内存管理

笔记源文件：Markdown，长图，PDF，HTML

1. 内存管理基础

1.1. 概述

1.1.1. 内存管理的功能

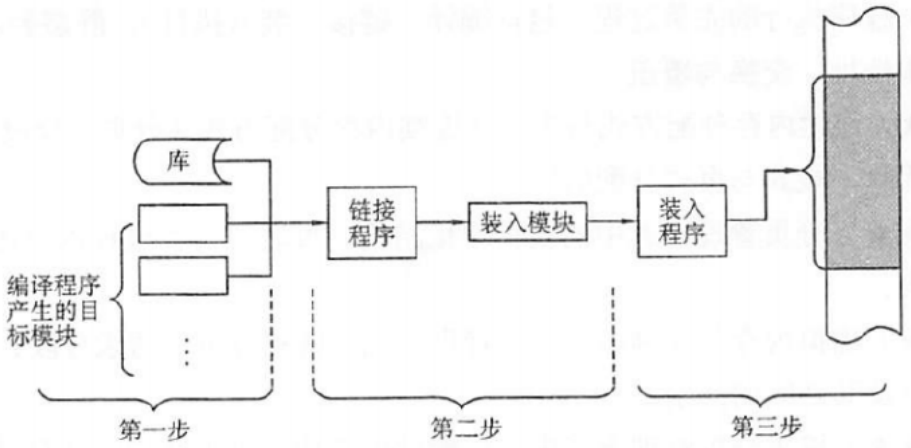
1. 分配回收内存：记住内存使用情况，内存分配，回收用户释放的内存
2. 地址变换：程序的逻辑地址 ⟺ 内存的物理地址
3. 扩充内容：基于逻辑层面的虚存技术
4. 存储保护：使各道作业在内存中独立运行，且不破坏系统程序

1.1.2. 其他背景知识

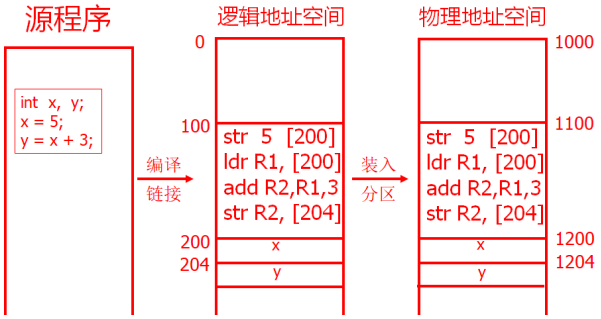
1. 程序执行的必要条件：输入内存，放入一个进程
2. 输入队列：磁盘上等待进入内存并执行的进程集合
3. 程序的加载：将程序代码/数据从磁盘读入内存，并准备开始执行
4. 动态加载：一个程序只有在调用时才会加载

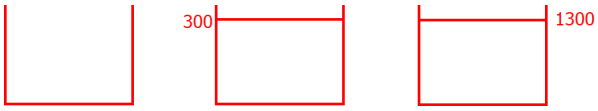
1.2. 程序的加载

1.2.1. 概览：编译-链接-装入



1.2.2. 地址变换





1 符号地址(对源程序而言)：编程中用变量/数据名指定的位置

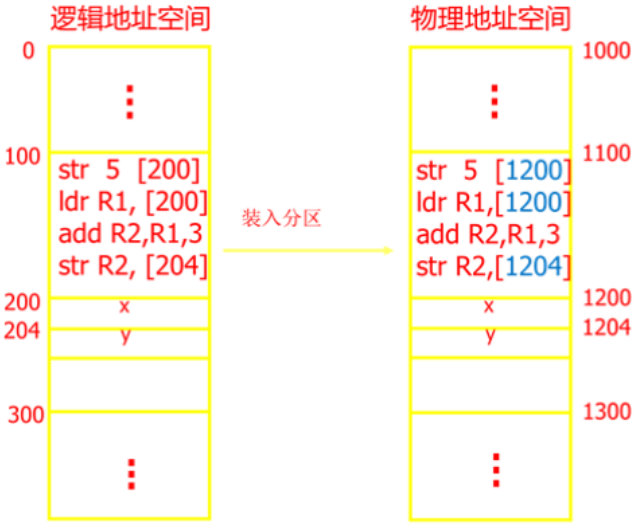
2 相对/虚拟/逻辑地址(对目标程序而言)：源程序编译后CPU生成的目标代码地址，从0开始

3 物理/绝对地址(对可执行程序而言)：程序加载后在内存的实际地址

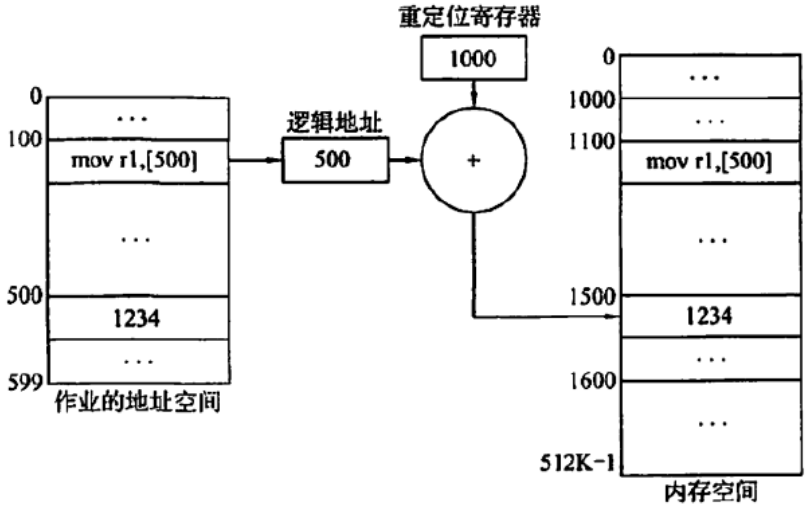
PS—逻辑地址对用户可见，物理地址对用户透明

1.2.3. 重定位：虚拟地址→物理地址

1 静态重定位：在装入时，逻辑地址全部转为绝对的地址，执行过程中不变



2 动态重定位：起始地址放入重定位寄存器，执行过程中，在CPU访问内存前，把要访问的数据/程序地址转化为内存地址(硬件实现：重定位寄存器+逻辑地址->物理地址)



1.2.3. 三种链接

1 静态链接：全部链接完再运行

2 装入时动态链接：边装入边链接

3 动态链接：一部分先运行，等需要某些块了再链接+装入，节省了内存(不需要的块就不装入了)

