操作系统:

基本概述

操作系统的两个基本功能;

第一部分: 控制硬件的程序, 主要控制三大硬件

IO 设备: 可以将 IO 设备统一当成文件,因为鼠标键盘的操作不过就是文件的读取

处理器: 进程的设计,使得程序间切换成为了可能,提高了 CUP 的工作效率

主存: 对所有程序而言,都好像独占内存,因为有神奇的内存映射-->虚拟地址空间.

总结: **文件对 IO 设备的抽象**,**进程是对程序的抽象**,**虚拟内存是对主存的抽象**.

第二部分 : 负责与用户交互

命令接口—允许用户直接使用

程序接口—允许用户通过程序使用

GUI – 图形用户接口

操作系统的特征

并发 :

宏观上**同时发生**, 微观上交替发生.

计算机上是同时**执行程序**,交替**执行进程**.

资源共享:

互斥共享: 比如写入资源就必须是互斥的.

同时共享: 微观上仍然是交替共享. 比如读取文件

虚拟:

虚拟内存: 计算机可同时运行内存大于实际**物理内存**的软件.结果就是用户觉得每个软件都能独占完整的物理内存. **空分复用**,

虚拟处理器: 利用进程,即使是单核处理器也能宏观上同时运行多个程序.**时分复用**.

异步:

异步是系统并发过程中,由于资源的互斥,导致进程走走停停,以一种无法预知的速度前进,导致的现象.

结论: 没有并发性,则共享性没有意义. 另外没有共享性,则并发性无法实现,应为处处受限,资源无法访问,仅能做一些计算性的工作. 仅有系统拥有了并发性,才可能导致异步性.

操作系统得到发展历程:

手工操作:

输入纸片. 缺点,单用户占用,输入时间长.CUP 资源利用率低

批处理系统: 单道批处理系统: 增加了磁带读写,相当于缓冲区,但资源利用率仍较低

多道批处理系统: 不提供人机交互,系统自己分配资源,资源利用率较高

分时操作系统: 优点:提供了人机交互,平均分配用户时间,但不利于优先处理事件

实时操作系统: 能优先处理紧急事件. 自动驾驶,导弹系统等

网络操作系统:

分布式操作系统:

个人计算机处理系统:

系统调用

系统调用主要负责对用户的交互. 涉及到调用的资源处理问题:打印机问题. 因此系统资源必须通过系统调用,由**系统分配**.不能由用户程序自己调用.

比如高级语言: 其中库函数中有系统调用指令,其用汇编语言中有一个重要的语句 int x, 表示 interrupt x, x 为编码,指定某种系统调用功能如进程的创建,文件的访问等. 从此程序从**用户态**转到运行在**核心态**.

系统调用分配 :

设备管理:

文件管理:

进程控制:

进程通信:

内存管理:

操作系统的内核:

内核: 计算机配置的底层软件是操作系统最核心最基本的部分

包括: **时钟管理**,**中断处理**,原语(执行到底),设备驱动,**进程管理**,存储器管理等.

大内核: 在微内核的基础添加 设备驱动,**进程管理**,存储器

微内核: **时钟管理**,**中断处理**,原语(cup 切换)等,

中断:

背景: 多到程序并发执行的**本质**. 中断可使 CPU 从用户态切换到核心态,使得操作系统获得计算机的控制权. 中断是唯一一种从用户态到核心态的**唯一**一种途径,而核心态到用户态仅需置某个标志位为 1 即可.

内中断: 程序的自愿中断如打印操作,或者异常(除/0).

外中断: 与当前 cpu 指令无关, 如外部打印机传来的中断信号.

进程:

组成:

程序段,数据段,PCB.

进程是: 系统进行**资源分配**和**调度**的一个独立单位.

PCB:

进程描述信息: 进程标识符 PID, 用户表示符 UID

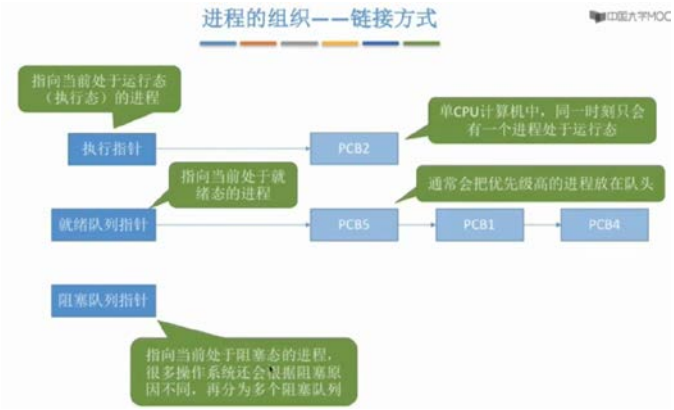
进程控制和管理信息: 进程优先级,进程当前状态

资源分配清单: 程序段指针,数据段指针,键盘,鼠标

处理机相关信息: 各种寄存器的值. 因此进程的切换需要保存变量信息.

进程的组织方式: 链接和索引

链接: 多个链表,每个链表都有相应的功能,执行,就绪,阻塞队列



进程的特征:

动态性: 进程动态地产生,变化和消亡

并发性: 多个进程实体,并发执行

独立性: 系统获取资源和调度的**基本单位**

异步性: 各进程按照自己独立的,**不可预知**的速度向前推进,(操作系统控制进程调用)

结构性: 每个进程都有一个 PCB, 程序段,数据段.

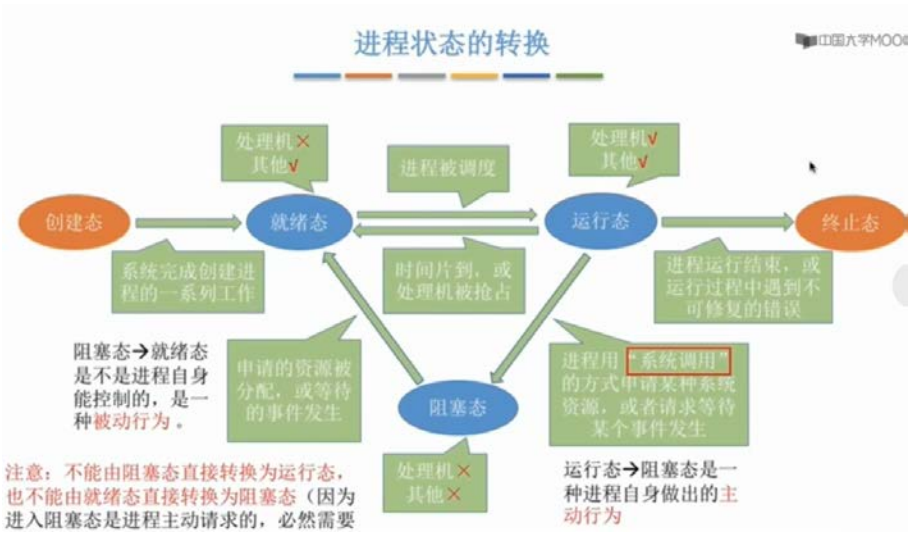
进程内部的状态转换:

三种状态: **运行态**,**就绪态**,**阻塞态.**

运行态: 单核 CPU 仅能一个进程运行.

就绪态: 已具备运行条件,等待空闲 cpu

阻塞态: 因等待某事件如 IO 操作的完成



进程控制:

原语实现的,一步到位. 目的是进程状态的转换. 原语运行在操作系统的核心态.

目标功能:

1. 更新 PCB 的信息;
    a) a 修改进程的标志状态,b
    b) 切换时保存 cpu 的运行环境.c
    c) 加载进程时,必须恢复 cpu 的原始运行环境
2. 将 PCB 插入合适的队列
3. 分配/回收资源

进程通信:

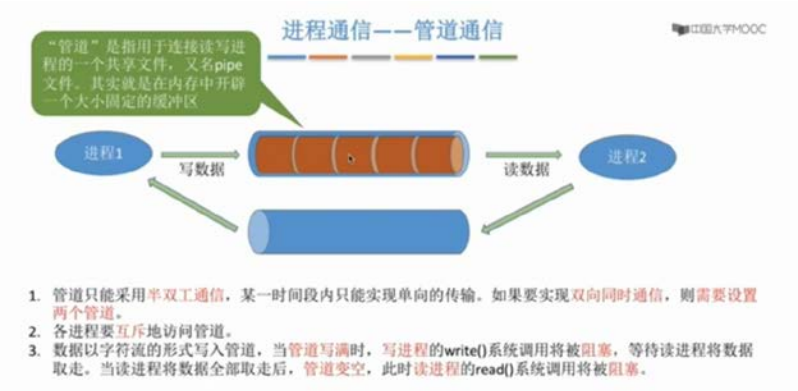
实现方法: **共享存储**,**消息传递**,**管道通信**.

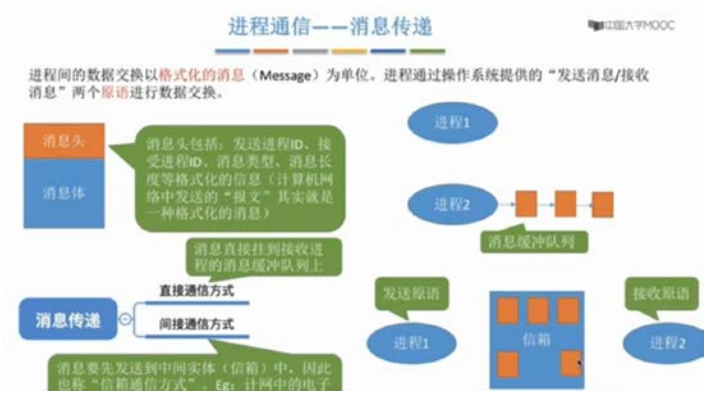
原理: 进程的地址(内存)空间不能直接访问. 相互独立. 想要访问必须通过接口

共享存储: 访问是互斥的,同时刻仅能一个进程访问. 进程可以决定共享区的大小,数据格式

管道通信: 一个特殊的共享空间,类似 swap,4kB, 半双工通信. 仍然互斥.

消息传递: 原语进行**数据交换**. 进程 1 将信息通过原语发送到进程 2 的消息缓冲队列.





线程:

线程是程序执行流的最小单位. 可以认为是轻量级的进程.进一步提高系统的**并发度**.

同一个进程内,线程并发执行. 注意系统资源是分配给进程而不是线程.

进程: 资源分配的基本单位, 线程: 调度的基本单位.

进程之间切换系统开销很大,线程之间切换开销较小

线程特征:

多 cpu 系统,线程可占有不同的 cpu

每个线程有线程 id, 线程控制块 TCP

线程也有 **就绪**,**阻塞**,**运行** 三种基本状态

同一进程内的线程共享进程资源.

由于线程之间共享内存地址空间,因此线程通信无序系统干预.

用户级线程: 不需要调用到核心态. 线程分配由用户程序决定,操作系统认为其为一个进程

优点: 线程切换在用户空间即可,不需要切换到核心态,开销小

缺点: 某个线程阻塞,则整个进程都会阻塞,并发度不高,不能在多核处理器上运行.

内核级线程: 线程切换由操作系统负责, 操作系统可将内核级线程分配到多核 cpu 上运行.但是用户级线程不能被操作系统分配到多核 cpu 上运行.

