操作系统（Operating Ststem， OS）是指**控制和管理整个计算机系统的硬件和软件资源**，并合理地组织调度计算机的工作和资源的分配，以提供给用户和其他软件方便的接口和环境，它是计算机系统中最基本的系统**软件**

操作系统的四个特征：

**并发 共享 虚拟 异步**

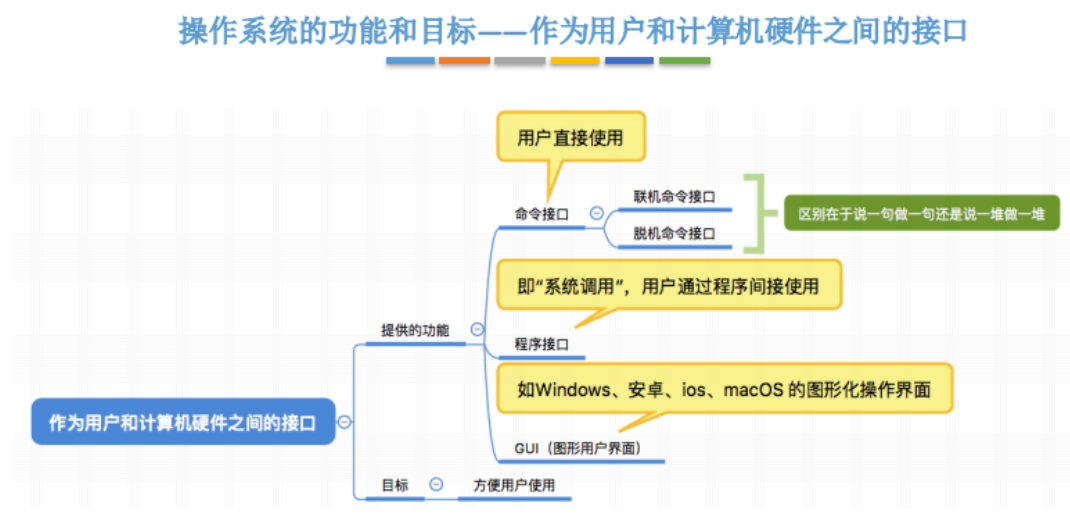
并行：两个或多个事件在同一时刻同时发生

共享即资源共享，是指系统中的资源内存中多个并发执行的进程共同使用

虚拟是指把一个物理上的实体变为若干个逻辑上的对应物

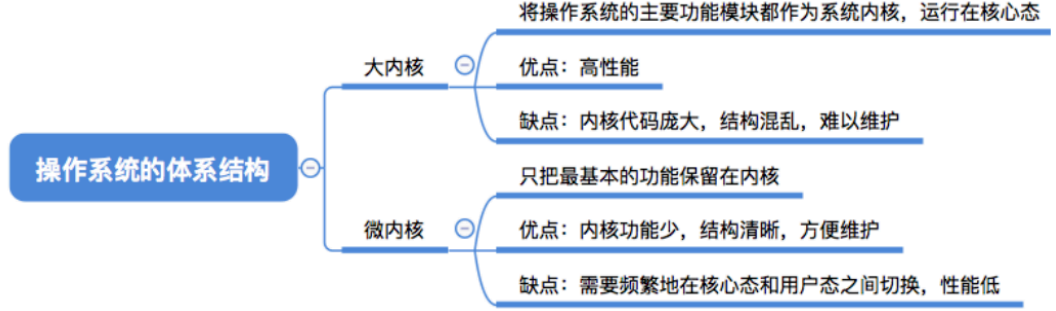
异步是指，在多道程序环境下，允许多个程序并发执行，但由于资源有限，进程的执行不是一贯到底的，而是走走停停，以不可预知的速度向前推进（有空分复用和时分复用）

