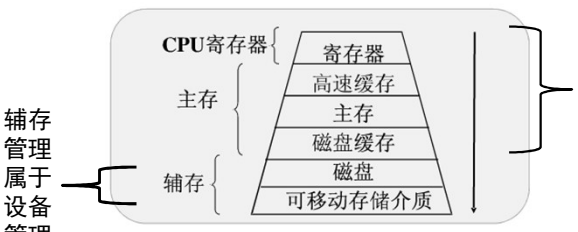
第四章　存储器管理

存储系统设计的三个问题

容量、速度和成本

- ■容量：需求无止境
- ■速度：能匹配处理器的速度
- ■成本问题：成本和其它部件相比在合适的范围之内

存储器的多层结构



- 容量愈来愈大
- 访问数据的速度愈来愈慢
- 价格愈来愈便宜

存储器的层次结构

- 操作系统的工作就是协调三级存储器的工作
- 操作系统中管理存储器的部分称为存储管理器
- 任务：
  - 跟踪哪些存储器正在被使用
  - 哪些存储器空闲
  - 在进程需要时为它分配存储器
  - 使用完毕后释放存储器
  - 在主存无法容纳所有的进程时管理主存和磁盘间的交换

寄存器

- CPU内的硬件
- 存放处理机运行时的数据
- 存取速度非常快
- 价格昂贵
- 8/32/64位
- 数十个到数百个
- 容量小

5

主存储器

- 内存或主存
- 计算机中的主要部件
- 用于存放进程运行时的程序和数据
- CPU的控制部件从内存中取得指令和数据
- 从内存中读取数据并装入寄存器
- 从寄存器数据存入内存中
- 从数十MB到数GB（VLSI）
- 内存数据存取速度快

6

可执行存储器

- 寄存器和主存储器被称为可执行存储器

进程可以在很少的时钟周期内使用一条load或store指令对可执行存储器进行访问。

- 对辅存的访问则需要通过I/O设备实现

访问中将涉及到中断、设备驱动程序以及物理设备的运行，所需耗费的时间远远高于访问可执行存储器的时间，一般相差3个数量级甚至更多。

7

高速缓存

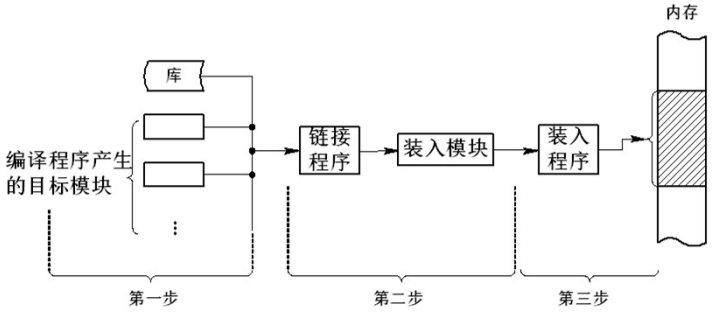
- 现代计算机结构中的一个重要部件
- 介于寄存器和主存储器之间的存储器
- 用于备份主存中较常用的数据
- 减少处理机对主存的访问次数
- 容量远大于寄存器
- 比内存约小两到三个数量级
- 从几十KB到几MB
- 访问速度快于主存储器

8

磁盘缓存

- 暂时存放频繁使用的一部分磁盘数据和信息
- 减少访问磁盘的次数
- 不是一种实际存在的存储器
- 利用主存中的部分存储空间暂时存放从磁盘中读出或写入的信息

- 多道程序环境下，程序要运行必须为之创建进程，而创建进程的第一件事就是分配内存



程序的装入

- 逻辑地址：

应用程序经编译后形成目标程序
目标程序经过链接后形成可装入程序
装入程序的地址都是从0开始
装入程序中的其他地址相对于起始地址计算

- 物理地址：

主存中一系列存储信息的物理单元的地址

- 重定位：

逻辑地址（相对地址）到物理地址（绝对地址）的映射

程序的装入

- 内存空间（或物理空间、绝对空间）

内存地址的集合

- 逻辑地址空间

由程序中逻辑地址组成的地址范围

程序的装入

- 绝对装入方式
- 可重定位装入方式
- 动态运行时装入方式

程序的装入

1. 绝对装入方式(Absolute Loading Mode)
   - 单道程序环境中，事先确定了程序将内存中的绝对地址
   - 编译生成的目标代码采用绝对地址进行编址
   - 绝对地址的产生
     - 程序员直接赋予
     - 通常在程序中采用符号地址，在编译或汇编时，再将符号地址转换为绝对地址。
     - 编译或汇编时产生

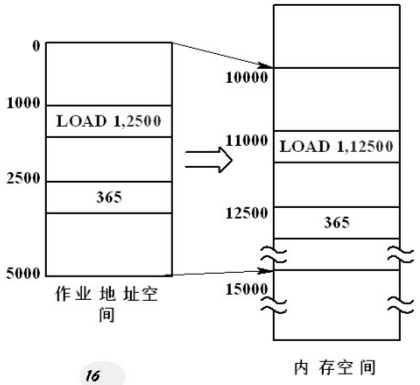
程序的装入

2. 可重定位装入方式(Relocation Loading Mode)
   - 根据内存的当前情况，将装入模块装入到内存的适当位置
   - 源程序编译生成的目标模块都采用相对地址进行编址，每个模块都从0开始编址
   - 链接后模块也采用相对地址编址

程序的装入

2. 可重定位装入方式(Relocation Loading Mode)

- 重定位
装入时对目标程序中指令和数据地址修改的过程
- 静态重定位
地址变换在装入时一次完成的，以后不再改变



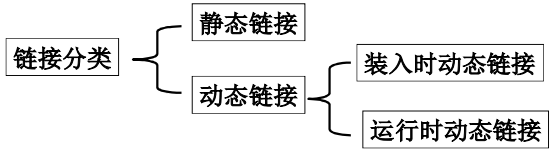
程序的装入

3. 动态运行时装入方式(Denamic Run-time Loading)

- 把装入模块装入内存后，不立即把装入模块中的相对地址转换为绝对地址
- 地址转换推迟到程序真正要执行时才进行
- 装入内存后的所有地址都仍是相对地址
- 绝对地址的转换有硬件支持
- 能保证进程的可移动性

程序的链接

链接是指多个目标模块在执行时的地址空间分配和相互引用



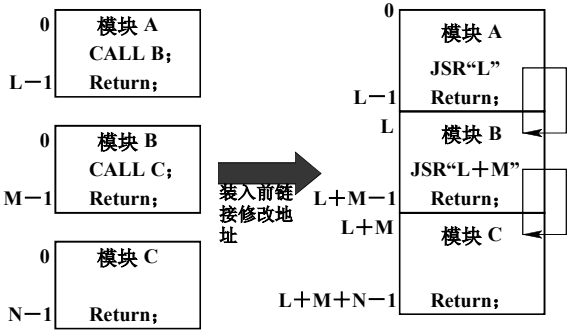
程序的链接

静态链接方式(Static Linking)

- 在程序运行前，先将各目标模块及所需的库函数链接成一个完整的装配模块，以后不再拆开
- 装配装入模块须解决两个问题
  - ✓ 对相对地址进行修改
  - ✓ 变换外部调用符号

程序的链接

静态链接方式(Static Linking)



