OS笔记（二）

内存管理

基本内存

预处理->编译->汇编->链接->加载

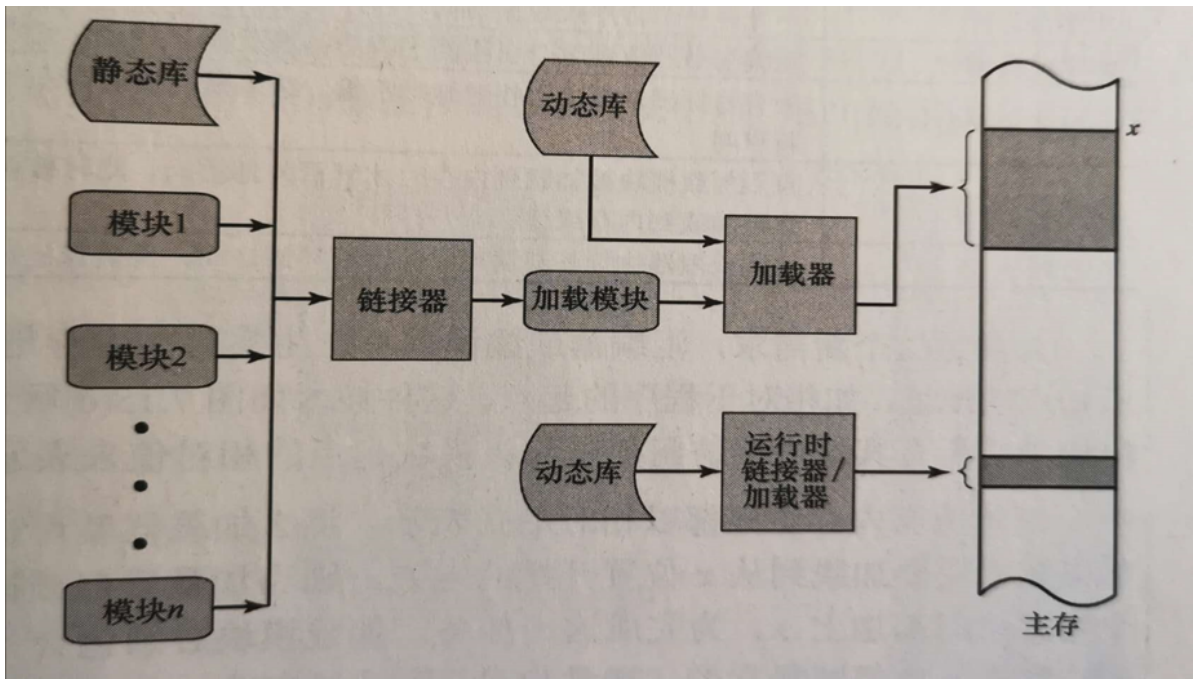
符号地址--编译或汇编-->逻辑地址（从0开始编址，相对地址是逻辑地址的特例）--加载或运行时-->绝对地址

加载

将可加载模块装入内存+地址重定位

- 绝对：给定模块要加载到内存中的固定位置，逻辑地址和实际内存地址完全相同，程序员要根据内存情况确定相对地址。
- 重定位：换出后再换入时往往不会换到之前的位置，所以需要重定位
    - 静态（可重定位加载）：编译后产生从0开始的相对地址，运行前把相对地址转换成绝对地址，并装入（加载），运行时不允许程序在内存中移动
    - 动态（运行时加载）：重定位寄存器用于保存程序在内存中的起始地址，绝对地址=逻辑地址+重定位寄存器中的值。界限寄存器用于存放作业界限地址，判断是否出界。不必连续装入内存，可移动，更方便**共享**

链接



静态：编译后加载前，要修改相对地址，从0开始->接着上一模块开始

不利于代码共享（A、B模块都call C，就只能连AC，BC），不利于模块的独立升级，可能会链接一些不会执行的模块，浪费存储空间（if call B else call C）

加载时动态：能共享、独立升级，不能解决链接不会执行的模块的问题，且加载后不能移动

运行时动态：支持分段系统，如Windows 的DLL

内存管理需求：重定位，保护，共享，逻辑组织（内存被组织成线性的地址空间），物理组织

分区

固定分区，大小和数量都固定

会产生分区内部的碎片，造成空间浪费，大小不等的分区可以缓解这一问题，但比较依赖放置算法

动态分区，malloc，按进程需要的空间大小来分区

会产生外部碎片->紧凑（压缩），使占用的空间连续，要改变地址空间，该过程中进程无法执行，会浪费处理器时间

动态分区算法

首次匹配：从头开始扫描，找到空间足够的首个可用块

前面会出现很多小碎片，后面才有大的空间

下次匹配：从上一次放置的位置开始扫描

最佳匹配：空闲分区按容量大小从小到大链接，从链首开始扫描

产生的外部碎片一般很小，难以再利用

最坏匹配：空闲分区按从大到小链接

伙伴系统

一半一半的划分，同一个根分出来的可以合并

折中，兼有固定分区和动态分区的特点

分页

页（针对进程），页框（针对内存）

每个进程都有它的页表

逻辑地址A=页号P|页内偏移量W，页面大小L=$2^{(W的位数)}$，P=floor(A/L), W=A mod L

页号--查页表-->页框号P'，物理地址=页框号P'|页内偏移量W=P'*2^{W}+W

页表存放在内存，PCB保存有页表的起始地址，页表寄存器存放当前运行进程的页表的起始地址

优点：不存在外部碎片，实现了离散（不连续）分配

缺点：存在页内碎片，需要快表等硬件支持，**不易实现共享**（动态链接和共享要以逻辑模块为单位，但是分页没管模块）

分段

通常是由编译器来分段

段长不固定，没法像分页一样用页号*页面大小算出起始地址，只能多记录一个段号对应的起始物理地址，所以分页的逻辑地址是拼接，分段则是相加

逻辑地址=段号|段内偏移量

物理地址=段首址+段内偏移量W（段内地址独立编址，0到W）

段表寄存器存段表的起始地址

由于偏移量的位数不固定，为了避免偏移量>段长，段表中还要加一个段长字段

优点：便于程序模块化设计->便于动态链接、保护和共享；支持动态增长的数据结构

缺点：分段的最大尺寸受到主存可用空间的限制，需要段表寄存器等硬件支持

分页和分段是内存分区的两种方式，分页是固定分区方式，分段类似动态分区方式

分页对用户透明，分段对用户可见

页--信息的物理单位--系统管理

段--逻辑单位--用户需求

分页中页号和偏移量是可以拼接的，所以分页的地址实质是一维表示，分段是二维

段是虚拟的？二级分区？

虚拟内存

任意时刻，进程驻留在内存的部分称为驻留集

内存失效：访问一个不在内存中的逻辑地址

处理：中断当前进程，操作系统产生一个磁盘I/O读请求，在执行磁盘 I/O期间，操作系统调度另外一个进程运行，磁盘I/O完成后产生中断，操作系统将相应的进程置于就绪状态