功能和目标：接口 程序接口就是系统调用



特权指令不允许用户使用 程序状态字寄存器PSW标志处于什么状态

内核：大内核和微内核



中断：

中断是指计算机运行过程中，出现某些意外情况需主机干预时，机器能自动停止正在运行的程序并转

入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

1. 当中断发生时，CPU立即进入核心态

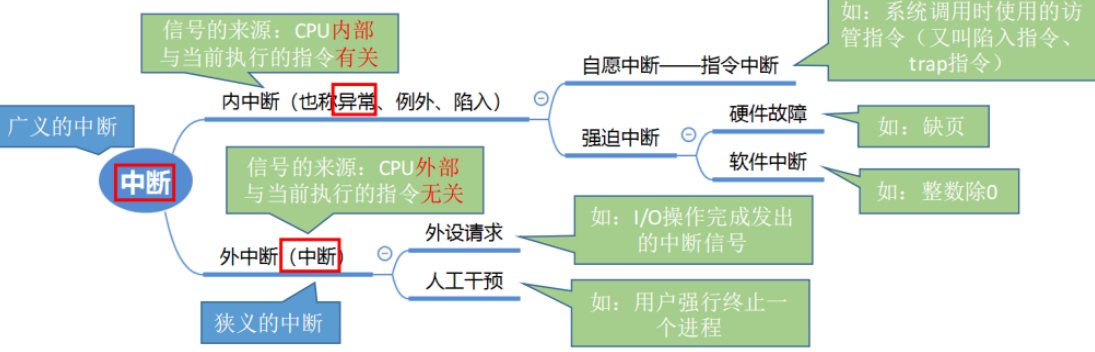
2. 当中断发生后，当前运行的进程暂停运行，并由操作系统内核对中断进行处理。

3. 对于不同的中断信号，会进行不同的处理。

4. 有了中断，才能实现多道程序并发执行。

5. “用户态→核心态”是通过中断实现的，并且中断是唯一途径。“

中断分类：



系统调用：

系统调用发生在用户态，对系统调用的处理发生在核心态

执行**陷入指令**会处理内中断

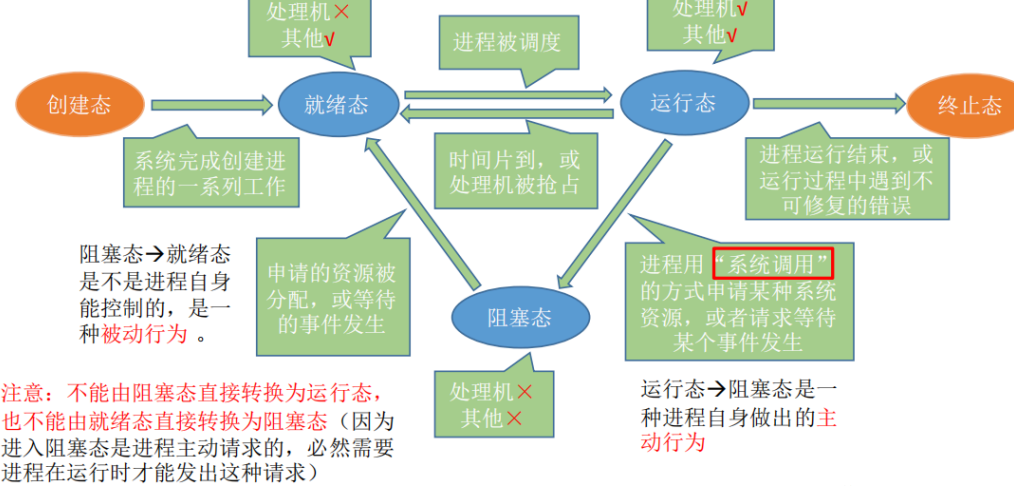
进程：

PCB，程序段，数据段三部分构成了进程实体（也叫作进程映像）。一般情况下，我们把进程实体简称为进程 PCB进程控制块是核心！也是进程存在的唯一标识

进程的组织：链接方式 索引方式

进程是资源分配的基本单位，注意进程有异步性

状态转换：



进程控制有原语实现，原语，一般是指由若干条指令组成的程序段，用来实现某个特定功能，

在执行过程中不可被中断

进程通信：操作系统提供了三种方法：共享存储，消息传递，管道通信。

关于管道通信！

1.**管道只能采用半双工通信**，某一时间段内只能实现单向的传输。如果要实现双向同时通信，则需要设置两个管道。

2.各进程要互斥地访问管道。

3.数据以字符流的形式写入管道，当管道写满时，写进程的write()系统调用将被阻塞，等待读进程将数据取走。当读进程将数据全部取后，管道变空，此时读进程的read()系统调用将被阻塞。

4.如果没写满，就不允许读。如果没读空，就不允许写。

5．**数据一旦被读出，就从管道中被抛弃，这就意味着读进程最多只能有一个**，否则可能会有读错数据的情况。

消息传递：通过通过操作系统提供的“发送消息/接收消息” 两个原语

分直接通信（消息直接挂在对方进程的消息缓冲队列上）和间接通信（先发到中间实体信箱）

线程：理解为轻量级的进程 线程是程序执行的最小单位

分类：

用户级线程：用户级线程由应用程序**通过线程库实现**。所有的线程管理工作都由应用程序负责（**包括线程切换**） 在操作系统内核看来，意识不到线程的存在

内核级线程：

内核级线程的管理工作由操作系统内核完成。线程调度、切换等工作都由内核负责，因此内核级线程的切换必然需要在核心态下才能完成。

可以这样理解，“内核级线程”就是“从操作系统内核视角看能看到的线程”。

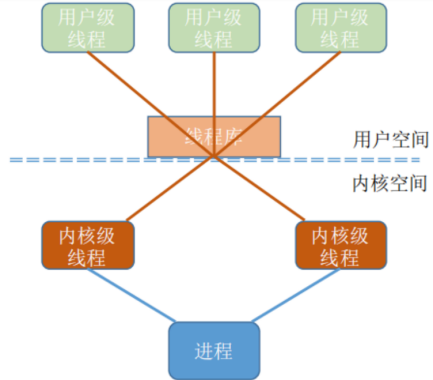
操作系统只“看得见”内核级线程，因此**只有内核级线程才是处理机分配的单位**。

多线程模型：

多对一模型:多个用户及线程映射到一个内核级线程。每个用户进程只对应一个内核级线程

一对一模型:一个用户级线程映射到一个内核级线程

多对多模型:n用户级线程映射到m个内核级线程(n >=m)。每个用户进程对应m个内核级线程。

注意看这里确实用户级线程切换由线程库完成！

处理机调度：

高级调度（作业调度)：按一定的原则从外存上处于后备队列的作业中挑选一个(或多个）作业，给他们分配内存等必要资源，并建立相应的进程（建立PCB)