调度规则

先到先得,后到先得,优先队列(大根堆),

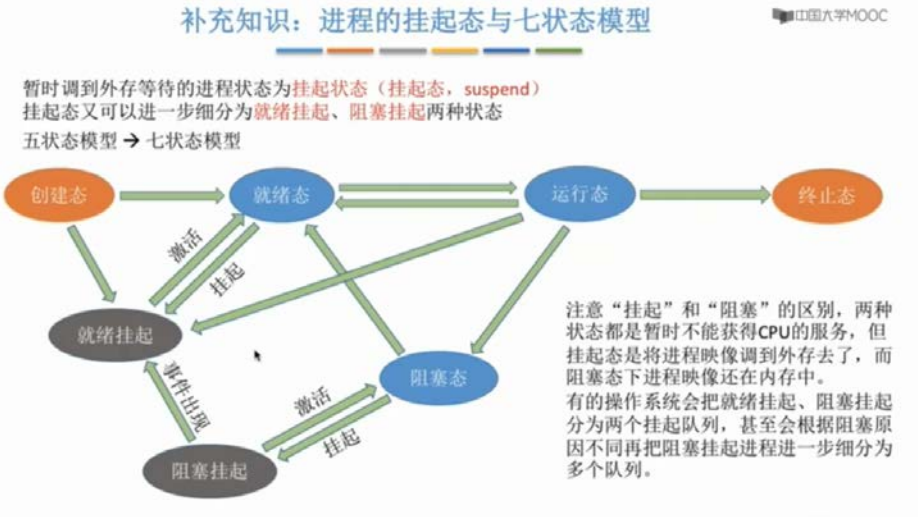
三种调度

高级调度(**作业调度**) : 何时从 外部存储设备 存到内存中

中级调度(**内存调度**): 将暂时不能运行的进程暂存在外设中,等具备运行条件后再重新存入.此时进程的状态为**挂起状态**,**PCB 仍保留**, 但是程序段和数据段会存在外设.

低级调度(**进程调度**): 最基本的调度.进程间调度.

七状态模型: 内存不够时, 就绪和阻塞态的进程会被挂起,放到外设中,关系如下:



	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度（作业调度）	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存（面向作业）	最低	无→创建态→就绪态
中级调度（内存调度）	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存（面向进程）	中等	挂起态→就绪态（阻塞挂起→阻塞态）
低级调度（进程调度）	按照某种规则，从就绪队列中选择一个进程为其分配处理机	内存→CPU	最高	就绪态→运行态

调度基本准则:

如果某进程的代码执行进入到内核态则不能执行进程调度,因此内核资源被锁定,会影响其他进程. 但是一般的临界资源还是可以的,比如打印机,重点是看资源的重要程度.

非剥夺调度方式: 进程仅能主动放弃,不能被抢占.

剥夺调度方式: 有优先级更高则抢占切换. 优先级可以是时间.因此可以研发出时间片的进程调度策略.

评价指标:

CPU 利用率:

= 忙碌时间/总时间

系统吞吐量:

=作业量/耗时 = 平均每秒完成多少作业

周转时间:

= 开始到结束的时间

也有平均周转时间 = mean(周转时间)

带权周转时间(主要反映了用户主观感受) = 周转时间 / 实际运行时间

等待时间: 周转时间-运行时间

响应时间:

调度算法:

正常有两种版本: 抢占式版本和非抢占式版本

先到先得 FCFS

主要从公平的角度考虑

短作业优先

非抢占式: 看哪个进程的程序段比较小. shortest job first

抢占式: short process first .每当有进程加入就绪队时,开始计算剩余时间,cpu 分配给剩余时间最短的进程.

短作业的良好表现在于减少了等待的进程个数, 比如让 50 个人等 1 分钟,则总体上的等待时间就很多,但是让最短的程序先运行,则能有效快速减少等待人数,因此此策略效率更好.

缺点是: 长进程可能饿死

高响应比优先:

响应比: = (等待时间+程序长度)/程序长度

解决了 sjf 的长进程饿死的情况.

时间片轮转调度算法(RR)

时间片(100ms) 强行剥夺,等待调度. 更加公平. 分配时间给进程,而不是线程

可抢占., 时钟中断.

优先级调度算法:

根据紧急程度决定处理顺序

多级反馈队列调度算法: (unix)

保证先处理短时的一级队列,

优点: 一开始进程进来时,优先级很高,都会执行,fcfs,相对比较公平. 对短进程也能较快运行完毕. 不必去估计进程的长短(有可能存在用户欺骗),此方法按运行时间来分配时间片.

缺点: 有可能导致饥饿.

进程同步:

异步: 进程之间独立,不可预知的速度前进

同步: 进程之间有先后顺序, 直接制约关系.

互斥: 临界资源的访问需要互斥的进行.

进程软件互斥的实现方法:

互斥原则: 空闲进入, 忙则等待,有限等待, **让权等待**.

单标志法：用一个全局变量设定 flag. 每个进程分配的 flag 不同,while(flag)只能有一个进程执行. 并且在进入之后,将 flag 还原.

缺点是进程仅能交互进行，若仅有单一进程,则永远无法访问.

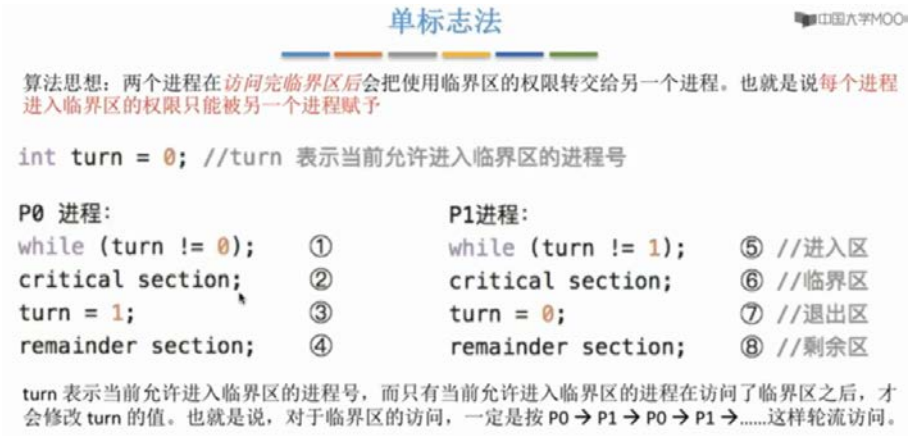
双标志检查法:

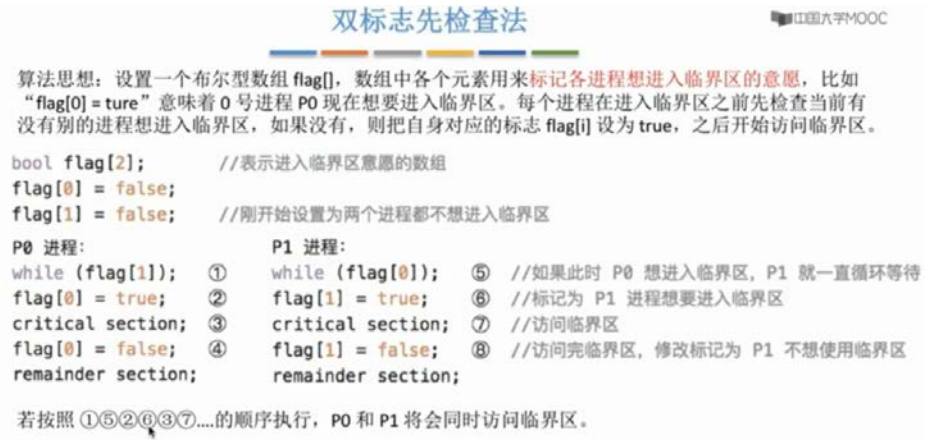
设置 flag 数组 flag[2]表示两个进程, flag[0]=true 表示第一个进程的意愿.

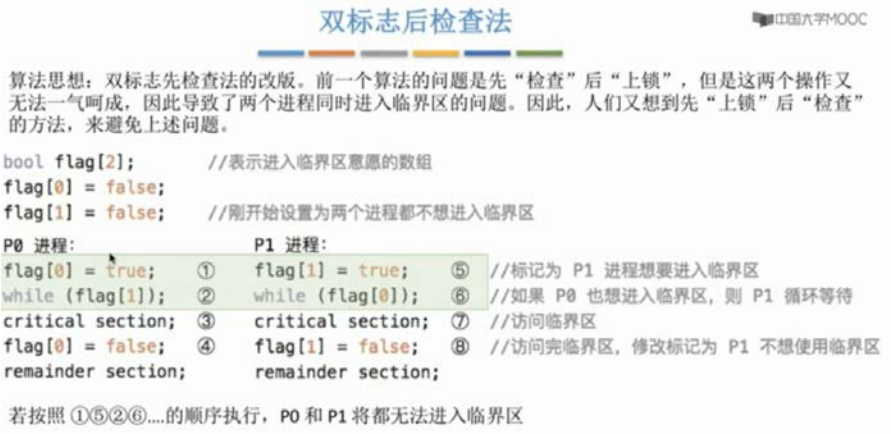
也就是只要有一个进程想用,那么当前进程就不能访问,算是单标志的加强版.而且不用交替运行. 缺点是由于进程的异步性，有可能 1 进程刚许可进入,还来不及置 true 就被切换,导致进程 0 又可以访问. 可以将语句 5,6 改为原子操作.可以改正这个缺点

双标志后检查法：之前是先检查,后上锁. 这个是先上锁后检查.但是仍然有可能饥饿

Peterson 算法：孔融让梨法. 结合双标志和单标志. 但默认先把控制权 turn 交给对方,然后看对方意愿 flag.是否为 true. 缺点是无法让权等待.







进程互斥硬件实现方法:

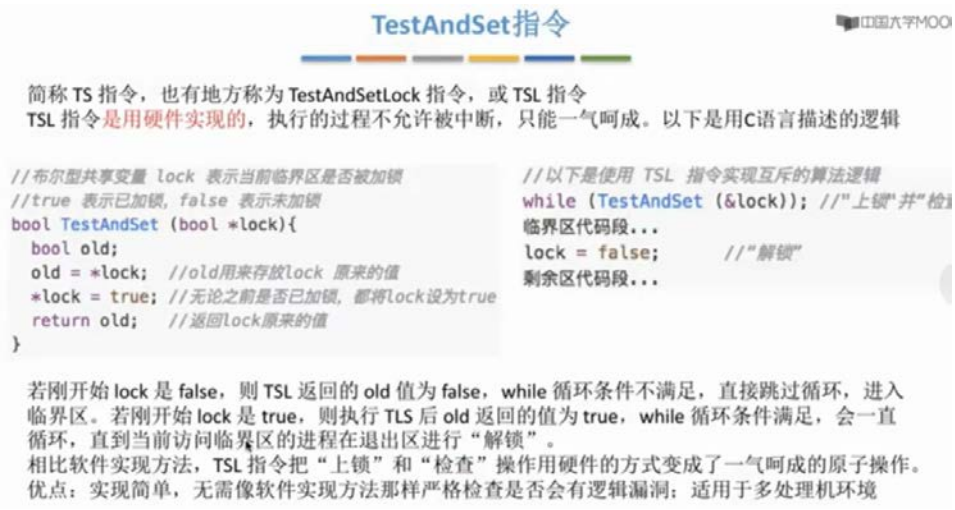
中断屏蔽算法：原语开关．不允许中断.