1.2.4. 三种装入：对应编译/装入/执行时绑定到内存地址

1️⃣绝对装入：编译时生成绝对代码(含物理地址)，决定了要装内存哪

2️⃣可重定位装入：装入时完成地址变换(物理地址=基地址+逻辑地址)，实现容易但程序地址要连续

3️⃣动态运行装入：程序运行时在内存中位移，程序运行某指令/访问某数据后才装入，地址可不连续

1.3. 内存保护：防止一个作业破坏另一个

1️⃣界限寄存器法

1. 上下界寄存器法：让上/下界寄存器分别存储作业的开始/结束地址，如果作业运行时访问的内存超出这个上下界就立马中断
2. 基址+限长寄存器法：分别存放作业的起始地址+作业长度，限长寄存器与相对地址进行比较超出就立马中断

2️⃣存储保护键方法：若干分区中每个分区有很多存储块，给每个存储块分配一个单独的保护键(锁)。进入系统的作业被赋予一个保护键(钥匙)，然后检查二者保护键是否匹配，不匹配就立即中断

1.4. 覆盖&交换技术

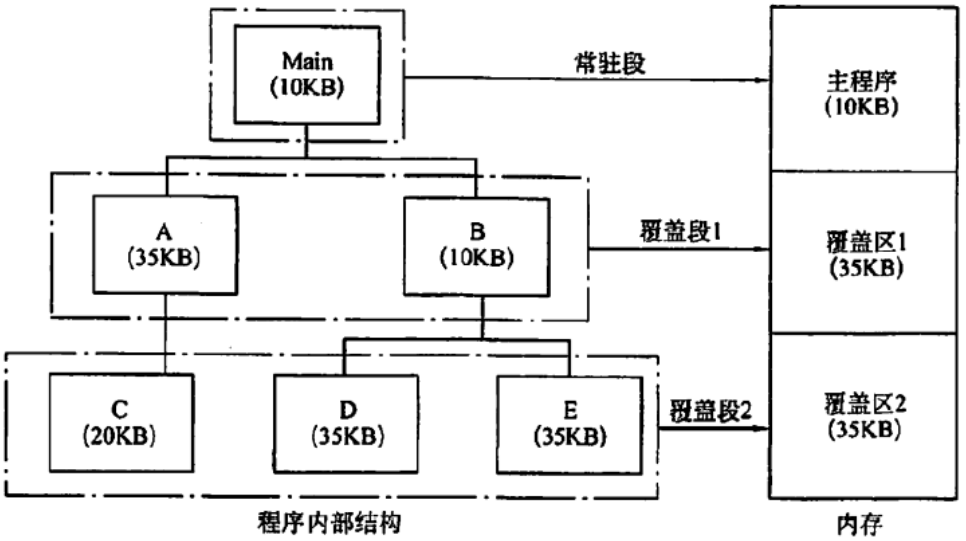
1.4.1. 覆盖技术：把大程序分为一系列覆盖

见于早期小内存OS

1️⃣基本概念

1. 覆盖：程序中相对独立的程序单位
2. 覆盖段：程序执行时不需要同时装入内存的一组覆盖
3. 覆盖区：与覆盖段一一对应的存储区域，将覆盖段分配到一个覆盖区。覆盖区的大小=覆盖段中最大覆盖的大小

2️⃣覆盖实例：一般都是由程序员提供覆盖结构



1.4.2. 内存交换(扩展)

1️⃣概述：把暂不用的程序/数据从内存移到外存(或移回来)

2️⃣兼容分时系统：内存中只有一完整作业，时间片用完后OS就把他丢到外存，放外存中另一作业来

3️⃣与覆盖技术的比较

1. 覆盖要求程序员给出程序段之间的覆盖结构，交换不需要
2. 交换发生在不同进程/作业之间，覆盖发生于用一进程/作业

4️⃣交换技术的特点

1. 从主存交换到什么设备：快速+空间够+直接访问，比如SSD
2. 从主存交换到哪里：交换空间(aka备份区，大小固定，独立于文件系统，可直接存取)
3. 什么进程被交换：优先级低的被交换出去(进程要空闲即休眠/不占CPU)，高的被交换进来
4. 转移时间：交换耗时，应该远低于进程执行时长
5. 合适交换：内存爆满

5️⃣挂起与交换：挂起进程会被丢到外存

1.5. 连续分配: 程序装入连续内存

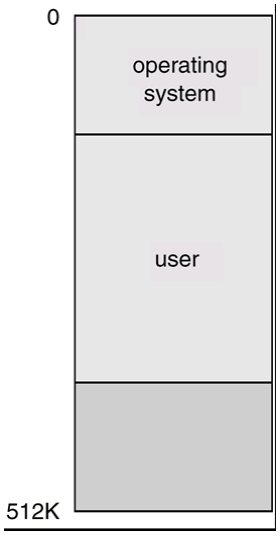
1.5.0. 内部/外部碎片

1️⃣内部碎片：已分给作业但不能被利用的内存(某个作业占用内存中没填满的部分)

2️⃣外部碎片：由于太小而无法分配给作业的内存碎片(不同作业之间剩余的内存)

1.5.1. 单一连续分配：单任务OS

1️⃣内存结构：低地址给OS，高地址给用户，再其余的浪费掉



2️⃣缺点：会产生内部碎片

1.5.2. 多分区分配：固定/静态分区(早期)

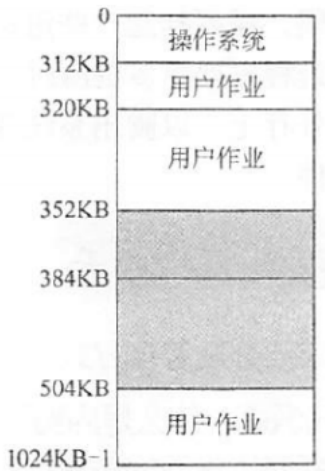
1 概述：OS分区+多个用户分区，用户分区大小在装入前预先确定，每个分区装一个程序

PS1：分区大小可以相等也可以不等

PS2：会产生内部碎片且分区有限，但是易于实现开销小

2 分区说明表：记录可分配的区号(及其大小/起止)，程序装入内存时检索一次分区表找出满足要求的空闲分区

分区号	大小	起始地址	状态
1	8KB	312KB	已分配
2	32KB	320KB	已分配
3	32KB	352KB	未分配
4	120KB	384KB	未分配
5	520KB	504KB	已分配



1.5.3. 多分区分配：动态分区

1.5.3.1. 概述

1 含义：作业进入主存时，再建立分区

2 分区大小=作业大小：

1. 作业进入主存时查找大于等于作业大小的空闲分区
2. 等于的话直接分配
3. 大于的话分成两半——和作业一样大的(占用)+剩余部分(空闲)

3 存在外碎片

1.5.3.2. 分区分配中的数据结构

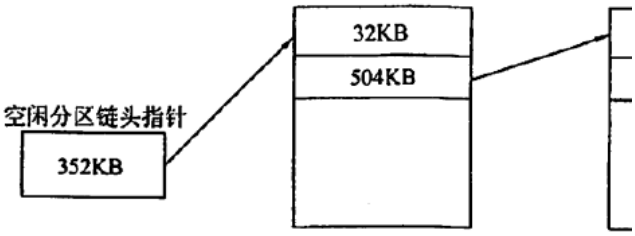
1 空闲分区表：登记空闲的分区，一个分区对应一项，一项中有分区号/大小/起始/状态