抖动

局部性原理，访问呈簇（cluster）性，时间、空间的局部性，可以通过对会访问的块进行猜测来避免抖动。

在页表中加入：存在位（地址转换的时候用），修改位（换出的时候用），其他控制位，页号是索引，不用加入页表，页号+寄存器中存储的页表首址即可索引到相应的表项，从这个意义上来看顶级页表的页号也是一个偏移量

页表装在内存中

多级分页

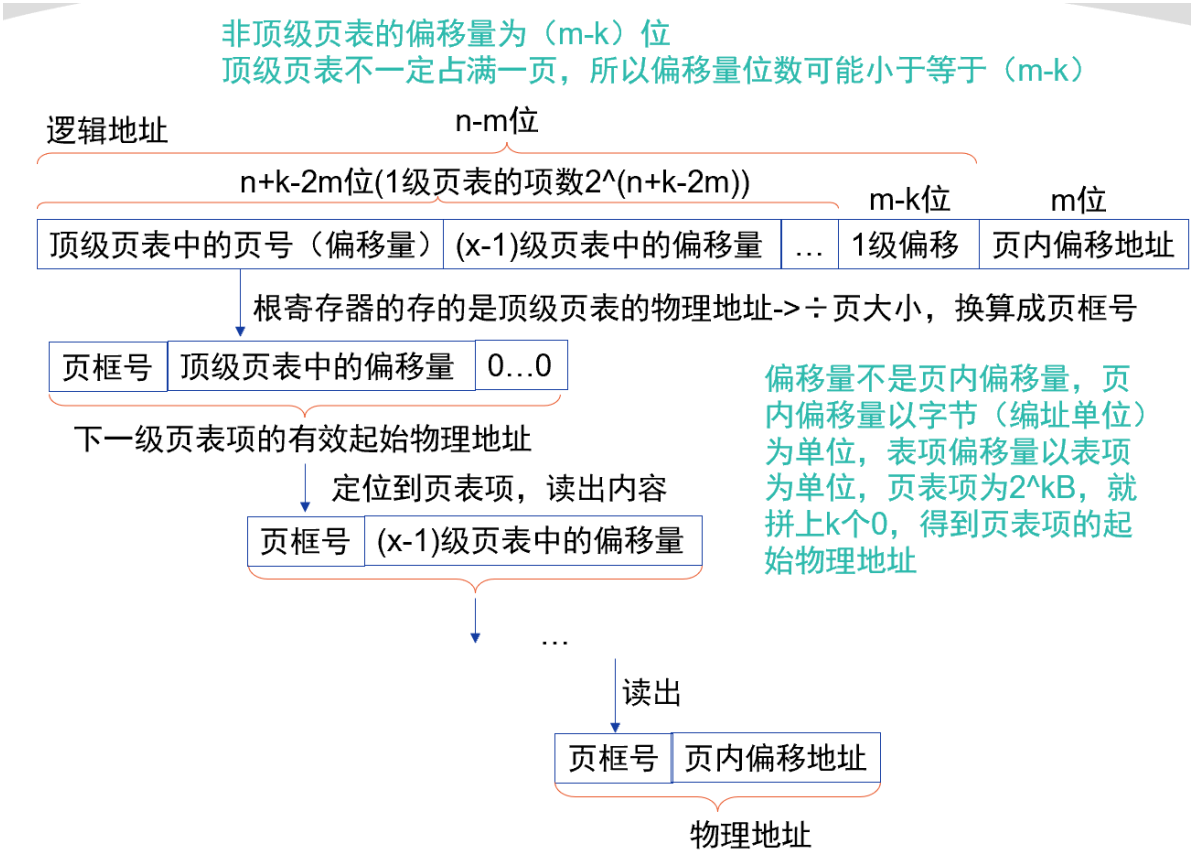
页表占据很多页，所以需要（把页表当成一块普通的空间）再建一级页表来索引页表

最终要形成能够装在一页中的根页表，寄存器中存根页表的页框号

页表只是逻辑上的分级，实际页表都是线性存储在内存中的

n位的逻辑地址 --> 2^n*编址单位大小的虚拟空间 -->页大小为$P=2^m$，虚拟空间的页数：$W1=2^n/2^m$ --> 页表有W1项，页表项大小为2^k，页表大小$V1=W1*2^k$ --> V>P？，占V1/P页，再建立高一级页表，低级页表的一页对应该级页表的一项 --> 高级页表的项数，$W2=V1/P=2^{(n+k-2m)}$，页表大小的计算同上（根页表是一级页表）

低级的一页在高级页表中是一项，地址中的高位是高级页表的页号，低位是偏移量



T：多级页表的地址转换

1. 先求出逻辑地址划分
2. 求划分中每项的内容
    - 直接变成二进制，划分之后求
    - 十进制
3. 按上图的访问和转换过程求出物理地址

倒排页表

另一种组织页表的方式

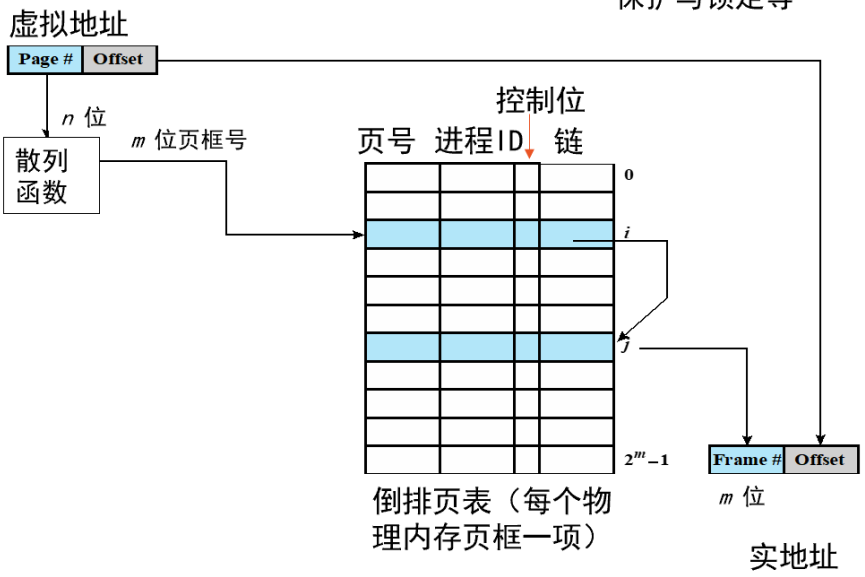
无论有多少进程、支持多少虚拟页，页表都只需要实存中的一个固定部分，所有进程共用一张页表

倒排：使用页框号而非虚拟页号来索引页表项

❖ 倒置页表结构

页号 | 进程标志符 | 控制位 | 链指针

- 使用该页的进程
- 一些标记，如有效、访问与修改、保护与锁定等
- 下一项的索引值,0到2^m-1（n>m，多个虚拟页号可能映射到同一个倒排页表项，采用链式结构处理溢出）

