# Lecture6 虚拟存储（管理）技术

## 大纲

1. ﻿硬件机制（地址转换）、页表/页表项、页错误处理
2. ﻿各种软件策略
3. ﻿读取策略
4. 放置策略(placement)
5. 置换策略(replacement)
6. 驻留集策略
7. 清除策略
8. 装载控制策略
9. ﻿内存映射文件机制

## 虚拟内存管理的目标

1. ﻿透明性（Transparency）

﻿运行程序/用户不感知内存虚拟化存在

1. ﻿效率（Efficiency）

﻿需要硬件支持以提高速度

1. ﻿保护（Protection）

﻿﻿使进程与进程之间、进程与操作系统隔离，确保进程不受其他进程影响，也不会影响操作系统

## 虚拟页式存储管理

1. 背景知识与相关术语
2. ﻿存储体系
3. ﻿由操作系统协调各存储器的使用，虚拟内存系统建立在存储体系的硬件基础上
4. ﻿目的

* 使虚拟内存速度尽量快，与CPU取指速度相匹配
* 使虚拟内存容量尽量大，能装下当前运行的程序与数据

1. ﻿相关术语辨识
2. ﻿虚拟内存

* ﻿把物理内存与磁盘结合起来使用，得到一个容量很大的“内存”，即虚存
* ﻿程序引用内存所使用的地址与内存物理地址是不同的，可被自动转换成物理地址
* ﻿虚存大小受计算机系统寻址机制（如32/64位）和可用磁盘容量的限制

1. 虚拟地址空间

* ﻿分配给进程的虚拟内存

1. 虚拟地址

* ﻿虚拟内存中某一位置的地址，该位置可以被访问，仿佛它是（实际）内存的一部分

1. 虚拟存储技术

* ﻿当进程运行时，先将其一部分装入物理内存，另一部分暂时保存在磁盘上。当要执行的指令或访问的数据不在物理内存时，由操作系统自动完成将它们从磁盘调入物理内存的工作
* 现代操作系统在进程运行时完全不将其装入物理内存，而是先建立内存映射（进程地址空间与磁盘之间）

1. 基本思想: ﻿操作系统中的资源转换技术——以CPU时间和磁盘空间换取物理内存空间
2. 装载程序到物理内存时，不是装入全部页面，而是装入几个甚至零个页面（只定义好进程地址空间而不实际分配物理页）
3. 如果进程执行时需要的页面不在内存（引发Page Fault），则动态装入所需页面（页面调度，调页）
4. 需要时，将内存中暂时不用的一些页面交换到磁盘，以便获得更多的内存空间
5. 调页机制
6. 两种调页方式
7. 请求调页（demand paging）
8. 预先调页（prepaging）

现代操作系统常常结合两者，通过Page Fault实现请求调页，同时一次性多读入几个页面到物理内存，实现预先调页（利用局部性原理）

1. 设计思想
2. 将进程的虚拟地址空间与物理地址空间分离
3. 3个重要策略（Coffman &Denning）
4. 取页策略（fetch policy）

系统何时把页面载入内存（如上面的两种调页方式）

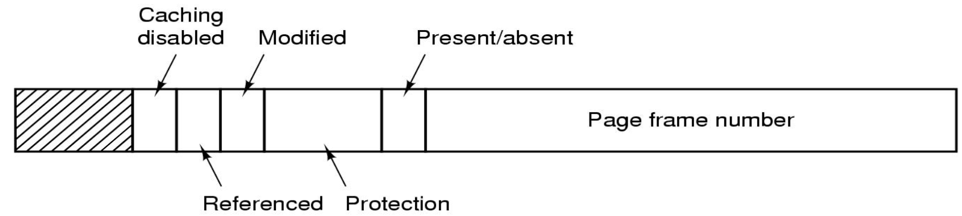
1. 放置策略（placement policy）

系统把页面放在何处

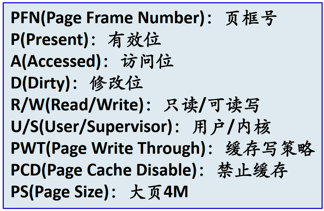
1. 置换策略（replacement policy）

执行放置操作时发生页框不足时， 如何选择从内存里“删除”的其他页框

1. 设计与实现时要解决的问题
2. 页表表项（PTE）的设计
3. 页框号、有效位、访问位、修改位、保护位等

* 页框号（内存块号、物理页面号、页帧号）
* 有效位（驻留位、中断位）：表示该页是在内存还是在磁盘（Valid/Present）
* 访问位：引用位（Referenced、 Accessed）
* 修改位：查看此页是否在内存中被修改过（Dirty/Modified）
* 保护位：读/写/执行（Protection）

1. x86页目录项和页表项

* 页目录项 PDE (Page Directory Entry)
* 页表项 PTE (Page Table Entry)

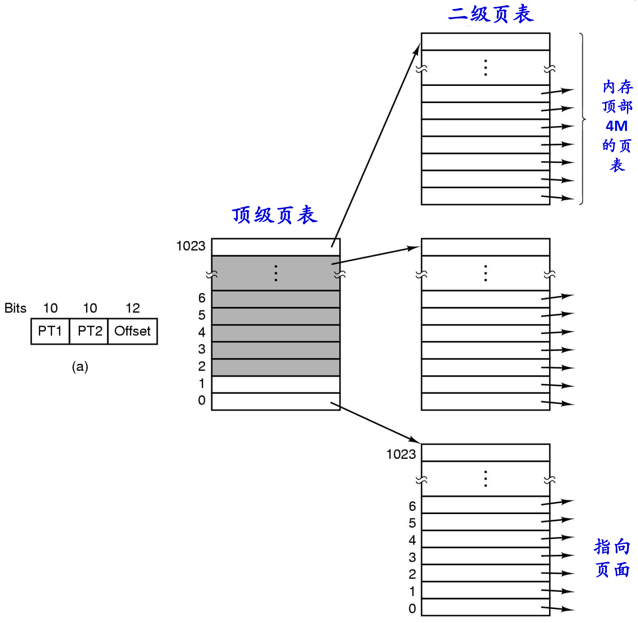
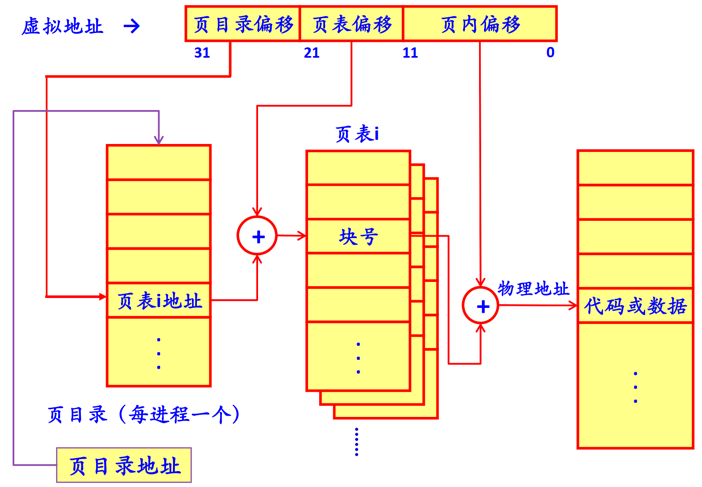
1. 如何处理页表巨大的问题？**﻿多级页表**
2. ﻿32位虚拟地址空间的页表规模

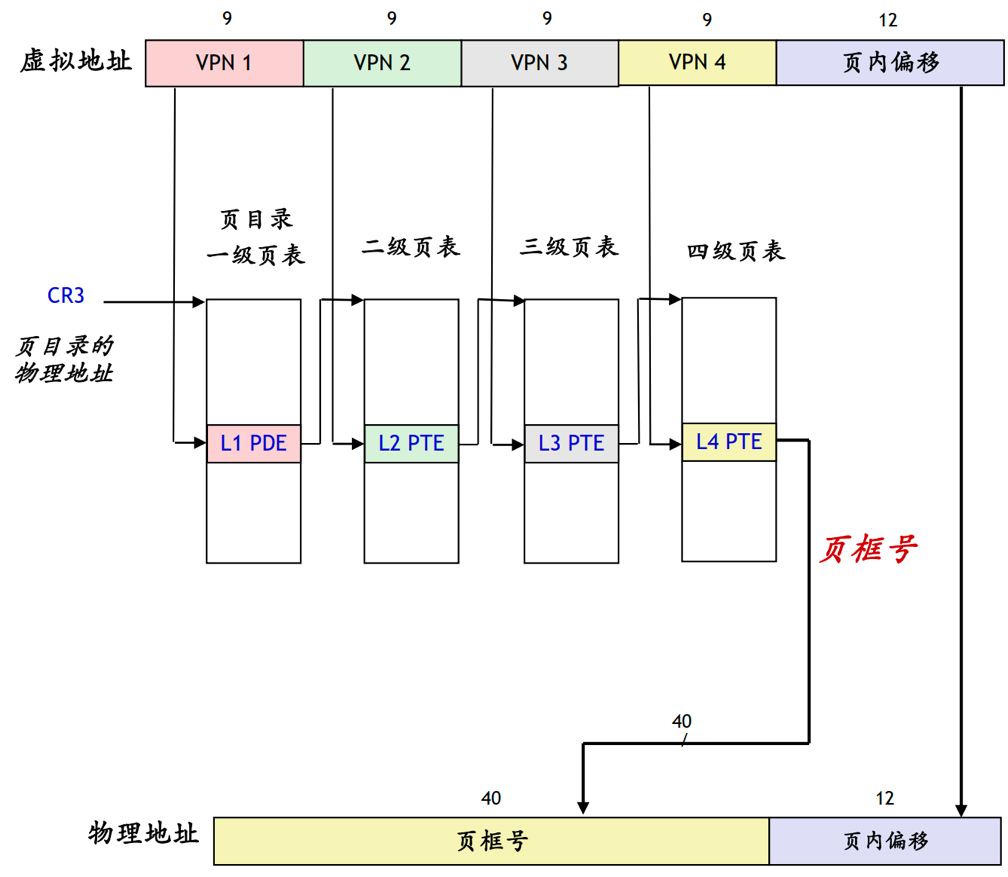
* ﻿页面大小为 4K，页表项大小为4字节，用户地址空间为2G = 231，则一个进程有219页；其页表需要占512页（注意计算过程）
* ﻿ps. 64位虚拟地址空间一张页表有33554432 GB

1. 页表在物理内存中的存储

* ﻿一个进程的页表的各页在内存中若不连续存放，则需要引入地址索引→页目录（一级页表） 离散化存储，一个页表只占一页
* ﻿注意：页表本身也映射到内核的虚存（进程运行时，页表在物理内存中，部分页表映射到内核的虚拟内存）

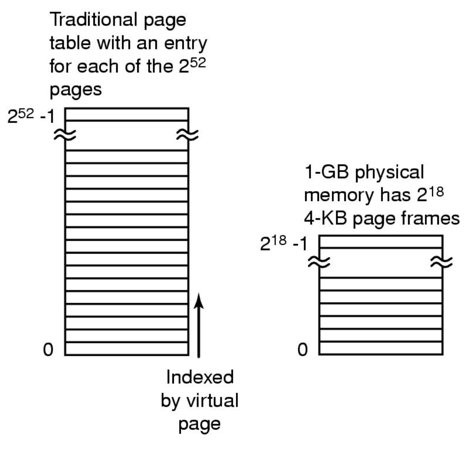
1. 两级页表

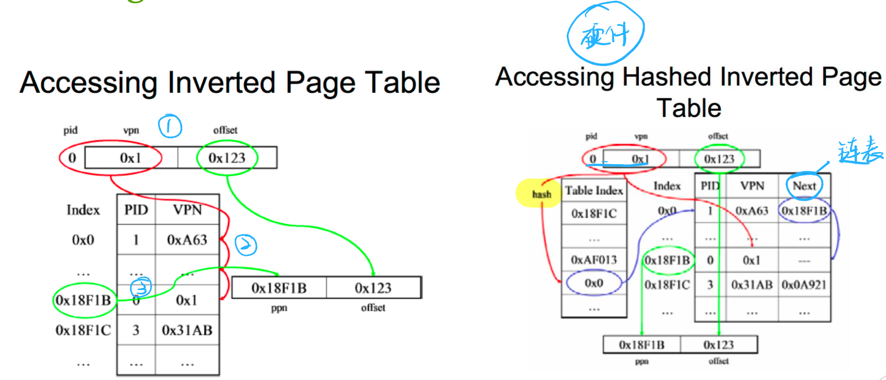
* 设计思想：数组→树
* ﻿二级页表结构及地址映射

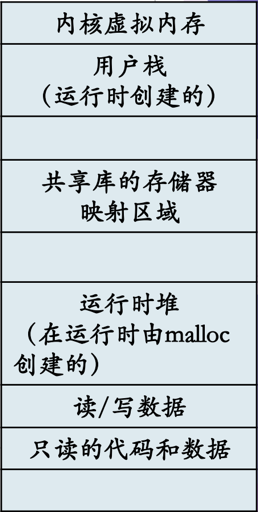
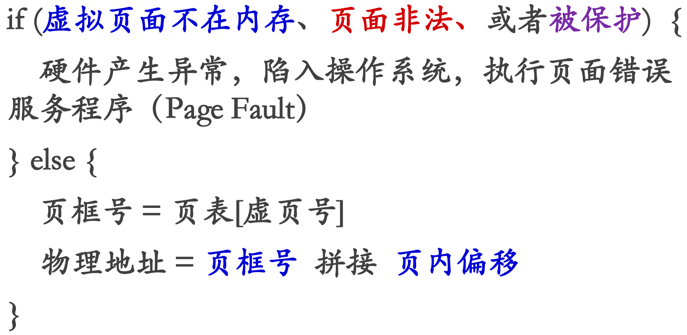
1. ﻿Core i7 四级页表结构（﻿虚拟地址空间248）