PS：内存从低到高——分区号从小到大

分区号	大小	起始地址	状态
1	32KB	352KB	空闲
2	…	…	空表目
3	520KB	504KB	空闲
4	…		空表目
5	…		…

2 空闲分区链：用指针链接所有空闲分区

PS：每空闲分区起始位，存放空闲分区大小+指向下一空闲分区指针



1.5.3.3. 分区分配算法：怎样把空闲区分给作业

1 首次适应算法：

1. 含义：在空闲分区链中从头按顺序找，选找到的第一个大小合适的空闲区
2. 优点：分配/释放速度快
3. 缺点是：低地址空闲块会越分越小，导致之后的查找成本大

2 下次适应算法：

1. 基于首次适应的改进：空闲分区链改为循环链表，每次从上次停留地方开始找，
2. 缺点：全局都难有大的空闲区

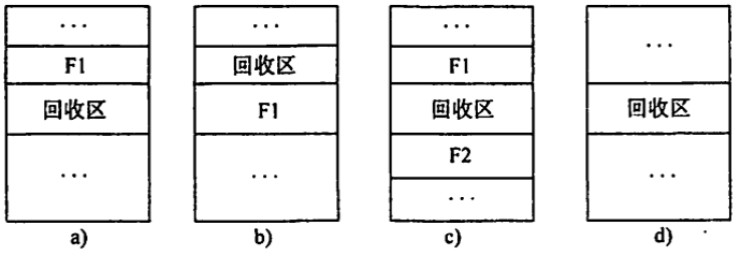
3 最佳适应算法：

1. 含义：空闲区从小到大排列，每次从小到大一个个试，试到差不多大小的块便分给进程
2. 缺点：会产生很多难以利用的碎片，除非用碎片拼接(aka紧凑)

4 最差适应算法：

1. 含义：空闲分区按照容量大到小排列，最大空闲分区优先分配
2. 缺点：大作业来到时，大空闲区已经被优先分配掉了

1.5.3.4. 分区回收



1 回收区上/下邻接空闲区：合为一空闲区，首地址为顶上那个

2 回收区上下邻接空闲区：合为一大空闲区，从链表中删除下面的分区

3 回收区上下无空闲区：独立为空闲区，加入空闲分区链表Z

1.5.3.4. 如何处理碎片?

1 核心问题：作业装入的内存要连续，但内存碎片总和总是大于作业大小

2 拼接技术(紧缩)：

1. 含义：向一个方向移动已分配的作业，碎片就此紧缩在另一端
2. 何时紧缩？：某个分区回收时(频率高)，找不到足够空间时(频率低但是实现复杂)

3️⃣动态分区分配+拼接→动态重定位分区分配：空闲区不够，但碎片总和够大时实行分区

1.6. 非连续分配概述：程序装入非连续内存

1️⃣核心：把程序打散存在主存里，然后用索引将其联系起来

2️⃣分类

```
├── 分区大小不定：分段存储管理
├── 分区大小固定：分页存储管理
                ├── 运行时把作业所有页装入内存：基本分页存储管理
                ├── 运行时把作业部分页装入内存：请求分页存储管理(见后虚拟存储)
```

1.7. 基本分页存储管理

1.7.1. 分页以&页表&地址变换

1.7.1.1. 简单分页/纯分页原理

1️⃣页&块

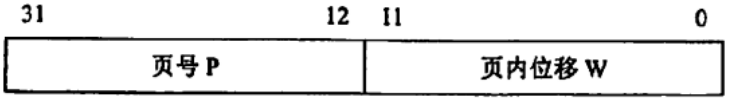
1. 页：作业中等大的逻辑内存空间
2. 块/帧：主存中，大小固定的物理内存空间，大小上块和页相等的
3. 页框：主存中大小与页一样大的块

2️⃣作业调度：以块为单位，将作业任一页丢到主存任一块，所有页要一次调入(块不够就等待)