高级调度是辅存（外存）与内存之间的调度。每个作业只调入一次，调出一次。作业调入时会建立相应的PCB，作业调出时才撤销PCB。

中级调度：引入了虚拟存储技术之后，可将暂时不能运行的进程调至外存等待。等它重新具备了运行条件且内存又稍有空闲时，再重新调入内存

注意**PCB是常驻内存的**！（与高级区别）

低级调度（进程调度）：按照某种方法和策略从就绪队列中选取一个进程，将处理

机分配给它

临界资源： 一个时间段内只允许一个进程使用的资源。各进程需要互斥的访问临界资源。

**临界区：访问临界资源的那段代码**。 （强调：代码 代码 代码）

内核程序临界区一般是用来访问某种内核数据结构的，比如进程的就绪队列。

进程的切换：（**切换是有代价的**，不能频繁切换）

1.对原来运行进程各种数据的保存

2.对新的进程各种数据的恢复(如:程序计数器、程序状态字、各种数据寄存器等处理机现场信息，这些信息一般保存在进程控制块)

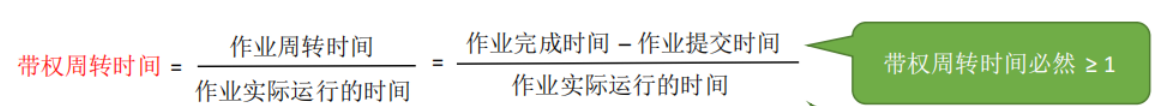
调度算法评价指标：

CPU利用率： CPU"忙碌"的时间占总时间的比例。

系统吞吐量：单位时间内完成作业的数量。

**周转时间：是指从作业被提交给系统开始，到作业完成为止的时间间隔。**

**周转时间=作业完成时的时间-作业提交时间**

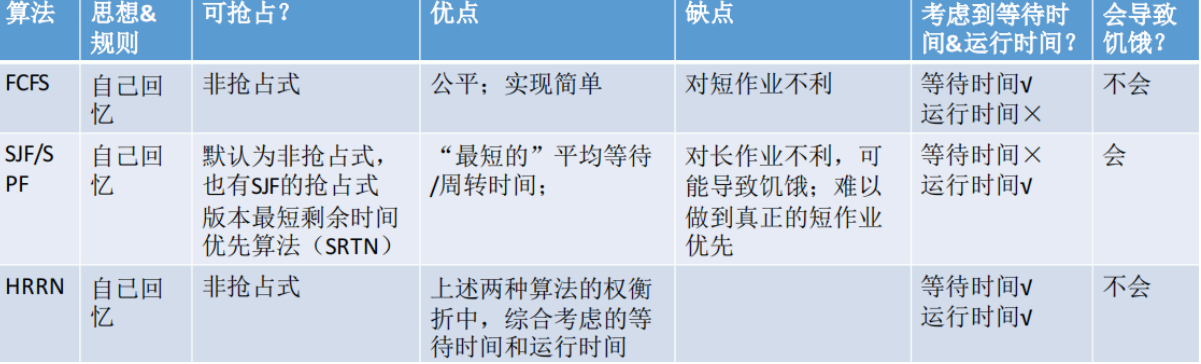


等待时间：指进程/作业处于等待处理机状态时间之和

（注意在等待I/O完成的期间其实进程也是在被服务的，所以不计入等待时间）

响应时间：指从用户提交请求到首次产生响应所用的时间

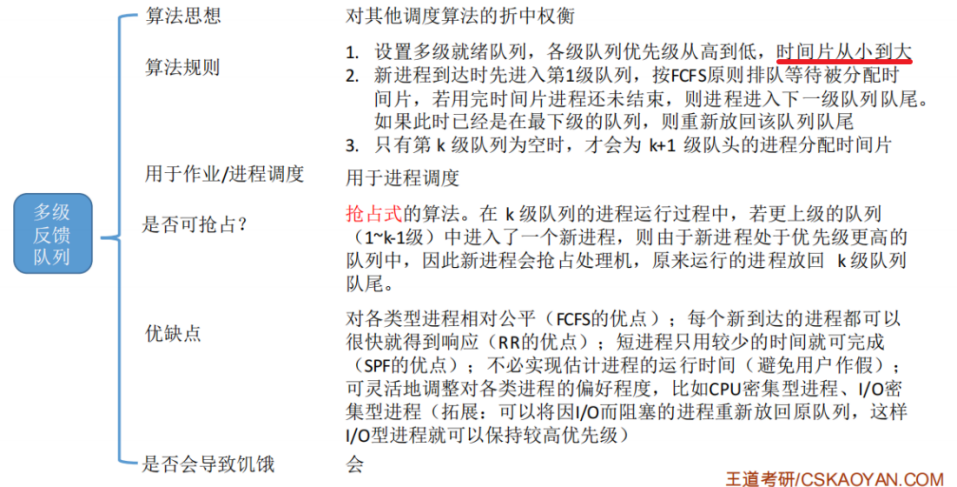
调度算法：



时间片轮转调度：抢占式 时间片大小的选取

优先级调度：静态和动态优先级

多级反馈队列调度：



注意如果用完还没结束，进入下一个队列的末尾！（这三种算法适合交互式系统）

进程同步与互斥：  
进程互斥指当一个进程访问某临界资源时，另一个想要访问该临界资源的进程必须等待

为了**实现对临界资源的互斥访问**，同时保证系统整体性能，**经典四原则！**

1. 空闲让进。临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区;

2. 忙则等待。当已有进程进入临界区时，其他试图进入临界区的进程必须等待;

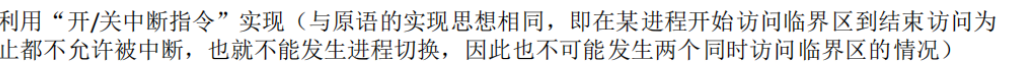
3. 有限等待。对请求访问的进程，应保证能在有限时间内进入临界区（保证不会饥饿)

