就减少了一位，查找速度也相应地提高了。
- 但是，这种方法会降低快表的命中率，降低虚、实地址的变化速度。
- 也可以采用普通的按地址来访问，可以使用顺序查找法、对分查找法和散列查找法等。

## 3 内部地址变换优化(续5)

- 目前，计算机系统都采用相联寄存器组、硬件的散列压缩、快慢表结构和多个相等比较器等方法，来提高系统的性能。

## 4 页面替换算法及实现

- 同高速缓冲存储器一样，在虚拟存储器中，当访问的指令和数据不在主存时，发生页面失效，需要从辅存中将包含该指令或数据的一页（或一段）调入到主存储器中。
- 虚存空间比主存空间大得多，必然也会出现主存中所有页面都已经被占用，或者所有主存空间都已经被占用的情况，这时，如果继续把辅存中一页调入主存，就会发生实页冲突。此时，只有从主存空间中选出一页替换出去，让出空间来接纳新调入的页。
- 应该选择哪一页进行替换呢？这就是页面替换算法要解决的问题。

## 4 页面替换算法及实现(续1)

- **页面替换算法**

(1)随机算法

随机算法是将软的或硬的随机数产生器产生的随机数作为主存储器中被替换的页的页号。这种算法最简单，且易于实现。但是，没有利用主存储器中页面使用情况的“历史”信息，也没有反映程序的局部性特点，从而命中率较低，较少被使用。

(2)先进先出算法

选择最早被调入主存储器的页作为被替换的页。它的优点是实现容易，并利用了主存储器中页面使用情况的“历史”信息，但是，不能正确反映程序的局部性。因为最先进入主存的页很可能也是现在经常要使用的页。

## 4 页面替换算法及实现(续2)

(3)近期最少使用算法，即LFU算法

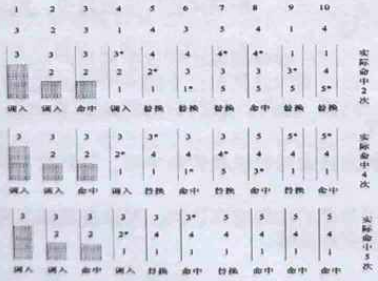
选择近期最少被访问的页作为被替换的页。该算法能比较正确地反映程序的局部性。

(4)最优替换算法，即OPT算法

选择将来一段时间内最久不被访问的页作为被替换的页。

## 4 页面替换算法及实现(续3)

- 页面替换算法性能分析：同一地址流命中率比较




## 5 提高主存命中率的方法

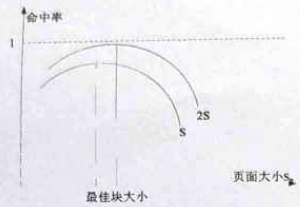
- 通常，影响主存命中率的主要因素有：

(1)程序在执行过程中的页地址流分布情况；
(2)所采用的替换算法；
(3)页面大小；
(4)主存容量；
(5)所采用的页面调度方法。

## 5 提高主存命中率的方法(续1)

- 页面大小的选择

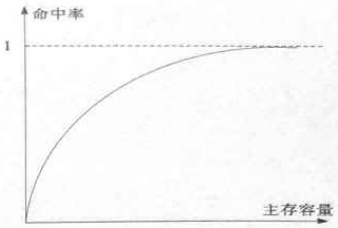
与Cache命中率与页面大小的关系一样，主存命中率与页面大小的关系也不是线性的。页面大小还与主存容量、程序的页地址流分布情况等因素有关。




## 5 提高主存命中率的方法(续2)

- 主存容量

与Cache类似，当主存容量增加时，主存命中率也会相应地提高。




## 5 提高主存命中率的方法(续3)

- 页面调度方法，页面调度就是系统给用户分配主存页面数的过程。
- 一般有三种方式：

(1)分页方式:

将整个程序先链接装配，将整个程序装入主存后才运行，其命中率为100%，但是主存的利用率较低；

(2)请求页式:

在发生页面失效时，才将所需要的页装入主存。其主存的利用率很高，但命中率将受到频繁的页面替换的影响；

(3)预取式:

根据程序的局部性特点，在程序被挂起之后又重新开始运行之前，预先调入相关的页面。这种方法可能会因为预先调入的页面用不上而造成时间和空间上的浪费。