3️⃣页/块大小的决定：

1. 过大会导致碎片太多，过小会导致页表过长(占用内存)+页面进出主存效率低
2. 通常为2幂大小，512B-4KB

4️⃣逻辑地址结构：[页号] [页内位移]，如下有$2^{20}=1M$页+每页$2^{12}=4K$大小

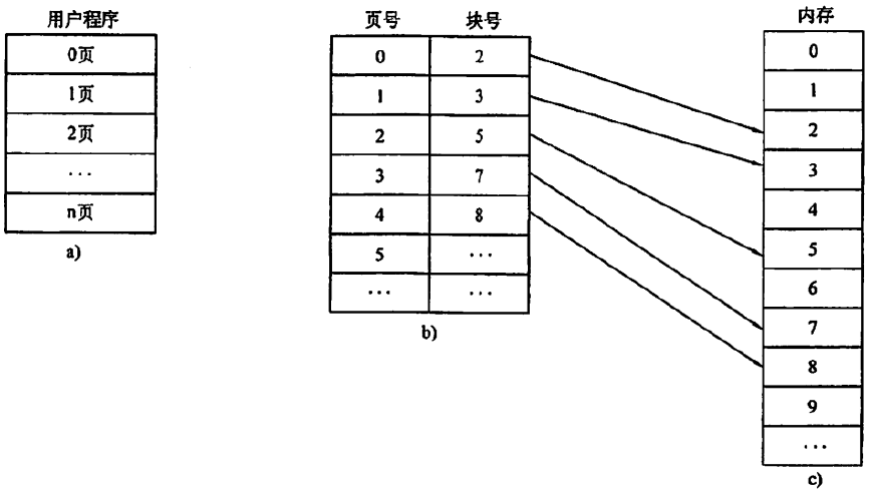


这两个参数都由CPU生成，存在如下关系

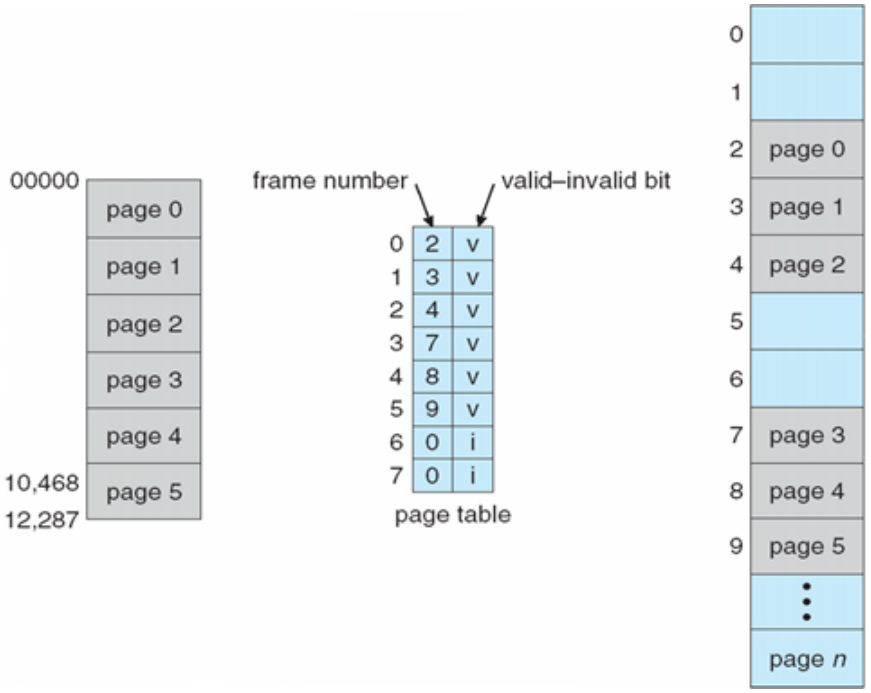
```
(int)[逻辑地址]/[页面大小]=[页号]
(int)[逻辑地址]%[页面大小]=[页内位移]
```

1.7.1.2. 页表: (用户程序的页)页号 ⟺ 块号(主存物理块)

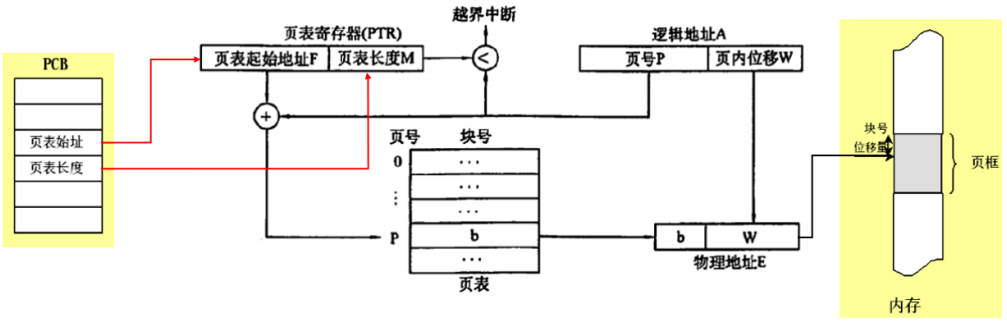
1️⃣概览：页表存在内存中，如图例子。页表项=页号+块号+其他(存在位/修改/访问权限)



2 有效-无效位：在页表表项中，有效表示有关页在进程的逻辑地址空间中



1.7.1.3. 基本地址变换机构: 基于硬件



1 页表寄存器(PTR)：**页表基址寄存器**(主存中的页表起始地址)+**页表限长寄存器**(页表长度)

2 逻辑地址→物理地址

1. 计算出页号和页内位移

```
(int)[逻辑地址]/[页面大小]=[页号]
(int)[逻辑地址]%[页面大小]=[页内位移]
```

2. 若页号>页表长度则越界终中断

3. 页表起始地址+偏移量(页号*页表每项长度)→得到地址，从该地址取出物理块号

4. 物理块号+页内位移(=块内位移)→物理地址

3 弊端：存取数据/变量要访问两次主存(第一次访问页表确定物理地址+第二次用物理地址访问指令or数据)，快表可以解决这一问题

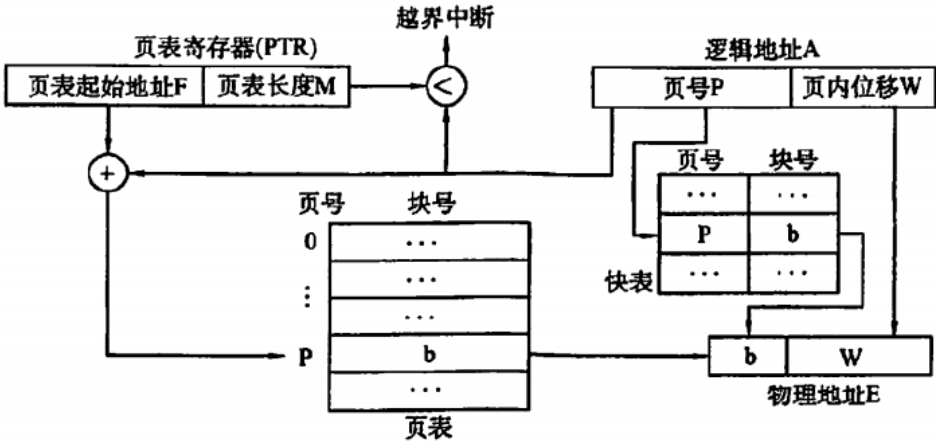
1.7.2. 其他类型的页表

1.7.2.1. 具有快表的地址变换机构

1 快表：储存作业当前/近期访问的页表项，类似于Cache

2 联想寄存器TLB：存储块表的寄存器

2 改进后逻辑地址→物理地址



1. 求出页号+页内位移

2. 先把页号和快表中的对比，对上了就得到对应块号，与页内位移组合成物理地址

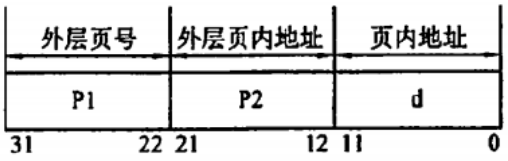
3. 否则就和原来一样

1.7.2.2. 两级页表

1 页表大小计算：页表长度(页表项目数)*页表每项大小(块号位数)

2 背景：$\text{页表长} = 2^{\text{页号位数}}$，页表长度爆炸式增长→占空间太大

3 两级页表逻辑地址



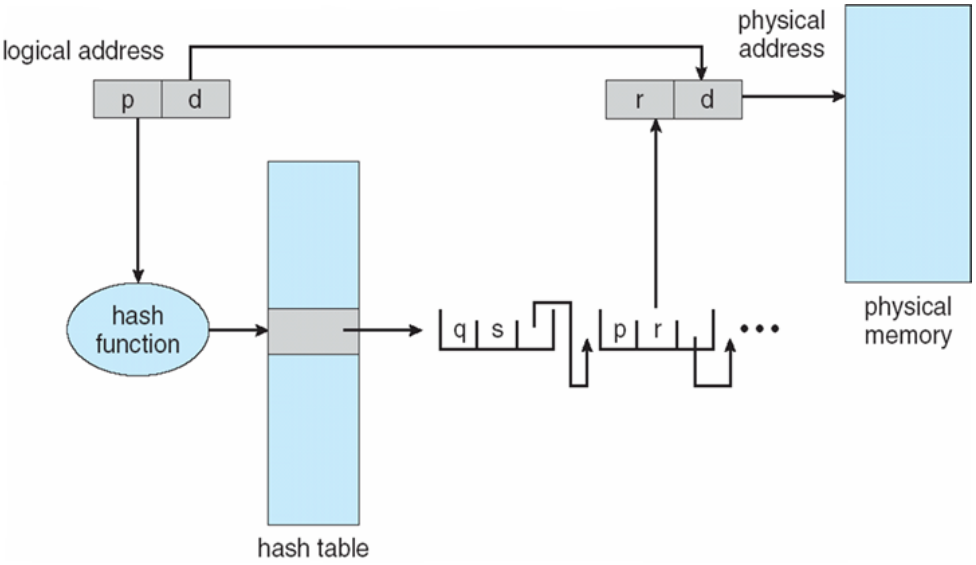
4 地址变换过程：

$$\text{外层页号} \xrightarrow{\text{在外部页找到}} \text{二级页表首地址} \xrightarrow[\text{得到}]{+\text{外层页内地址}} \text{物理块(首)地址} \xrightarrow[\text{得到}]{+\text{页(块)内地址}} \text{物理地址}$$