4. 让权等待。当进程不能进入临界区时，应立即释放处理机，防止进程忙等待。

**进入区 临界区 退出区 剩余区**

进程互斥的软件实现方法：单标志 双标志 Peterson算法（双方都争着想进入临界区，那可以让进程尝试“孔融让梨”，主动让对方先使用临界区）

硬件实现：中断屏蔽



Testandset swap指令

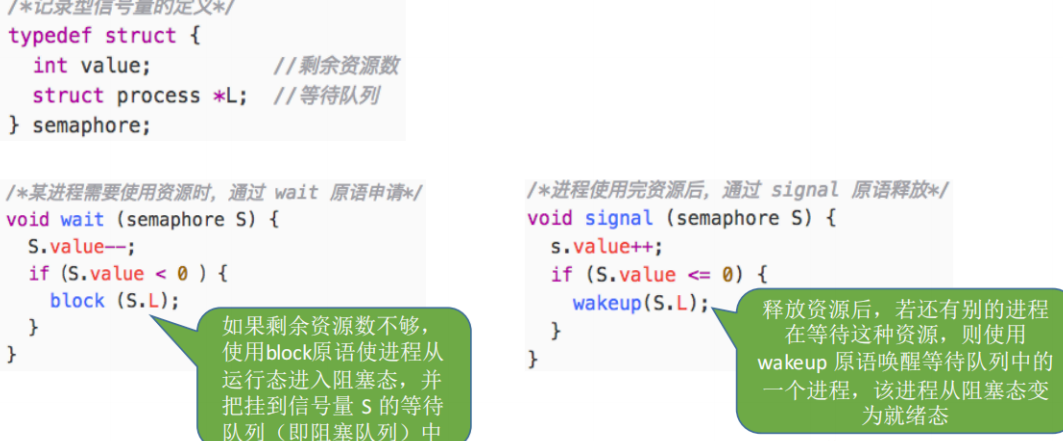
信号量：

一对原语: wait(S)原语和 signal(S)原语，可以把原语理解为我们自己写的函数，函数名分别为 wait和 signal，括号里的信号量s其实就是函数调用时传入的一个参数。

wait、signal原语常简称为P、V操作

**强调P是wait代表消耗 V是signal代表生产！**

纪录型信号量：（整型信号量的缺陷是存在“忙等”问题）



实现同步互斥的一些细节：  
互斥：在临界区之前执行P(mutex) 在临界区之后执行V(mutex) P、V操作必须成对出现

同步：

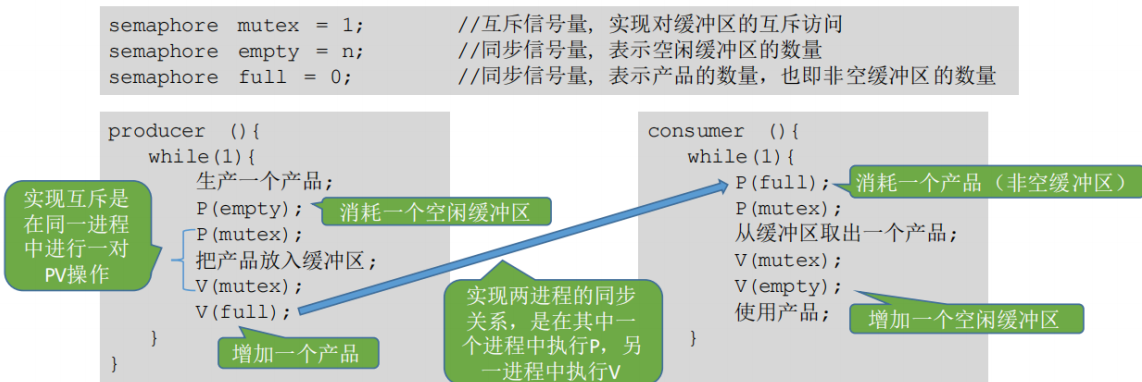
1. 分析什么地方需要实现“同步关系”，即必须保证“一前一后”执行的两个操作（或两句代码)

2. 设置同步信号量s,初始为0

3. 在“前操作”之后执行v(S)

4. 在“后操作”之前执行P(S)

生产者消费者：



……

管程：管程相当于对临界区资源进行抽象而编写的一个类

1.局部于管程的共享数据结构说明; （一个类）

2.对该数据结构进行操作的一组过程; （类中的方法）

3．对局部于管程的共享数据设置初始值的语句; （类中的变量）

4.管程有一个名字。 （类名）

基本特征！

1．局部于管程的数据只能被局部于管程的过程所访问; （类中变量有自己的作用范围）

2.一个进程只有通过调用管程内的过程才能进入管程访问共享数据; 这种互斥特性是由编译器来实现的。

3．每次仅允许一个进程在管程内执行某个内部过程。

死锁：在并发环境下，各进程因竞争资源而造成的一种互相等待对方手里的资源，导致各进程都阻塞

**死锁的四个条件！互斥 不剥夺 请求和保持 循环等待**

预防策略：

1. 预防死锁。破坏死锁产生的四个必要条件中的一个或几个。

2. 避免死锁。用某种方法防止系统进入不安全状态，从而避免死锁（银行家算法)