* 定义好一页的大小就可以计算出页表可以划分为几级（假定PTE大小和地址空间大小已知）

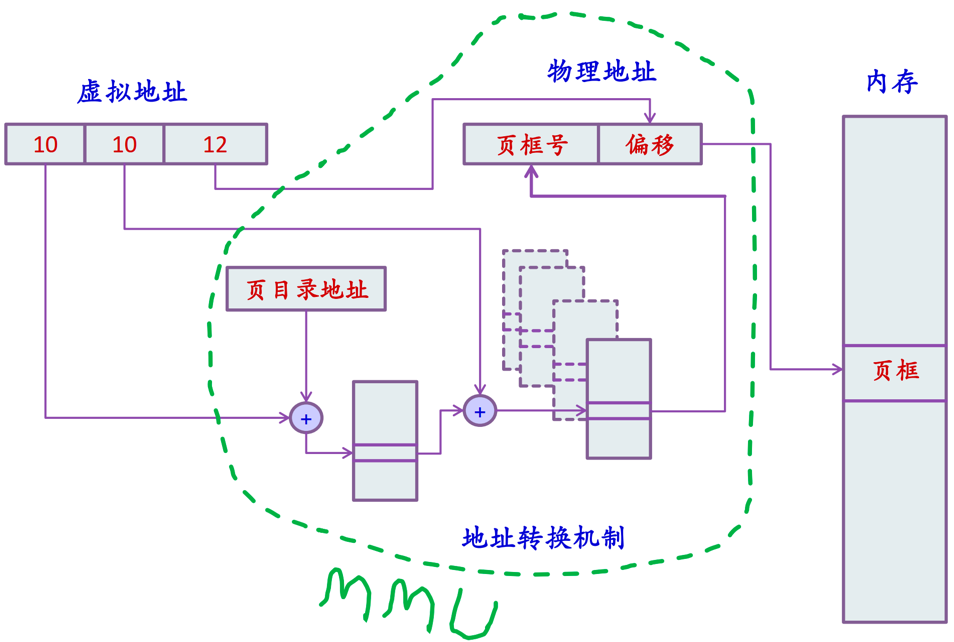
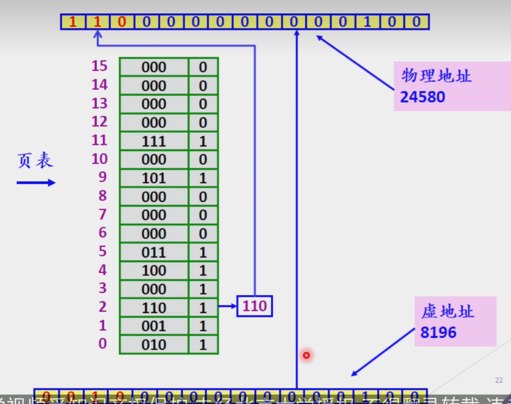
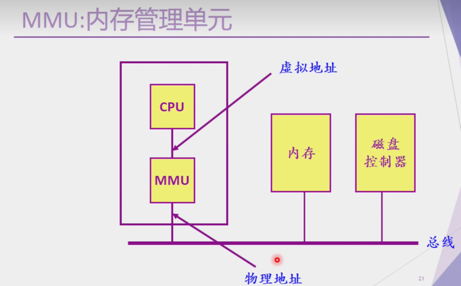
(212 bytes/page) / (8 bytes/PTE) = 29 PTE/page

1. ﻿反转(反置、反向)页表 用于节省页表占据的物理内存空间

* ﻿一般的地址转换
* ﻿从虚拟地址空间出发：虚拟地址 → 查页表 → 得到页框号→ 形成物理地址
* ﻿每个进程一张页表
* ﻿反转页表的地址转换
* ﻿从物理地址空间出发，系统建立一张页表
* ﻿页表项记录进程i的某虚拟地址(虚页号)与页框号的映射关系（pid + 虚拟页号作为索引，可能是一对多映射，即共享页）
* 示例（需要硬件支持）
* 小结
* ﻿PowerPC、UltraSPARC和IA-64等体系结构采用
* ﻿将虚拟页号部分与进程pid映射到一个散列值；﻿散列值指向一个反转页表；需要拉链解决冲突问题（？）
* ﻿反转页表大小与实际内存大小成固定比例，与进程个数无关（故只有一个，无需进行页表的切换）

1. 地址重定位（地址转换）
2. ﻿逻辑过程（硬件机制）

* 页面非法指访问未被映射到物理页的虚拟页
* 被保护指没有访问权限

1. 实际过程
2. 操作系统对存储的抽象：（虚拟）地址空间
3. ﻿﻿如何加快地址映射速度，以改善系统性能？**快表(TLB)**
4. ﻿问题

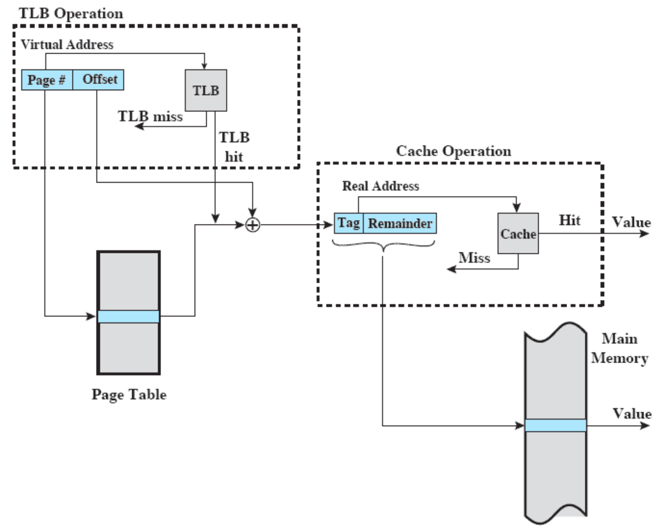
* ﻿页表 → 两次或两次以上的内存访问
* k级页表要访问k+1次物理内存才能完成对物理页的索引（k次页表+1次页框）
* CPU的指令处理速度与内存指令的访问速度差异大， CPU的速度得不到充分利用

1. 原理：﻿程序访问的局部性原理
2. 简介

* ﻿﻿TLB: Translation Look-aside Buffer，相联存储器（associative memory）
* 特点
* ﻿按内容并行查找
* 保存正在运行进程的页表的子集(部分表项)
* ﻿工作原理：采用联想映射技术按内容同时查找
* ﻿作用：﻿加快地址映射速度，改善系统性能

1. 一些问题

* ﻿快表的大小、位置
* 很小，在MMU模块中
* ﻿快表的置换问题
* 和页面置换类似，有多种策略
* ﻿快表TLB的刷新问题
* ﻿进程切换， TLB刷新，TLB未命中（冷不命中），性能下降（属于进程切换的间接开销）
* ﻿PCID（Process Cpntest IDentifier）/ASID（Address Space IDentifier）功能

1. 工作流程

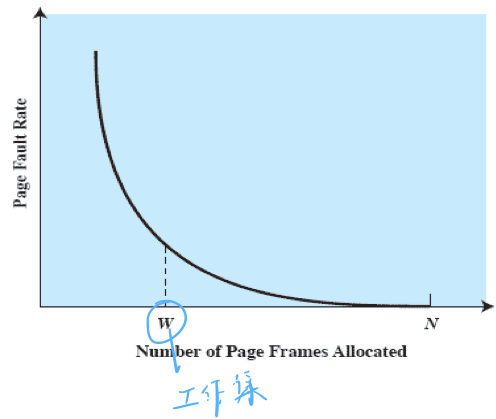
软件只负责定义进程地址空间到磁盘的映射（如填写PCB中的相关 数据结构）和创建进程的页表

1. 缺页异常
2. 简介

* 是﻿一种Page Fault，Page Fault包括页错误、页面错误、页故障、页失效等情况
* ﻿在地址映射过程中，硬件检查页表时发现所要访问的页不在内存（PTE有效位为0），则产生缺页异常

1. **处理**

* ﻿操作系统执行缺页异常处理程序：获得磁盘地址**（PCB中包含该信息）**，启动磁盘（I/O，DMA），将该页调入内存（考虑预取策略）
* ﻿如果内存中有空闲页框，则分配一页，将新调入页装入内存，并修改页表中相应页表项的驻留位及相应的页框号
* ﻿若内存中没有空闲页框，则要置换某一页（同样需要修改页表项）；若该页在内存期间被修改过（检查PTE的dirty位），则要将其写回磁盘（策略：尽量选没被修改过的页进行置换）

1. 驻留集管理
2. 问题背景：过去设备物理内存紧张，需要考虑﻿驻留集大小的设定，即给﻿每个进程分配多少页框？
3. 策略

* ﻿固定分配策略
* ﻿进程创建时确定，一个进程分配固定数量的物理页，如果在进程运行过程中物理页不够用了，即使物理内存有空闲空间也不多分配，而是置换原来已分配的物理页
* ﻿可以根据进程类型（交互、批处理、应用类）或者基于程序员或系统管理员的需要来确定
* ﻿可变分配策略：﻿根据缺页率评估进程局部性表现
* ﻿缺页率高→增加分配的页框数
* ﻿缺页率低→减少分配的页框数
* 缺点：开销比较大

1. 置换策略（见下：页面置换算法）
2. 清除策略
3. 加载控制

## 页面置换

1. ﻿简介
2. ﻿置换范围与两种置换策略

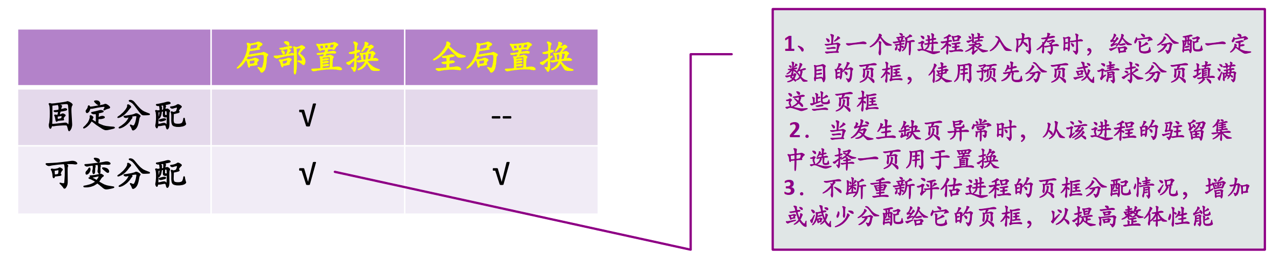
﻿计划置换页面的集合是局限在产生缺页异常的进程，还是所有进程的页框？

1. ﻿局部置换策略

﻿仅在产生本次缺页的进程的驻留集中选择

1. ﻿全局置换策略 现在多用

﻿将内存中所有未锁定的页框都作为置换的候选

1. 两种置换策略与驻留集分配策略的关系

可变分配 + 全局置换现在最常用

1. ﻿置换策略简介
2. 目标

* ﻿决定置换当前内存中的哪一个页框
* ﻿尽可能置换最近最不可能访问的页（理论上最优）

1. 设计思想

* ﻿根据局部性原理，最近的访问历史和最近将要访问的模式间存在相关性，因此，大多数策略都基于过去的行为来预测将来的行为（﻿典型设计思路）
* ﻿注意：置换策略设计得越精致、越复杂，实现的软硬件开销就越大（tradeoff）

1. ﻿约束条件：不能置换被锁定的页框（和内存紧缩不能作用于参与I/O的界面类似）
2. ﻿页框锁定
3. 原因

采用虚存后页面置换的开销使程序运行时间变得不确定

1. 实现

给每一页框的PTE增加一个锁定位，﻿通过设置相应的锁定位，不让操作系统将进程使用的页面换出内存，避免产生由交换过程带来的不确定的延迟

1. 例子

* ﻿操作系统核心代码
* 操作系统关键数据结构（PCB，一级页表）
* 操作系统I/O缓冲区
* ﻿正在进行I/O的内存页面
* ﻿Windows中的VirtualLock和VirtualUnLock函数

1. ﻿**页面置换（replacement）算法**
2. 简介
3. ﻿又称页面淘汰（替换）算法
4. 历史

* ﻿最优算法
* 最近未使用
* 先进先出
* 第二次机会
* 时钟算法
* 最近最少使用（LRU，最常用）
* 最不经常使用
* 老化算法
* 工作集
* 工作集时钟

1. 理想的置换算法（OPT）

* ﻿设计思想

﻿置换以后不再需要的或最远的将来才会用到的页面

* 实现

不可能实现

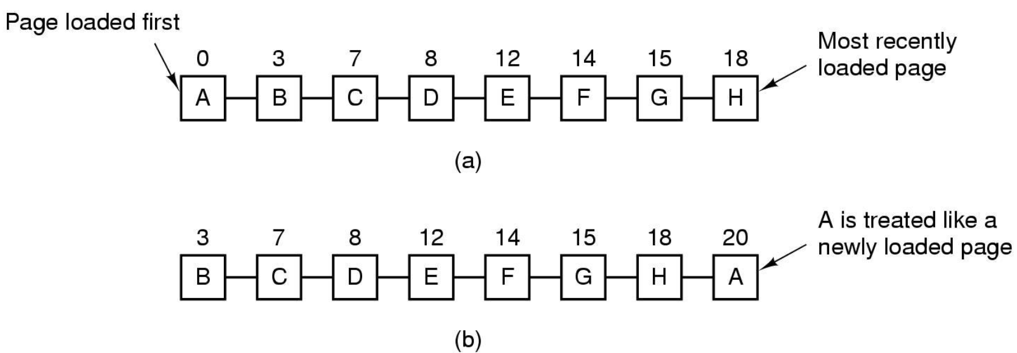
* 作用

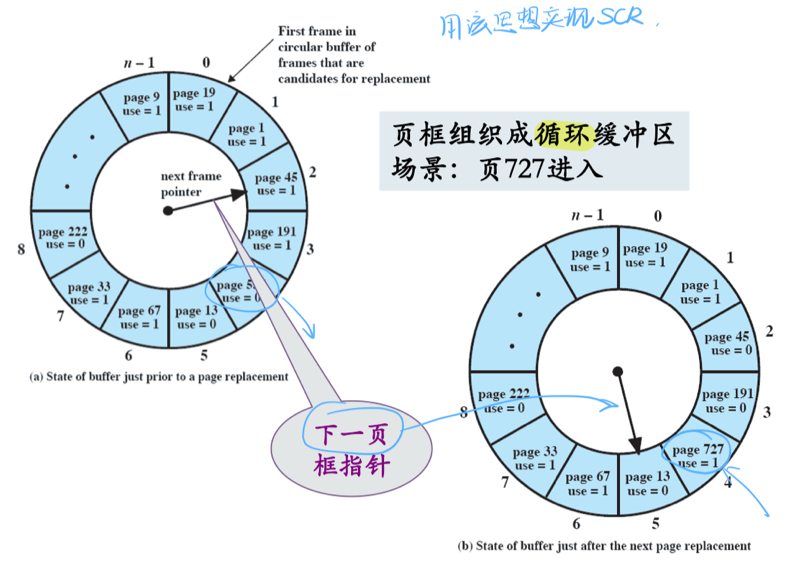
为我们设计更好的置换算法提供思路

1. ﻿**先进先出页面置换算法（FIFO）**
2. ﻿设计思路：选择在内存中驻留时间最长的页并置换它
3. 反例：超市撤换商品，在超市中驻留时间越长其实说明其越畅销，不应被替换为其他商品（即最先被调入内存的也可能是更频繁被使用的）
4. ﻿实现：页面链表法
5. ﻿特殊情况：Belady现象