➕多级页表：逻辑和两级页表一样，多见于64位系统，缺点是地址变换耗费资源

1.7.2.3. HASH页表

通过哈希表来完成逻辑地址到物理地址的映射



1.7.2.4. 反转页表

1 背景：传统上每个进程设一张页表，浪费内存

2 解决方案：页表按物理内存块组织(而非进程)，反转页表中**包含了内存中所有物理块地址→其逻辑的地址的映射**

3 关于表项：

1. 内存中每一块在表中占一项
2. 每项包含：进程逻辑页号(存储在物理内存中)+进程标识
3. 使用HASH表来搜索表项

4 特点：减少了页表占用空间，但查找时间增加

1.7.3. 页的共享与保护

1 分页中共享的实现：共享用户地址空间中的页指向相同的物理块

2 分页中的保护：

1. 地址越界保护：比较地址变换机构中的页表长度和逻辑地址中的页号
2. 访问控制：程序访问一个页面时，OS检查该操作是否有权限(只读/只写/可执行)，无权就中断

1.7.4. 基本分页存储管理的利弊

1 利：内存利用率高+离散分配+便于存储访问控制+无外部碎片

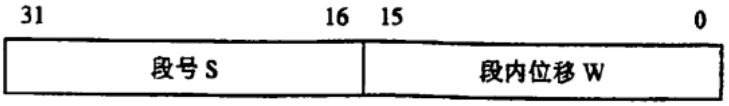
2 弊：需要硬件支持(如快表)+内存访问效率低+共享困难(对比分段)+有内部碎片

1.8. 基本分段存储管理

页是信息的储存单位，段是信息的逻辑单位

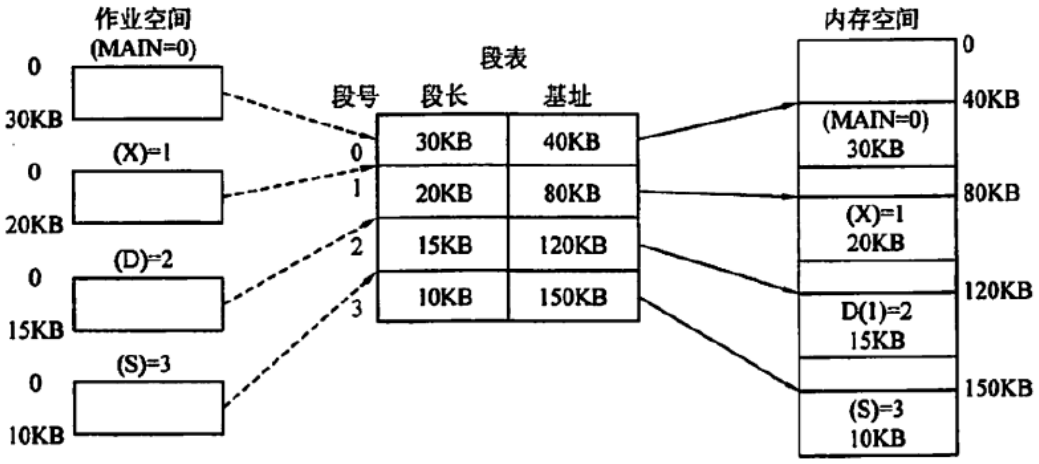
1.8.1. 分段存储原理

1. 作业&内存分段：每个分段都有段名，每段地址从0开始，每段的地址连续(段间可不连续)
2. 分段存储的逻辑地址结构：段数=$2^{段号位数}$，段长=$2^{段内位移位数}$

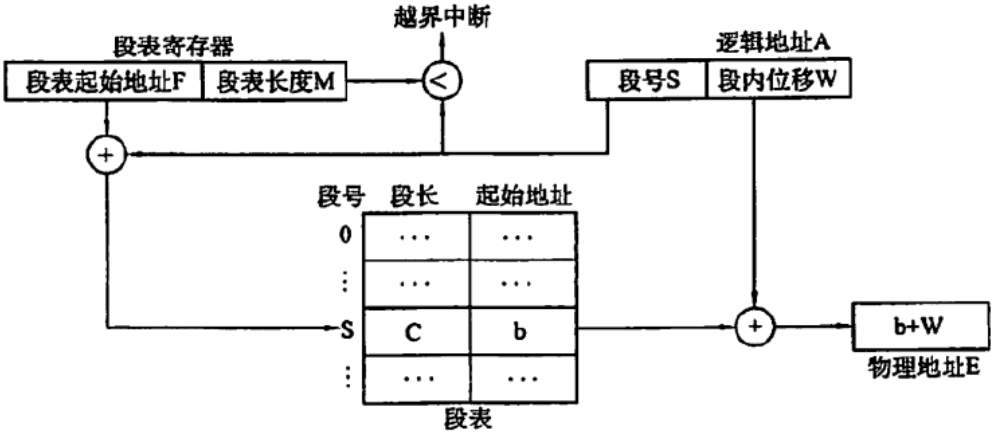


3. 分页与分段的区别
   1. 分段：段号由用户定义，每段大小和含义不同，地址是二维的(用户给出段号+用户给出偏移)
   2. 分页：页号由OS生成，页号无特殊含义，地址是一维的(OS给出页号+用户给出偏移)

1.8.2. 段表: [段号]+[段长]+[段在内存的起始地址]



1.8.3. 逻辑地址到物理地址的转换



1. 先对比段号AND段表长度，若段号超出，则中断
2. 算出段表表项位置=段起始地址+段号*段表表项长度
3. 读取表项内容，若段长<段内位移，则中断(动态增长段除外)
4. 根据表项中的段起始地址+段内位移→物理地址

1.8.4. 段的共享与保护

1️⃣共享：多个作业段表中相应表项指向被共享物理段的同一物理副本

2️⃣保护的含义：

1. 一个作业在共享段读数据时，防止另一个作业修改内容
2. 不可修改的数据/代码共享，可修改的不共享

PS1：纯代码/可重入代码：不可修改的代码

PS2：共享的一个规则：不可修改代码与不可修改数据可共享，但可修改代码与可修改数据不可共享

PS3：大多系统中，程序都被分为代码区/数据区

3️⃣保护的方式：地址越界保护(段号>段表长就中断，偏移>段长就中断)+访问控制保护(读写权限)