(*a*) 目标模块(外存) (*b*) 装入模块（外存）

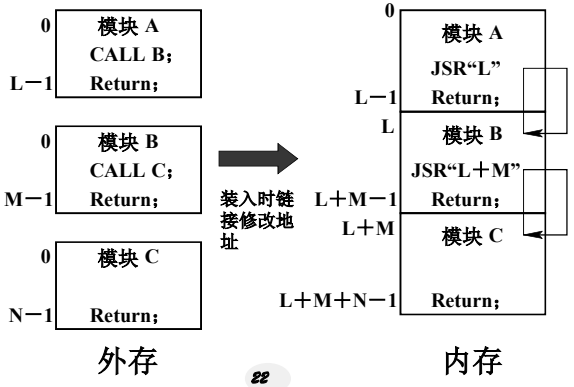
程序的链接

装入时动态链接(Load-time Dynamic Linking)

- 将用户的源程序编译后所得的一组目标模块在装入内存时采用边装入边链接的方式
  - ✓ 便于修改和更新
  - ✓ 便于实现对目标模块的共享

程序的链接

装入时动态链接(Load-time Dynamic Linking)



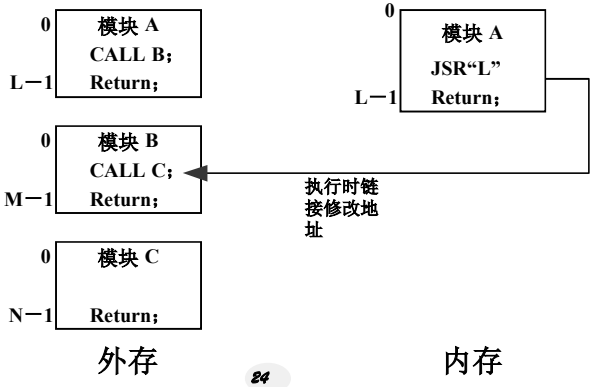
程序的链接

运行时动态链接(Run-time Dynamic Linking)

- 对某些模块的链接推迟到执行时才进行，
- 在执行过程中，发现被调用模块尚未装入内存时，由OS找到该模块并将之装入，链接到调用者模块
- 凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，
- 加快程序的装入过程，节省大量的内存空间

程序的链接

运行时动态链接(Run-time Dynamic Linking)



连续分配方式

为一个用户程序分配一个连续的内存空间

- 程序中代码或数据的逻辑地址相邻
- 体现在内存空间分配时物理空间的相邻

- 单一连续分配
  - 用于单用户，单任务中
- 分区式分配
  - 固定分区分配
  - 动态分区分配
  - 动态可重定位分区分配

单一连续分配

- 用于单用户、单任务的操作系统中
- 把内存分为
  - 系统区：OS使用，通常放在内存低址部分
  - 用户区：用户可使用的全部内存空间
- 用户区内存中，仅装有一道用户程序
- 整个内存的用户空间单一程序独占

固定分区分配

- 最简单的可运行多道程序的存储管理方式
- 内存用户空间划分为若干个固定大小的区域
- 每个分区中只装入一道作业

固定分区分配

3. 存储保护——界地址保护法

- 固定分区存储管理中，采用静态地址重定位
- 要防止用户程序对操作系统形成的侵扰
- 防止用户程序之间形成侵扰
- 在 CPU 中设置一对专用寄存器用于存储保护
- 低界限寄存器：作业所在分区的低边界地址
- 高界限寄存器：作业所在分区的高边界地址

固定分区分配

- 优点：
  - 易于实现，开销小。
- 缺点：
  - 内碎片造成浪费
  - 分区总数固定，限制了并发执行的程序数目

动态分区分配

1. 分配中数据结构

- 配置相应的数据结构
  - 记录内存的使用情况
  - 为分配提供信息
  - 实现分区分配。
- 常用的数据结构：空闲分区表、空闲区链表

动态分区分配

1. 分配中数据结构

空闲分区表

序号	分区大小	分区始址	状态
1	64KB	44KB	0
2	24KB	132KB	0
3	40KB	210KB	0
4	30KB	270KB	0

- 从空闲分区表中找一个足以容纳该作业的空闲区
- 若这个分区比较大，则一分为二
- 一部分分配给作业，另一部分仍作为空闲区留在表中

动态分区分配

1. 分配中数据结构

空闲分区链表

在每个空闲分区的两端设置附加信息：

(1) 状态信息： “0”表示该区空闲，“1”表示已分配

(2) 分区大小(以字或块为单位)