* ﻿例子：系统给某进程分配 m个页框，初始为空，﻿页面访问顺序为1 2 3 4 1 2 5 1 2 3 4 5，﻿采用FIFO算法，计算当m=3和m=4时的缺页异常次数
* ﻿m=3时，缺页异常9次；m=4时，缺页异常10次
* ﻿注：FIFO页面置换算法会产生异常现象（Belady现象），即：当分配给进程的物理页面数增加时，缺页次数反而增加

1. ﻿**第二次机会置换算法(SCR, Second Chance)**
2. 设计思路

* ﻿按照先进先出算法选择某一页面，检查其访问位R（硬件确定，软件可设置）
* 如果为0，则置换该页
* 如果为1，则给第二次机会，并将访问位置0
* R位**定期清零**，避免偶尔被访问一次的页面得到第二次机会

1. ﻿**时钟算法(Clock)**
2. ﻿**最近未使用算法(NRU, ﻿Not Recently Used)**
3. 设计思想

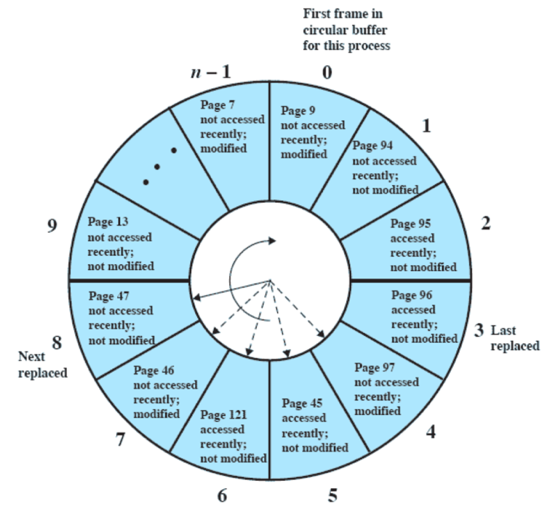
﻿选择在最近一段时间内未使用过的一页并置换

1. 基本实现

* ﻿设置页表表项的两位﻿：访问位（R），修改位（M）。如果硬件没有这些位，则可用软件模拟（做标记）
* ﻿启动一个进程时，R、M位置0，R位被定期清零（复位）
* ﻿发生缺页异常时，操作系统检查R，M
* ﻿第0类：无访问，无修改
* 第1类：无访问，有修改
* 第2类：有访问，无修改
* 第3类：有访问，有修改
* ﻿算法随机从编号最小的非空类中选择一页置换

1. ﻿时钟算法实现（﻿注意：与上面介绍的NRU有区别）

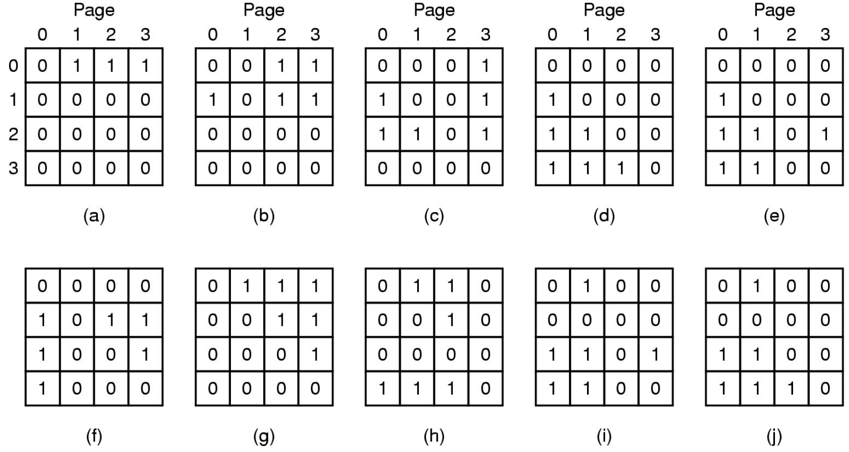
* ﻿从指针的当前位置开始，扫描页框缓冲区，选择遇到的第一个页框（r=0；m=0）用于置换(本扫描过程中，对访问位不做任何修改)
* ﻿如果第1步失败，则重新扫描，选择第一个（r=0； m=1）的页框(本次扫描过程中，对每个跳过的页框，将其访问位设置成0)
* ﻿如果第2步失败，指针将回到它的最初位置，并且集合中所有页框的访问位均为0。重复第1步，并且，如果有必要，重复第2步。这样将可以找到供置换的页框



1. **﻿最近最久未使用算法(LRU，﻿Least Recently Used)**
2. 设计思想

* ﻿选择最后一次访问时间距离当前时间最长的一页并置换，即置换未使用时间最长的一页

1. ﻿﻿实现

* ﻿时间戳或维护一个访问页的栈
* 一种硬件实现：页面访问顺序0, 1, 2, 3, 2, 1, 0, 3, 2, 3

**思考题：解释其实现机制**

* 有4个页框，因此设置一个4x4的矩阵
* 若访问的页面在内存中，则将对应的行置1，列置0
* 若访问的页面不在内存中，则需要进行页面置换，应当替换1最多的列对应的页面

1. 优缺点

* 优点：﻿性能接近OPT
* 缺点：﻿开销大

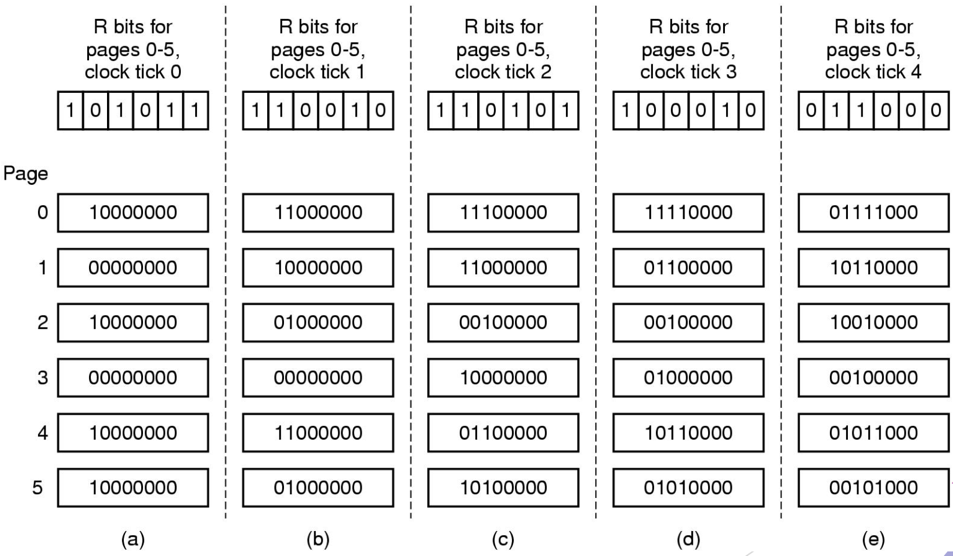
1. **﻿最不经常使用算法(NFU，﻿Not Frequently Used)**
2. 设计思想

* 选择访问次数最少的页面置换
* 是LRU的一种软件解决方案，但不完全等价于LRU

1. 实现（软件）

* ﻿软件计数器，一页一个，初值为0
* ﻿每次时钟中断时，计数器加R（若页面在该时间片内被访问，则加1；否则加0）
* ﻿发生缺页异常时，选择计数器值最小的一页置换

1. **﻿老化算法(Aging)**
2. 设计思想

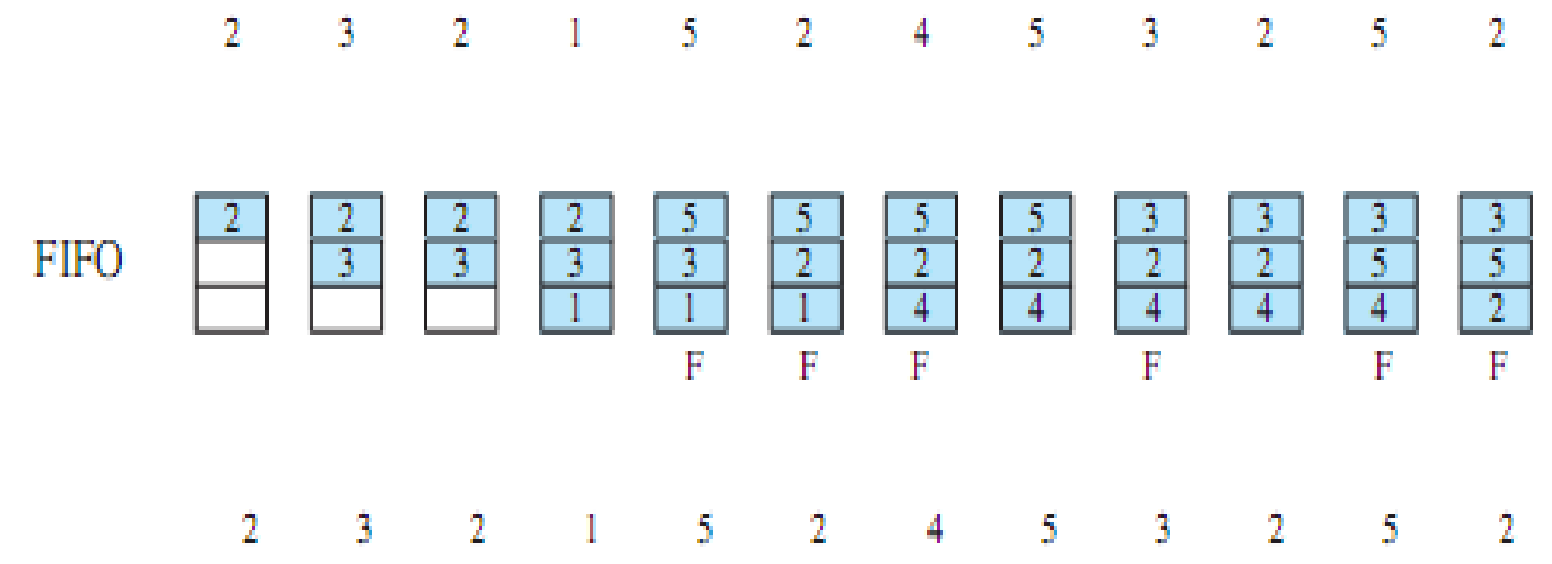
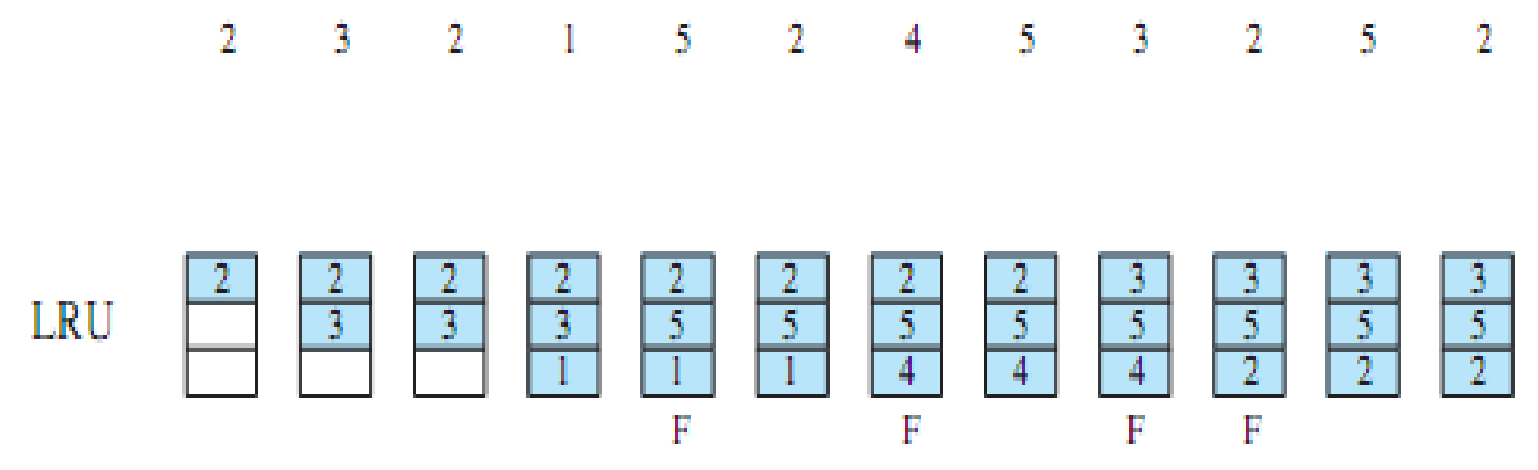
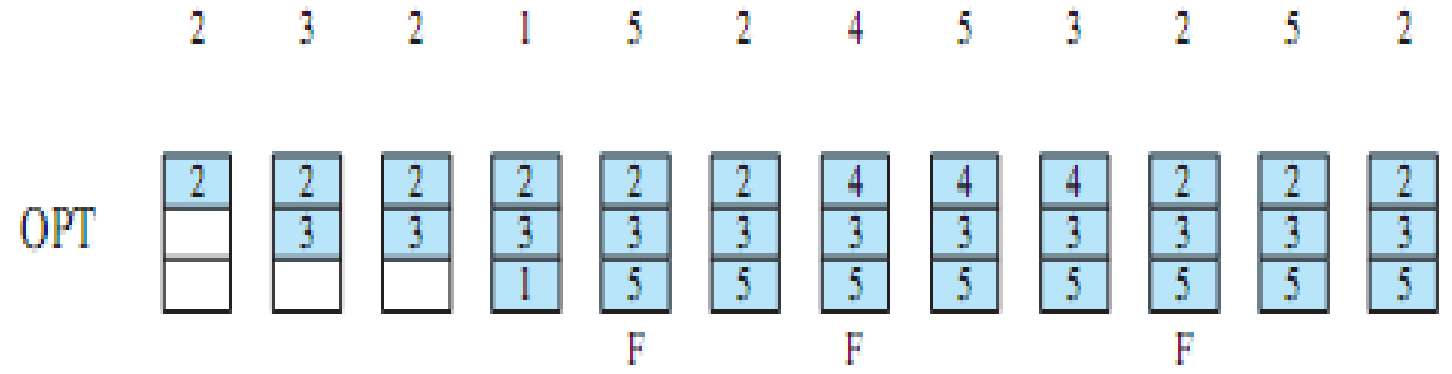
* ﻿用一种更好的方法模拟LRU
* ﻿每一页的计数器在加R前先右移一位，R位加到计数器的最左端