3. 死锁的检测和解除。允许死锁的发生，不过操作系统会负责检测出死锁的发生，然后采取某种措施解除死锁。

**银行家算法步骤:**

①检查此次申请是否超过了之前声明的最大需求数

②检查此时系统剩余的可用资源是否还能满足这次请求

③试探着分配，更改各数据结构

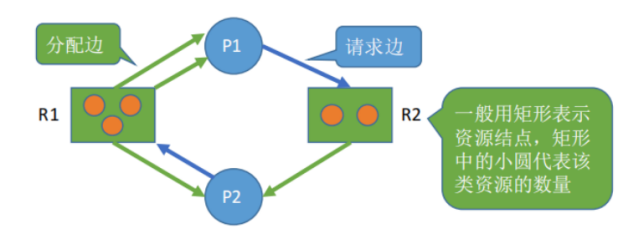
④用安全性算法检查此次分配是否会导致系统进入不安全状态

安全性算法步骤:

检查当前的剩余可用资源是否能满足某个进程的最大需求，如果可以，就把该进程加入安全序列，并把该进程持有的资源全部回收。

不断重复上述过程，看最终是否能让所有进程都加入安全序列。

死锁的检测——资源分配图



如果系统中剩余的可用资源数足够满足进程的需求，那么这个进程暂时是不会阻塞的，可以顺利地执行下去。如果这个进程执行结束了把资源归还系统，就可能使某些正在等待资源的进程被激活，并顺利地执行下去。相应的，这些被激活的进程执行完了之后又会归还一些资源，这样可能又会激活另外一些阻塞的进程.

如果按上述过程分析，最终能消除所有边，就称这个图是可完全简化的。此时一定没有发生死锁（相当于能找到一个安全序列)。

检测出死锁后，死锁的解除：

1. 资源剥夺法。挂起（暂时放到外存上）某些死锁进程，并抢占它的资源，将这些资源分配给其他的死锁进程。但是应防止被挂起的进程长时间得不到资源而饥饿。

2. 撤销进程法（或称终止进程法）。强制撤销部分、甚至全部死锁进程，并剥夺这些进程的资源。 这种方式的优点是实现简单，但所付出的代价可能会很大。因为有些进程可能已经运行了很长时间，已经接近结束了，一旦被终止可谓功亏一篑，以后还得从头再来

3. 进程回退法。让一个或多个死锁进程回退到足以避免死锁的地步。这就要求系统要记录进程的历史信息，设置还原点。

如何选择对哪些进程动手？

1. 进程优先级 (优先级低的)

2. 已执行多长时间 （执行时间短的）

3. 还要多久能完成 （时间长的进行处理）

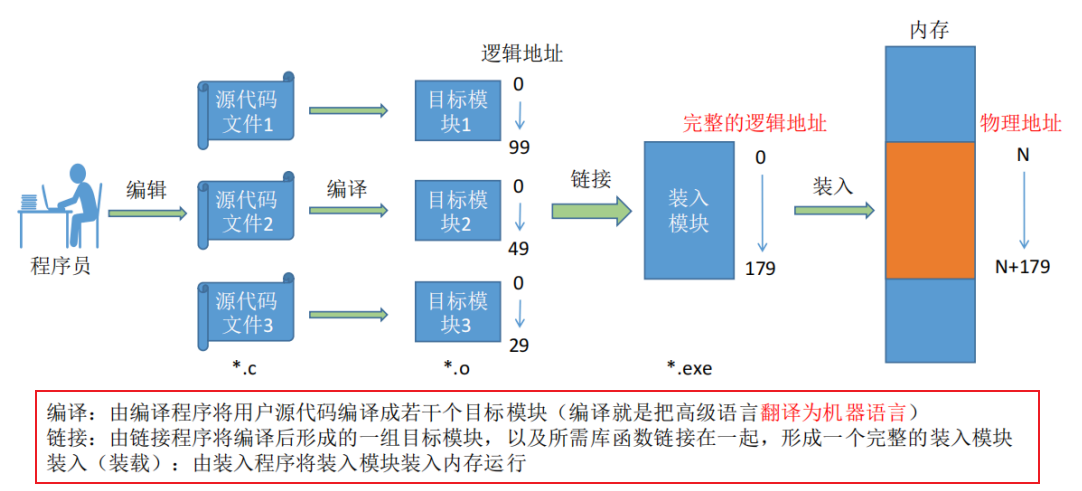
4. 进程己经使用了多少资源 （资源多的）

5. 进程是交互式的还是批处理式的 （进行批处理的）

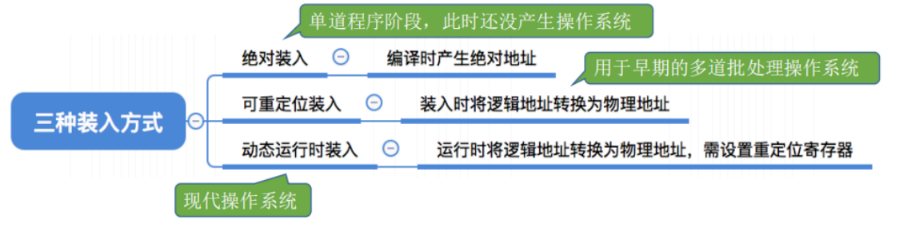
内存管理：

强调：**编译时产生的指令只关心“相对地址”，实际放入内存中时再想办法根据起始位置得到“绝对地址”**。

Eg: 编译时只需确定变量x存放的相对地址是100（也就是说**相对于进程在内存中的起始地址而言的地址**）。CPU 想要找到x在内存中的实际存放位置，只需要用进程的起始地址+100即可。 **相对地址又称逻辑地址，绝对地址又称物理地址**。



三种装入：



内存保护的方法：

1. 上下限寄存器 2.重定位寄存器（基址寄存器）+界地址寄存器（限长寄存器）

覆盖技术：

覆盖技术的思想 : 将程序分为多个段（多个模块）。常用的段常驻内存，不常用的段在需要时调入内存。

内存中分为一个“固定区”和若干个“覆盖区”。需要常驻内存的段放在“固定区”中，调入后就不再调出 （除非运行结束)不常用的段放在“覆盖区”，需要用到时调入内存，用不到时调出内存