(3) 指针：

- 指向其上或其下分区的位置
- 首字指针(又叫前向指针)指向下一分区
- 尾字指针(又叫后向指针)指向其上一分区位置

动态分区分配

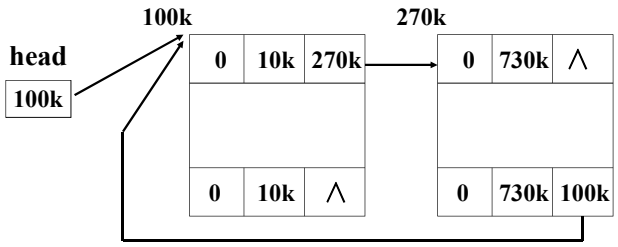
1. 分配中数据结构

状态位	分区大小（N+2）	前向指针	字
大小为N的已分配区或空闲区			
状态位	分区大小（N+2）	后向指针	字

所有空闲区连成一个双向链
设置一个链表头指针head，指向链中第一个空闲区

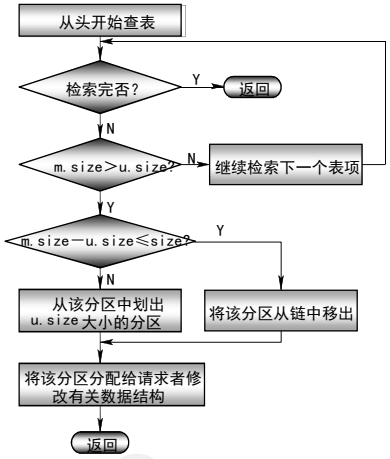
动态分区分配

1. 分配中数据结构



动态分区分配

2. 分区分配操作

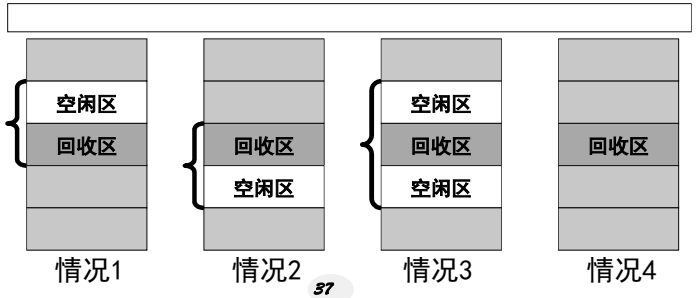


动态分区分配

3. 分区内存回收

进程运行结束释放内存时，系统根据回收区的首地址，把它插入到空闲链表中。

根据回收区的位置，有四种情况需处理：



动态分区分配

3. 分区内存回收

(d) 若释放区R上下都不邻接空闲区

将其插入空闲区链的适当位置

动态分区分配

4. 保护

界地址保护法

➢采用基址寄存器（存放分区的物理起始地址）和限长寄存器（存放分区长度）进行存储保护

➢基址寄存器和动态重定位的定位寄存器合并使用一个寄存器（节约硬件成本）

➢程序运行时，硬件自动进行地址比较，若地址溢出，发生地址越界中断

存储键保护法

➢为分区设置保护键，相当于一把锁

➢为进程分配存储键，相当于一把钥匙，存放在 PSW 中

➢每次内存访问检查锁和钥匙是否匹配，不匹配则保护性中断

动态分区分配

5. 碎片问题

分配回收后内存中存在的很小难以使用的空闲块

➢ 外部碎片

目前所有孔的空间之和可以满足请求，但是这些孔互相之间并没有毗邻

➢ 内部碎片

给进程的孔比进程的请求略微大一些。多出的部分存在于分区内，但不能被其他进程使用

动态分区分配

5. 碎片问题

问题：开销大；移动时机

紧凑技术：移动内存中作业的位置，合并小空闲区域

操作系统
用户程序1
10kb
用户程序3
30kb
用户程序6
14kb
用户程序9
26kb

操作系统
用户程序1
用户程序3
用户程序6
用户程序9
80kb

紧凑

动态分区分配

6. 分配算法

- 基于顺序搜索的动态分区分配算法
- 基于索引搜索的动态分区分配算法
- 动态可重定位分配算法

基于顺序搜索的动态分区分配

- 寻找某个空闲分区，其大小需大于或等于程序的要求
- 若是大于要求，则将该分区分割成两个分区
- 其中一个分区为要求的大小并标记为“占用”
- 另一个分区为余下部分并标记为“空闲”
- 分区的先后次序通常是从内存低端到高端
- 分区释放算法：需要将相邻的空闲分区合并成一个空闲分区

基于顺序搜索的动态分区分配

- 首次适应算法(first fit, FF)
- 循环首次适应算法(next fit, NF)
- 最佳适应算法(best fit, BF)
- 最坏适应算法(worst fit, WF)

基于顺序搜索的动态分区分配

1）首次适应算法FF
- 空闲分区链以地址递增顺序链接
- 分配时从链首开始查找，找到一个大小可满足的空闲分区，划出一块给请求者
- 优点：
  - 简单；
  - 优先利用低地址空闲区，保留高地址大空闲区
- 缺点：
  - 造成在低地址部分很多难以利用的小空闲分区
  - 查找效率低

基于顺序搜索的动态分区分配

2）循环首次适应算法NF
- 每次分配时从上一次找到空闲分区的下一个空闲区开始查找
- 优点：
  - 减少查找空闲分区开销
  - 空闲分区分布更均匀
- 缺点：
  - 缺乏大的空闲区

基于顺序搜索的动态分区分配

3）最佳适应算法
- 空闲区按**容量由小到大**排序
- 每次分配时，把能满足要求、又是最小的分区分配给作业
- 优点：
  - 不缺乏大的空闲区
- 缺点：
  - 会在存储器中留下“零头（或碎片）”
  - 查找效率低

基于顺序搜索的动态分区分配

4）最差适应算法
- 空闲区按**容量由大到小**排序
- 分配把能满足要求、又是最大的分区分配给作业
- 优点：
  - 剩余的空间最大化
  - 不出现太小的“零头”
- 缺点：
  - 缺乏大的空闲区

基于顺序搜索的动态分区分配

优点：

- 便于动态申请内存
- 便于共享内存
- 便于动态链接

缺点：

- 碎片问题(外碎片)
- 要求连续的内存空间
- 内存利用率不高
- 受实际内存容量限制

基于索引搜索的动态分区分配

- 快速适应算法
- 伙伴系统
- 哈希算法

基于索引搜索的动态分区分配

1）快速适应算法（quick fit）

- 分类搜索算法，将空闲分区根据其容量大小进行分类
- 每一类相同容量空闲分区单独设立一个空闲分区链表
- 系统中存在多个空闲分区链表
- 在内存中设立一张管理索引表
  - 每一个表项对应了一种空闲分区类型
  - 记录了空闲分区链表的表头指针

基于索引搜索的动态分区分配

1）快速适应算法（quick fit）

优点：

- 查找效率高
- 能保留大的分区
- 不会产生外碎片

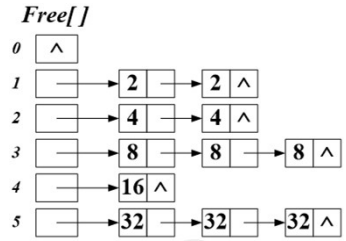
缺点：

- 分区归还主存时算法复杂，系统开销大
- 空间浪费较为严重

基于索引搜索的动态分区分配

2）伙伴系统（Buddy System）

- 尺寸为2^i的空闲块被分为两个大小为2^{i-1}的空闲块——伙伴
- 两个大小为2^{i-1}的空闲伙伴块可被合并为长2^i的空闲块
- 两个伙伴的物理地址连续
- 辅助数据结构：位图，空闲链表数组free[ ]



基于索引搜索的动态分区分配

2）伙伴系统（Buddy System）

伙伴系统中，大小为2^k，地址为x的内存块

伙伴块的地址用$buddy_k(x)$通式表示为：

$$buddy_k(x) = \begin{cases} x + 2^k (若 x \ MOD \ 2^{k+1} = 0) \\ x - 2^k (若 x \ MOD \ 2^{k+1} = 2^k) \end{cases}$$

基于索引搜索的动态分区分配

2）伙伴系统（Buddy System）

作业需求：A/80KB，B/50KB，C/100KB，D/60KB；作业释放：B，D

128k	256k		384k	512k	640k	768k	896k	1M
A	128		256		512			
A	B	64	256		512			
A	B	64	C	128	512			
128	B	64	C	128	512			
128	B	D	C	128	512			
128	64	D	C	128	512			
256			C	128	512			
1024								

基于索引搜索的动态分区分配

3）哈希算法（hash）

分析：

- 分类搜索算法和伙伴系统算法将空闲分区根据分区大小进行分类，对于每一类具有相同大小的空闲分区，单独设立一个空闲分区链表。
- 为进程分配空间时，需要在一张管理索引表中查找到所需空间大小所对应的表项，从中得到对应的空闲分区链表表头指针，从而通过查找得到一个空闲分区