思考题：和LRU比较

老化算法通过定期右移计数器并更新R位的方式，实现了对页面最近使用情况的模拟，从而达到类似LRU算法的效果。不同之处在于老化算法通过硬件计数器进行实现，而LRU通常通过维护一个链表或其他数据结构来记录页面使用的顺序

1. ﻿页面置换算法的应用
2. ﻿计算、比较不同的算法效果

例子：系统给某进程分配3个页框(固定分配策略)，初始为空进程执行时，页面访问顺序为：﻿2 3 2 1 5 2 4 5 3 2 5 2，﻿计算应用FIFO、LRU、OPT算法时的缺页次数（不要忘记第一次访问时一定缺页）

1. ﻿FIFO
2. ﻿LRU
3. ﻿OPT
4. ﻿影响缺页次数的因素
5. ﻿页面置换算法

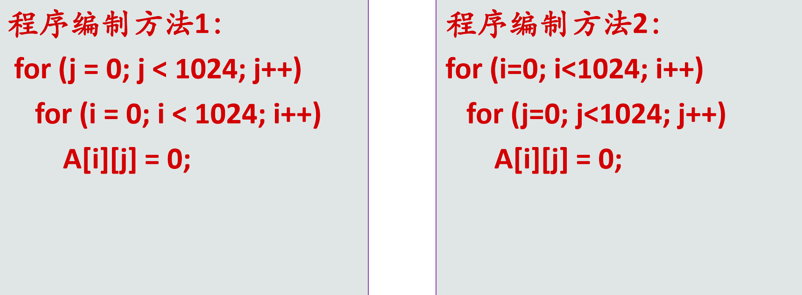
颠簸（Thrashing，抖动）：虚存中，页面在内存与磁盘之间频繁调度，使得调度页面所需的时间比进程实际运行的时间还多，这样导致系统效率急剧下降，这种现象称为颠簸或抖动

1. ﻿页面的大小
2. ﻿确定页面大小对于分页的硬件设计非常重要而对于操作系统是个可选的参数
3. ﻿要考虑的因素

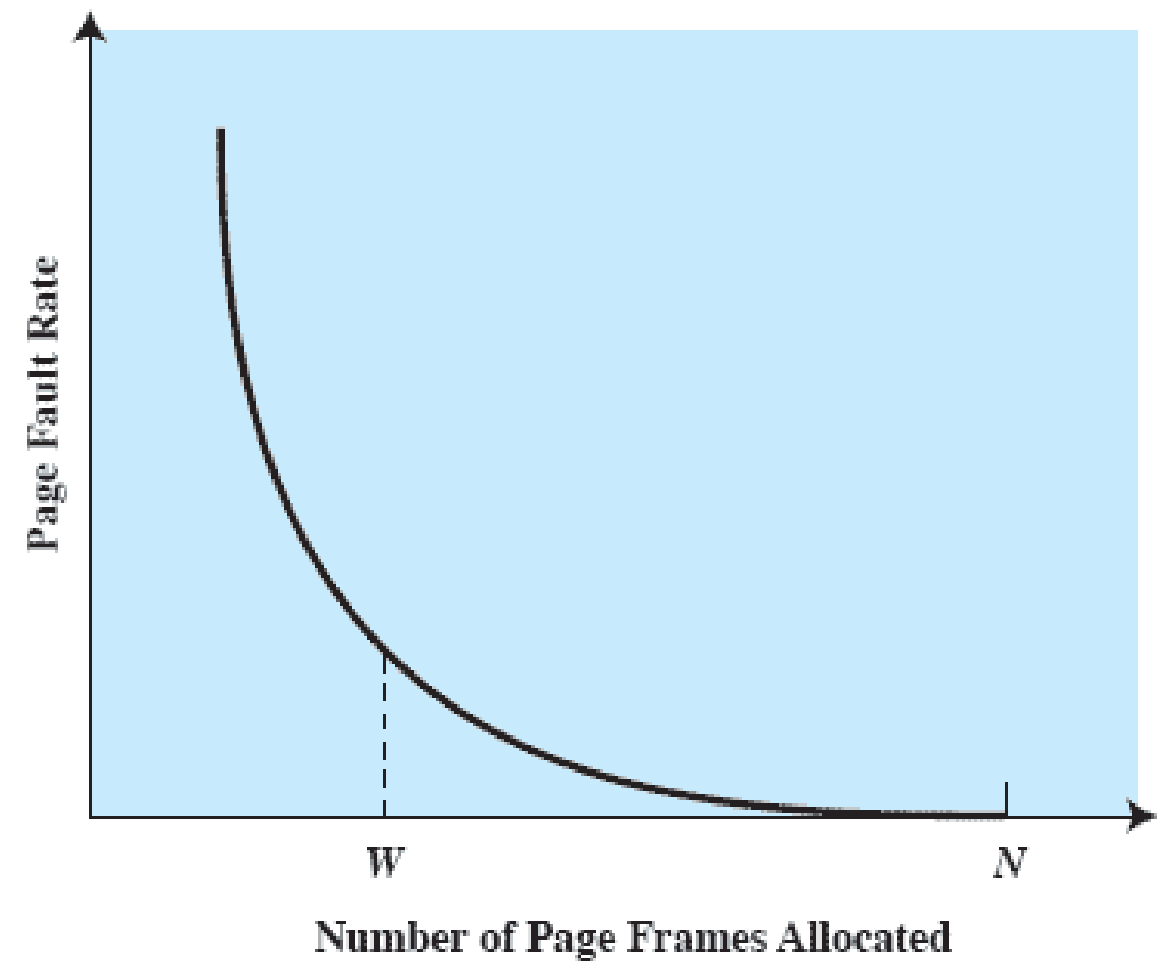
* ﻿内部碎片
* 页表长度
* 辅存的物理特性

1. 大页与小页

* ﻿最优页面大小P = sqrt(2se)
* s:页表项的大小
* e:程序运行期间页面故障的平均处理时间
* ﻿Intel 80x86/Pentium：4096或4M
* 支持多种页面尺寸的优缺点
* ﻿为有效使用TLB带来灵活性
* ﻿给操作系统带来复杂性

1. 程序的编制方法
2. ﻿例子：系统给某进程分配了一个页框；页面大小4K；矩阵A[1024][1024]；行优先存储，问两种编程方法﻿缺页几次？

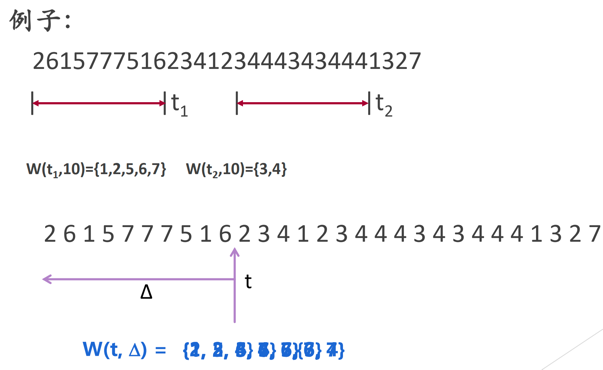
* A的一行恰好为一页
* 方法1缺页1024次
* 方法2缺页1024x1024次

1. 分配给进程的页框数

应当平衡进程占用的空间和缺页导致的时间开销

1. ﻿工作集(Working Set)模型
2. ﻿简介
3. ﻿由Denning提出(1968)，可以看做是进程运行过程中的一种性质
4. 基本思想

* ﻿根据程序的局部性原理，一般情况下，进程在一段时间内总是集中访问一些页面，这些页面称为活跃页面，如果分配给一个进程的页框太少了，使该进程所需的活跃页面不能全部装入内存，则进程在运行过程中将频繁发生中断（Page Fault）
* ﻿如果能为进程提供与活跃页面数相等的页框数，则可减少缺页异常次数

1. 模型机制
2. ﻿工作集W（t，Δ）﻿= 该进程在过去的Δ个虚拟时间单位中访问的虚拟页面集合
3. ﻿工作集的内容取决于三个因素

* ﻿访页序列特性
* ﻿当前时刻t
* ﻿工作集窗口长度(Δ)，﻿窗口越大，工作集就越大

1. 工作集与驻留集
2. 驻留集：﻿当前时刻，进程实际驻留在内存当中的页框集合
3. 二者的区别

* ﻿工作集是进程在运行过程中固有的性质
* ﻿驻留集取决于系统分配给进程的页框数和页面置换算法

1. ﻿工作集算法
2. ﻿基本思路

﻿找出一个不在工作集中的页面并置换它

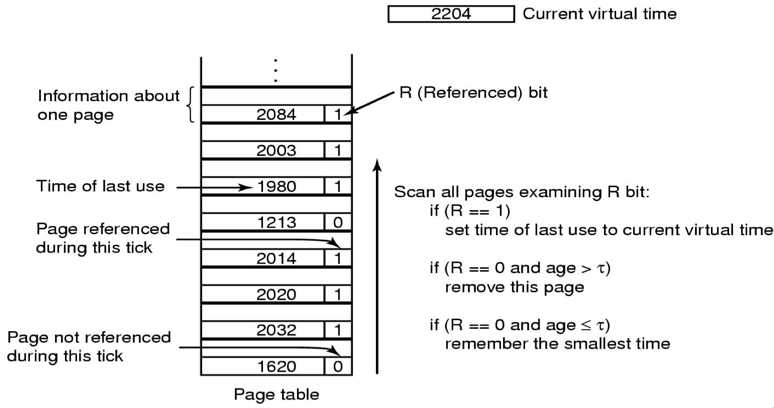
* ﻿监视每个进程的工作集
* 周期性地从一个进程的驻留集中移去那些不在它的工作集中的页（可使用LRU策略）
* 只有当一个进程的工作集在内存中时（即进程的驻留集包括了它的工作集）才可以更好地执行该进程

1. 基本实现

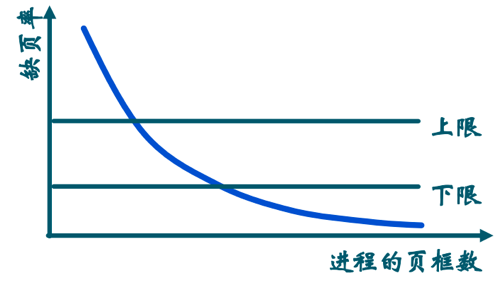
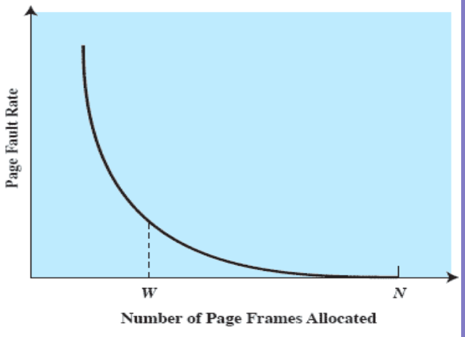
* ﻿每个页表项中有一个字段：记录该页面最后一次被访问的时间
* ﻿设置一个时间值T（时间戳）
* ﻿根据一个页面的访问时间是否落在“当前时间-T”之前或之中决定其在工作集之外还是之内

1. 具体实现

﻿扫描所有页表项

* ﻿如果一个页面的R位是1，则将该页面的最后一次访问时间设为当前时间，将R位清零
* ﻿如果一个页面的R位是0，则检查该页面的访问时间是否在“当前时间-T”之前
* ﻿如果是，则该页面为被置换的页面
* ﻿如果不是，记录当前所有被扫描过页面的最后访问时间里面的最小值。扫描下一个页面并重复检查操作

1. ﻿讨论

* 工作集策略的思想是有效的，许多操作系统都试图采用近似工作集策略
* ﻿其中的一种方法是考虑进程的缺页率，通过监视缺页率达到类似的结果——缺页率算法
* ﻿当增大一个进程的驻留集时，缺页率会下降，工作集的大小会降到图中W点所标记的位置
* ﻿方法： 设置一个进程的缺页率的最小阈值和最大阈值，根据此增减驻留集大小（尽量让驻留集动态近似工作集）