优点：简单,高效.

缺点：不适用于多处理机,因为中断仅适用于一个 cpu,另一个 cpu 可正常执行,有可能访问.必须进入内核态,比较危险.

TestAndSet 指令：硬件实现,一气呵成．先上锁后解锁，其中 testandset 硬件实现,且不可中断．但是不满足**让权等待**.另外 while true 也会忙等,cpu 无端消耗.

SWAP 指令：也不允许中断，交换两者的值 基本和 TSL 一样，由于是 flag 变量,因此适用于多处理机．但仍不满足让权等待.



信号量机制:

目标: 软件方法的一气呵成，以及让权等待.

原理: 为了一气呵成,使用操作系统提供的一对原语来对信号量进行操作.

P = wait = test. V=signal.=release

整形信号量:

缺点是不满足**让权等待**.

int a=1; 表示可用资源数.

test(a){while (a<=0);}

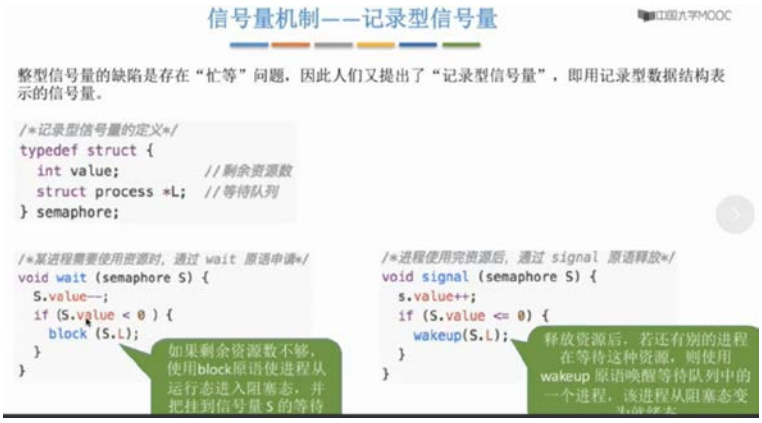
记录型信号量:

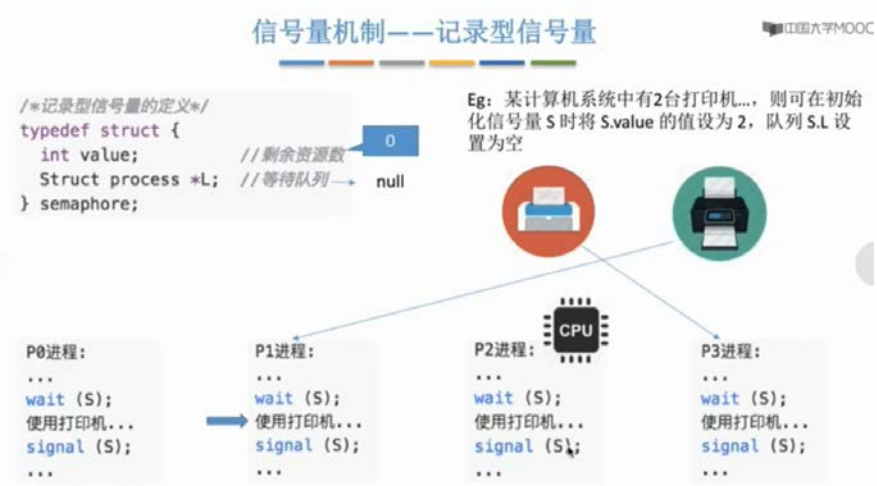
为了解决空转的问题 while 循环. 设定一个状态量.结构体 . 该结构体不仅存储 flag,还有一个是等待队列. 由于是全局变量,因此可控制多个处理器.

但其实 block 函数内部的实现机制也是 while 吧.

其中 wait 和 signal 函数都是**原语函数.** 不能中断.

有排队进程必然有 s<0; 因此如果发现 s++<=0.那么必然有等待队列,因此 pop L.





信号量机制实现互斥操作:

由于纯软件法有缺点，纯硬件法 无法实现**让权操作**.

于是通过信号量机制实现互斥操作. 由于有个等待队列.可用大根堆等方法实现优先队列,可以实现让权等待的功能.

这个跟软件实现很像,不同之处在于 wait 和 signal 操作都是原语，另外也不需要交替执行.

信号量机制实现进程同步:

进程前后关系的确定

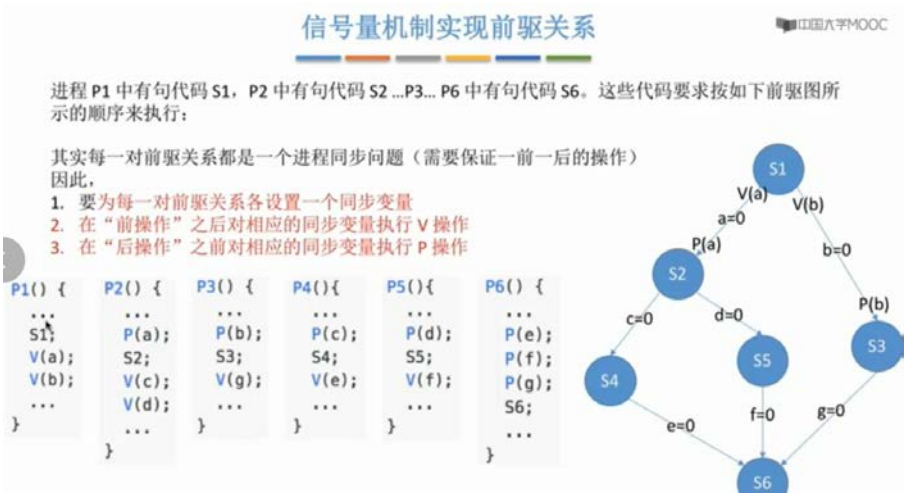
先设定 S=0，然后进程 A 执行完之后执行 V 操作 S++. 然后 B 操作一开始检查 P(S),等到 A 进程释放之后才能执行 B 进程一下的代码.

A(){ code1;code2;V(S);}

B(){P(S); code1;code2;}

注意：当 B 先执行时,会将 B 添加到等待队列，因此当 A 释放资源时,系统会主动 wakeup B **进程**!!

多重前驱设定：等待队列确实是不错的设计. wakeup 直接切换.



同步互斥综合应用

单生产者单消费者:

关系 1: 互斥 拿和取仅能一个进程访问 变量为 mutex

关系 2: 非满 才能存 empty = n >0

关系 3: 非空 才能拿 full = 0 >0 ; 一开始是空的.

注意相邻的 PV 操作不能调换，有可能会引起死锁.

比如 P{mutex};P{empty} 放入产品 V{mutex},V{full}; 如果资源慢了，进程 A 先锁住后发现 empty=0,然后唤醒 B 释放空间，但是 B 又被锁住,无法释放,因此造成死锁.

多生产者多消费者:

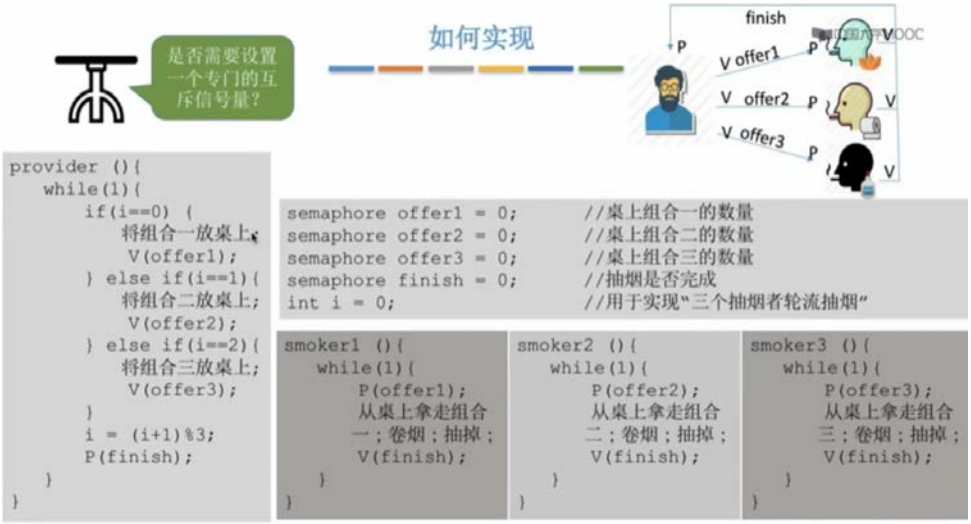
生产者 AB，对应消费者 CD，那资源就用 P 去测试，用完之后就释放 V 资源

关系 1: 仅有一个盘子，因此 AB 与 CD 互斥访问. mutex = 0; AB 用 P(mutex)检查

关系 2: A 供应 C，因此 A 测试 P(plate),释放 V(apple), C 拿 P(apple),释放 V(plate)

关系 3: B 供应 D，因此 B 释放 V(orange), C 用 P(orange) 拿 orange.

轮流吸烟问题:



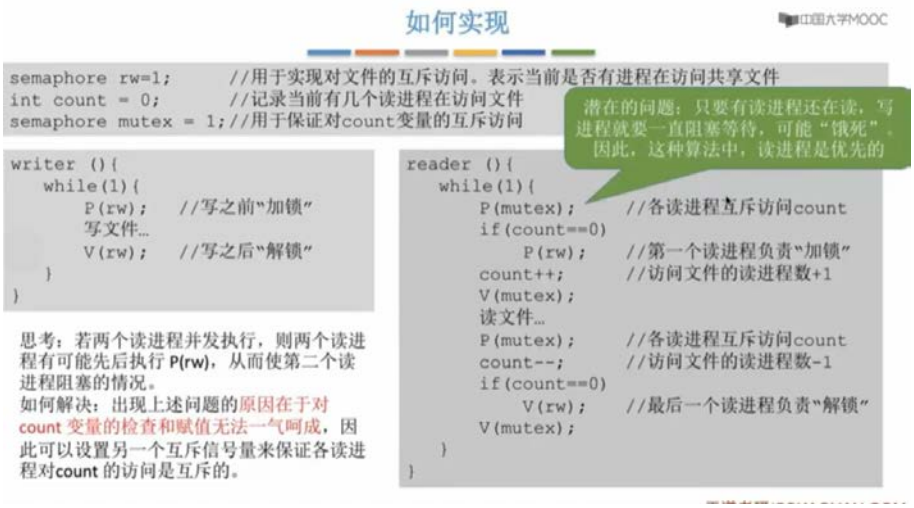
多读者,多写者操作的实现

互斥关系:

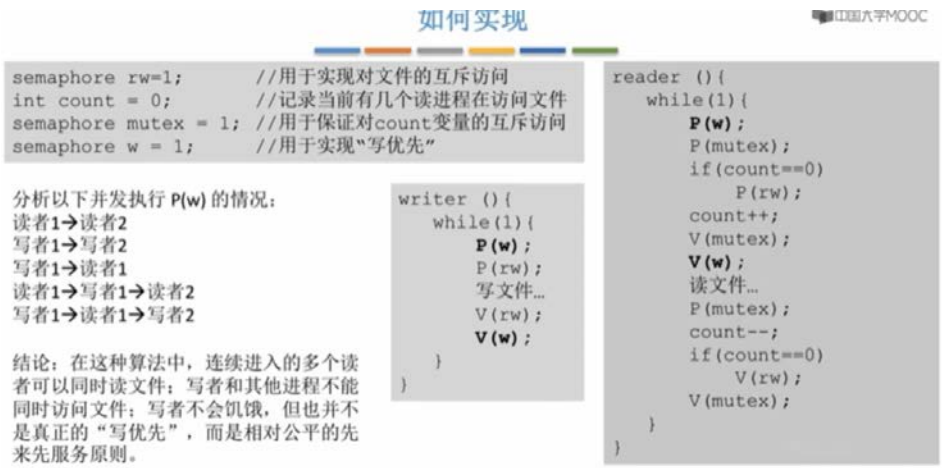
1 写进程之间互斥，写度进程之间互斥，读读进程之间**不互斥**.