交换技术：

内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存

再次强调：PCB会常驻内存，不会被换出外存

连续分配管理：  
单一连续分配，固定分区分配（分区说明表 没有外部碎片但有内部碎片），动态分区分配

**动态分区分配**：

**这种分配方式不会预先划分内存分区，而是在进程装入内存时，根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要**（空闲分区表/链）

强调一下回收：会把相邻的空闲区域合并为一个。

**内部碎片**，分配给某进程的内存区域中，如果有些部分没有用上。

**外部碎片**，是指内存中的某些空闲分区由于太小而难以利用。（紧凑技术解决）

动态分区分配算法：  
首次适应：

每次都从低地址开始查找，找到第一个能满足大小的空闲分区（**空闲分区以地址递增的次序排列**。每次分配内存时顺序查找空闲分区链）

最佳适应：

优先使用更小的空闲区 空闲分区按容量递增次序链接

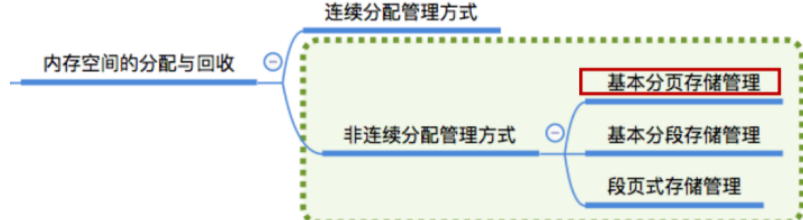
最大适应：

每次分配时优先使用最大的连续空闲区 空闲分区按容量递减次序链接

临近适应：

每次都从上次查找结束的位置开始检索，就能解决上述问题 空闲分区以地址递增的顺序排列

分页存储管理：

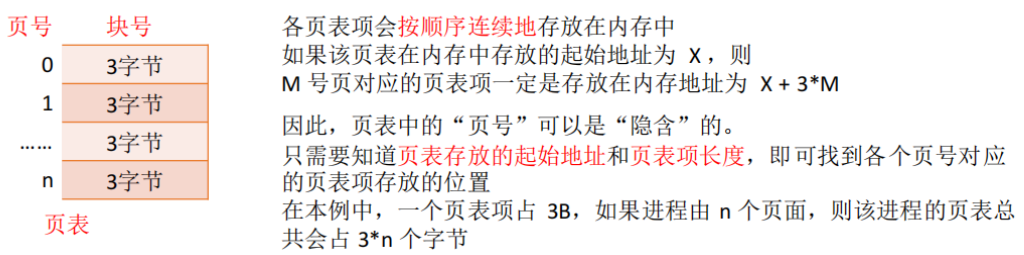


将内存空间分为一个个大小相等的分区（比如:每个分区4KB），每个分区就是一个“页框”，或称“页帧”、“内存块”、“物理块”。每个页框有一个编号，即“页框号”(或者“内存块号”、“页帧号”、“物理块号”)

页框号从0开始。

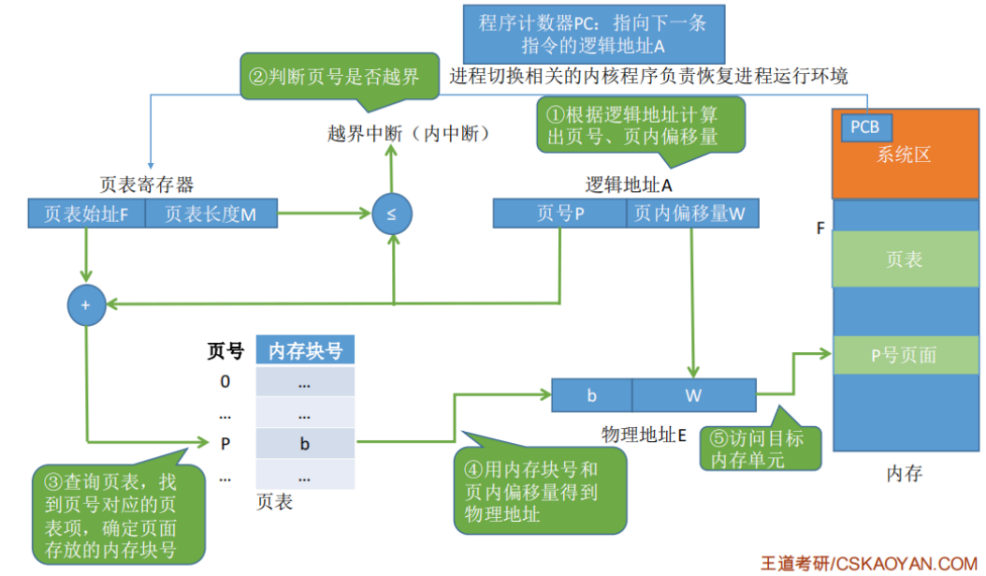
将用户进程的地址空间也分为与页框大小相等的一个个区域，称为“页”或“页面”。每个页面也有一个编号，即“页号”，页号也是从0开始。