1. 页面置换算法小结

注意R位定期清零

1. 清除策略
2. 简介
3. ﻿分页系统工作的最佳状态：发生缺页异常时，系统中有足够多的空闲页框，这样可以尽可能地减少缺页时页面的置换，从而降低缺页处理的开销
4. ﻿结论：保存一定数目的页框供给比使用所有内存并在需要时搜索一个页框有更好的性能
5. 设计思想：分页守护进程
6. ﻿设计一个分页守护进程（paging daemon），多数时间处于睡眠状态，可定期唤醒以检查内存状态
7. ﻿如果空闲页框过少，分页守护进程通过预设的页面置换算法选择页面换出内存，被置换的页面只修改相应进程的PTE，而不实际进行页面置换
8. ﻿如果页面装入内存后被修改过，则将它们写回磁盘
9. ﻿当需要使用一个已置换出的页框时，如果该页框还没有被新内容覆盖，则将它从空闲页框缓冲池中移出（只需要修改PTE，见下“页缓冲技术”）即可恢复该页面

﻿分页守护进程可保证所有的空闲页框是“干净”的

1. ﻿实现(A.S.Tanenbaum)
2. ﻿使用一个双指针时钟，前指针由分页守护进程控制：当它指向一个“脏”页面时，就把该页面写回磁盘，前指针向前移动；当它指向一个干净页面时，仅仅向前移动指针（前指针不实际进行置换操作，只是修改PTE和更新磁盘）
3. ﻿后指针用于页面置换，与标准时钟算法一样
4. 由于分页守护进程的工作，后指针命中干净页面的概率会增加
5. ﻿页缓冲技术
6. ﻿目的：提高性能
7. ﻿思路
8. ﻿不丢弃置换出的页，将它们放入两个表之一：如果未被修改，则放到空闲页链表中，如果修改了，则放到修改页链表中
9. 被修改的页以簇方式写回磁盘(不是一次只写一个，减少I/O操作的数量，从而减少了磁盘访问时间)
10. 被置换的页仍然保留在内存中，一旦进程又要访问该页，可以迅速将它加入该进程的驻留集合(代价很小)

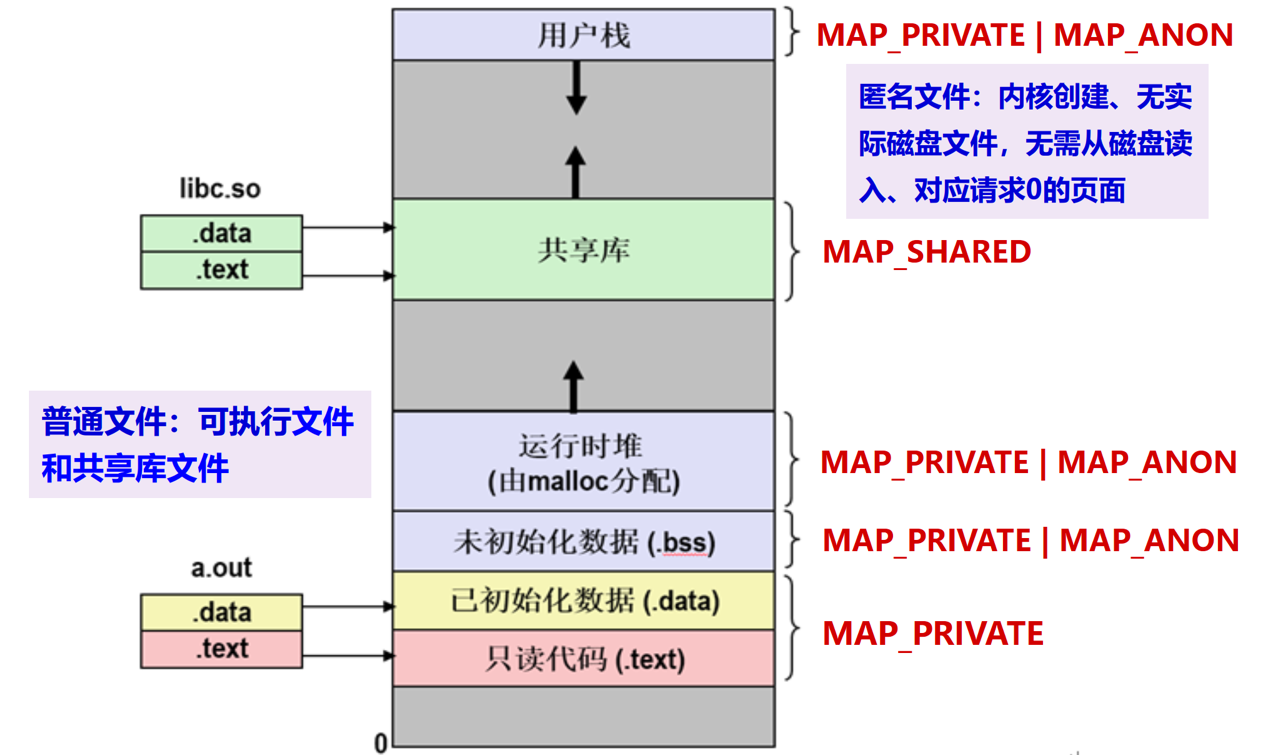
逻辑上的页面置换只需要通过修改PTE有效位即可实现

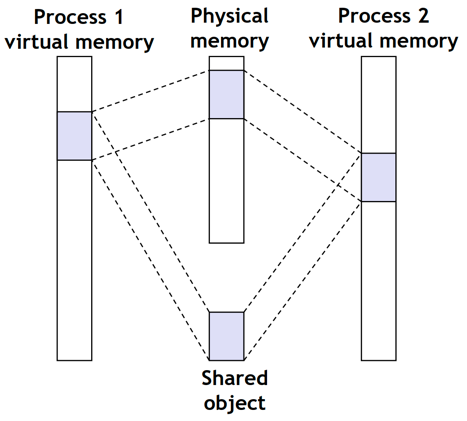
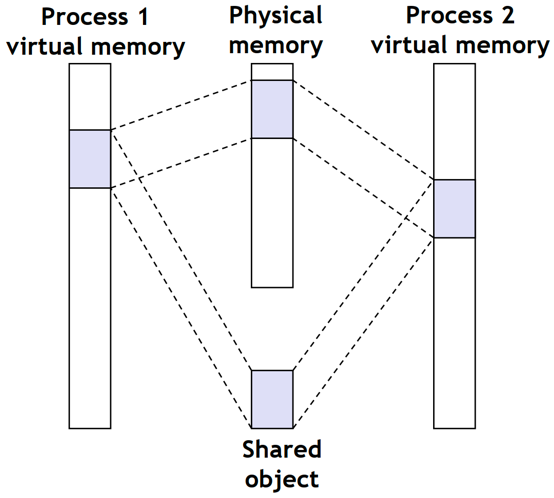
## 内存映射

1. 简介
2. ﻿基本思想
3. ﻿进程通过一个系统调用（例如mmap） 将一个文件或文件中的一部分映射到其虚拟地址空间的一个区域，访问这个文件就像访问内存中的一个大数组，而不是对文件进行读写
4. ﻿在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储。当进程退出或显式地解除文件映射时，所有被修改页面会写回文件
5. 系统调用mmap
6. ﻿**void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);**

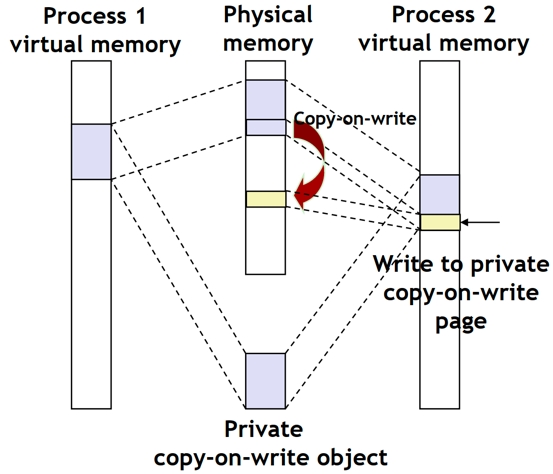
* ﻿将指定文件fd中偏移量offset开始的长度为length个字节的一块信息映射到虚拟空间中起始地址为start、长度为length个字节的一块区域
* ﻿过程：得到vm\_area\_struct（回顾Linux虚拟内存组织）结构的信息，并生成相应页表项，建立文件地址和区域之间的映射关系
* ﻿prot指定该区域内页面的访问权限位，对应vm\_area\_struct结构中的vm\_prot字段
* ﻿PROT\_EXE：页面内容由指令组成
* PROT\_READ：区域内页面可读
* PROT\_WRITE：区域内页面可写
* PROT\_NONE：区域内页面不能被访问
* ﻿flags指定所映射的对象的类型，对应vm\_area\_struct结构中的vm\_flags字段
* ﻿MAP\_PRIVATE：私有的写时拷贝对象，对应可执行文件中只读代码区域（.init、.text、.rodata）和已初始化数据区域（.data）

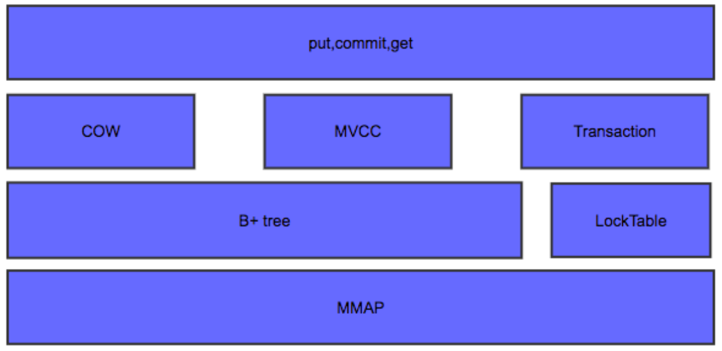
生成随机数可能会修改代码段

* ﻿MAP\_SHARED：共享对象，对应共享库文件中的信息
* ﻿MAP\_ANON：请求0的页，对应内核创建的匿名文件，相应页框**用0覆盖并驻留内存**
* ﻿MAP\_PRIVATE | MAP\_ANON：未初始化数据（.bss）、堆和用户栈等对应区域