该写法的问题在于 count 也有可能中断，导致 count 和 P 操作无法连贯执行，因此需要额外进行一个 PV 变量 mutex 使其保持互斥. 也就是说突然跳转到别的进程时,仍然不能访问 count 的值.

这样写法有个障碍是 读,写,读的顺序不能被保证,,会变成读读读写,因为读进程直接 count++



按照排队的写法，另外加一个 pv 操作变量 w. 作用是当有一个读时,后面的读不能超前写进程.

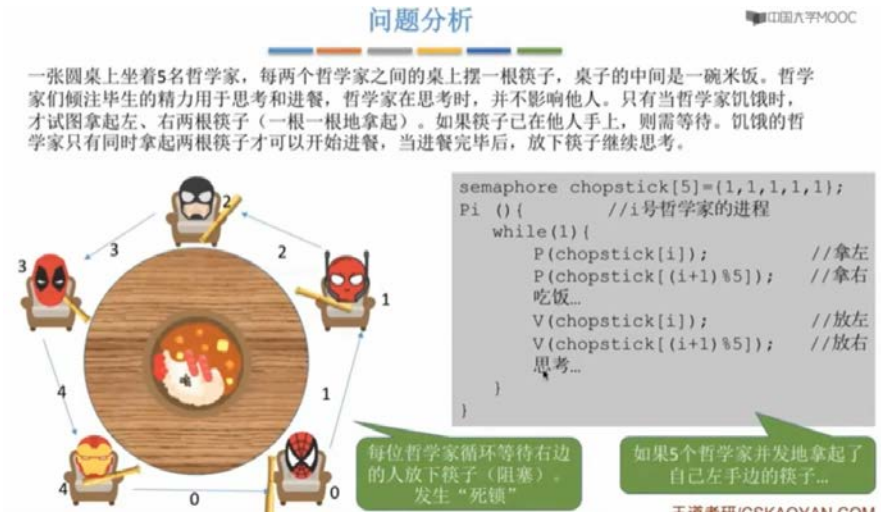


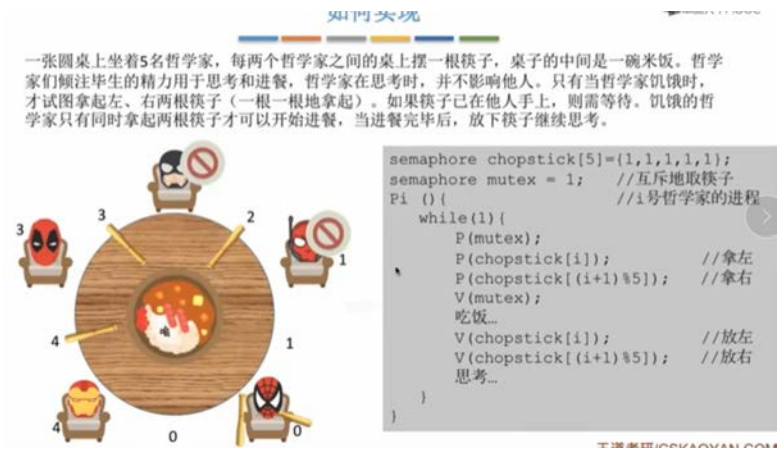
哲学家的筷子问题

忍让是无法原子操作造成的困扰

如果每个进程都仅执行第一句,那么会造成死锁现象.

因此加一个 mutex 互斥可以使得两个 P 语句顺利进行. 但此方法仅能保证有**一个人**吃饭,系统分配不合理,实际上可以两个人**吃饭**的. 但目前仅给出这种方案



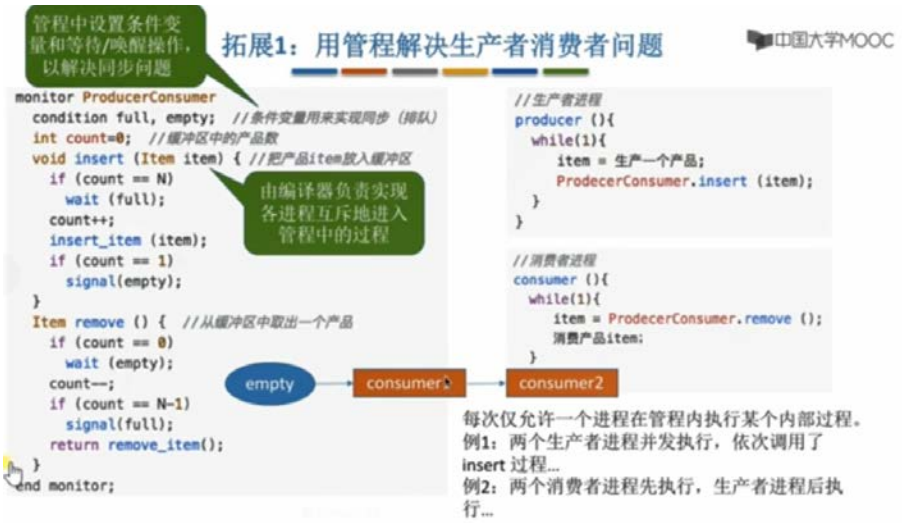


管程

PV 操作的另一种软件写法，更方便程序员使用.

管程的定义就像是 class ,进程就是 class 的实例，一次仅允许一个实例通过 class 提供的方法修改全局变量.

由编译器实现，内部具体**操作不明**.



死锁:

基本概念:

饥饿: 一个进程的饥饿. 长期得不到处理. 可能是阻塞态也有可能是**就绪态**.

死锁: 进程之间互相等待对方手里的资源而导致的. 多个进程之间发生死锁. 一定是**阻塞态**.

死循环: 代码逻辑错误导致的 ,比如 while (true),

死锁的必然前提:

1 互斥条件. 2 不会强制剥夺,只能主动释放. 3 **循环等待链.**

死锁的处理策略 :

1 破坏上述必要条件.2.代码层避免死锁,3 检测和解除

破坏 互斥条件:

SPOOLing 技术，进程访问资源 以一个队列的形式，按顺序执行. 这样就不用互斥访问了.

但应用场景不广

破坏 不剥夺原则:

方案一: 当资源不能就绪时,释放手里全部资源

方案二: 通过操作系统协助,优先级更高的可以剥夺

缺点是**实现复杂**,之前的工作会破坏,申请和释放的高频率操作降低系统吞吐量.

破坏 请求和保持条件:

请求和保持: 手里已经有一个资源, 然后提出新的资源请求,但是被阻塞,并且不释放自己的资源.

静态分配方法: 运行前,一次性全部申请所有资源, 如果不能够则不投入运行.

缺点: 实现简单. 如果能结合动态释放则效果应该不错.

破坏 循环等待:

编号方法: 破坏循环链, 规定大编号不能申请小编号的资源.

每个进程按编号递增的顺序请求资源, 只能申请更高的

原理: 因此只要进程的资源一定是不断往上升, 不会说突然需要小编号的情景. 因此总会有一个进程能够拿到最大编号的资源, 此进程能够顺利进行,最终释放.

缺点: 资源的编号不好分配, 可能有物理现实联系. 每个操作系统的编号可能不一样.

动态策略: 安全序列

安全序列一定不会发生死锁, 不安全序列不一定会发生死锁.

银行家算法:

进程总的需求资源	进程已有资源	进程请求资源	系统可用资源

1 检查 请求+已有是否>总的申请资源

2 检查 系统可用资源> 请求资源

3 尝试分配+ 启动安全算法:

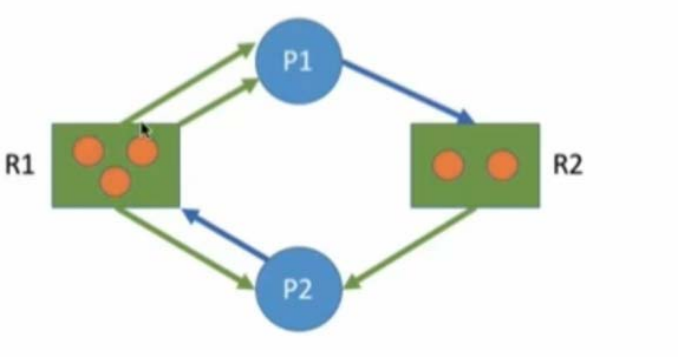
安全算法: 将可用资源以进程所需全部分配给某个进程(递归),然后收归该进程资源,迭代,看能不能完成所有进程. 若有一条递归路径能实现,则当前步骤安全. 步骤 3 可确认分配.

死锁检测和解除:

死锁的循环等待很适合用图来解决. 因此定义如下图:

然后不断消除完成进程相连的边,最终能完成清除所有节点,那么不会发生死锁.

总结:消除不阻塞进程的边, 直到无边可消.



死锁的解除:

1 资源剥夺法: 挂起进程,抢占资源.

2 撤销进程法: 撤销进程法, 代价比较大.

3 进程回退法, 要求操作系统记录历史信息,不容易实现.

可以按优先级剥夺进程资源.

优先级: 已运行时间, 资源的数量,剩余时间,进程优先级,交互程序(**用户体验**)或是批量处理

内存:

之前进程和调度规则都是 关于 CPU 处理器, 现在开始讲内存, 然后将 IO.

进程包括: **PCB**,**程序段**,**数据段**.

逻辑地址与物理地址:

编译时不是直接对变量赋予物理地址,而是放相对地址. 因此可解决物理地址不同这一问题.

三种方式 从逻辑地址到物理地址

绝对装入

编译器负责: 此时还没有操作系统

如果**已知起始点**,那么编译器其实可以直接写成物理地址. 灵活性很低.

静态重定位

装入模块属于操作系统

编译,链接后. **装入模块**(程序机器代码从外存到内存)负责将 全部地址转换为物理地址.. 装入程序由操作系统控制. 分配其全部地址空间, 并且程序在运行过程中不能再移动,也不能再申请空间

动态重定位

操作系统负责.

重定位模块 与 装入模块不同, 动态重定位将逻辑地址加载到内存, 然后送到 cpu 前,如果代码设计到地址,那么与重定位模块的起始值相加得到真实物理地址. 移动程序在内存的位置仅需设置重定位寄存器的起始位置即可.

链接操作:

程序如 c 语言-->编译.s 汇编,然后生成机器码.o,有一些系统调用模块.o ,将这些.o 文件链接起来装入内存.

静态链接

将各个模块的逻辑地址重新整合成一个逻辑地址. 第一个模块的结尾就是第二个模块的起始地址. 然后载入内存

装入时动态链接:

类似静态重定位. 装入模块负责 整合逻辑地址. 也相当于静态链接.