1.8.5. 基本分段的特点

1️⃣划为多模块：如代码段/数据段/共享段，分别编写/编译/保护，进行共享

1. 共享：把需要共享的代码/数据放在一段
2. 保护：段信息独立，保护段就是保护信息

2️⃣碎片：没内碎片，外碎片可通过内存紧缩消除

2️⃣缺点：需硬件支持，段最大尺寸受主存限制

1.8.6. 分页分段对比

	分 页	**分 段**
目的	提高内存利用率	更好满足用户需要
单位划分	**页是信息的物理单位，页大小固定(由OS确定)**	段是**信息的逻辑单位**，其含义完整。段长不固定(用户确定)
作业地址空间	一维(页内偏移)	二维(段名+段内偏移)
内存分配	以页为单位离散分配，无外碎片	以段为单位离散分配，有外碎片(需要紧缩)

1.9. 基本段页式存储管理方式

1.9.1. 分段分页&分块

1️⃣作业分段分页：先给作业的地址空间逻辑分段(每段有段号)，再给每段内分页

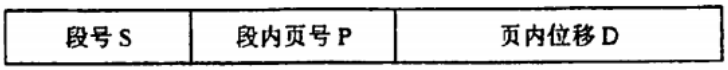
2️⃣主存的分块(和分页管理一样)：分为如讴歌和页大小一样的块

1.9.2. 段表与页表

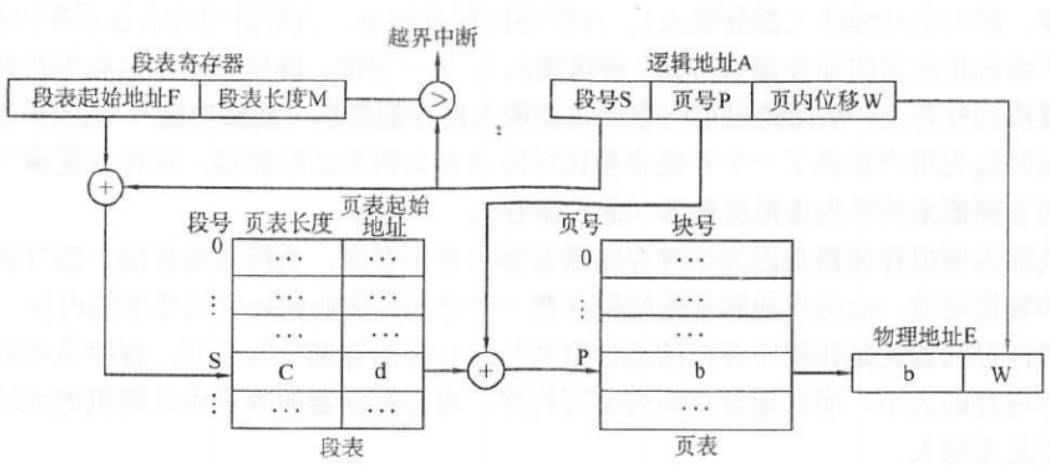
1. 段表：每进程拥有一张段表，结构为——段号+对应页表始址+页表长
2. 页表：每个段表有一张页表，结构为——页号+块号

1.9.3. 地址与地址变换

1. 段页管理的虚拟地址结构



2. 从虚拟地址到逻辑地址的变换



1. 先对比段号和段表长度，如果段号大就中断
2. 段起始地址与段号相加，得到所需段表项(表项：段号+页表长度+页表起始地址)
3. 如果页号>页表长度就中断
4. 在通过页表起始地址+页号得到页表项目地址，通过该地址取出块号
5. 块号+页内位移就是物理地址了，然后访问内存

1.9.4. 特点

1. 内部碎片太多：页式平均一个程序有半页碎片，段页式平均一段就有半页碎片(一个程序很多段)
2. 为了获取一条指令或数据，需**三次访问内存**

2. 虚拟内存管理

2.0. Pre

2.0.1. 引入虚存的原因

1. 一次性(全装入后才执行)+驻留性(运行完后作业才脱离内存)，对于大作业难以满足要求
2. 程序执行时有些代码用的较少(错误处理部分)，而程序IO又耗费世家

2.0.1. 局部性原理

1. 时间局部性：同一指令/数据短时间内被高频执行/访问**(这可以归因于大量的循环操作)**
2. 空间局部性：某条指令被访问后，其附近的指令有极大概率也被访问

2.1. 虚存基本概念

2.1.1. 虚存的定义

1. 部分装入：一部分装入内存，一部分放外存
2. 请求调入：执行时访问信息不在内存时，再要求OS将其调入内存
3. 置换功能：OS将内存中暂时不用的内容置换到外存
4. 虚拟内存：逻辑上扩充内存容量的系统

2.1.2. 虚存的特征

1. 离散性(最基本)：程序被打散存储在内存中
2. 多次性(最重要)：一个作业被分成多次调入内存
3. 交换性：作业在运行时可以不断从内存中换入换出
4. 虚拟性：用户使用的内存远大于实际内存

2.1.3. 其他有关虚存

1. 虚存实现手段：请求分页，请求分段
2. 硬件机构：外存外存都要足够大，有中断机构(访问内容不在内存时就中断程序)，地址变换机构，段/页表
3. 好处：较小内存执行较大程序，并发性提高，比覆盖技术编成更简单

2.2. 请求分页存储管理方式

2.2.1. 请求分页原理：局部性原理

1. 请求分页=基本分页+请求调页功能+页面置换功能
2. 页面调入策列
    1. 预调页策略：进程首次调入时，把预计很快要被访问的页，主动调入内存
    2. 请求调页策略：进程运行时，将需要的页调入内存(通过**页面置换**把暂时不用的页置换出去)

PS: 内存中每一页都在外存上保留一份副本

3. 从何处调入页面：硬盘分为文件区+对换区，对换区IO快于文件区
    1. 对换区足够大：全从对换区调入所需页面，运行前就把进程必要块放到对换区
    2. 对换区不够大：不会被修改的文件都从文件区调入(减少IO)，需要修改的放到调换区
    3. UNIX方式：未运行的页放在文件区，运行过又被换出的放对换区