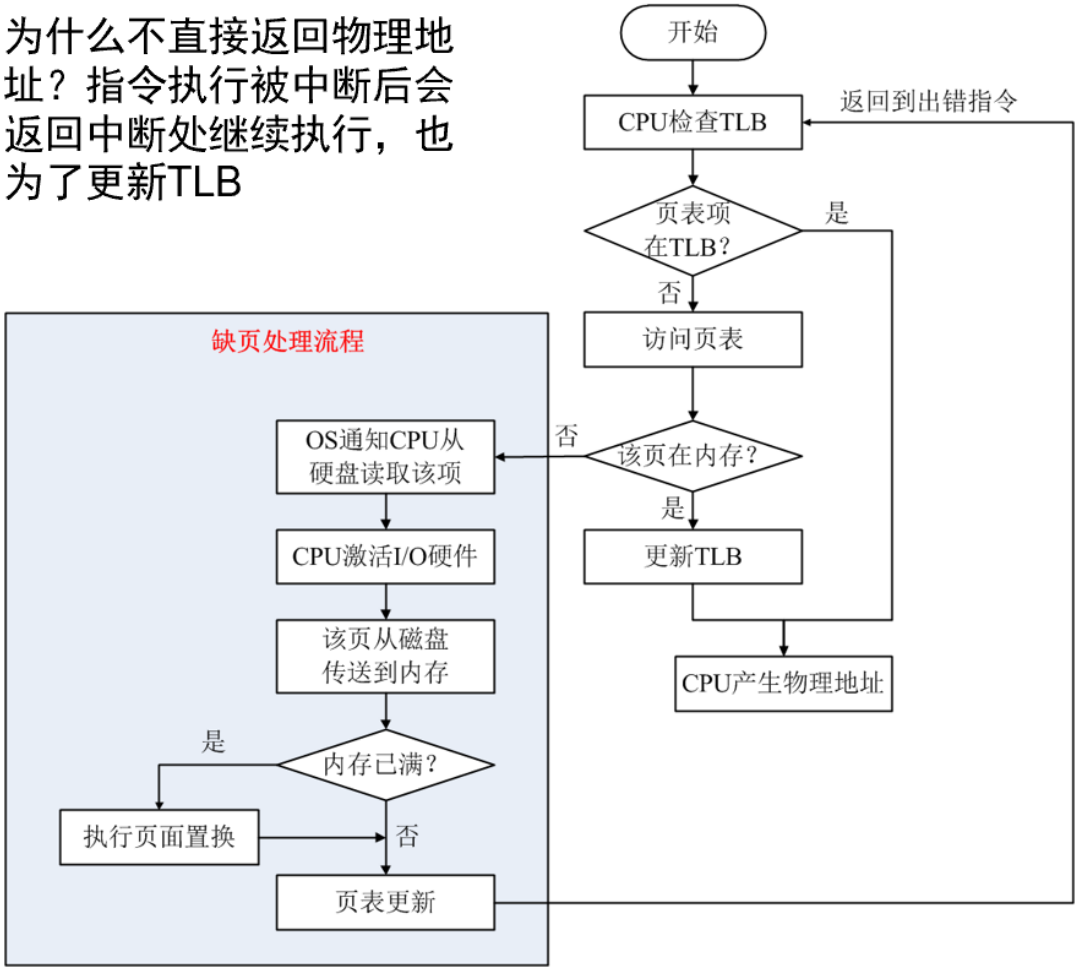


TLB

硬件实现多个TLB表项同时比对查找

具有快表的地址转换流程图

为什么不直接返回物理地址？指令执行被中断后会返回中断处继续执行，也为了更新TLB



cache（内存高速缓存，不是内存，内存是它的上一级存储）

页尺寸，考虑内部碎片和页表大小两方面

页表（任意一级）也有可能不在内存中，分段式也有可能发生缺段

段页式

先分段，段内分页

❖ 段页式的逻辑地址

- 用户地址空间被程序员划分成若干段，每段划分成若干页
- 程序员的角度：逻辑地址 = 段号+段内偏移量
- 系统的角度：段内偏移量 = 页号+页内偏移量

❖ 段表和页表

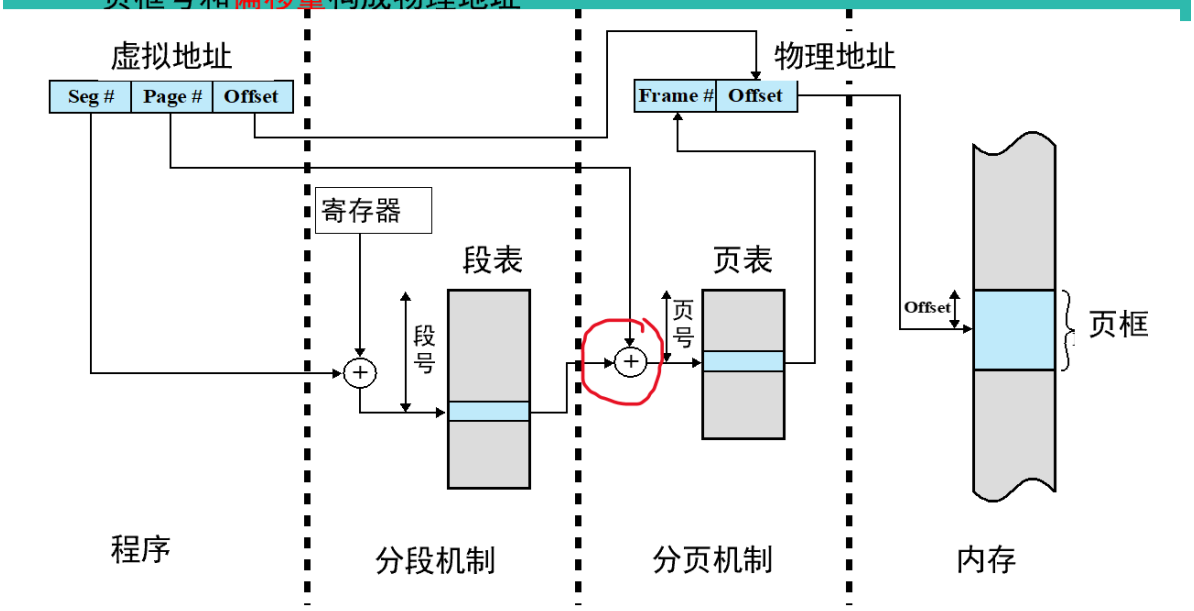
- 每个进程一个段表
- 每个段一个页表
- 段表项：含段长和对应页表的起始地址
- 页表项：含页框号、存在位P、修改位M等