基于索引搜索的动态分区分配

3）哈希算法（hash）

问题：

- 如果对空闲分区分类较细，则相应索引表的表项也就较多，因此会显著地增加搜索索引表的表项的时间开销

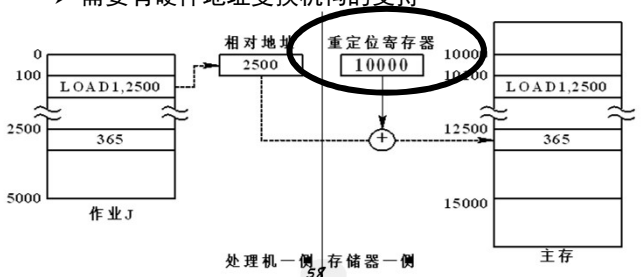
求解：

- 利用空闲分区表的规律，建立哈希函数，构造哈希表

动态可重定位分区分配

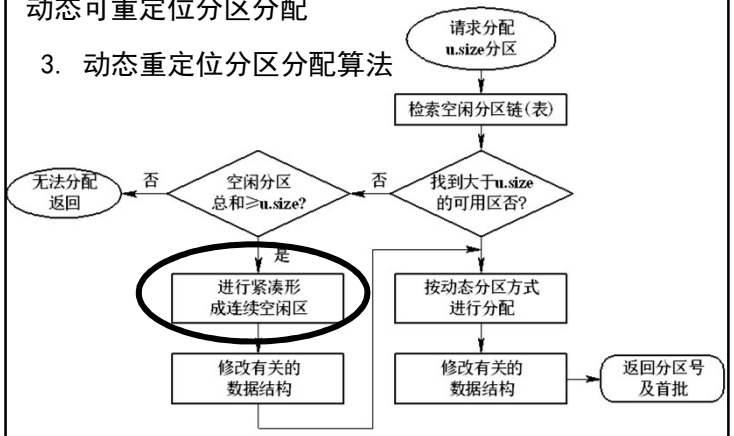
2. 动态重定位的实现

- 作业装入内存后的所有地址仍是相对地址
- 指令执行时将相对地址转换成物理地址
- 需要有硬件地址变换机构的支持



动态可重定位分区分配

3. 动态重定位分区分配算法



动态可重定位分区分配

优点：

- 解决了可变分区分配所引入的“外碎片”问题
- 消除内存碎片，提高内存利用率。

缺点：

- 提高硬件成本
- 紧凑时花费ＣＰＵ时间

	单道连续分配	多道固定连续分配	多道可变连续分配
作业道数	1	<=N（分区块数）	多道
内部碎片	有	有	无
外部碎片	无	无	有
硬件支持	1、界地址寄存器 2、越界检查机构	1、上下界寄存器；2、越界检查机构 3、基地址寄存器；4、长度寄存器； 5、动态地址转换机构	
可用空间管理			1、数组；2、链表
解决碎片方法			紧凑
解决空间不足	覆 盖		
提高作业道数	交 换		

- 单一连续分配
- 分区式分配
  - 固定分区分配
  - 动态分区分配
    - 基于顺序搜索的动态分区分配算法
      - 首次适应算法(first fit, FF)
      - 循环首次适应算法(next fit, NF)
      - 最佳适应算法(best fit, BF)
      - 最坏适应算法(worst fit, WF)
    - 基于索引搜索的动态分区分配算法
      - 快速适应算法(quick fit, QF)
      - 伙伴系统(Buddy System)
      - 哈希算法(Hash)
    - 动态可重定位分配算法

主存不足的存储管理技术

- 移动
- 对换
- 覆盖

主存不足的存储管理技术

移动技术

- 移动条件

  碎片浪费可用空间
- 移动时机

  进程释放分区时

  进程装入分区时
- 移动算法

主存不足的存储管理技术

对换技术

1. 对换的引入

◈指把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据，调入内存

◈“整体对换”或“进程对换”：以整个进程为单位

◈“页面对换”或“分段对换”：以“页”或“段”为单位

主存不足的存储管理技术

对换技术

2. 对换空间的管理

主要目标

◈提高进程换入和换出的速度

对换区中空闲盘块的管理

◈在系统中配置相应的数据结构，记录外存的使用情况

◈空闲分区表中的每个表目中应包含两项：对换区的首址及其大小（单位：盘块号和盘块数）

对换区的分配采用连续分配方式

◈首次适应算法

◈循环首次适应算法

◈最佳适应算法

主存不足的存储管理技术

对换技术

回收操作

（1）回收区与插入点的前一分区F1相邻接；

（2）回收区与插入点的后一分区F2相邻接；

（3）回收区还同时与F1和F2二个分区相邻接；

（4）回收区的前、后没有与之相邻接的空闲分区。

处理方法也与动态分区分配式的方法相同

主存不足的存储管理技术

对换技术

3. 进程的换出与换入

进程的换出

◈系统选择处于阻塞态且优先级最低的进程作为换出进程

◈启动盘块将进程程序和数据传送到磁盘的对换区

◈若传送未出现错误，便回收其所占用的内存空间

◈对该进程的进程控制块做相应的修改

进程的换入

◈系统应定时查看进程状态，找出就绪状态换出进程

◈将其中换出时间(换出到磁盘上)最久的进程作为换入进程

◈换入，直至已无可换入的进程或无可换出的进程为止

主存不足的存储管理技术

对换技术

- ◈ 选出被换出进程：
  - 因素：优先级，驻留时间，进程状态
- ◈ 换出过程：
  - ◈ 对于共享段：计数减1， 是0则换出，否则不换
  - ◈ 修改PCB和MCB（或内存分配表）
- ◈ 选择换入进程：优先级，换出时间等
- ◈ 换入过程：
  - ◈ 申请内存
  - ◈ 换入

主存不足的存储管理技术

对换技术

对换的作用

- ◈解决主存容量不足
- ◈平衡系统负载

对换进程的选择

- ◈时间片耗尽，优先权较低
- ◈代码、数据、堆栈
- ◈时机：时间片结束，I/O操作
- ◈地址重定位

UNIX对换器

Windows对换器

主存不足的存储管理技术

覆盖技术

覆盖的作用

- ◈解决进程所需空间超过物理空间大小或分区大小的问题

覆盖的实现技术

- ◈程序划分固定区/覆盖区，按调用结构和次序分批调入覆盖区

覆盖技术的不足

- ◈程序员进行内存管理
- ◈代码量不可超出主存容量

离散分配方式

1、分页存储管理

2、分段存储管理

3、段页式存储管理

- 存储器管理
  - 连续分配方式
  - 离散分配方式
    - 分页存储管理
      - 基本分页存储管理
      - 请求分页存储管理
    - 分段存储管理
      - 基本分段存储管理
      - 请求分段存储管理
- 段页式存储管理
  - 基本分页存储管理
  - 基本分段存储管理
- 虚拟存储器 ← 页面置换算法
  - 请求分页存储管理
  - 请求分段存储管理

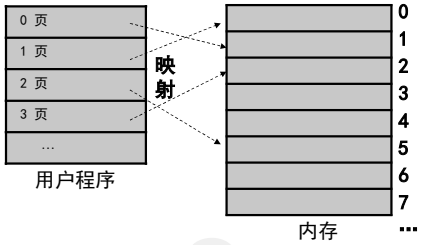
基本分页存储管理方式

基本的（纯）分页管理方式

- 不具备页面对换功能的分页存储管理
- 不具有支持实现虚拟存储器的功能
- 单个作业全部装入内存后方能运行

基本思想

- 进程逻辑空间分成若干大小相等的片，称为页面或页
- 内存空间分成与页大小的若干个存储块，称为物理块或页框
- 进程分配内存以块为单位装入多个可不相邻接的块中



页面与页表

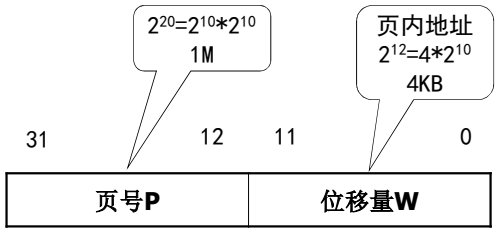
- 页面：将一个**进程的逻辑地址空间**分成若干个大小相等的区域，称为页面或页，并加以编号，从0开始编制页号，页内地址是相对于0编址。
- 物理块：**内存**按页的大小划分为大小相等的区域，称为物理块（物理页面，页框(frame)，帧），同样加以编号，如0＃块、1＃块等等
- “页内碎片”：为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于**进程的最后一页**经常装不满一块而形成了不可利用的碎片