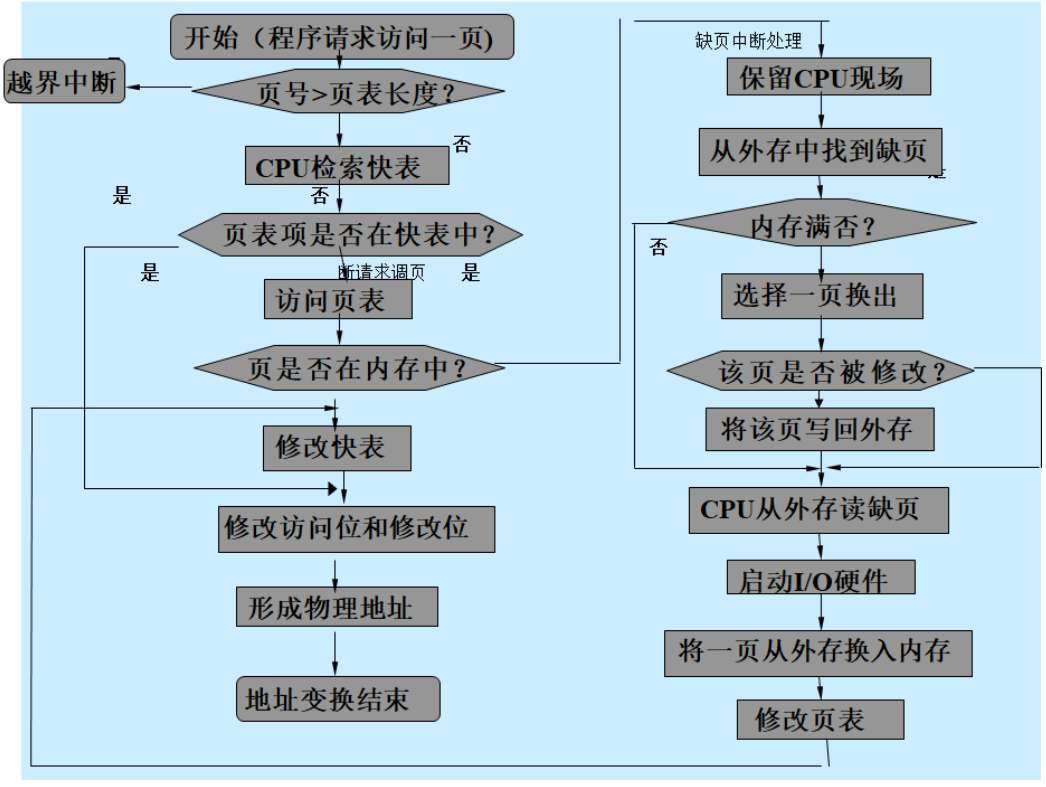
2.2.2. 页表结构

页号	物理块号	状态位	访问字段	修改位	外存地址

1. 页号与物理块号：与之前相同，虚拟地址页<—映射—>物理地址快
2. 状态位(存在位)：判断页是否在主存，不在时触发**缺页中断**
3. 访问字段：记录局部性参数——页面一段时间内访问次数/最近多久未被访问
4. 修改位：记录页面调入内存后是否被修改
    1. CPU以写方式访问页时，修改位被设置
    2. 页在内存中没被修改，页换出时就不再写到外存(节省读写次数)
5. 外存地址：页面在外存的地址，调入该页时使用

2.2.3. 缺页中断与地址变换

1. 如果访问的页在内存中，地址变换与分页存储管理相同
2. 访问页不在内存，进程就请求OS从外存调入所需页



➕ 缺页中断与一般中断的区别

1. 缺页/一般中断，发生在指令执行期间/结束后
2. 一指令可触发多次缺页中断，如指令的两个操作数都在外存
3. 缺页/一般中断，返回到**本指令开始/下条指令**重新执行

2.2.4. 请求分页的性能

1. 优点：离散存储碎片少，提供虚拟存储主存利用率高
2. 缺点：硬件复杂，会抖动，最后一页仍有内部碎片
3. 主动动作：处理缺页中断，从磁盘读页(开销最大)，重新开始执行被中断程序
4. 缺页率=访问访问失败次数/进程访问内存总次数

2.3. 页面置换(淘汰)算法

2.3.1. 最佳置换(OPT)

1. 页面号引用串：进程执行时，按时间顺序引用的页面序列
2. OPT：已知页面号引用串的情况下(其实不可能的)，淘汰以后不再使用/最迟被使用的页
3. 特点：缺页率最低，但无法实现(仅作为理论最低缺页率的参考)

2.3.2. 先进先出(FIFO): 最简单

1. 队列结构：指针指向最先进入的页，每次淘汰指针指向的页
2. 缺点：会产生Belady异常，源于最早进来(也是最早淘汰)的页使用最频繁

2.3.3. 最近最少使用(LRU)

1. 原理：
   1. 最近最少使用的页$\xrightarrow[\text{性能接近}]{\text{近似的等同于}}$往后最迟被使用的页(OPT)
   2. 基于假设：刚访问的页倾向于马上又被访问
2. 含义：淘汰最久不被使用的页
3. 硬件支持：每个页表项有一个计数器，记录访问情况，作为被淘汰依据

2.3.4. 最不常用/最常用置换(LFU/MFU)

1. 含义：选择当前为止访问最少次/最多次的页面淘汰
2. 思想：淘汰最少使用的→访问多的倾向于再被访问；淘汰最多使用的→访问少的页往往最新调入
3. 实现方式：设置一个计数器，被访问一次该页的计数器就+1

2.3.5. LRU近似算法

2.3.5.1. 时钟置换(CLOCK)/最近未使用(NRU)/二次机会

LRU+FIFO的折中

1. 数据结构：每页设一个访问位(被访问后置1)，内存中所有页构成一个**循环链表**
2. 访问页在链表中：访问位改为1
3. 访问页不在链表中：
   1. 指针指向上次被淘汰页的下一页

2. 指针顺序&循环遍历**循环链表**

2.1. 指针指向页的访问位=1时，就把该访问位清零，然后继续遍历

2.2. 指针指向页的访问位=0时，就淘汰该位然后访问位变1

4️⃣ 示例：

内存及控制信息			输入串	指针移动情况及帧替换信息	是否缺页
内存	访问位	指针	7	指针所指的位置恰好有访问位为 0 的	
6	0			于是就淘汰这个帧，指针下移	√
4	0	←			
3	1				
内存	访问位	指针	4	内存中没有 4，需要找到一个帧放入 4	
6	0			指针所指的位置的访问位为 1	
7	1			将其变成 0，再下移（回到开头）	
3	1	←			√
内存	访问位	指针		指针所指的位置恰好有访问位为 0 的	
6	0	←		于是就淘汰这个帧，指针下移	
7	1				
3	0				
内存	访问位	指针	3	内存中有 3，于是 3 所在帧的访问位变为 1	
4	1			指针不变	
7	1	←			
3	0				
内存	访问位	指针	6	内存中没有 6，需要找到一个帧放入 6	
4	1			指针所指的位置的访问位为 1	
7	1	←		将其变成 0，再下移	
3	1				
内存	访问位	指针		指针所指的位置的访问位为 1	
4	1			将其变成 0，再下移（回到开头）	
7	0				
3	1	←			√
内存	访问位	指针		指针所指的位置的访问位为 1	
4	1	←		将其变成 0，再下移	
7	0				
3	0				
内存	访问位	指针		指针所指的位置恰好有访问位为 0 的	
4	0			于是就淘汰这个帧，指针下移	
7	0	←			
3	0				