运行时动态链接:

需要时才将程序调入. 如遇到 printf 才将 printf.o 模块装入内存,动态进行链接.

内存管理

管理空闲区域, 程序装入哪里? 如何回收?

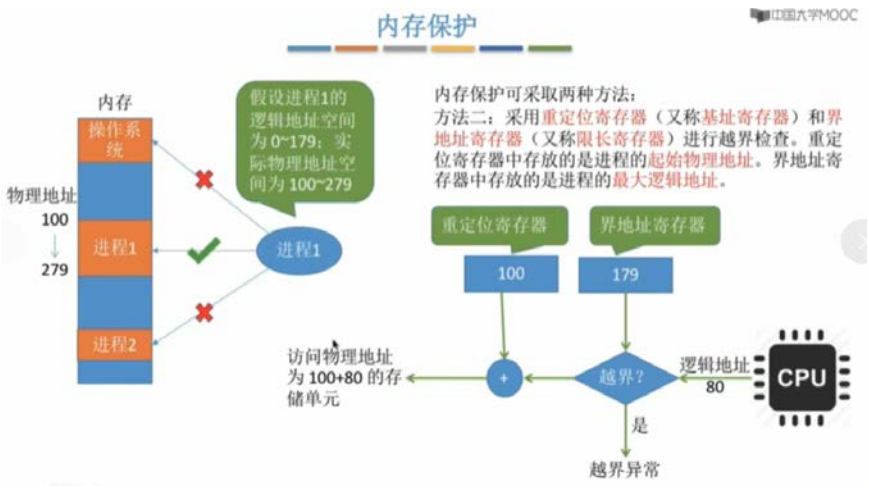
为编程方便, 仅需考虑逻辑地址, 至于逻辑地址与物理地址的映射关系式操作系统要管的事.

内存保护措施

方法一: 上下限寄存器, 保证访问不会超出这个范围.

方法二: 重定位寄存器加界地址寄存器(保存最大逻辑地址)

因此从 cpu 出来访问的逻辑地址,判断有无超过程序逻辑所需,如果正确,那么再叠加重定位地址进行访问.

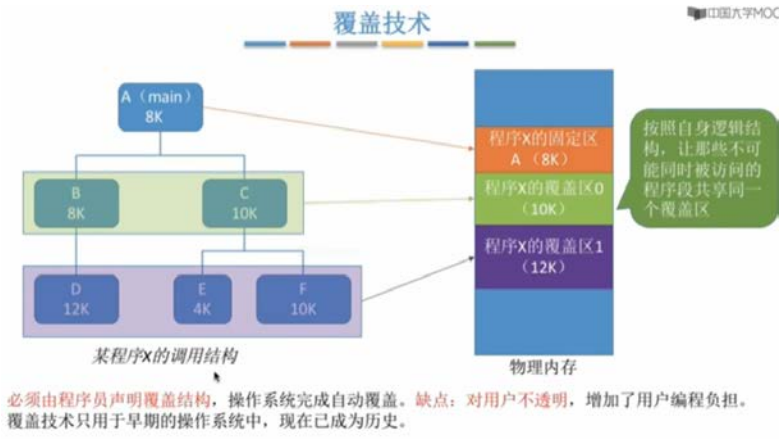


内存扩充:

覆盖技术

由**程序员负责**, 常用程序片段存在内存固定区,不常用的存在覆盖区.需要时再载入.

层级结构, 如果两条支路不可能同时执行,那么可以将这两条支路加入覆盖区.



交换技术

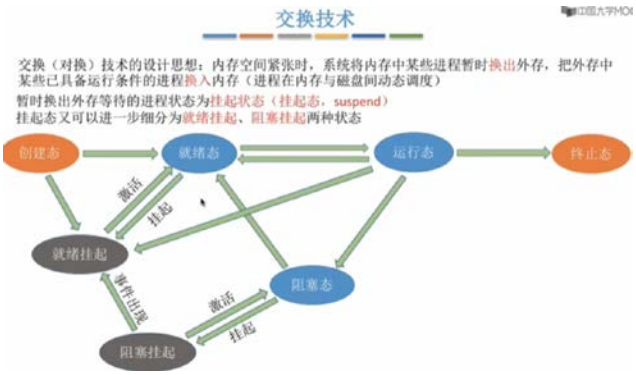
将某些进程转入外存,但是 PCB 仍留下,同时 PCB 记录外存位置. 此进程变为**挂起状态**.

挂起进程规则可以是优先级,内存驻留时间等作为优先队列.

对磁盘而言,可以划分出一个 swap 区(linux) 专门负责交换.对换区一般是连续分配—索引速度更快. 但注意 swap 分区是**页面组织**的形式,不是直接放内存,这样也方便页面同一调度(页面会在下

面讲到). 另外 swap 空间有限(128MB 或 2GB),不能装下 GTA,因此 swap 区主要存储动态数据,原始数据实际上可以直接从 exe 文件里读取,也就是外存里读取. 因此内存放常用进程,不够时放一些到 swap, 一次性数据可以从 exe 文件外设读取.

另外 swap 区是连续分配的, 而文件区是离散分配.但两者都是外存.



连续分配管理

单一连续分配: (非常早期)

二元化. 系统区,用户区. 实现很简单,但是仅能**运行一个程序**. 内存利用率低.

固定分区分配:

分区大小相等, 等大小分割内存. 缺乏灵活性,有时还需要合并等操作.内存浪费.

分区大小不等, 比较有灵活性.

利用表格记录占用信息.

动态分区分配:

根据进程动态分配大小.. 很 6.

记录方法:

空闲分区表.记录大小,起始位置.**状态**.

空闲分区链，链表的表头记录表大小和下一个空闲分区的地址,相当于用链表记录表格.

进程的地址分配：动态分析算法.....

内部碎片：空间太大,进程分配后导致浪费

外部碎片：空间太小，进程根本不会选这个空间.造成浪费.

动态分区算法:

首次适应算法:

每次从**最小地址**开始找，对空闲分区表开始搜索.适合就使用.可以一开始就对表进行排序

前端会留下外部碎片,导致搜索时间上升..但是后端会有大空间. **全场最佳!!!**.

算法开销小,因为不需要重新排序.

最佳适应算法:

默认**留下大空间**. 可以对表格进行容量排序，**默认拿下小的分区**.

缺点是会留下很多难以利用的外部碎片.

最坏适应算法:

完全和最佳适应算法相反. 默认拿下最大的.

由于动态划分大小,因此外部碎片会比较少,但是大进程不好搞.

邻近适应算法:

在首次适应算法的基础上，记录上次搜索的地址,按此地址往后查找，适合即放.

但是后端的大分区也会被分解.

分页存储:非连续分配

连续分配方式：固定分区会造成内部碎片. 动态分区又会造成外部碎片. 因此考虑用分页存储的方式来解决.

每个页框 4KB. 以页框为单位为进程分配. 难点在于逻辑地址与物理地址的转换，也就是记得进程的各部分被分配到哪里.

使用表格记录该进程每页的号码.内存中以 4KB 间隔顺序排放,知道序号就知道起始位置了.

计算程序偏移:比如每页 50 行,则逻辑地址 80,相当于第二页的第 30 行指令.

操作

1. 知道逻辑地址页号(程序内部的顺序)
2. 知道偏移量
3. 知道页号在内存的页面.(内存的页面顺序)

4 通过页面和偏移量直接索引程序.

页表寄存器(寄存器) PTR 记录 页表在内存的起始地址和页表长度 M. 放入 PCB 中.

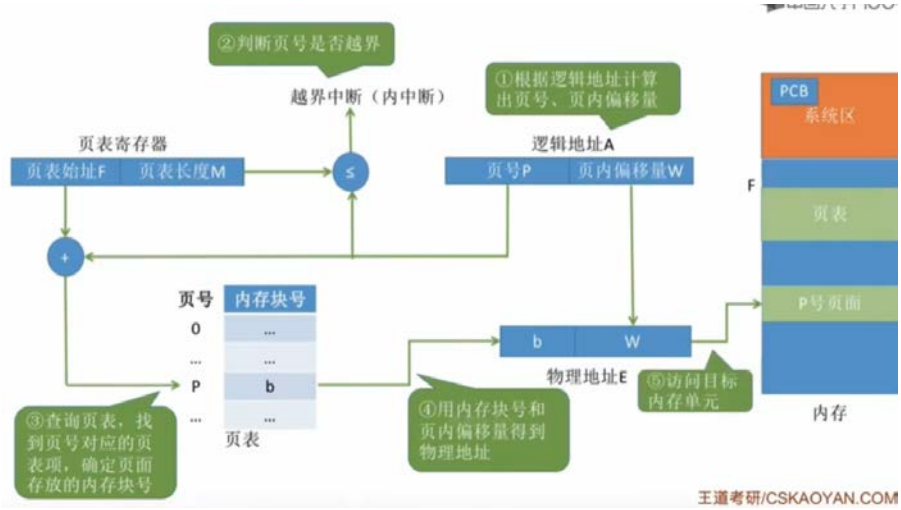
同时 PCB 也有**页表**.(表格) 如图:.M 表示最多有几个页面.

页号是程序逻辑页号. 可通过(逻辑地址//size)得到页号. size 一般为 4KB

页面是内存物理页面.

页表是页号和页面的联系, 一个线性表格. 索引页也很简单.不多说了.

页表寄存器 仅仅是辅助作用,查看是否越位等.



有些基础工作的是 页面线性表的索引, 页号大小为 M, 则一般需要 log2(M)位来存储. 可能为 3 字节. 如果**页表太大**, 某个页框全部用来存该线性表,那么必然会留下一个字节. 为了解决这个问题,可以强制规定其为 4 字节. 使其能统一.

快表:

局部性原理:(时间局部性,空间局部性)

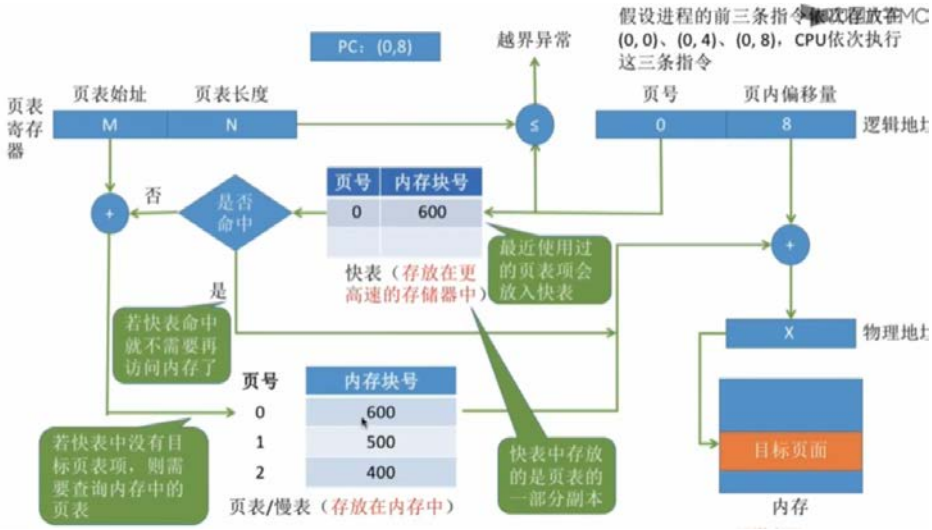
时间: 变量很可能会访问多次.

空间: 内存的页框,存储单元很可能再次被访问.