每个进程一个段表，每个段一个页表（只分页不分段，一个进程对应一个页表（可能多级））

注意，段号、页号与寄存器做的不是普通的加法，要看段表项、页表项的长度，如果表项长度是2^kB，按字节编址，应该是寄存器中的起始地址+表项长度*号

段页式的地址转换

- 虚拟地址（逻辑地址） = 段号 + 页号 + 偏移量，寄存器存放段表起始地址
- 根据段表起始地址和段号查找段表，得到对应段的页表起始地址
- 根据页表起始地址和页号查找页表，得到页框号
- 页框号和偏移量构成物理地址



保护：偏移量<段长的检查可避免非法越界访问

共享：共享部分作为一个段，可以在多个进程的段表中被引用

OS软件

读取策略

- 请求调页
- 预调页：减少刚开始的缺页，但是额外读取的页面可能不被使用

放置策略

即前面讲到的动态分区的四种算法

置换策略

页框锁定（不能置换），增加锁定位，操作系统内核和重要的数据结构保存在锁定的页框中

换出最近最不可能访问的页面，降低分页率

OPT

置换**下次访问**距当前时间最长的页面

理想算法，缺页率最小

LRU

根据局部性原理，置换最长时间未引用的页面

时间戳；建立链表，被访问的插入头部；维护访问次数数组，没访问到就-1，访问到就+1，移出次数最小的

FIFO

按循环队列管理页框序列

放入页框后不能随意移动，可以用指针指示队首和队尾

Clock

设置与页绑定的使用位U

没装满，装入页面，U=1，指针后移

不出现缺页，指针不移动，被访问到的页面U=1

当出现缺页时，指到U=1的页面U清零，指针后移，U=0的页面被换出，换进来的U=1，指针后移

改进的Clock：考虑修改位，优先置换最近**未被访问也未被修改**的，这样就方便累积多次修改后一次写回。

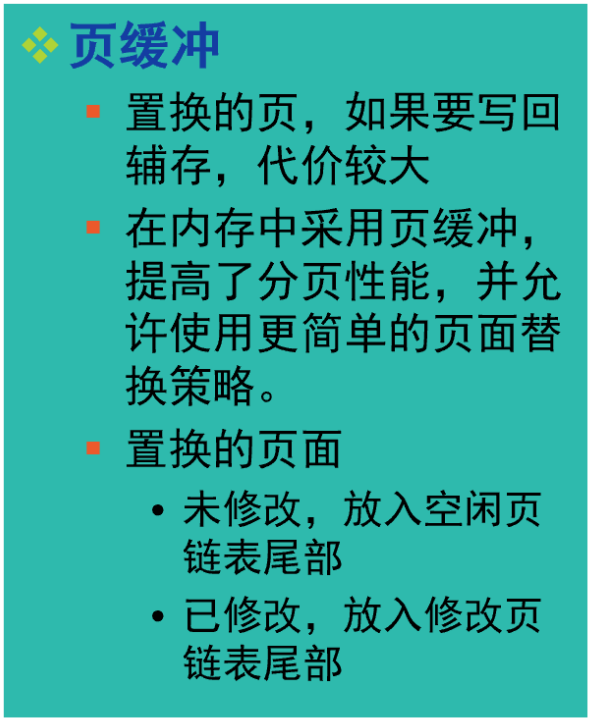
T：画出置换图

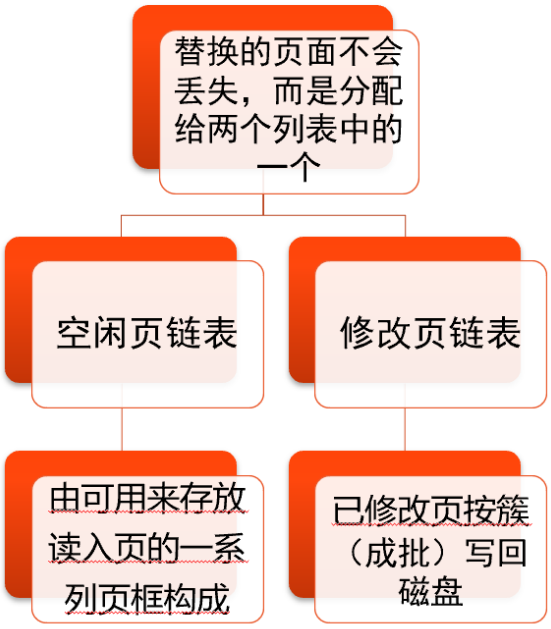
页缓冲

当一个进程被分配的页框已经被占满了，需要局部置换的时候，就把换出的页插入空闲页链表的头部，从尾部取一个页框，该页框可能是有内容的或空的，反正有内容也没被修改，可以直接用置换进来的页覆盖

优点：1. 批量写回。2. 有的页放入了空闲页链表中还未被覆盖，可以不用置换，直接找到，减少置换时间

页在内存中不会物理性移动，插入链表的是页表项，置换的时候修改页表项





驻留集管理

- 固定分配
- 可变分配

- 局部置换：只能换出属于该进程的页
- 全局置换

进程在虚拟时间t，驻留集表示**现在内存中**的页的集合，工作集W（t, Δ），表示该进程在**过去**的Δ个虚拟时间单位**被访问到**的页集合

过去一段时间可能多次访问同一个页，工作集大小≤窗口大小

工作集大小的快速变化和稳定阶段交替出现

根据工作集确定驻留集的大小=>简化：用缺页率来判断，缺页率大于某个阈值，则增加驻留集

$平均访问时间 = (1-p) * m_a + p * d_a$，p为缺页率，内存的访问时间为ma，发生缺页时的访问时间为da（中断服务+写出+调入+访存）

有效访问时间：从地址发出指定逻辑地址的访问请求到进程取出指令或数据所花费的时间

清除策略

按需清除，只有当一页被选择用于置换时才被写回辅存

预清除

使用页缓冲是比较好的混合策略

加载控制

多道程序度：驻留在内存中的进程数量

随着并发度的增加，CPU利用率先上升，后下降

- L=S准则：发生缺页的平均时间L等于处理缺页故障的平均时间S，此时处理器的利用率最大
- 监测clock算法的时针转动速度

T：地址转换易错点

一般假设按字节编址

！！先确定地址的位数和划分，从低到高划分

❖ 虚存分页系统地址转换示例

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
对应的物理地址为？

答案：125C

转换成二进制通常会写成16位，如果取高四位当做页号那就错了！

每页1KB，页内偏移量10位，0010才是页号

养成从后往前数的习惯，前面的0是可以在进制转换的时候补上的

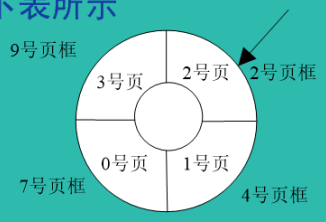
！！分段注意检查偏移量是否大于段长

T：综合题

❖ 驻留集管理示例示例

设某计算机的逻辑地址空间和物理地址空间均为64KB，按字节编址。若某进程最多需要6页存储空间，页的大小为1KB。操作系统采用固定分配局部置换为此进程分配4个页框（Page Frame），如下表所示