2.3.5.2. 改进时钟(CLOCK)/增强的二次访问

1️⃣ 改进的核心：增加了修改位(被修改了置1)，访问位同为0时优先淘汰没被修改的位(IO代价更小)

2️⃣ 算法步骤：进程启动将所有(访问位=0, 修改位=0)

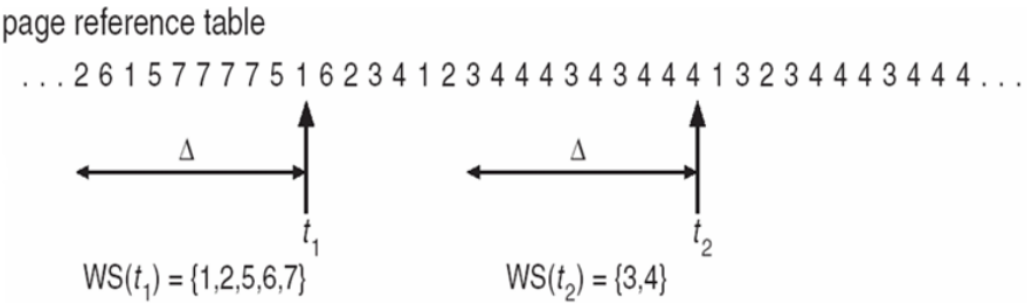
1. 从指针位置开始首次循环遍历
2. 先试图找到第一个(**访问位=0**, **修改位=0**)页替换，若没找到⬇️
3. 再试图找到第一个(**访问位=0**, **修改位=1**)页替换，所扫描过之处皆置**访问位=0**
4. 如还没找到，就重复上述过程(重复下去一定能找到)

2.3.7. 页面缓冲(PBA)

1. 按照FIFO算法选择被置换页
2. 对那些要被换出的页面，为其在内存中建立两个链表(缓冲)
   1. 对于被换出的未修改的未修改的页，丢到空闲页链表尾
   2. 对于被换出的已修改的未修改的页，丢到已修改页链表尾
3. 对于缓冲中的页，在未来一段时间内如果要访问他们，可以快速响应(免去IO)
4. 直到修改页链表到达一定规模后，再将他们一同IO到磁盘

2.4. 工作集理论：基于局部性原理

1. 目的：解决抖动，提高CPU利用率
2. 原理：预知程序在特定时间内要访问的页面(活跃页面)并提前加载它们
3. 基本概念
   1. 工作集：最近n次内存访问的页面集合，或者说落入工作集窗口的页面集合
   2. 工作集窗口(Δ)：对于给定的访问序列选取定长的区间，其大小选定很重要，Δ过小则不能包含整个局部，Δ过大则可能包含多个局部



4. 工作集模型：
   1. OS跟踪每个进程的工作集(如何跟踪是难点)，为其分配大于其工作集的物理块数
   2. 若还有空闲物理块，启动其他进程
   3. 若所有进程的工作集之和>可用物理块总数，OS会暂停&换出一个进程，释放的物理块给其他进程

2.5. 页面分配策略：如何给进程一定空闲页

2.5.1. 最少页数

1. 含义：保证进程运行的最小物理块数
2. 取值依据：指令格式，功能，寻址方式

➕某进程物理块数<最少页数：进程频繁缺页，进而崩溃

2.5.2. 分配&替换策略

1. 分配(固定分配)：平均(每个进程块数一样)+按比例+按优先级
2. 替换：全局(进程之间可争躲页，块数会增加)+局部(进程只能从自己那获得页，块数不变)

➕全局置换时，无法控制页错误率，系统吞吐率会更高

2.5.3. 组合

	固定分配：每个进程分得相同个物理块	可变分配：OS给进程动态分配物理块
局部置换：进程只在自己内存块中置换页面	各进程物理块数相同，进程间不争夺块，块少进程频繁换页/块多进程浪费内存(核心在于进程块数分配)	先给每进程一定物理块，进程互相枪内存。增加频繁换页进程块数，减少缺页率过低进程块数
全局置换：进程需要更多内存时，可以替换掉其他进程的内存页	无	OS维护一空闲物理块队列，进程每次缺页就从队里取一个，取完了就去抢其他进程的块

2.7. 抖动与缺页率

2.7.1. Belady异常

1. 含义：FIFO算法中缺页率会随内存中块数增加而增加
2. 成因：FIFO算法的策略与内存动态特征违背，总会换掉进程要访问的页
3. PS：LRU算法和最佳置换算法永远不会出现Belady异

2.7.2. 抖动/颠簸现象

1. 含义：某一页面刚被换出又被访问(重新调入)，频繁调入调出，CPU利用率低下
2. 成因：进程分到的物理块太少
3. 解决方案：
    1. 全局置换会造成颠簸，局部置换能限制颠簸
    2. 根本上要给进程足够的物理块

2.7.3. 缺页率

作业有n页，系统给作业分了m页

如果作业运行时要访问A次页面，当所访问的页不在内存中时，需要将该页调入内存F次

则定义

1. 缺页率：f=F/A
2. 命中率：1-f
3. 控制缺页频率：缺页率太低/高，回收/分给一些进程的页框

2.8. 请求分段存储管理系统

1. 将当前需要的若干段装入主存便可运行，访问分段不在主存中时再调入，将不用分段也置换出去
2. 段表表项结构

段号	段长	内存始址	访问字段	修改位	状态位	外存地址

PS. 总结

PS.1. 三种离散分配方式

对比及联系	内存管理方式		
	分页存储管理	分段存储管理	段页式存储管理
有无外部碎片	无	有	无
有无内部碎片	有	无	有
优点	内存利用率高，基本解决了内存零头问题	段拥有逻辑意义，便于共享、保护和动态链接	兼有两者的优点
缺点	页缺乏逻辑意义，不能很好地满足用户	内存利用率不高，难以找到连续的空闲区放入整段	多访问一次内存

PS.2. 几种内存管理

比较的方面	单一连续分配	分区		分页		基本分段	基本段页式
		固定分区	可变分区	基本分页	请求分页		
内存块的分配	连续	连续		离散		离散	离散
适用环境	单道	多道		多道		多道	多道
地址维数	一维	一维		一维		二维	二维
是否需要全部程序段在内存	是	是		是	否	是	是
扩展内存	交换	交换		交换	虚拟存储器	交换	交换
内存分配单位	整个内存的用户可用区	分区		页		段	页
地址重定位	静态	静态	动态	动态	动态	动态	动态
重定位机构	装入程序	装入程序	重定位寄存器	页表 页表控制寄存器 加法器		段表 段表控制寄存器 加法器	段表 页表 段表控制寄存器 加法器
信息共享	不能	不能		可以，但限制多		可以	可以