操作系统为每一个进程创建一个页表 每个页表项的长度是相同的，页号是“隐含的



基本地址变换机构——借助进程的页表将逻辑地址转换为物理地址 页表寄存器PTR

**页表寄存器(PTR），存放页表在内存中的起始地址F和页表长度M。**



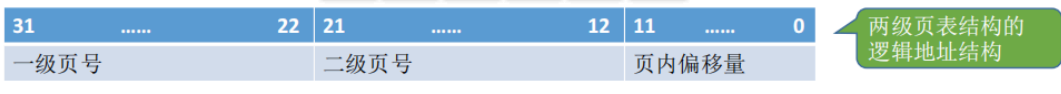
快表：

快表，又称联想寄存器（TLB），是一种访问速度比内存快很多的高速缓冲存储器，用来存放当前访问的若干页表项，以加速地址变换的过程。与此对应，内存中的页表常称为慢表

问题一:页表必须连续存放，因此当页表很大时，需要占用很多个连续的页框。

问题二:**没有必要让整个页表常驻内存**，因为进程在一段时间内可能只需要访问某几个特定的页面。（可以在页表项中新增一个标志位）

——引入两级页表



分段管理：

进程的地址空间:按照程序自身的逻辑关系划分为若干个段，每个段都有一个段名（在低级语言中，程序员使用段名来编程)，每段从0开始编址。

内存分配规则 : 以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻

段号+段内地址：段号的位数决定了每个进程最多可以分几个段。

段内地址位数决定了每个段的最大长度是多少。

一个程序——一个段表 各个段表项长度相同，段号隐含，记录了基址和段长

二者对比：

页是信息的物理单位。分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，对用户是不可见的。

段是信息的逻辑单位。分段的主要目的是更好地满足用户需求。一个段通常包含着一组属于一个逻辑模块的信息。分段对用户是可见的，用户编程时需要显式地给出段名。

页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。分页的用户进程地址空间是一维的，程序员只需给出一个记忆符即可表示一个地址。

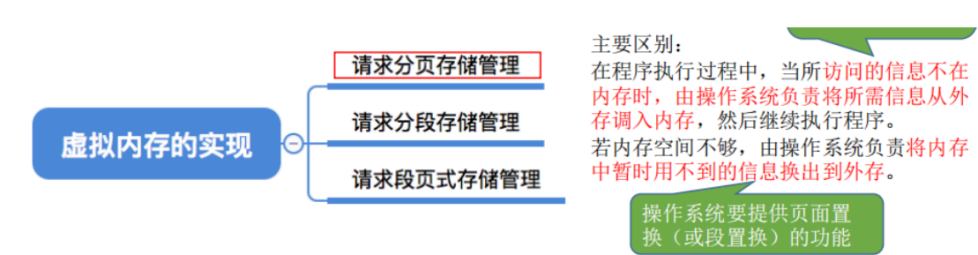
分段的用户进程地址空间是二维的，程序员在标识一个地址时，既要给出段名，也要给出段内地址。

**分段比分页更容易实现信息的共享和保护（主要是长度可以不固定，这是它的优点，也是它是二维的原因）**

段页式：

寻址：段表+页表

虚拟内存： 这么一梳理确实清晰了



请求分页管理

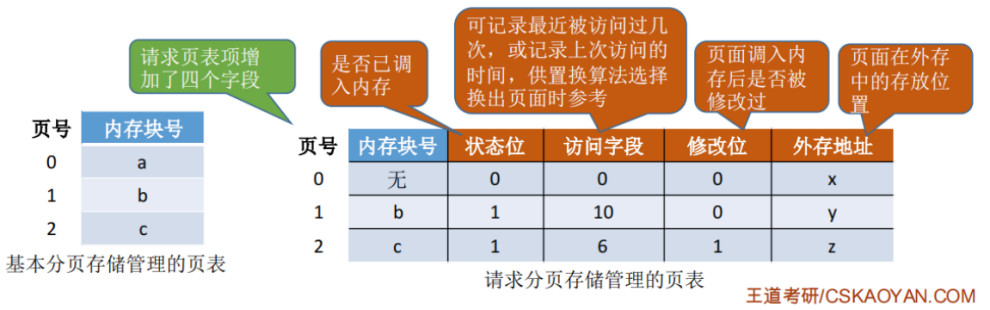
1. 操作系统需要知道每个页面是否已经调入内存;

2. 如果还没调入，那么也需要知道该页面在外存中存放的位置。

3. 当内存空间不够时，要实现“页面置换”，操作系统需要通过某些指标来决定到底换出哪个页面;

4. 有的页面没有被修改过，就不用再浪费时间写回外存。有的页面修改过，就需要将外存中的旧数据覆盖，因此，操作系统也需要记录各个页面是否被修改的信息。

因此页表会增加四个字段来上面的信息



在请求分页系统中，每当要访问的页面不在内存时，便产生一个缺页中断，然后由操作系统的缺页中断处理程序处理中断。

**缺页中断属于内中断中的故障！**（陷阱是有意而为的异常，终止是不可恢复的致命错误）

**页面置换算法：**

最佳置换算法（OPT，Optimal):每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。（但**实际无法实现**）

先进先出置换算法（FIFO):每次选择淘汰的页面是最早进入内存的页面

**Belady异常―一当为进程分配的物理块数增大时，缺页次数不减反增的异常现象**

**只有FIFO算法会产生Belady异常**。

最近最久未使用置换算法（LRU，least recently used):每次淘汰的页面是最近最久未使用的页面。（用访问字段记录该页面自上次被访问以来所经历的时间t）

**时钟置换算法（或称最近未用算法NRU）：**

简单的CLOCK 算法实现方法:为每个页面设置一个访问位，再将内存中的页面都通过链接指针链接成一个循环队列。当某页被访问时，其访问位置为1。当需要淘汰一个页面时，只需检查页的访问位。如果是0，就选择该页换出;如果是1，则将它置为0，暂不换出，继续检查下一个页面，若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描（第二轮扫描中一定会有访问位为0的页面，因此简单的CLOCK算法选择一个淘汰页面最多会经过两轮扫描)