页号	页框号	装入时刻	访问位
0	7	130	1
1	4	230	1
2	2	200	1
3	9	160	1



若该进程执行到260时刻时，要访问逻辑地址为17CAH的数据，请回答

(1) 该逻辑地址对应的页号是多少？

(2) 若采用先进先出(FIFO)置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程。

(3) 若采用时钟(CLOCK)置换算法，该逻辑地址对应的物理地址是多少？要求给出计算过程（设搜索下一页的指针沿着顺时针方向移动，且当前指向2号页框，如上图所示）

寻址（地址转换和寻址流程）+置换策略

注意是装入时刻，越小留的越久，不是装入时长

！！驻留集大小固定为2，访问页号为1，发生缺页时要置换，把换入的页放入CA2H页框，修改页表项为0 - 1；1 CA2H 0

！！访问流程，缺页之后是回到第一步重新走一遍的

IO管理

IO设备

IO控制

程序IO

编程方式一般是写循环来实现的，所以是忙等，CPU不断查询IO状态

中断IO

不用一直查询，中断来了才处理

CPU向IO模块发出IO命令，如果是非阻塞的IO指令，则CPU继续执行后续指令，如果是阻塞的IO指令，则将当前进程设置为阻塞态并调度其他进程

DMA方式

CPU把控制权交给DMA控制器，CPU只在启动和结束时进行处理

DMA与中断IO的区别

- 中断频率
- 传送单位：中断方式一次只传输一个单位（如字节、字符、字）的数据，DMA方式一次传输一块数据

通道控制方式

IO通道有自己独立的指令集、处理器和存储器，本身就是一台小型计算机

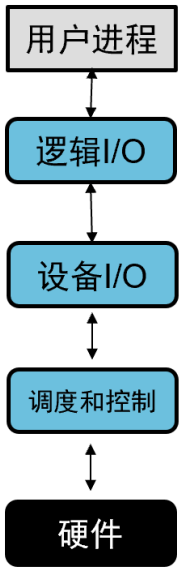
实现 CPU、通道和 I/O 设备三者的并行操作

IO组织

IO设计的目标：效率（与内存和处理器相比，大多数I/O设备的速度都很低）和通用性

I/O功能组织模型

- 逻辑I/O：把设备当作逻辑资源处理，不关心控制设备的细节。
- 设备I/O：请求的操作和数据被转换成对应的I/O指令、通道指令和控制器指令，并可以使用缓冲提高效率。
- 调度和控制：与硬件真正发生交互的软件层。处理中断，收集和报告I/O状态。



设备独立性/无关性：应用程序独立于具体使用的物理设备

IO缓冲

块设备：以块传输，按块号访问，eg，磁盘

流设备：以字节流传输，eg，打印机，鼠标，终端

单缓冲

预读

每块数据的处理时间为：Max(T,C)+M

双缓冲

Max(T, M+C) ≈Max(T, C)

循环缓冲

希望I/O操作跟上进程执行速度时，使用循环缓冲

缓冲的作用：

- 缓解I/O设备速度与CPU速度不匹配的矛盾
- 减少IO传输对CPU的中断频率
- 在多道程序环境中，当存在多种I/O活动和多种进程活动时，缓冲可以提高CPU和IO设备的并行性，提高单个进程的性能

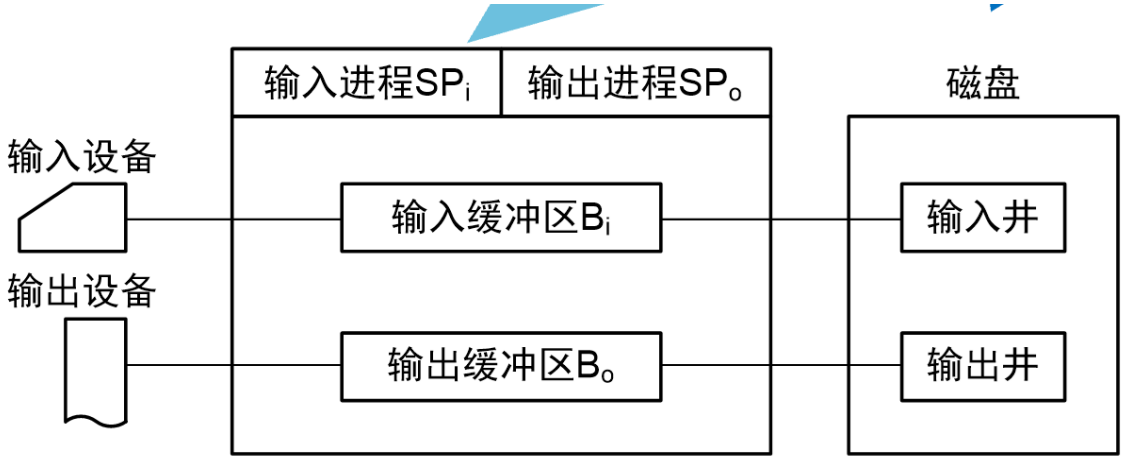
spooling技术

虚拟设备：通过虚拟技术将一台独占物理IO设备虚拟为多台逻辑IO设备，从而允许多个用户共享一台物理I/O设备。

内存可能放不下同时从不同设备输入的很多数据，就先存在磁盘（虚拟）中，再依次输出

实现共享的同时还可以提高IO的速度，把井也看作缓冲区，对低速I/O设备进行的操作，演变为对输入/输出井中数据的存取。

SPOOLing系统的组成：下图中除输入输出设备以外的部分



磁盘

磁道：一个圆环；扇区；不同盘片的相同磁道称为柱面

访问时间

T: 访问时间=（磁头）寻道时间T_s +（盘）旋转延迟T_r，平均1/2r + 数据传输时间$T_t = b/rN$

也有磁头固定的磁盘，每个磁道有一个磁头

！！文件按扇区顺序存放，且放在相邻的磁道上，第一个扇区以后就没有寻道时间，读每个磁道之前加上一个旋转延迟即可；随机存放，则每个扇区都要寻道，要有旋转延迟（换磁道的时候的旋转等待时间还是算上了）

不能跨越磁道或扇区，就要单独算每个道/扇区能存几条记录

- 磁盘访问时间比较

  考虑一个典型的磁盘，平均寻道时间为4ms，转速为7500r/m，每个磁道有500个扇区，每个扇区有512个字节。假设有一个文件存放在2500个扇区上，估算下列两种情况下读取该文件需要的时间。

  （1）2500个扇区分别位于5个相邻磁道上，且文件按扇区顺序存放；

  （2）2500个扇区随机分布。

T_s=4ms; r = 7500r/m = 0.125r/ms;

T_r =1/(2r)=1/(2*0.125) = 4ms

T_t = b/(rN) = (500*512)/(0.125*500*512) = 8ms (文件按扇区顺序存放时，每个磁道数据的传输时间)

T_t = b/(rN) = 512/(0.125*500*512) =0.016ms (文件2500个扇区随机分布时，每个扇区数据的传输时间)