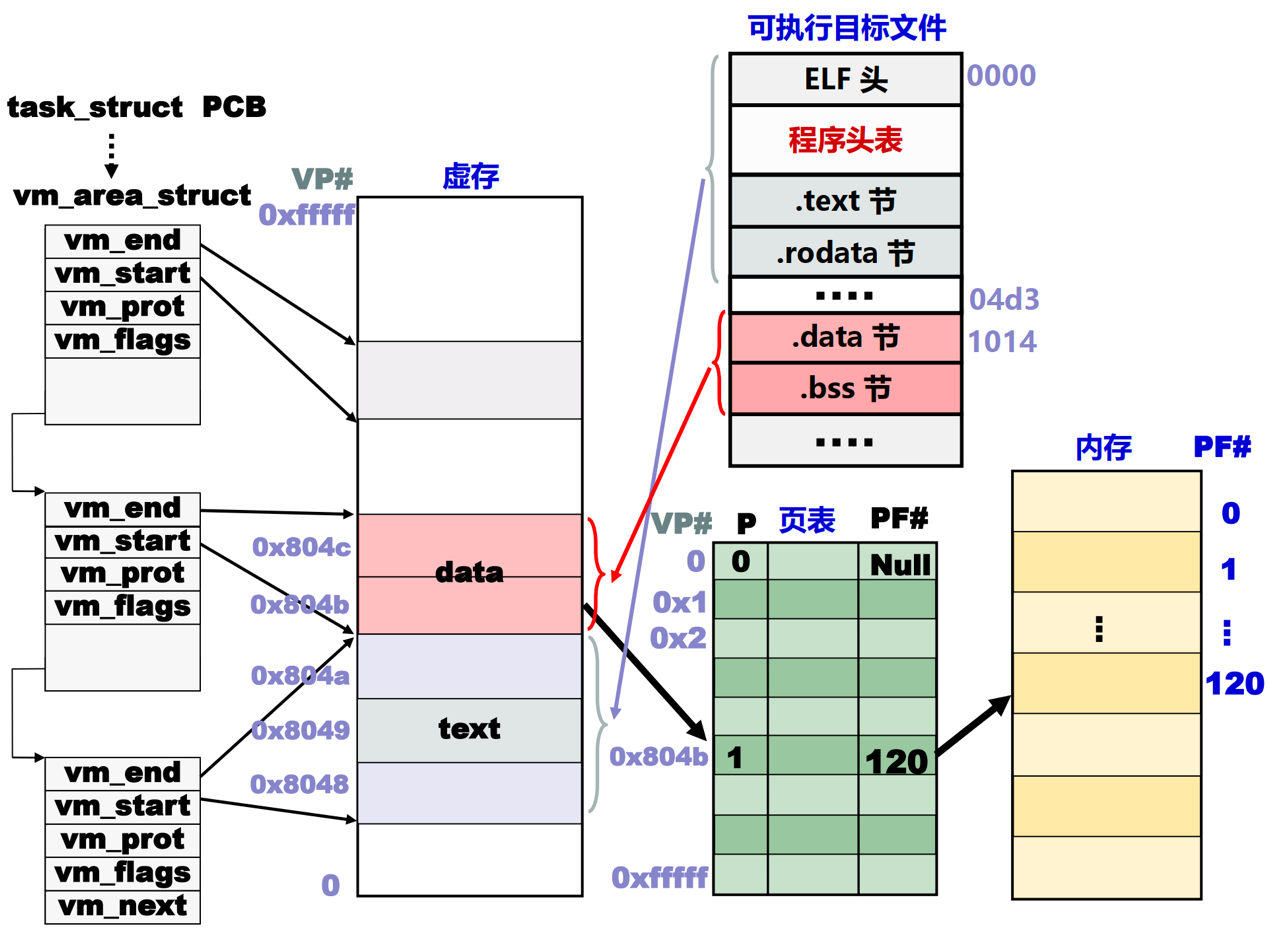
1. ﻿共享库文件中的共享对象
2. ﻿设计思想

多个进程调用共享库文件中的代码，但共享库代码在内存和硬盘都只需要一个副本

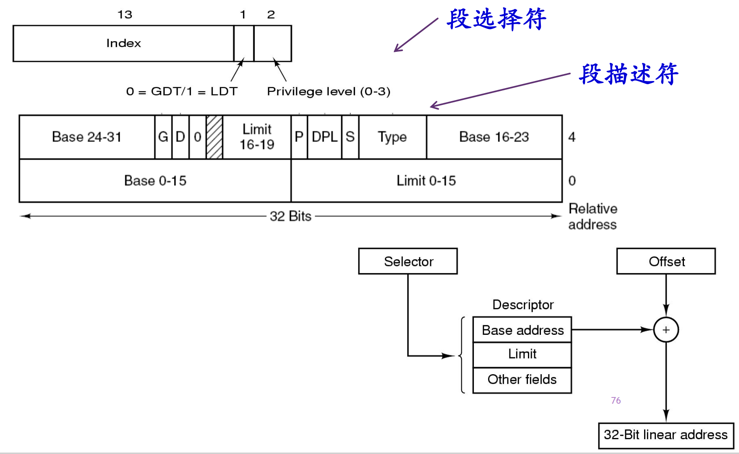
1. 实现机制
2. ﻿进程1运行过程中，内核为共享对象分配若干页框
3. ﻿进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上即可实现共享
4. ﻿一个进程对共享区域进行的写操作结果，对于所有共享同一个共享对象的进程都是可见的，而且结果也会反映在硬盘上对应的共享对象中
5. ﻿私有的写时拷贝对象（COW）
6. 设计思想
7. ﻿同一个可执行文件对应不同进程时，只读代码区一样， 可读可写数据区开始也一样，但属于私有对象
8. ﻿为节省主存，多采用写时拷贝技术
9. ﻿实现机制
10. 进程1运行过程中，内核为对象分配若干页框，并标记PTE为只读
11. ﻿进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上，并标记PTE为只读
12. 若两个进程都只是读或执行，则在内存只有一个副本，节省主存
13. ﻿若进程2进行写操作，则发生访问违例（Page Fault），此时，内核从mm\_area\_struct中判断异常原因是进程试图写私有的写时拷贝页，就会分配一个新页框，把内容拷贝到新页框，并修改进程2的页表项
14. ﻿内存映射文件的应用
15. ﻿LMDB（Lightning Memory-Mapped Database）

﻿内存映射型数据库，通过使用内存映射文件，可以提供更好的输入/输出性能，对于用于神经网络的大型数据集( 比如ImageNet )，可以将其存储在LMDB中

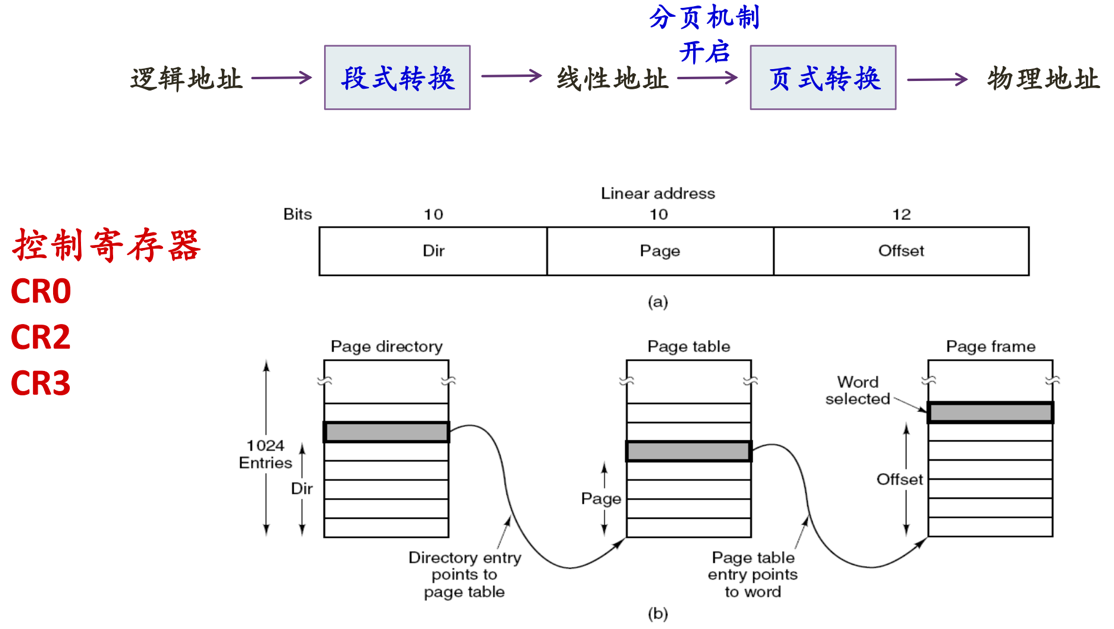
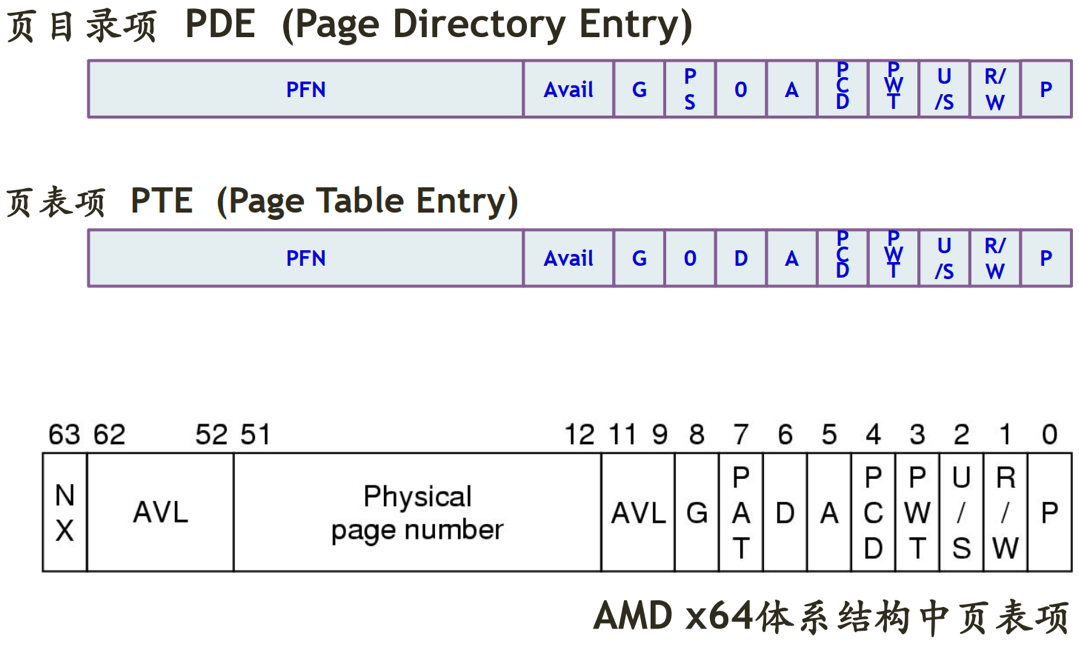
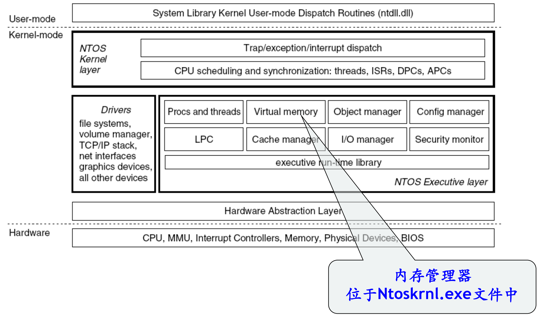
## 虚拟存储管理小结



## Windows虚拟内存

1. ﻿Intel x86虚拟内存机制
2. ﻿保护模式下的寻址方式
3. 思考题

* ﻿x86提供几种工作模式？
* 实模式/保护模式
* 保护模式下的寻址方式？
* 基地址+偏移
* 地址保护如何完成？
* 页表
* 软件如何绕过分段机制？
* 将整个程序编译为一段

1. ﻿逻辑地址到物理地址的转换
2. CR0：分页开关
3. CR2：Page Fault时存储线性地址
4. CR3：存储根目录基地址
5. ﻿x86页目录项和页表项
6. Windows的体系结构
7. 内存管理器的组成部分
8. ﻿基本组成

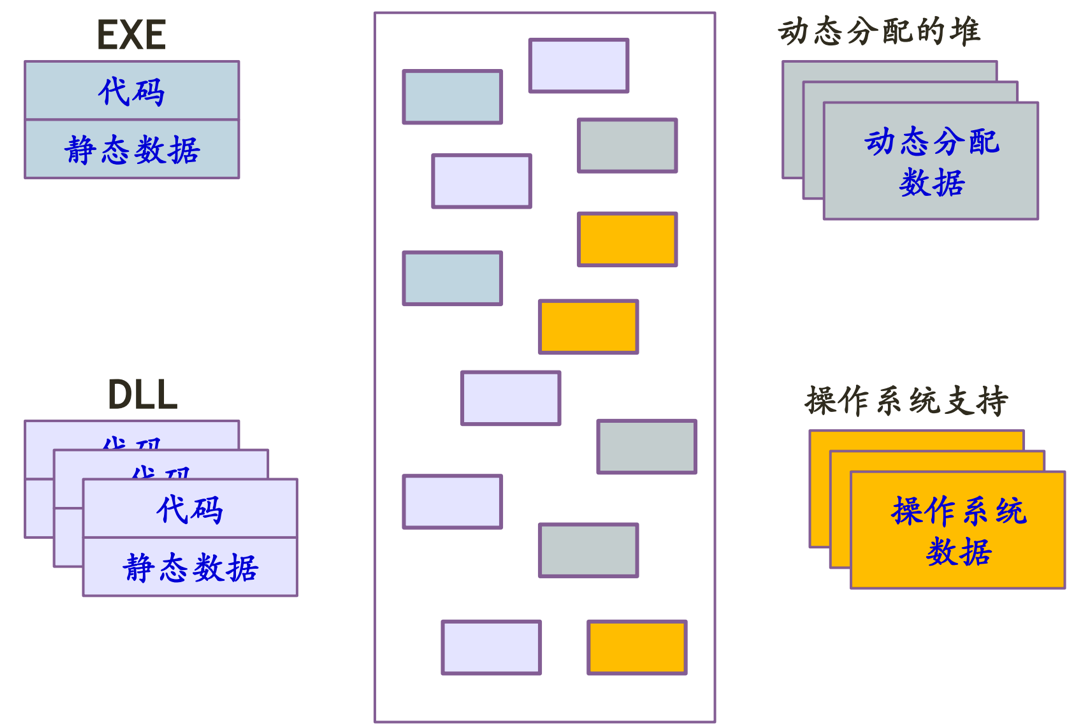
* 一组执行体系统服务程序（函数）
* 用于虚存的分配、回收和管理
* 大多数服务是通过Win32 API或内核态的设备驱动程序接口的形式呈现
* 一个页面错误陷阱处理程序 Page Fault不等同于缺页
* 用于解决硬件检测到的内存管理异常
* 负责分配页框，或把磁盘上的数据装入内存