页面与页表

- 页面大小应适中，2的幂，通常为1KB~8 KB
- 页面若太小
  - 虽然可使内存碎片减小，从而减少了内存碎片的总空间，有利于提高内存利用率，但也会使每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存； 此外，还会降低页面换进换出的效率
- 页面较大
  - 虽然可以减少页表的长度，提高页面换进换出的速度，但却又会使页内碎片增大

页面与页表

地址结构



页面与页表

地址结构

特定机器地址结构是一定，给定一个逻辑地址空间中的地址A，页面的大小为L，则页号P和页内地址d计算如下：

$$P = INT\left[\frac{A}{L}\right]$$

$$d = [A]\ MOD\ L$$

例： 其系统的页面大小为1KB，设A=2170B，则由上式可以求得P=2 ，d=122。

$$\begin{array}{r} -2048 \\ \hline 122 \end{array}$$

页面与页表

地址结构

例：系统页面大小为1KB，逻辑地址为2170，求页号与页内偏移量

- 页号 P=INT[2170/1024]=2
- 页内偏移量d=2170 mod 1024 =122
  - 第0页 0~1023
  - 第1页 1024~2047
  - 第2页 2048~3071
- 表示为(2,122)

页面与页表

例：L=1000B，则第0页对应0-999，第1页对应1000-1999。

设A=3456，

则P=INT[3456/1000]=3，d=[3456] mod 1000=456

故　A=3456→（3，456）

一般来说，页面尺寸应该是2的幂。这样的优点是可以省去除法，由硬件自动把地址场中的数拆成两部分来决定对应的页号和页内地址。

页面与页表

例：页的大小为1KB，则逻辑地址4101的页号、页内地址：

$1K=1024=2^{10}$（页内地址位数为10）

$4101=2^{12}+2^2+2^0$ ，逻辑地址字如下：

000100 | 0000000101

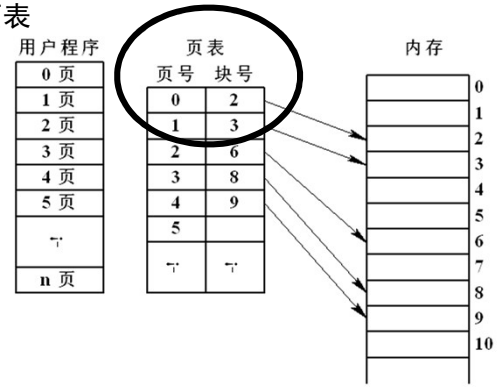
页号　　页内地址

故　A=4101→（4，5）

页面与页表

- ❑主存分配
  - ❖用户程序的任一页分配到内存中的任一物理块
  - ❖非连续的内存分配
- ❑问题
  - ❖如何管理、如何进行地址变换

页面与页表

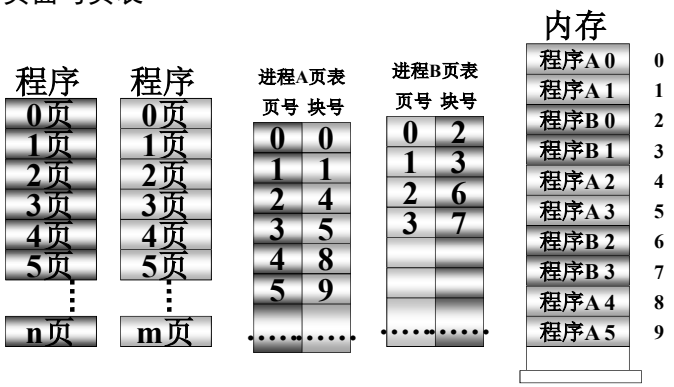


实现页号到物理块号的映射

页面与页表

- 页表
  - 列出了用户程序的逻辑地址与其在主存中的物理地址间的对应关系。
  - 一个页表中包含若干个表目，表目的自然序号对应于用户程序中的页号，表目中的块号是该页对应的物理块号。
  - 页表的每一个表目除了包含指向页框的指针外，还包括一个存取控制字段。
  - 表目也称为页描述子。

页面与页表



页面与页表

例：某虚拟存储器的用户编程空间共32个页面，每页为1KB，内存为16KB。假定某时刻一用户页表中已调入内存的页面对应的物理块号如下表：

页号	物理块号
0	5
1	10
2	4
3	7

则逻辑地址0A5C（H）所对应的物理地址为：_____

页面与页表

页号	物理块号
0	5
1	10
2	4
3	7

1. 0A5C＝0000, 1010, 0101, 1100
2. 页号为2
3. 对应块号为4，有：
4. 物理地址：0001，0010, 0101, 1100
5. 即：125C

地址变换

- 实现从逻辑地址到物理地址的转换
- 通过页表将逻辑地址中页号转换为内存中物理块号

物理地址(绝对地址)	
页面号	页内地址

虚拟地址(有效逻辑地址)	
页号	页内地址

地址变换

- 页表的实现
  - 寄存器：变换速度快、成本高，适应小型系统
  - 页表驻留在内存：速度较低、成本低，适应大系统
- 页表驻留内存

页表长度	页表始址

  - 设置页表寄存器PTR(Page Table Register)
  - 存放页表在内存中的始址和页表的长度
- 进程未执行时，页表的始址和页表长度存放在PCB中
- 调度到进程时，将数据装入页表寄存器

地址变换

页表存放在内存中，CPU每次存取一个数据要两次访问内存

第一次访问内存中的页表，从中找到该页的物理块号，将此块号与页内偏移量W拼接以形成物理地址。

第二次访问内存时，从第一步所得地址中获得所需数据（或向此地址中写入数据），并将此页号与高速缓存中的所有页码进行比较。

地址变换

- 地址变换过程
  - 把虚拟地址2500转换成页号P=2，位移量W=452
  - 如果页号大于页表大小则中断；否则继续
  - 页号2与页表起址1000运算（1000+2*20，设页描述子大小为20）得到页描述子地址为1040
  - 从页描述子中读取块号8；
  - 根据页描述子的“存取控制”判断该指令是否被允许访问内存，如果不允许，则中断；否则继续
  - 块号8与位移量452运算（8*1024+452=8644，1024为页面大小）得到物理地址8644
  - 执行LOAD操作

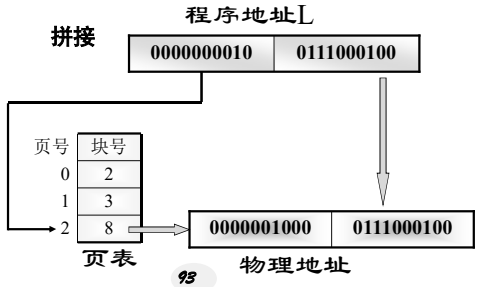
地址变换

执行 Load 1, 2500

2500 = 10 01, 1100, 0100 = 2048 + 452

第2页对应物理第8页，2500所对应的物理地址：

10, 00 00, 0000, 0000+01, 1100, 0100=8196+452=8644



地址变换

例2．设有8页的逻辑空间，每页有1024字节，它们被映射到32块的物理存储区，那么逻辑地址的有效位是__13__位，物理地址至少__15__位。

例3．在采用页式存储管理的系统中，某作业J（或某进程）的逻辑地址空间为4页（每页为2048字节），且已知该作业的页表如下，求出有效逻辑地址4965所对应的物理地址，并画出地址变换图。

页号	块号
0	1
1	3
2	5
3	7

地址变换

例4：有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、10、5块，试将虚地址7145，3412转换成内存地址。