放在相邻的磁道上，第二次以后就没有寻道时间

（1）（4+4+8）+ 4*（4+8）= 64ms=0.064s

(2) 2500*(4+4+0.016)=20040ms=20.04s

数据连续存储，有利于提高存取效率。

磁盘调度

名称	说明	注释
根据请求者（进程）选择		
随机	随机调度	用于分析和模拟
FIFO	先进先出，根据请求的先后顺序	最公平的调度
PRI	进程优先级	属于磁盘管理之外
LIFO	后进先出，优先处理新来的请求	局部性最好，资源利用率最高，但可能会饥饿
根据被访问的磁道选择		
SSTF	最短服务时间优先，选择磁道离当前最近的IO请求	利用率高，队列小
SCAN（电梯）	在磁盘上往复，磁头只沿着一个方向移动，在移动途中满足所有未完成的请求 Look(最后一个请求，不必到达最里/外)	服务分布比较好，最里/外的对局部性利用较好
C-SCAN	单向，快速返回	服务变化较低
N-Step-SCAN	避免磁头臂黏着，每次只针对长度为N的子序列处理	服务保证
FSCAN	N-Step-SCAN，N等于扫描开始时的队	负载敏感

RAID

三个特性

- RAID 是一组物理磁盘驱动器，操作系统把它视为单个逻辑驱动器（典型的虚拟设备）
- 条带化
- 使用冗余，奇偶校验信息等来恢复数据

disk cache

磁盘高速缓存，位于内存，包含了磁盘某些扇区的副本

置换策略

LRU算法

LFU算法

被访问后计数+1，换出访问次数最少的

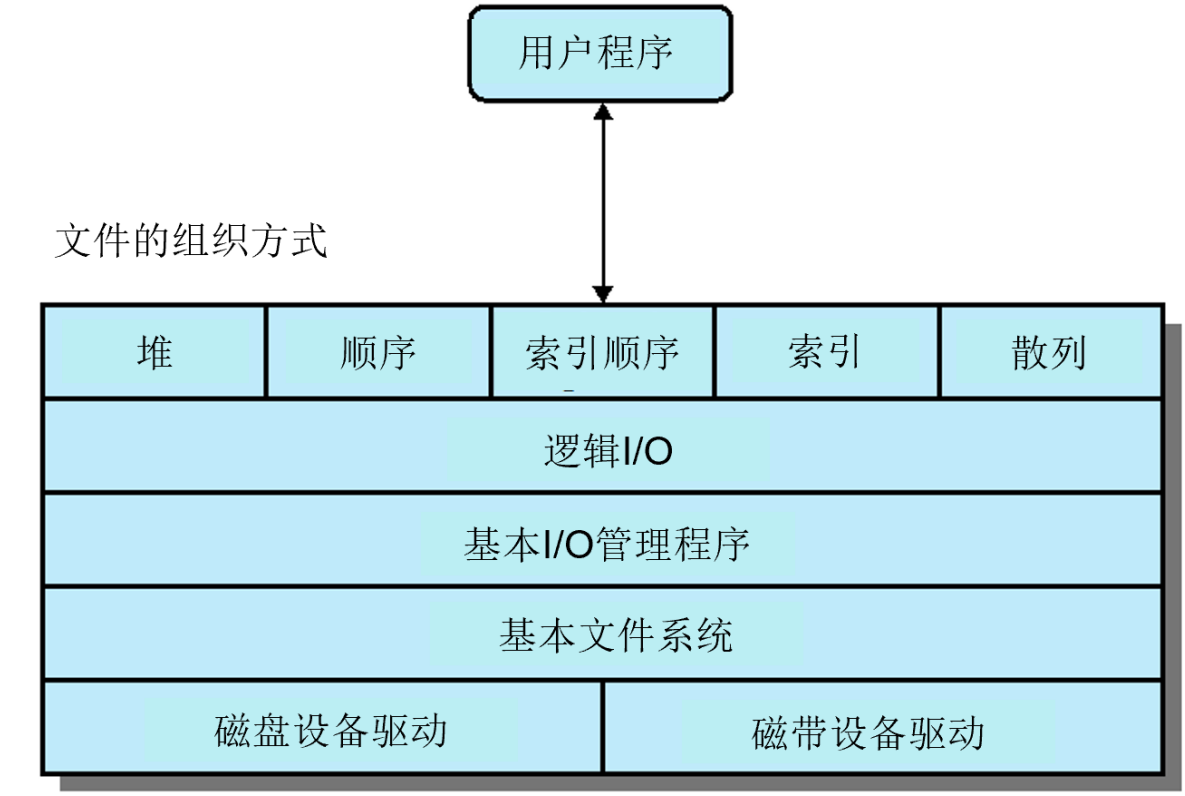
进来时间长的被访问的次数很多，可能一直不会被换出去

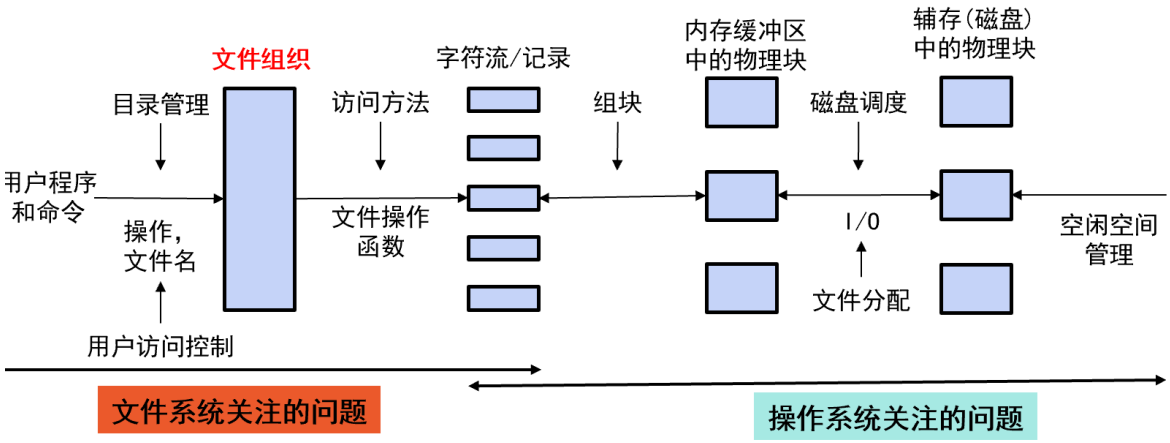
解决：划分新区和老区，或者没被访问的-1

文件系统

域，记录，文件，数据库

分层架构





文件组织

原则：快速访问；易于修改；节约存储空间；维护简单；可靠性

堆文件：按照到达的顺序收集，每条记录的格式、长度不定，通过穷举检索

顺序文件：类似数据库中的表，记录有固定的格式（域），按关键域（主键）存储和排序

索引顺序文件：根据关键域（主键）来建立索引，索引可以有多级；溢出文件：每条插入开销较大，要往文件中插入记录时，可以先将其放在溢出文件中，并由主文件中它的前一个记录通过指针指向，采用批处理合并