块表: 联想寄存器(TLB) 访问速度比内存快很多的**高速缓冲存储器**. 加速地址变换的过程.

快表的访问速度可以是内存的 1%.

其实就是快速版的表格而已. 因为页表对应的页面每次都要查询,而且该关系是固定的,因此可以将该表格的某些项,邻近项存在该高速缓冲器中.另外局部性原理.也决定了不需要记录太多的表格,记最近的几项就好了



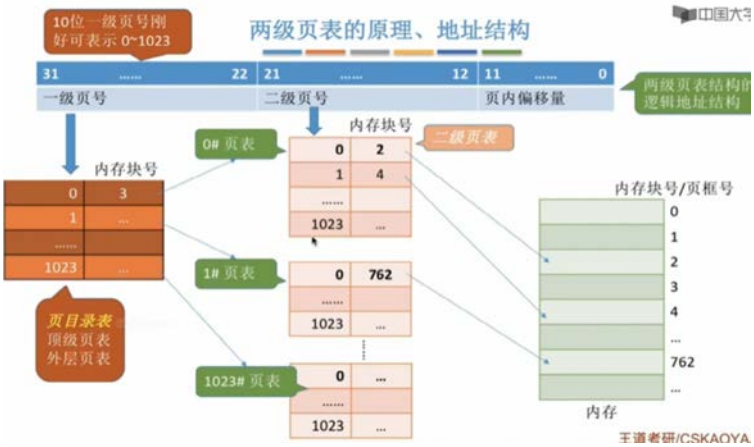
二级页表:

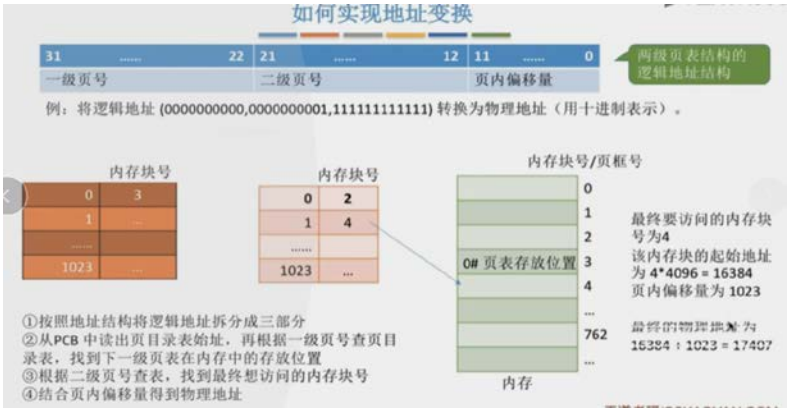
单集页表的存储问题:

页表有可能很大，可能占用很多页框，失去了分页存储的优点.

两级页表: 顶级页表: 页表下面的页表. 算力消耗是多消耗 n 倍..

正常 4 字节对于地址而言太多了，因此分为二级页表,前 10 位存顶级页号,后 10 位存第二级页号,第三级存偏移量. 因此地址表示法仍不变，先从顶级页表中读出二级页表存储位置,然后通过二级页号读出对应的页面. 然后通过偏移量计算出实际物理位置.





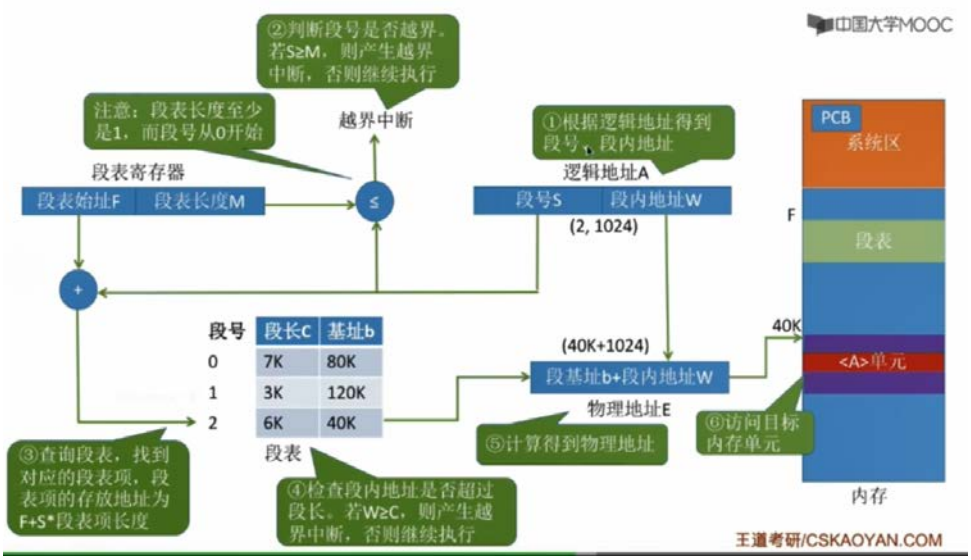
分段存储管理:

分段: 类似分页,从程序本身出发, **连续存储**. 不同段则是离散的. 这又要涉及内存动态分区分配问题.

分段过程如下: 首先索引变量 A 的逻辑地址-->发现他是**程序段** 2,段内地址为 W. 因此根据段表寄存器找到段表的初始地址,然后根据段表(每项的字节长度,默认是 40B) * 2 得到第二段的内存地址.

然后可以根据段表直接索引到第二段的起始**物理地址**,(目前仅用到 S), 然后再加上偏移量得到真实物理地址.

注意: 长度限制, 段号有没有>段表长度 M. 第二次长度限制:段长 C>W,因为偏移地址肯定不能超过段长的.



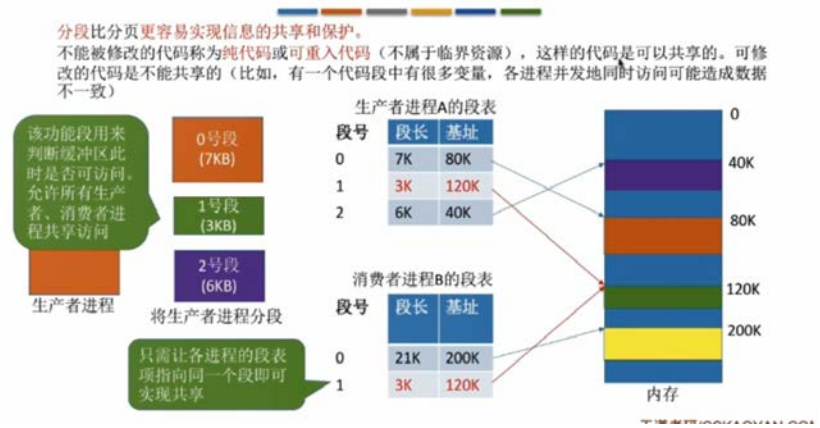
与分页对比:

分段多了一个段长的限制, 分页是固定长度的.

分段优点：更容易实现信息的**共享和保护**.

共享保护场景:进程间通信时,可以直接共享某段. 类似 c 语言中的.h 文件,class

下图的绿色部分为 class. 方便访问. 但是分页的话有可能跨页访问,信息会泄露



段页结合的分配方式:

分页式管理:

特点：不会产生外部碎片,但是不方便按**逻辑模块**实现信息保护和共享,

分段式管理:

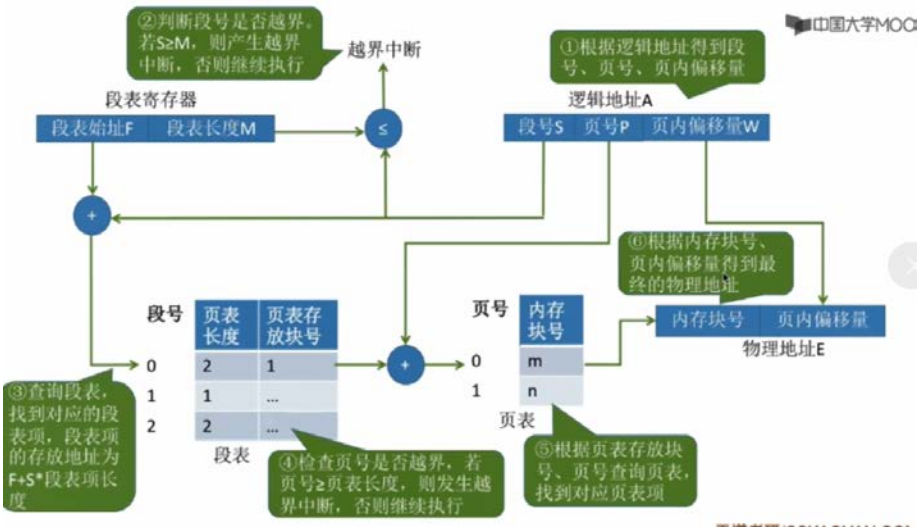
特点：方便逻辑模块的共享和保护，但是容易产生外部碎片．因为连续空间分配

融合之后：对每一段进行页面拆分．因此共享段的特性**仍然保留**．用户来说分段可见,但是分页不可见,由操作系统设计.

32 位逻辑地址地址：段号:页号:**页内偏移量**.

一个进程有一个段表．一个**段表项**对应一个页表．二维线性表

索引过程需要三次，先访问段表,然后再访问页表,最后访问物理地址



虚拟内存

基于局部性原理: 时间局部和空间局部

传统方式:

1. 一次性: 将作业装入内存才能运行. 大作业无法运行.
2. 驻留性: 作业会一直驻留, 但实际上不需要.

原理是将内存调入外存, 因为本质上都是存储. 只是访问速度快慢而已.

虚拟内存特性:

多次性: 无需一次装入,分成多次调入内存

对换性: 将作业换入换出

虚拟性: 逻辑上扩充了内存的容量.

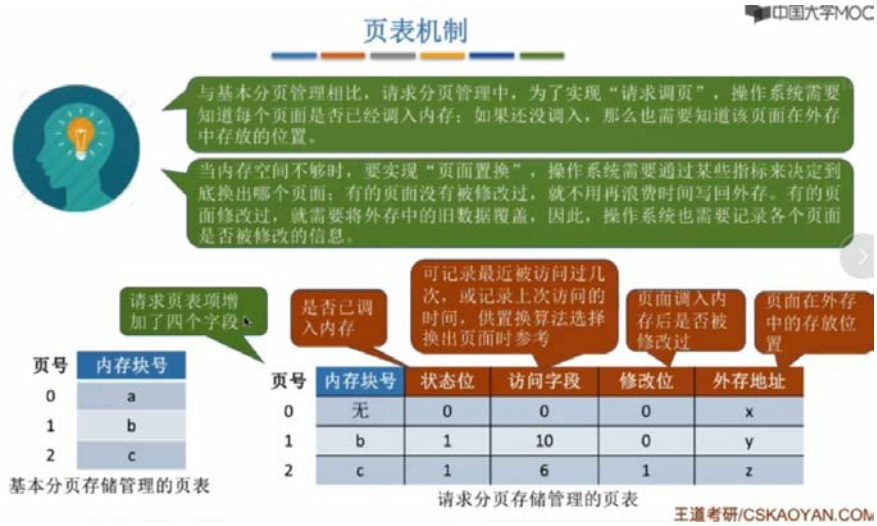
与传统方法新增: 请求分页功能(将外存的数据调入),页面置换功能(撤下不用作业).