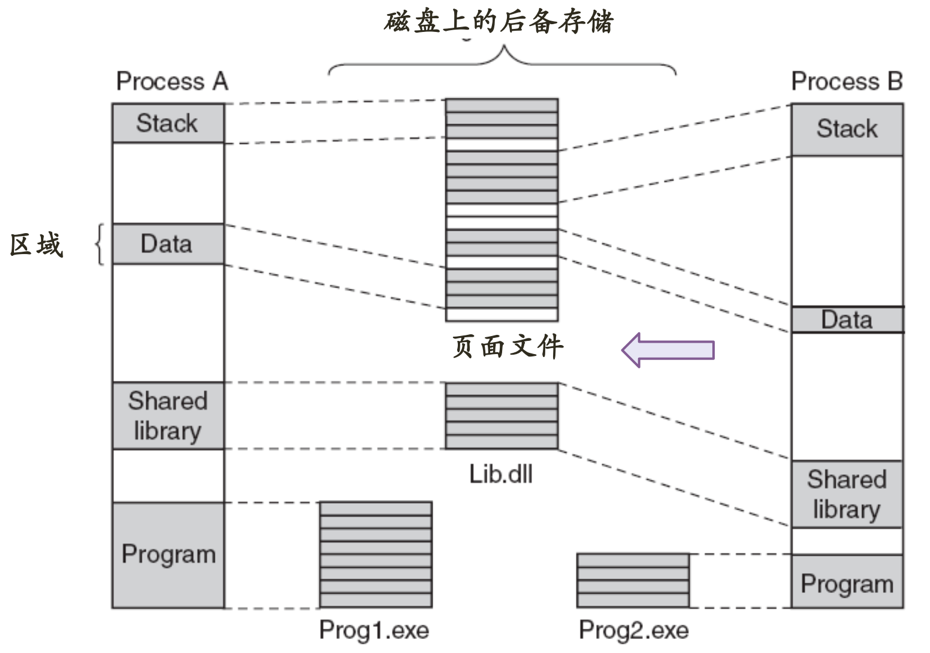
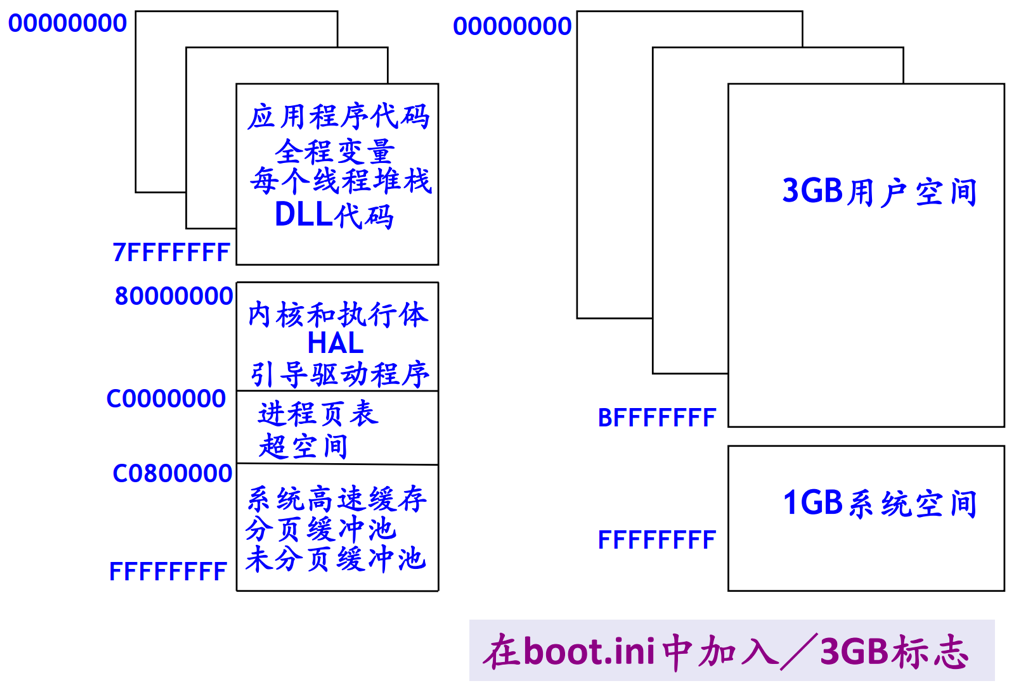
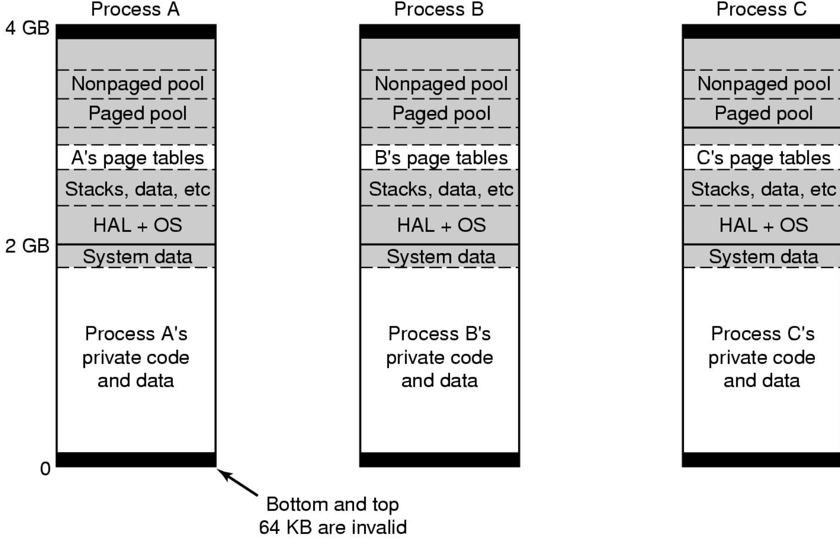
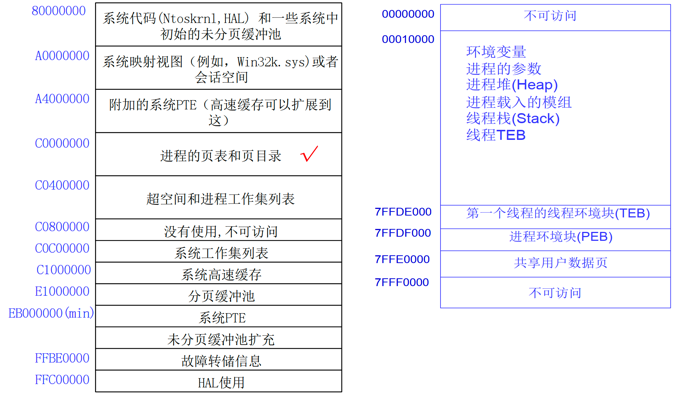
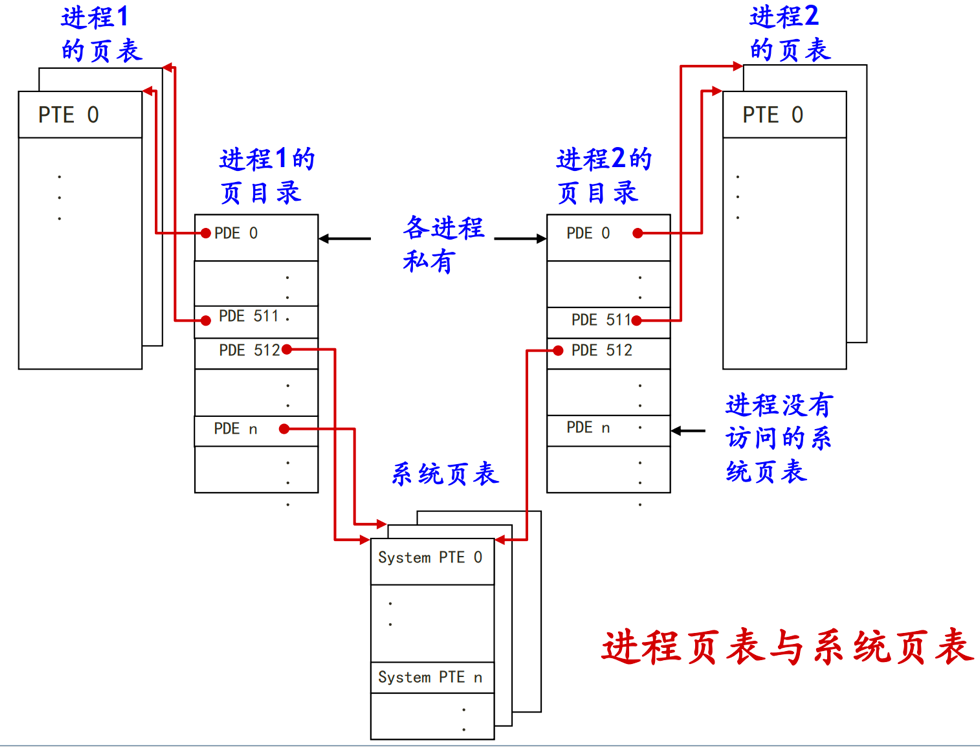
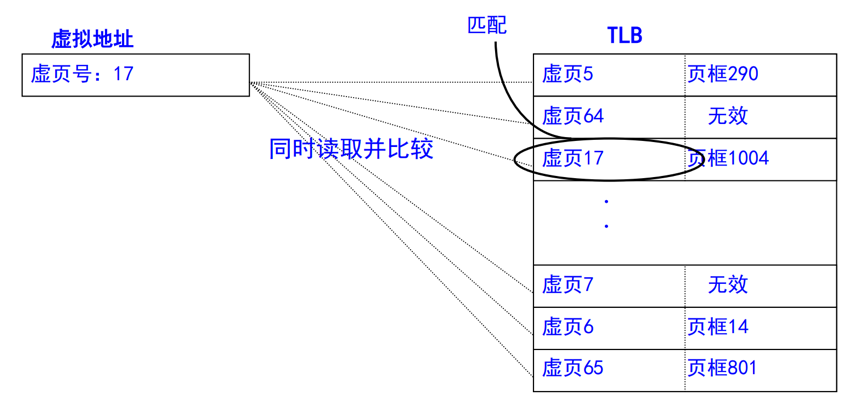
1. ﻿关键组件（线程）

* ﻿工作集管理器(MmWorkingSetManager，16)

优先级较低（16 – 31属于内核优先级）

* ﻿当空闲内存低于某一界限时，便启动所有的内存管理策略
* ﻿如：工作集的修整、老化和已修改页面的写入等
* ﻿进程/栈交换器(KeSwapProcessOrStack，23)
* ﻿完成进程和内核线程堆栈的换入和换出操作
* ﻿修改页面写出器(MiModifiedPageWriter，17)
* ﻿将“脏”页写回到适当的页文件
* ﻿映射页面写出器(MiMappedPageWriter，17)
* ﻿将映射文件中的“脏”页写回磁盘
* ﻿零页线程(MmZeroPageThread，0)
* ﻿将空闲链表中的页框清零

1. ﻿Windows虚拟内存管理
2. ﻿虚拟内存结构

1. ﻿磁盘上的相关内容
2. ﻿地址空间布局
3. **地址转换机制**
4. 进程地址空间分为用户地址空间和系统地址空间，反映在页表的映射关系上
5. 快表TLB
6. 具体机制

* ﻿页目录项PDE和页表项PTE的最低位为1，有效(Valid)，表示该页映射了物理内存
* ﻿当要访问的虚拟页在物理内存中时，该虚拟页对应的 PDE、 PTE都有效，CPU根据相应的PDE、PTE自动把虚拟地址转换成物理地址，完成访问——这一过程操作系统不需介入
* 如果要访问的虚拟页不在物理内存，此时的PTE无效（最低位为0）。对无效PTE的格式的定义（如写只读页...），由操作系统负责

1. ﻿Windows中对页面失效（无效的PTE）的定义

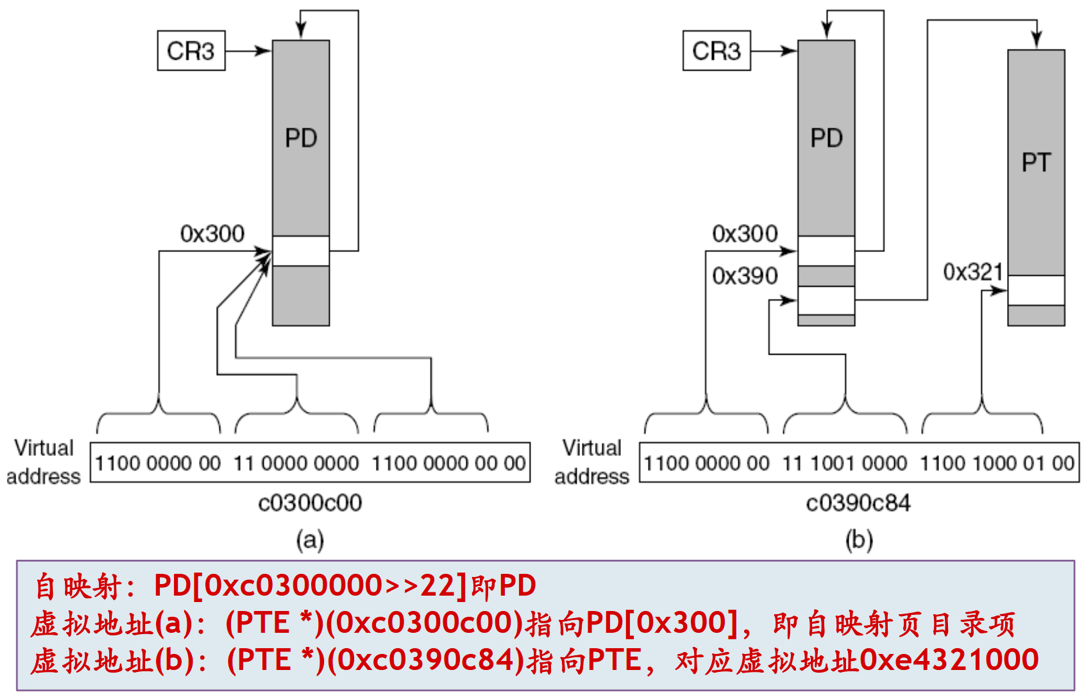
* ﻿所引用的页面没有被提交（没有加载进入内存）
* 违反权限的页面访问（写.rodata段，用户越界访问内核）
* 修改一个私有共享的写时复制页面
* 需要扩大栈
* 所引用的页已被提交但尚未被映射
* 如该页面被置换但尚未被换出内存，页面在物理内存中但PTE有效位为0
* 请求一个零页面

1. ﻿缺页异常处理
2. ﻿CPU 会自动通过页目录和页表把虚拟地址转换成物理地址。当某条指令访问的页不在物理内存中时，在地址转换过程中，CPU将会发现对应地址的页表项无效，从而引发页面错误（Page Fault）异常，该异常的中断号是0xe，有时也称为缺页中断
3. 发生缺页异常时，CPU自动将引发异常时访问的虚拟地址存入寄存器CR2
4. 在发生异常时， CPU自动根据中断号，在中断描述符表中找到相应的中断描述符，根据中断描述符中的地址，转到异常处理程序ntoskrnl!KiTrap0E
5. 异常处理程序KiTrap0E是操作系统提供的，它将会调用MmAccessFault，MmAccessFault 通过 CR2 中的访问地址，计算出相应的PDE／PTE地址，通过分析PTE中的内容（无效的 PDE／PTE的格式由操作系统定义），可以知道是哪种情况引起的异常，并根据情况作出相应的处理
6. ﻿页目录和自映射机制
7. ﻿页目录
8. 简介