索引文件：不只能根据关键域建立索引，还可以为其他可能成为查找条件的域建立索引。完全/部分索引

散列或直接文件

文件目录

单级结构：查找速度慢，不允许重名

两级目录：主目录和用户目录

树状结构

无循环图结构：在树型目录的基础上，允许多个目录项指向同一个数据文件或者目录文件

组块分配

定长/变长

跨越（磁盘）式/非跨越式

辅存管理

文件分配

块：一组连续的扇区，≥扇区（簇也是一组连续的扇区，不过是在大文件存储的背景下提出的）

分区：一组连续已分配的块，≥块

卷：磁盘上的逻辑分区，一组在辅存上可寻址的扇区的集合，逻辑上连续，物理上不需要连续

预分配OR动态分配，预分配要求在文件创建时声明文件的最大尺寸

	连续	链式	索引	
是否预分配	需要	可能	可能	
分区大小是固定还是可变	可变	固定块	固定块	可变
分区大小	大	小	小	中
分配频率	一次	低到高	高	低
分配需要的时间	中	长	短	中
文件分配表的大小	一个表项	一个表项	大	中

连续分配

缺点：外部碎片->紧凑；不利于文件长度的动态调整

优点：容易查找，起始地址+长度；适合顺序文件

链式分配

- 隐式：每个块都包含指向下一个块的指针

  缺点：局部性原理不再适用，难查找，每次都要寻道和旋转->周期性地合并

- 显式：用于链接文件各物理块的指针，显式地存放在文件分配表FAT中，该表在整个磁盘分区中仅一张，如果扇区太多，可以不以扇区而以簇（一组连续的扇区Cluster=2^n Sectors，其实和块的概念一样，不过这里叫簇）为基本分配单位