请求分页:

与传统页表相比 新增了 4 个关键字段:

最近访问次数, 是否被修改,是否在内存. 外存对应地址.

如果有状态位为 0 的,那么表示有剩余空间,如果没有空闲块,**发生缺页中断机制**,撤出一个不常用作业,同时观察此作业是否有修改过,如有修改,则需更新外存. 一次操作可能有多次中断.此时程序进入阻塞态.



页面置换算法

缺页但是有空闲,则直接调用即可，缺页但是没空闲,则需要页面置换算法

会启动磁盘 IO,因此会产生较大的开销. 追求最少的缺页率

最佳置换算法 OPT: 今后最长时间不被访问的的页面.但是的提前预知以后要访问哪些页面.需要记录一个访问表. 是一种理想化的算法.

先进先出 FIFO:

最久未使用 least recently used: 因此访问字段记录的就是最近的访问次数,可以顺便使用. 目前最佳性能好 ,但是开销大.因为要遍历搜索.至少也是 logn

时钟置换算法(CLOCK): LRU 需要硬件支持,算法开销大. 这个就很简单，设定一个访问位 flag，访问该页则该页置为 1. 然后循环遍历当前内存，找到第一个为 0 的页面进行换出. 如果是 1,则将 1 置位 0. 因此如果一个页面不访问,那么最快会在第二轮换掉. 但是前面还有页面,因此也有可能在很多论才换出. 因为有用到循环队列,因此很像一个时钟.

改进型时钟置换算法: 二元数组的选择. 选择**没有被修改**且**没有访问**的页面进行换出.

过程 1: 找(0,0) 没有访问,没有修改的页面,失败则过程 2

过程 2: 找(0,1),并且顺便将访问后就将访问位设为 0. 失败则过程 3

过程 3: 找(0,0) 因为过程 2 顺便置 0 了,因此有可能有(0,0)

过程 4: 找(0,1) 用于替换. 最多就到过程 4,肯定会中,

页面分配策略:

驻留集: 由于先前讲的,可将进程一部分放入 swap 分区,也可以在 exe 直接读取文件. 进程分为程序段和数据段，数据段可以放在 swap 也可以是直接从文件中读取(比如图片资源)，程序段程序

段也可以直接在 exe 中读取. 因此**没必要将进程全部读入内存**,如果每个进程的空间小,那么进程就多,CPU 利用率高. 但是太小,比如仅保留 PCB,那么缺页频繁,调度开销大.因此驻留集的大小值得研究.

	局部置换	全局置换
固定分配	仅在自己的物理块置换	xxxx 不存在此方式
可变分配	仅在自己的物理块置换	操作系统负责调度

固定分配: 简单, 可由操作系统提前预判

可变分配-全局置换: 操作系统会保存一个空闲物理块的队列, 可分配给此进程

可变分配-局部置换: 如果频繁发生局部置换,那么操作系统自动变为全局置换

工作集: 工作集大小是进程实际访问页面, < 驻留集, 可根据工作集的大小分配驻留集的大小.

比如某进程一直访问页面 123123123,很好的局部性,因此可以适当减小驻留集为 3.

文件系统

讨论问题:

1 文件自身的定义以及属性

2 文件内部数据存储格式

3 文件之间在磁盘的存放关系

文件属性:

文件名

标识符 exe,txt

存储格式:

无结构文件: 流式文件

有结构文件: 数据库之类的问题.

文件之间的组织: 根目录

文件的逻辑结构

记录方式分为: **定长记录**,**可变长记录**.

存储方式为: **链式存储**: (不能随机存取).

顺序存储(物理地址顺序): 可变长记录(也不能随机存储) ,定长记录(可以随机存储).

有一种方法可以实现变长记录,又能快速索引(但还是不能随机存储) **索引速度 logn**.

索引表: 一个表:**记录关键字**, 然后是**起始位置**,**文件长度**. 索引表一行就 12B,是额外消耗.

索引顺序文件: 由于索引表是额外消耗,有时空间利用率较低. 现在为索引表瘦身, 简要来说就是分组, 比如姓名 A 开头为一组,B 开头为一组.这样最多就只有 26 组了.快速索引,二级结构

文件目录

文件控制块

树形目录结构

有向无环图, 不同目录可以指向同一个文件, 有向边.但是删除操作不好搞. 共享计数器,删除仅删除当前目录下的**记录**,不会删除真正的文件,用计数器是否为 0,==0 就删除文件.

文件保护

口令保护: 就是用户登录密码, 优点是验证时间开销小,但是存在内存,容易被盗.

密码保护:

异或加密,对原始文件循环 异或 如 10010, 两次异或操作能恢复出原始信息.

1 xor 1 xor1 = 1; 1 xor 0 xor 0 = 1; 0 xor 1 xor1 = 0;

访问控制:

对用户分组, 然后让文件的访问授权到组内.

文件系统的实现

内存(RAM) 称作随机的含义是 随机访问. 不像是链表,必须顺序访问. 我觉得添加随机反而引起歧义,不如直接叫内存.

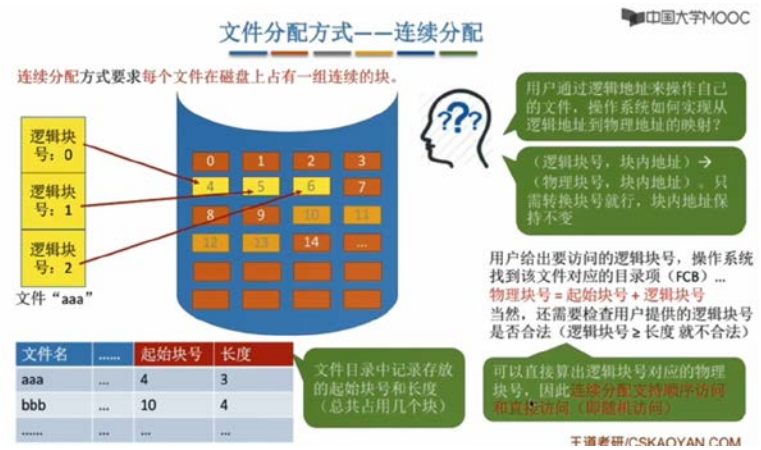
内存和文件系统都是存储器,存储方式大同小异. 区别就是内存分页, 外存分块, 块会大一些.另外注意外存经常遇到存储量增加的情况. 而内存的进程大小相对固定,而且可以覆盖载入,换页撤出,外存不能,因此文件内的链接结构还是略有不同的.

文件其实跟程序也有相似性, 比如 excel 访问文件, 用户肯定是按逻辑地址访问, 那么就需要逻辑地址和物理地址的转换. 而程序也一样, 需要逻辑地址和物理地址的转化.

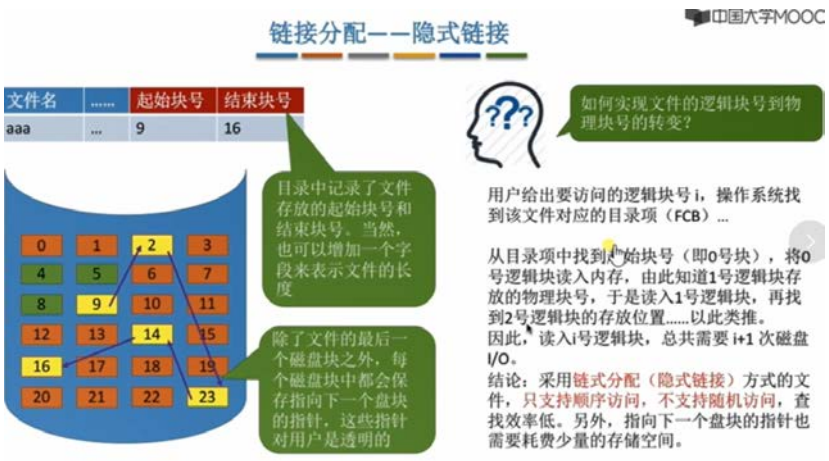
连续存储: 仅需知道初始地址即可，然后直接加上偏置.

优点是随机访问,还有连续域可以更快访问,甚至不需要换磁道.

缺点是不好**扩展添加**,因为下一块可能被占. 产生内部碎片



隐式链接: 仅支持顺序访问. 查找效率低速度慢. 但是增加块方便.空间利用率高.



显示连接(File Allocation Table):FAT 格式,将隐式连接的连接信息整合到一个表中，可以直接映射,达到随机访问. 访问速度较快. 不会有碎片问题，缺点是得建立一个表,这个表占一定空间.

注意文件分配表是加载到内存中的. 因此访问速度快.

索引分配: 不同于显示链接(仅有一张 FAT),索引分配也有一张 FAT 不过是二级的 FAT,首先先查出每个文件对应的索引块,再去找该索引块内的逻辑地址转换表,

相比于链接存储，索引分配的索引块大小是固定的. 当文件较小时,浪费空间,因此需要对索引表的大小进行讨论. 如果索引表大小为 1KB,索引物理地址需要 4B,那么最多只能存放 256 项地址映射,因此大文件就不好搞. 有以下三种方式.

假设磁盘块 1KB,索引项 4B,则最多放 256 个索引,256*1KB=256kB,最多能存的文件数.(**太小了**)

采用二级索引可以达到 256*256*1KB=64MB. 还可以. 3 层索引可以 16GB.

1 **链接方式**: 前一个索引块最后一项是下一个索引块的地址.但会索引太多次.

2 **二级索引**：第一个是索引块的表格，第二块才是索引块.

3 **混合索引**：前 15 块就是直接索引,后面是按级分配. 很不错 UNIX 系统结构.

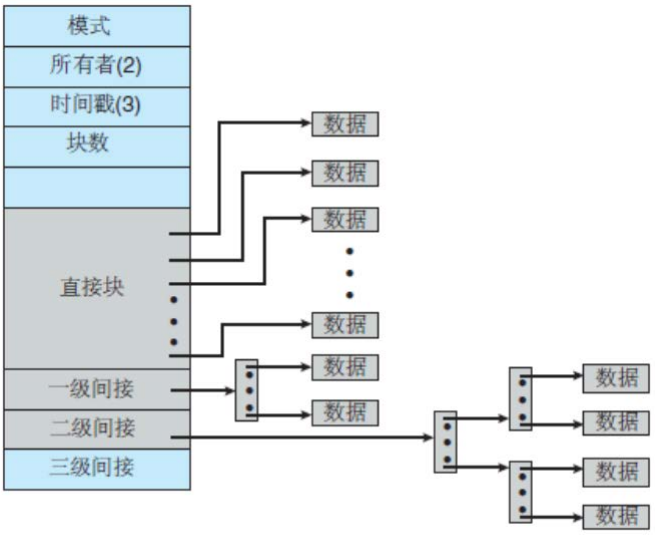


图 5 UNIX 的 inode

空闲文件的管理.

怎么记录空闲磁盘块,怎么分配磁盘块,怎么回收磁盘块.

空闲表法：一个表格，每一项记录空闲块的块号和块数. 分配空闲区时,检索块数的大小,刚好能满足文件. 回收时注意空闲表的增加或是合并. 有四种情况,前合并,中合并,后合并,不合并. 空闲表太大了，得不偿失

空闲链表法：就是空闲表的链表形式，增删比较方便，但是遍历时间就长了.