页号	块号
0	7
1	9
2	10
3	5

解：求虚地址 3412

P＝3412 % 2048 ＝1

W＝ 3412 mod 2048 ＝1364

MR=9*2048+1364=19796

求虚地址 7145：

P＝7145 % 2048 ＝3

W＝7145 mod 2048 ＝1001

MR=5*2048+1001=11241

地址变换

例5：有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、A、5块，试将虚地址0AFEH，1ADDH转换成内存地址。

页号	块号
0	7
1	9
2	A
3	5

解：求虚地址0AFEH的物理地址：

0000 1010 1111 1110

P＝1 W＝010 1111 1110

MR＝0100 1010 1111 1110＝4AFEH

求虚地址1ADDH的物理地址：

0001 1010 1101 1101

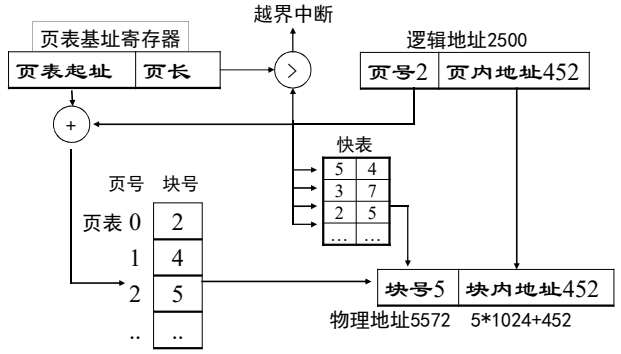
P＝3 W＝010 1101 1101

MR＝0010 1010 1101 1101＝2ADDH

地址变换

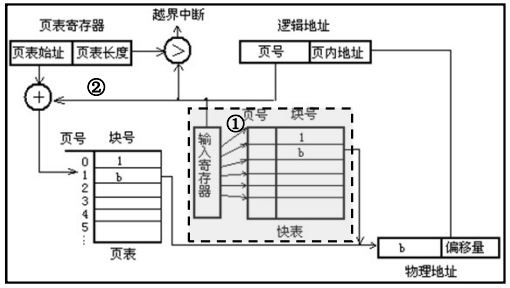
- 具有快表的地址变换机构
  - 每次CPU存取一个数据要两次访问内存
  - 在地址变换机构中增设一个具有并行查询能力的高速缓冲寄存器，又称为“联想寄存器”（Associative Memory）或“快表”，用以存放当前访问的那些页表项
  - 快表通常可存放16-512个表项，如果设计得当，命中率可达90%以上

地址变换



例4：具有快表的分页地址转换机构设每页1KB(1024)

地址变换



设检索联想存储器需20ns，访问内存需100ns

若检索联想存储器时命中：120ns，否则为220ns

命中率为80%时有效访问时间：120*80%+(1-80%)*220=140ns

地址变换

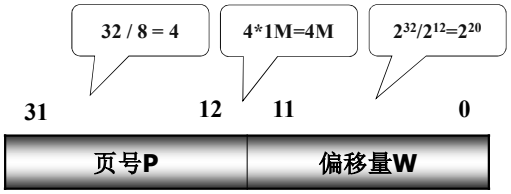
例5：有一页式系统，其页表存放在主存中：
①如果对主存的一次存取需要1.5 μs,试问实现一次页面访问的存取时间是多少?
②如果系统加有快表,平均命中率为85%,当页表项在快表中时,其查找时间忽略为0, 试问此时的存取时间是多少?

答：若页表存放在主存中，则要实现一次页面访问需两次访问主存：一次是访问页表，确定所存取页面的物理地址（称为定位）。第二次才根据该地址存取页面数据。

- 页表在主存的存取访问时间
  =1.5*2=3(μs)
- 增加快表后的存取访问时间
  =0.85*1.5+(1-0.85)*2*1.5=1.725(μs)

两级和多级页表

具有32位逻辑地址空间的分页系统，若规定页面大小为4KB，每个进程页表中的页表项可达1M个。若每个表项占用4个字节(32bit)， 进程页表要占用4 MB连续内存空间



两级和多级页表

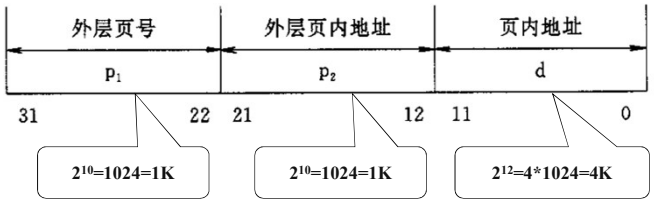
- 采用离散分配方式来解决连续大内存空间问题
- 只将当前需要的部分页表项调入内存
- 其余的页表项仍驻留在磁盘上，需要时再调入

两级和多级页表

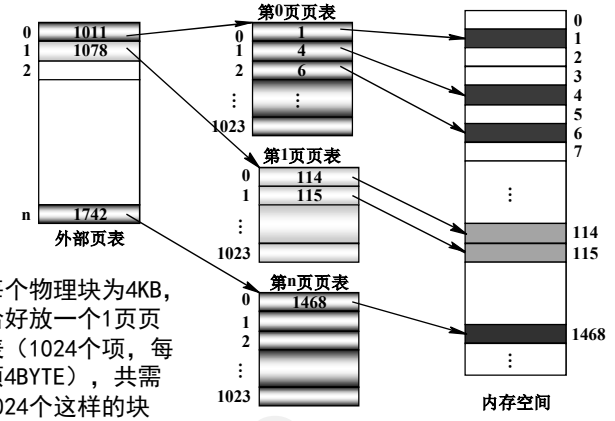
Two-Level Page Table

- 页表分页
- 离散存放页面在不同的物理块
- 为离散分配的页表建立页表

外层页表（Outer Page Table）



两级和多级页表



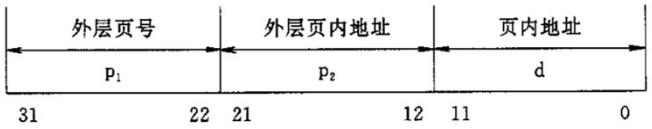
每个物理块为4KB，恰好放一个1页页表（1024个项，每项4BYTE），共需1024个这样的块

两级和多级页表

- 对于64位的机器
- 页面大小仍采用4 KB即2^{12} B
- 剩下52位仍按物理块的大小(2^{12}位)来划分页表
- 余下的42位用于外层页号
- 外层页表中可能有4096 G个页表项
- 占用16384 GB的连续内存空间

必须采用多级页表，将外层页表再进行分页，也是将各分页离散地装入到不相邻接的物理块中，再利用第2级的外层页表来映射它们之间的关系

两级和多级页表



2^{42}=4096G **2^{10}=1024** **2^{12}=4KB**

对于64位的计算机，如果要求它能支持2^{64}(=1844744 TB)规模的物理存储空间，则即使是采用三级页表结构也是难以办到的；而在当前的实际应用中也无此必要