* ﻿内存管理器创建的特殊页，用于映射进程所有页表的位置
* 其物理地址保存在KPROCESS中
* ﻿专用寄存器（x86中为CR3）用于保存页目录的物理地址

1. 访问

* ﻿硬件访问页目录、页表和页：通过VPN、PFN完成
* ﻿内核：通过虚地址来对它们进行访问（x86中，它还同时被映射到地址0xC0300000处）

1. 页目录的自映射机制
2. 工作集模型与页面置换算法
3. 工作集模型
4. 简介

﻿驻留在物理内存中的虚拟页面的子集

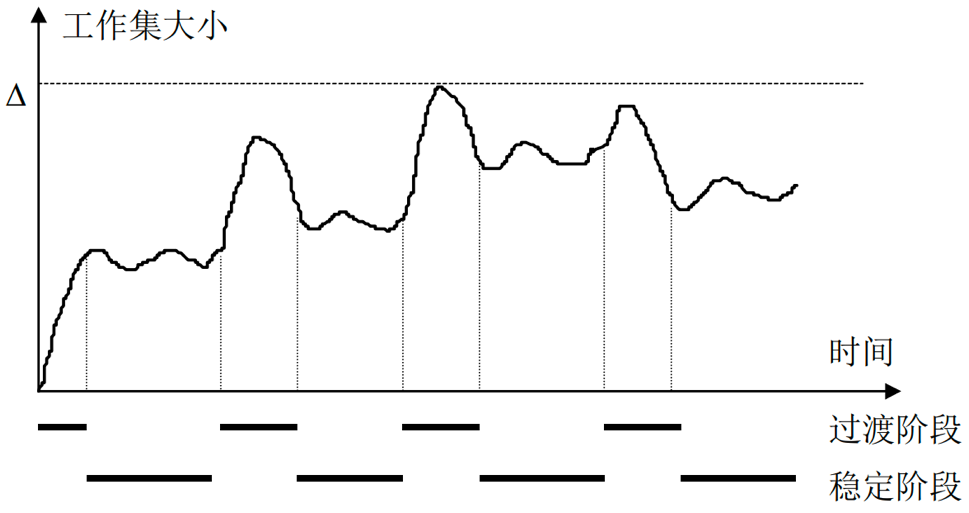
1. 分类

* ﻿进程工作集：为每个进程分配的一定数量的页框
* 系统工作集：为可分页的系统代码和数据分配的页框

1. 页面分配

﻿内存管理器利用请求式页面调度算法以及簇方式将页面装入内存

1. 工作集的大小变化

* ﻿进程开始执行后，随着访问新页面逐步建立较稳定的工作集
* ﻿当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定
* 局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值

1. 页面置换算法
2. 置页策略

选择页框应使CPU内存高速缓存不必要的震荡最小

1. 实现

* ﻿基于工作集模型
* ﻿由已装入内存的页框组成
* 最小值/最大值
* 当可用页框数量降低到一定程度时，启动工作集修整策略（﻿动态改变工作集大小和构成）
* ﻿平衡集管理器线程调用工作集管理器，﻿周期性检查物理内存状态
* ﻿大量可用内存
* 内存开始紧张
* 内存紧缺

1. ﻿用户空间内存分配方式
2. ﻿以页为单位的虚拟内存分配方式：函数（Virtualxxx）
3. 机制

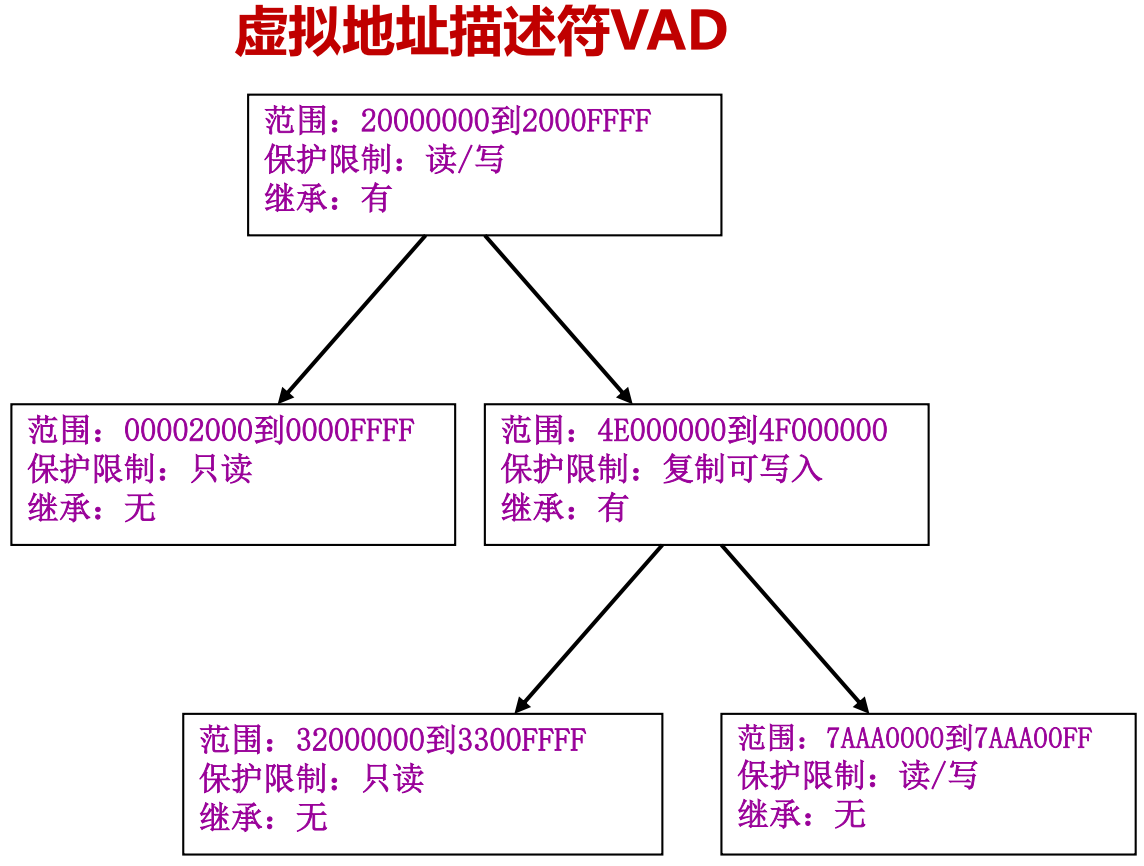
* ﻿进程地址空间0x0~0x7FFFFFFF，用户程序必须经过“保留”和“提交”两个阶段使用一段地址范围
* VirtualAlloc和VirtualAllocEx函数实现这些功能，用户程序可以首先保留地址空间，然后向此地址空间提交物理页面
* 保留地址空间是为线程将来使用所保留的一块虚拟地址。在已保留的区域中，提交页面必须指出将物理存储器提交到何处以及提交多少；提交页面在访问时会转变为物理内存中的有效页面
* VirtualFree或VirtualFreeEx函数回收页面或释放地址空间

1. ﻿问题

使用VirtualAlloc可以在用户地址空间中保留或者提交指定地址和大小的一段地址空间，那么系统如何知道指定的这段地址空间是不是已经被分配（保留或者提交）？

* 对于指定地址空间是否已经被提交了物理内存，可以通过页目录和页表来判断，不过这样做很麻烦；而对于指定地址空间是否已经被保留，通过页目录和页表是没有办法判断的
* Windows中使用VAD来解决这个问题

1. ﻿虚拟地址描述符VAD

* ﻿对于每一个进程，Windows 的内存管理器维护一组虚拟地址描述符（VAD）来描述一段被分配的进程虚拟空间的状态
* ﻿虚拟地址描述信息被构造成一棵自平衡二叉树以使查找更有效率
* ﻿VAD二叉排序树的根的地址保存在进程结构EPROCESS中

1. ﻿内存映射文件：函数（CreateFileMapping, MapViewOfFile）
2. 简介

Windows利用区域对象, ﻿通过区域对象服务提供的大数据流和内存共享服务实现内存映射文件

1. ﻿机制

* 区域对象（section object）在Win32子系统中称为文件映射对象
* 表示可以被两个或更多进程所共享的内存块，区域对象也可以基于页文件
* CreateFileMapping创建区域对象
* MapViewOfFile函数映射区域对象的一部分（叫做区域视图），并指定映射范围
* 当两个进程对同一区域对象建立视图时，就发生了对该区域对象的共享

1. 效果

* 使用区域对象将一个文件映射到进程的地址空间，之后访问这个文件就象访问内存中的一个大数组，而不是对文件进行读写
* 当程序访问一个无效的页面（不在物理内存中的页面）时，引起缺页中断，内存管理器会自动将这个页面从映射文件调入内存
* 如果应用程序修改了页面，内存管理器在页面进行常规调度时将修改写回文件（或者应用程序可以通过利用Win32的FlushViewOfFile函数来刷新一个视图）

1. 内存堆方法：Heapxxx和早期的接口Localxxx和Globalxxx
2. 简介

* ﻿内存堆方法适合于大量的小型内存申请
* ﻿堆(heap)是保留的地址空间中一个或多个页组成的区域，这个地址区域可以由堆管理器按更小块划分和分配
* ﻿堆管理器是执行体中分配和回收可变内存的函数集

1. 实现

* ﻿进程启动时带有一个缺省进程堆，通常是1MB大小。为了从缺省堆中分配内存，线程必须调用GetProcessHeap函数得到一个指向它的句柄。有了句柄后，线程可以调用HeapAlloc和HeapFree来从堆中分配和回收内存块
* ﻿进程也可以使用HeapCreate函数创建另外的私有堆。当进程不再需要私有堆时，可以通过调用HeapDestroy释放虚拟地址空间

1. ﻿物理内存管理
2. 基本方式
3. ﻿页框号数据库（PFN数据库）
4. 结构MMPFN（24字节）

保存每一个物理页的相关信息

1. 全局变量MmPfnDatabase

保存页框号数据库的首地址

1. ﻿页框的状态
2. ﻿活动（active）/有效（valid）

* ﻿该页框在某个进程的工作集中
* 此进程的对应页表项是有效的，从高20bit可获得PFN

1. ﻿过渡（transition）

* ﻿系统正在从一个文件将内容读入该页框，或者正在向一个文件写出该页框的内容

1. ﻿空闲（free）

* ﻿该页框中的内容不再被需要

1. ﻿零初始化（zeroed）

* ﻿该页框空闲并且已经被用零初始化

1. ﻿坏（bad）

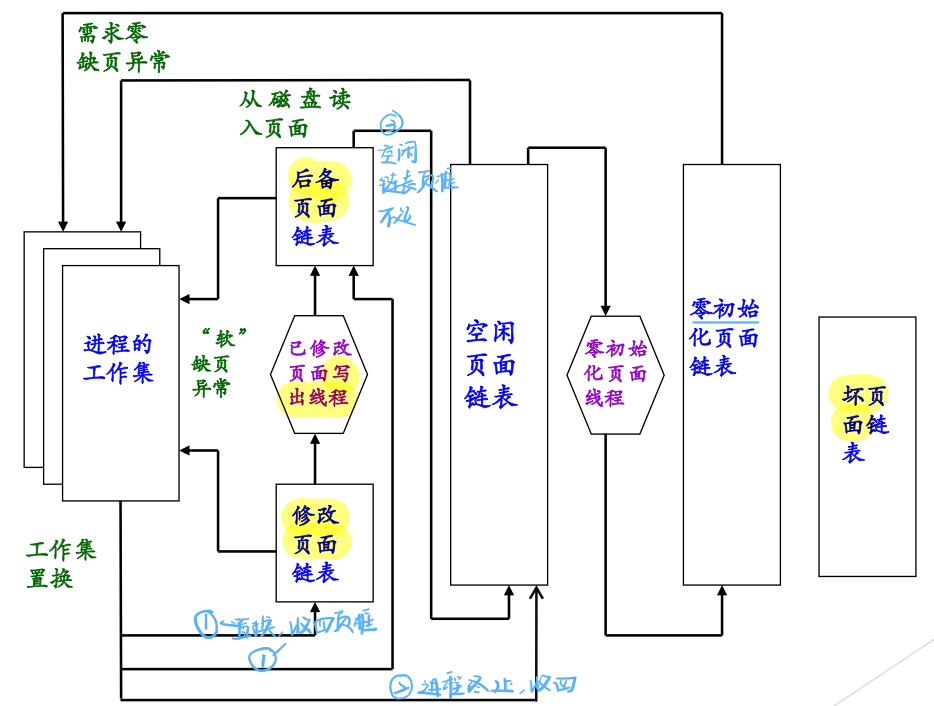
* ﻿该页框存在硬件错误，不能被使用

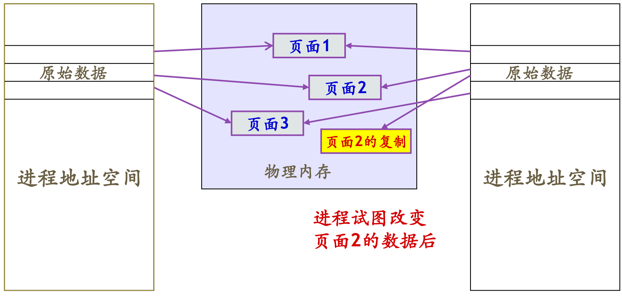
1. 后备（standby）

* ﻿该页框曾经在某个进程的工作集中，且该页框的内容在被此进程使用时没有改变
* 该页框现在已被移出该进程的工作集，但页框中的内容仍是此进程的内容，即对应的PTE中的高20bit仍然是该页框号，只是该PTE被标记为invalid和transition
* 当此进程需要再次访问这一页内容时，只需要重新设定该PTE的标志，并把该PTE变为有效，即把该页框从 standby 状态变为active(valid)状态即可

1. ﻿修改（modified）

* ﻿该页框曾经在某个进程的工作集中，且该页框的内容在被此进程使用时有改变
* 该页框现在已被移出该进程的工作集，但页框中的内容仍是在此进程的内容，即对应的PTE中的高20bit仍然是该页框号，只是该PTE被标记为invalid和transition
* 当此进程需要再次访问这一页内容时，只需要重新设定该PTE的标志，并把该PTE变为有效，即把该页框从modified状态变为active(valid) 状态就可以了
* 在该页框被系统作为其他用途使用之前，需要将该页框中的内容写入硬盘中的页文件中

1. ﻿页表与页框号数据库
2. 实现
3. 应用：﻿支持写时复制技术

* ﻿例如两个进程共享三个页，每个页都标志成写时复制
* ﻿新复制的页面对执行写操作的进程是私有的，对其他共享写时复制页面的进程是不可见的

## 重点小结

1. ﻿基本概念
2. ﻿虚拟内存、虚拟地址、虚拟地址空间
3. 页表表项、多级页表、 反转页表（倒排页表）
4. 快表TLB
5. ﻿虚拟内存管理
6. ﻿取页策略（按需、预取）
7. 驻留集管理（固定、可变）
8. 置换范围与策略（局部、全局）
9. 清除策略（页缓冲）
10. 加载控制
11. ﻿内存映射文件

## 虚拟内存总结

1. 虚拟地址空间是对内存的抽象
2. 虚拟内存管理机制
3. 虚拟内存管理的目标