位示图法：这个还挺简单的,就直接用 0101 代表空闲与否. 技术难度不大,

成组链接法：先找到 100 个空闲块当成超级块.每块由一个 N,和 100 项地址组成. 每个地址指向一个空闲块,但**仅有第一个地址**指向下一个**类超级块**..

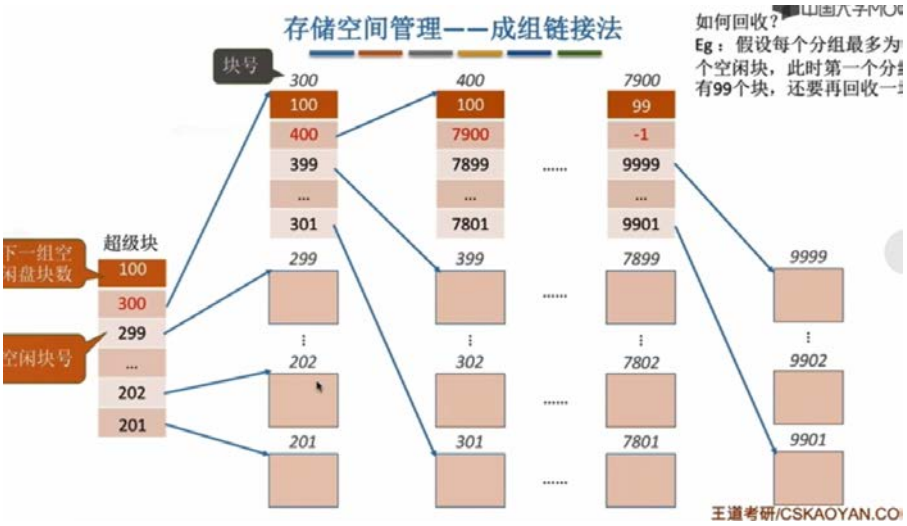
注意这个 N 当前块的空闲个数，并且仅有第一项有下一项的**地址指针!!!**.

分配时:从顶层开始分配.如果有请求到来: n 小于 N,则直接将倒序 n 分配.

若 n 大于 N,则需要将超级块的 100 释放,同时将第一个**类超级块当成超级块**.

如此循环,可将文件分配清楚.

回收时：直接补上超级块，若超过 100,则新建一个超级块.当前这个成为超级块.

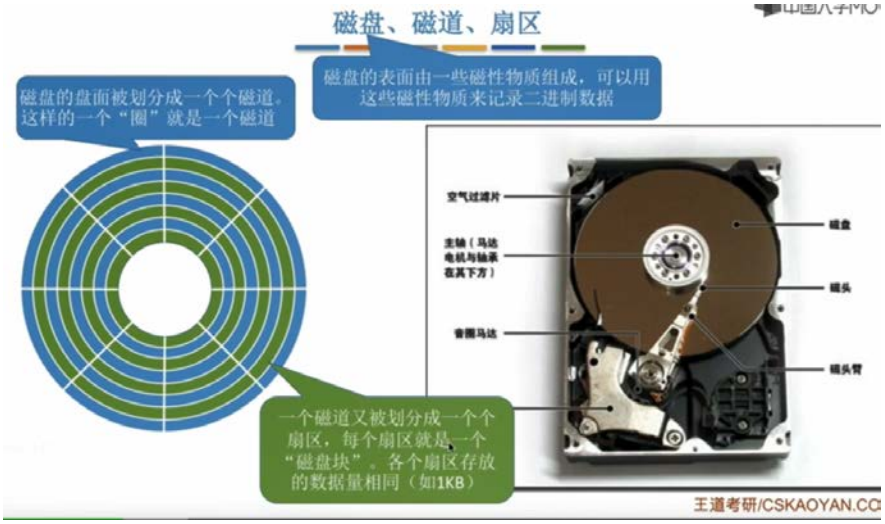


磁盘组织与管理:

磁盘 磁道,扇区. 每个扇区的数据量大小相同,因此最内侧的扇区存储密度最高.

而且不止一盘,很多盘.

要索引到某段数据, 需要三要素, 柱面号(确认磁道),盘面号(确认盘号),扇区号(确认扇区)



磁盘请求服务算法:

先到先得 FCFS

最短寻找时间有先算法: 但总体上不一定全局最优,类似吃豆子. 每次都计算当前最近的位置的进程服务. 有饥饿现象.

扫描算法: 电梯算法, 得跑到最外层才能往内走, 类似**往返跑**. 不会饥饿.

LOOK 算法: 在扫描算法的基础上, 如果当前任务队列没有往外的访问, 那么就可以返回.

循环扫描算法: 跑到最外围直接返回 0 地址.,类似单向跑.

减少时延的方法

一次读写操作需要的时间:

寻找时间(磁头移动到磁道时间), 延迟时间(磁道转到扇区),传输时间(数据读取)

交替编号: 因为磁头启动有一定延时,如果交替编号,则刚好足够磁头读取.

IO 设备

IO 控制器

IO 控制器是**操作系统的一部分**!!!. 负责与 CPU 进行交互,然后驱动外部的机械设备.

功能有:

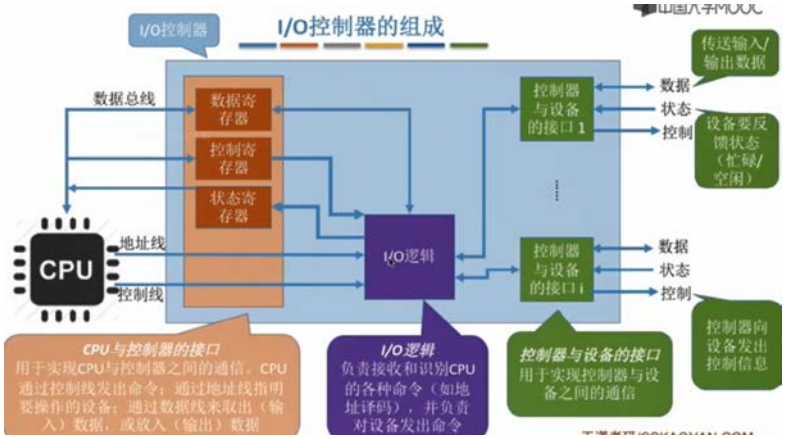
接收和识别 CPU 发出的命令: read/write

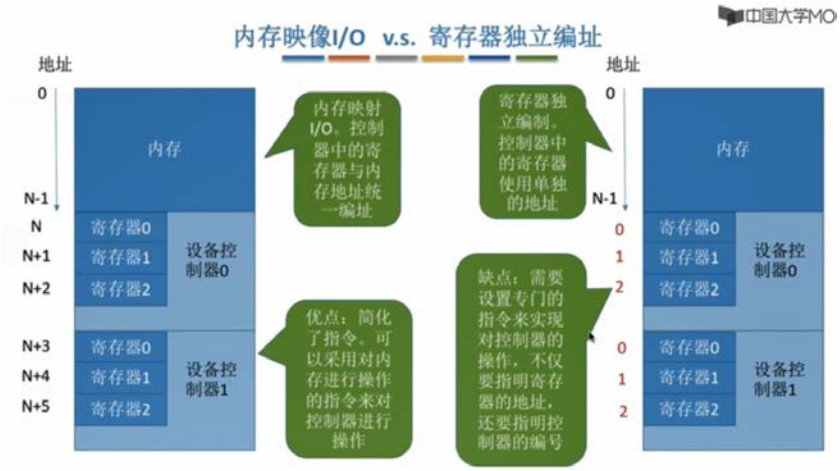
向 CPU 报告设备的状态: 某一个设备是否空闲或者忙碌.

数据交换: IO 设备本身有寄存器, 通过寄存器暂存 CPU 的指令.和数据

地址识别: 控制指令的发送对象.

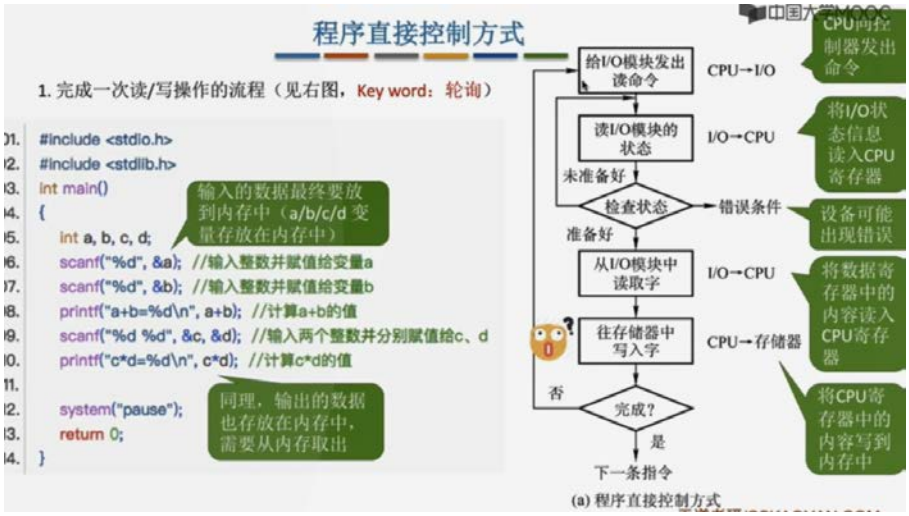
一个 IO 控制器对应多个地址. 每个设备对应一个状态寄存器,或者在内存开辟一个映像 IO,作为 IO 的专用地址,这样就不用专门的寄存器了.





控制流程

轮询: CPU 速度很快,一直访问状态寄存器. 缺点是 CPU 一直等待.浪费时间



中断驱动方式: CPU 可以中断去做别的事情，当前进程变为阻塞态. 等待中断.

过程: IO->CPU->内存,因此一次仅能读取一个字,不是字节.因为 CPU 寄存器就这么多.

优点是不用再空等

缺点是每次仅能读取一个字,如果中断频繁则需要消耗较多 CPU 时间

DMA 方式:

因为实际上 IO 数据仅需在内存即可. 因此 IO 直接导入内存是不错的方式.

数据传输的单位是块.cpu 告诉 DMA 模块需要读入多少数据,存到哪个内存模块.相当于多了一个小型 cpu 了,然后 DMA 模块处理完之后一个中断,表示读取结束.

提高数据读取的大小,不用 CPU 太过干预.

通道控制方式:

	完成一次读/写的过程	CPU干预频率	每次I/O的数据传输单位	数据流向	优缺点
程序直接控制方式	CPU发出I/O命令后需要不断轮询	极高	字	设备→CPU→内存 内存→CPU→设备	每一个阶段的优点都是解决了上一阶段的最大缺点。 总体来说，整个发展过程就是要尽量减少CPU对I/O过程的干预，把CPU从繁杂的I/O控制事务中解脱出来，以便更多地去完成数据处理任务。
中断驱动方式	CPU发出I/O命令后可以做其他事，本次I/O完成后设备控制器发出中断信号	高	字	设备→CPU→内存 内存→CPU→设备	
DMA方式	CPU发出I/O命令后可以做其他事，本次I/O完成后DMA控制器发出中断信号	中	块	设备→内存 内存→设备	
通道控制方式	CPU发出I/O命令后可以做其他事。通道会执行通道程序以完成I/O，完成后通道向CPU发出中断信号	低	一组块	设备→内存 内存→设备	

为什么需要不同设备驱动程序: 因为不同厂家的空闲和忙碌状态不同, 有的 0 为空闲有的 0 为忙碌.