反置页表（Inverted Page Table）

- 基本分页存储：

页表按照进程的逻辑地址顺序排序，内容为物理块号（页框号）

- 反置页表

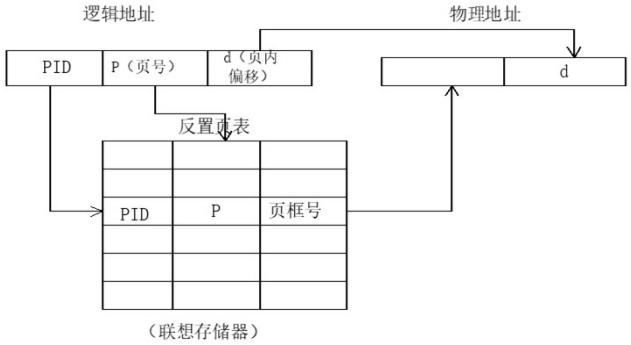
页表按照物理块号的顺序排序，内容为隶属的进程ID及其页号

- 实例：

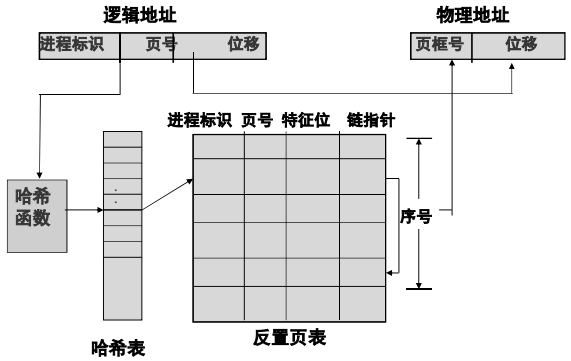
IBM AS/100、IBM RISC SYSTEM 6000等

- 反置页表进行地址变换
  - 利用进程ID和页号，检索反置页表
  - 可利用联想存储器来检索

反置页表（Inverted Page Table）



反置页表（Inverted Page Table）

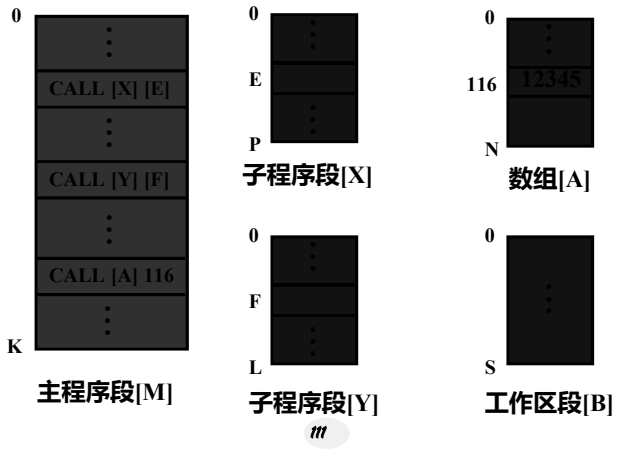


优点：

- 有效地解决了存储器的零头问题
- 提高了存储器利用率

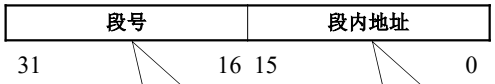
缺点：

- 采用动态地址变换机构，增加了计算机的成本
- 必须用内存存储各种表格，管理费时
- 分区间的碎片消除了，出现了页内碎片
- 作业地址空间受内存容量限制



- 分页存储管理的主要目的是为了提高内存利用率
- 分段存储管理的主要目的是为了满足用户在编程和使用上的要求
- 分段管理的主要目的
  - 方便编程
    - 用户作业通常按逻辑关系分若干个段
      - LOAD 1, [A]|<D>;
      - STORE 1,[B]|<C>;
  - 信息共享
    - 程序与数据的共享是以信息的逻辑单位为基础
  - 信息保护
  - 动态增长
  - 动态链接

1. 分段
   - ❖分段存储管理方式中，作业的地址空间被分成若干个段(segment)，每个段定义了一组逻辑信息
   - ❖分段地址中的地址具有如下结构



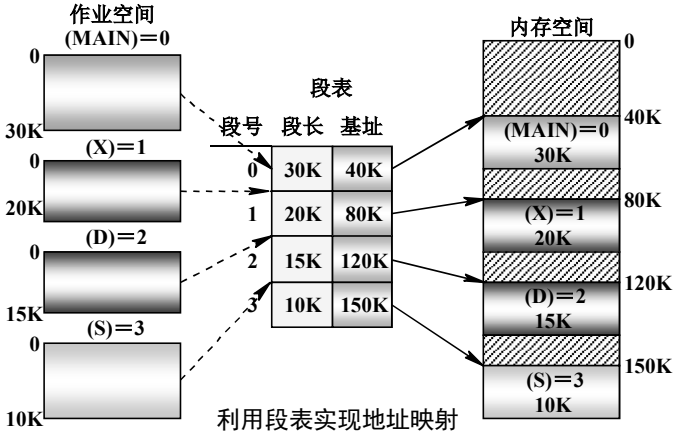
   - ❖分 $2^{16}=2^6*2^{10}=64K$ 多维 $2^{16}=2^6*2^{10}=64K$

2. 段表
   - ❖在分段式存储管理系统中，为每个分段分配一个连续的分区，而进程中的各个段可以离散地移入内存中的不同的分区中
   - ❖系统为每个进程建立一张段映射表，简称为“段表”
   - ❖每个段在段表中占一个表项，其中记录了该段在内存中的起始地址（又称为“基址”）和段的长度

❑段表

段号	段首址	段长度
0	58K	20K
1	100K	110K
2	260K	140K

记录了段号，段的首（地）址和长度之间的关系
每一个程序设置一个段表，放在内存



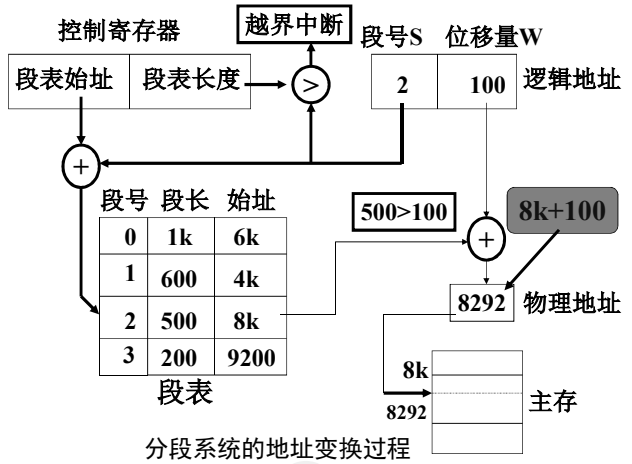
利用段表实现地址映射

3. 地址变换机构

- 系统设置一对寄存器
- 段表始址寄存器

  用于保存正在运行进程的段表的始址
- 段表长度寄存器

  用于保存正在运行进程的段表的长度



分段系统的地址变换过程

例，有一个多用户系统，可同时接纳40个用户，他们都执行一个文本编辑程序。如果文本编辑程序有160KB的代码和另外40KB的数据区，则总共需有8MB的内存空间来支持40个用户。

如果160KB的代码是可重入的，则无论在分页系统还是在分段系统中，该代码都能被共享，在内存中只需保留一份文本编辑程序的副本，此时所需的内存空间仅为：

40*40+160=1760KB，而不是8000KB。

假定每个页面的大小为4KB，那么，

160KB的代码将占用40个页面，数据区占10个页面。

信息共享

- 分段存储的一个优点是易于实现段的共享，即允许若干个进程共享一个或多个分段
- 分页系统中虽然也能实现程序和数据的共享，但远不如分段系统方便
- 可重入代码（Reentrant Code）又称为“纯代码”（Pure Code）是一种允许多个进程同时访问的代码。可重入代码是一种不允许任何进程对它进行修改的代码