索引分配

- 基于块，用单独的索引块存放所有指向文件中的其他块的指针
- 基于长度可变分区，指向的可能不是一个块，而是连续的分区

支持顺序访问和随机访问，使用最普遍

一个索引块容纳不下，引入多级索引

空闲空间管理

文件分配表和磁盘分配表

位示图

位示图大小，bit数=块数，换算成字节÷8

位示图过大，查找时间较长

链接空闲分区

使用指向每个空闲区的指针和它们的长度值，可将空闲区链接在一起

索引

将空闲空间视为一个文件，并使用索引表为文件分配空间，基于可变长度而不是块来建立索引

空闲块列表

可将表的一小部分存在内存中

Unix文件系统

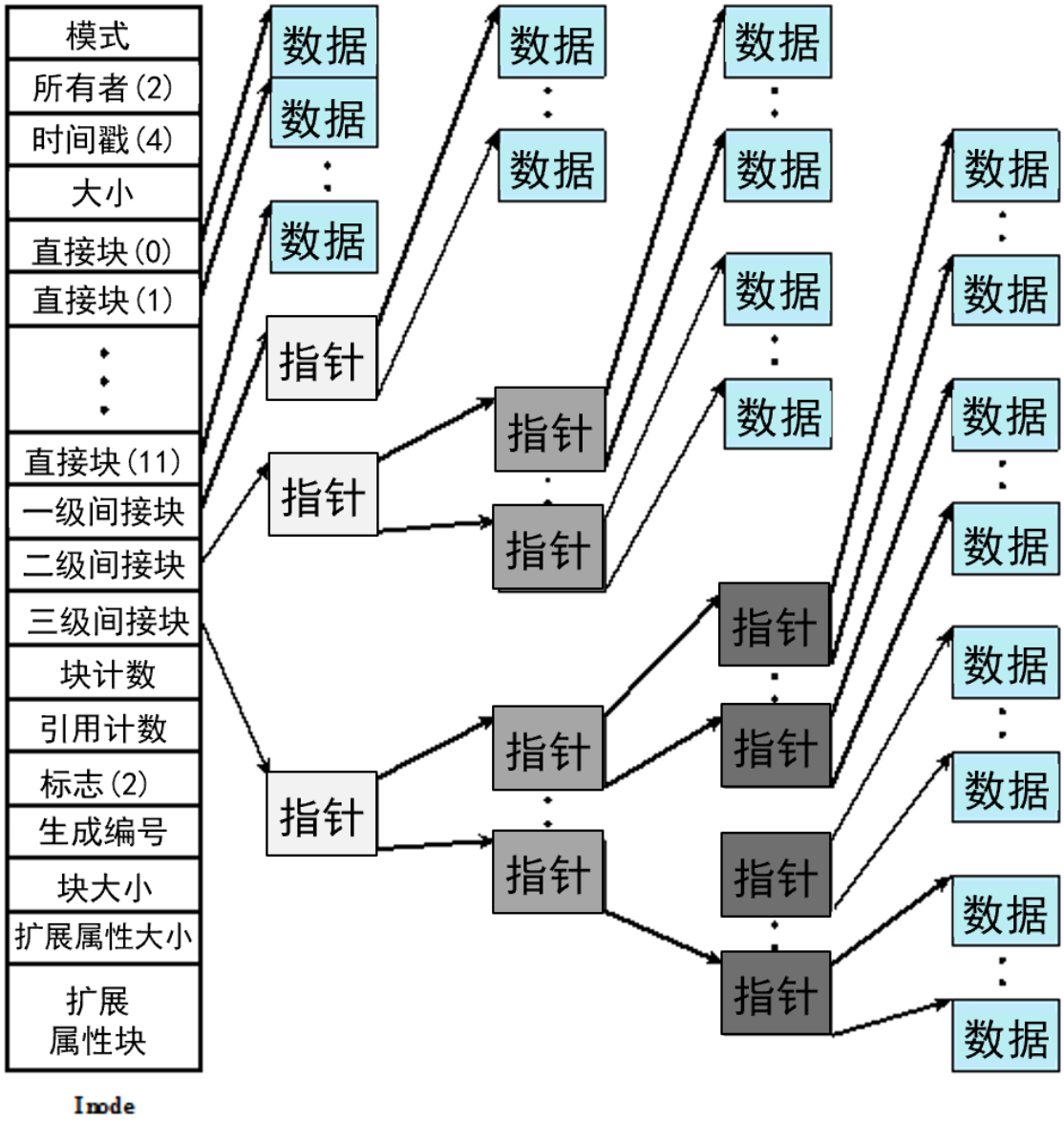
基于块，动态分配

目录结构是无循环图结构

inode

为什么要设置inode？为了减小目录文件的大小，目录项中只需要指向inode

引用计数，有几个目录项指向这个inode



混合索引，含直接索引13个，一级、二级、三级间接索引各一个

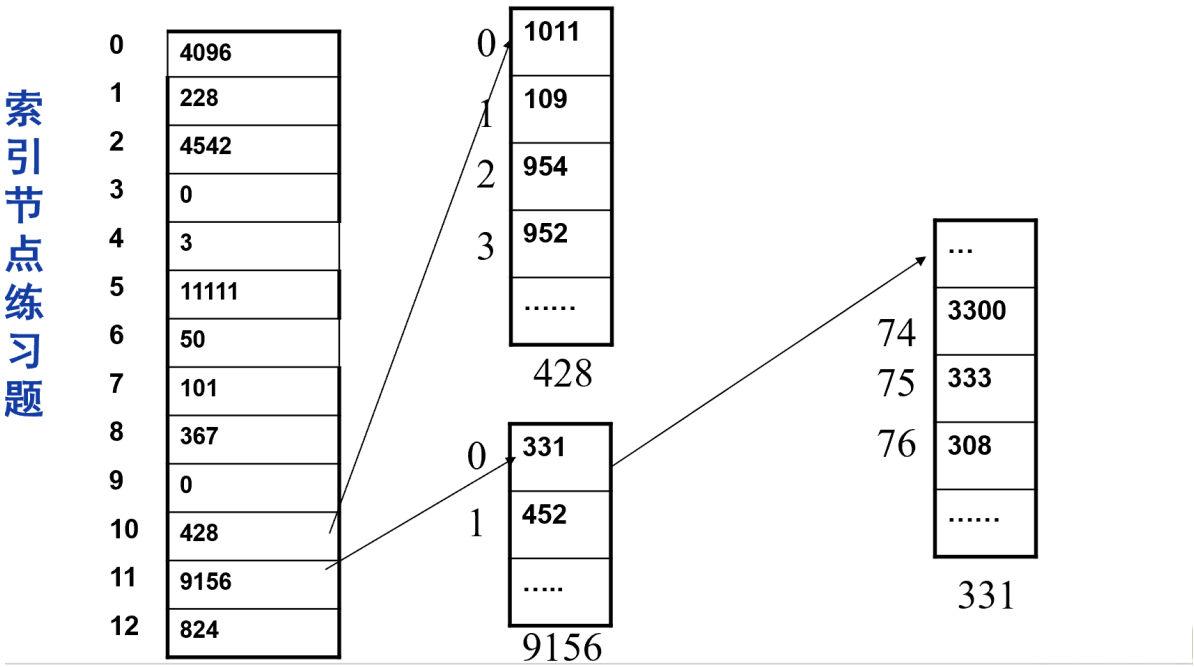
每块能存储的地址项数=块大小/地址项大小

T：将文件中的字节偏移量转换为物理地址

转换过程类似多级分页的地址转换

假定磁盘块的大小为1KB，每个盘块号占4个字节，文件索引节点中的磁盘地址明细表如图所示，如何将下列文件的字节偏移量转换为物理地址？

1． 9000　　2． 14000　　3． 350000



目录项的删除

- 若对应的文件只有一个引用时，同时删除该文件。
- 若对应的文件存在多个引用时，只删除目录项，而不删除文件，引用计数-1。只有在所有文件引用都被删除后才删除文件。

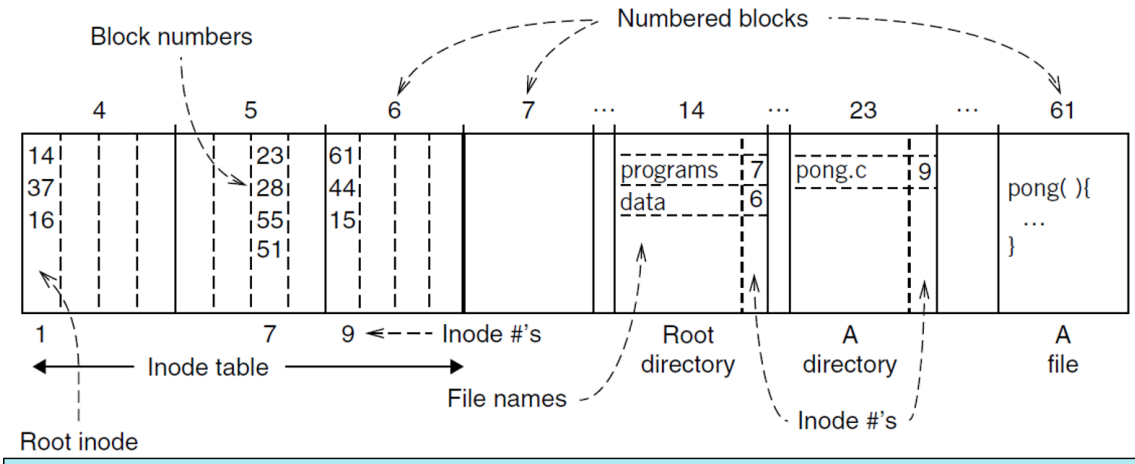
UNIX文件系统驻留在单个逻辑磁盘或磁盘分区，所有路径的起始都是根目录，不像windows分CDE盘

引导块|超级块|inode table|数据块

目录项 文件名|inode编号（inode在inode table中的位置）

目录即文件，第一个inode记录了根目录的存储位置

UNIX中的路径解析——/programs/pong.c



1. 找到索引节点表里第1个表项，记录了根目录文件的存储位置，第一个块为14块
2. 找到14块，看到programs文件的inode号为7
3. 在索引节点表里找第7个表项，记录了programs目录文件的存放位置，第一个块为23块
4. 找到23块，看到pong.c文件的inode号为9
5. 在索引节点表里找第9个表项，记录了pong.c文件的存放位置，第一个块为61块
6. 依次访问61，44，15块，可得到Pong.c文件的内容

注意区分块号和inode号，目录项里记录的是inode号，inode里记录的是块号

根inode->目录块号->文件inode号->文件块号

文件共享

硬链接：多个文件名链接到同一个索引节点

链接文件（目录项）和被链接的文件必须在同一个文件系统中

不能建立指向目录的硬链接（容易形成循环）

```
ln a b     //a, b链接到同一个索引节点
```

软链接(符号)

类似快捷方式，指向inode，指向写着路径的文件

可以跨盘、跨主机

允许对目录建立

```
ln -s a b  //为a创建一个符号链接b
```

删除a，b还存在，不过路径变成无效的了，删除b，a不受影响

对打开文件的管理

UNIX的文件系统中打开文件结构由以下三部分组成：

(1)进程打开文件表。每个进程都有一个进程打开文件表，其中每一项是一个指针，指向系统打开文件表。

(2)系统打开文件表。系统打开文件表也叫打开文件控制块。进程打开的文件都会在系统打开文件表中创建一个表项，其中主要包含：

・f-count：指向该系统打开文件表的进程数。

・f-inode：指向一个打开文件的内存i节点。

(3)内存i节点。其中主要包括：

・i-addr：文件在盘上的物理位置信息。

・i-count：与此内存i节点相连的系统打开文件表的个数。

不同用户对打开文件的共享只需将系统打开文件表中的指针f-inode指向同一个内存i节点即可。

Windows文件系统

分卷，基于簇，支持大文件，有可恢复性