分页和分段的主要区别

(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。分段的目的是为了能更好地满足用户的需要

(2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定，决定于用户所编写的程序，在对源程序进行编译时根据信息性质划分

(3) 分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址； 而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名， 又需给出段内地址

分页和分段的主要区别

	分块方式	使用	碎片	长度	目的
分页存储管理	物理分块系统需要	对程序员是不可见，使用简单	每个进程只有一个内部碎片，大小不超过1页	固定	提高内存的利用率
分段存储管理	逻辑分块，大小与信息块有关，满足用户需要	对程序员可见，使用方便，但难度大	每个进程会产生多个外部碎片	不确定	便于信息保护与共享，方便用户

信息共享

- 分段存储的一个优点是易于实现段的共享，即允许若干个进程共享一个或多个分段
- 分页系统中虽然也能实现程序和数据的共享，但远不如分段系统方便
- 可重入代码（Reentrant Code）又称为“纯代码”（Pure Code）是一种允许多个进程同时访问的代码。可重入代码是一种不允许任何进程对它进行修改的代码

例，有一个多用户系统，可同时接纳40个用户，他们都执行一个文本编辑程序。如果文本编辑程序有160KB的代码和另外40KB的数据区，则总共需有8MB的内存空间来支持40个用户。

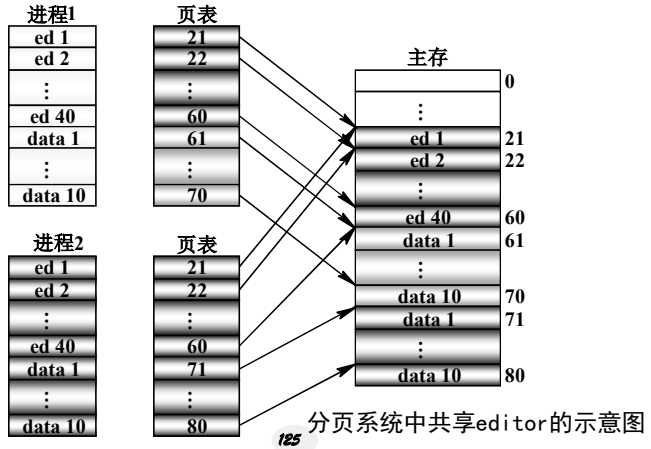
如果160KB的代码是可重入的，则无论在分页系统还是在分段系统中，该代码都能被共享，在内存中只需保留一份文本编辑程序的副本，此时所需的内存空间仅为：

40*40+160=1760KB，而不是8000KB。

假定每个页面的大小为4KB，那么，

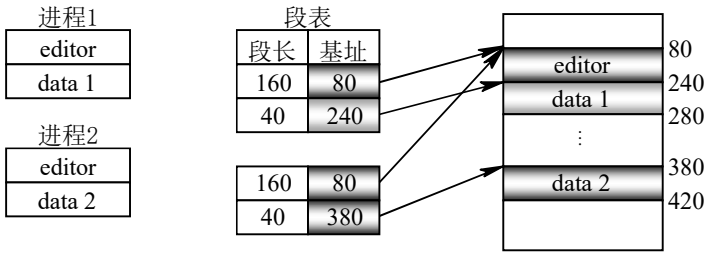
160KB的代码将占用40个页面，数据区占10个页面。

分页系统中对程序和数据的共享



分页系统中共享editor的示意图

分段系统中对程序和数据的共享



分段系统中共享editor的示意图

优点：

- 便于程序模块化处理和处理变化的数据结构。
- 便于共享分段。
- 便于动态链接。

缺点：

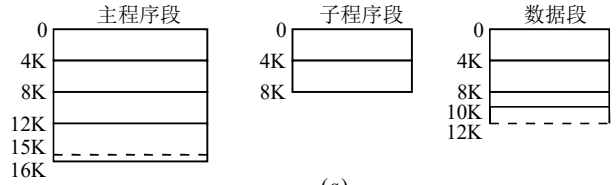
- 地址变换费时，需硬件支持，并且管理表格要提供附加的存储空间。
- 为满足段的动态增长和减少碎片，要用拼接技术。
- 段长不定，管理困难。
- 段长受内存可用区的限制。

段页式存储管理方式

1. 基本原理

- ❖是分段和分页原理的结合
- ❖将用户程序分成若干个段，再把每一段分成若干个页，并为每一段赋予一个段名
- ❖段页式管理中，地址机构由段号、段内页号及页内地址三部分所组成

段页式存储管理方式



(*a*)

地址结构

段号(S)	段内页号(P)	页内地址(W)

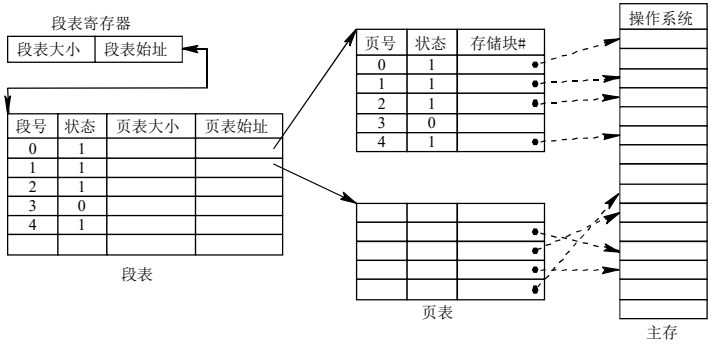
(*b*)

作业地址空间和地址结构

段页式存储管理方式

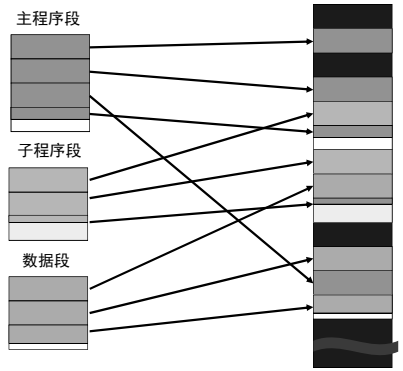
- 在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需同时配置段表和页表。
- 由于允许将一个段中的页进行离散分配，因而使段表的内容略有变化：它不再是段的内存始址和段长，而是页表始址和页表长度。

段页式存储管理方式



利用段表和页表实现地址映射

段页式存储管理方式



段页式存储管理方式

2. 地址变换过程

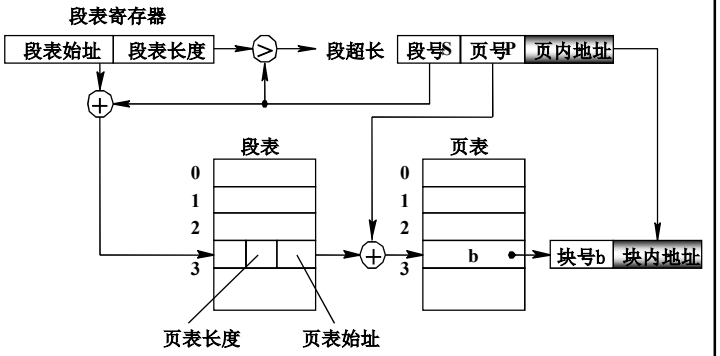
在段页式系统中，需三次访问内存。

第一次访问，是访问内存中的段表。

第二次访问，是访问内存中的页表。

第三次访问，才是真正从第二次访问所得的地址中，取出指令或数据。

段页式存储管理方式



段页式系统中的地址变换机构