页面分配策略：

**驻留集:指请求分页存储管理中给进程分配的物理块的集合**。

**固定分配:**操作系统为每个进程分配一组固定数目的物理块，在进程运行期间大小不变。即，驻留集大小不变（与之相对是可变分配）

由此还有：固定分配局部置换、可变分配局部置换、固定分配全局置换、可变分配全局置换

预调页策略……

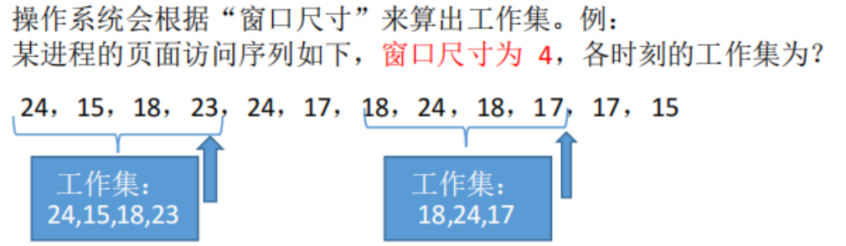
**抖动现象与工作集：**

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数（分配给进程的物理块不够)。

驻留集:指请求分页存储管理中给进程分配的内存块的集合。

工作集:指在某段时间间隔里，进程实际访问页面的集合。 （**即实际工作的**）

一般来说，驻留集大小不能小于工作集大小，否则进程运行过程中将频繁缺页



文件系统：

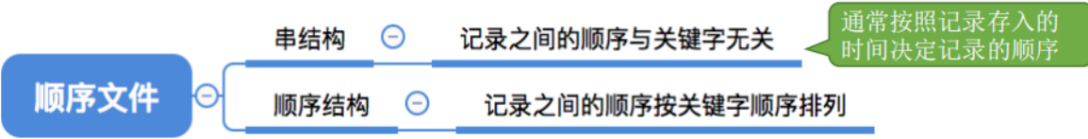
无结构文件:文件内部的数据就是一系列二进制流或字符流组成。又称“流式文件”。如:**Windows操作系统中的.txt文件**。

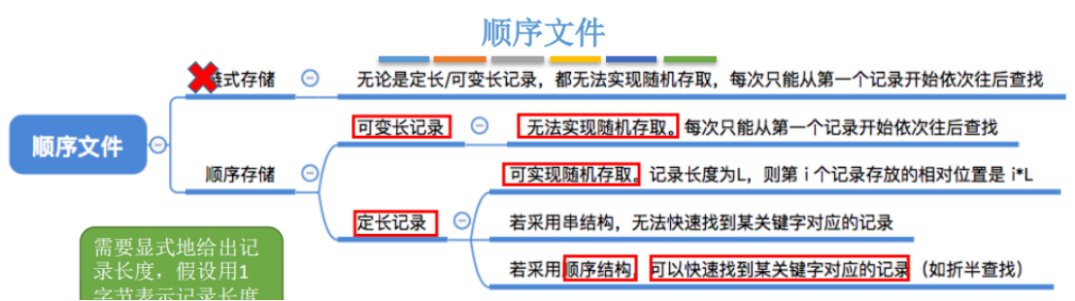
有结构文件:由一组相似的记录组成，又称“记录式文件”。每条记录又若干个数据项组成。如:数据库表文件。一般，每条记录有一个数据项可作为关键字（作为识别不同记录的ID)

有结构文件的逻辑结构：顺序文件、索引文件、索引顺序文件

顺序文件:文件中的记录一个接一个地顺序排列（逻辑上），记录可以是定长的或可变长的。

串结构：记录顺序与关键字无关





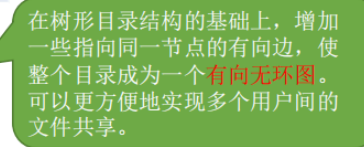
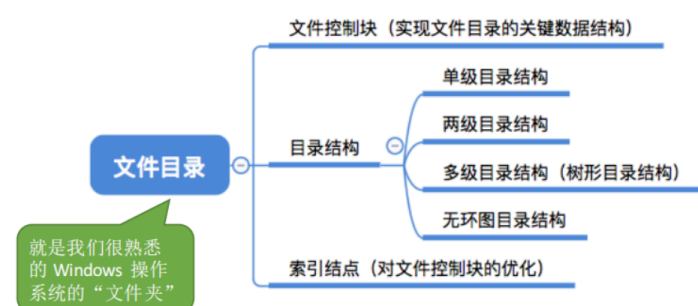
索引文件——索引表

索引顺序文件：

思考索引文件的缺点:每个记录对应一个索引表项，因此索引表可能会很大。比如:文件的每个记录平均只占8字节，而每个索引表项占32个字节，那么索引表都要比文件内容本身大4倍，这样对存储空间的利用率就太低了——如何解决？索引顺序文件

索引顺序文件是索引文件和顺序文件思想的结合。索引顺序文件中，同样会为文件建立一张索引表， 但不同的是:并不是每个记录对应一个索引表项，而是一组记录对应一个索引表项。

文件目录：



目录本身就是一种有结构文件，由一条条记录组成。每条记录对应一个在该放在该目录下的文件。目录文件中的一条记录就是一个“文件控制块（FCB)

单级目录 二级目录 多级目录 无环图目录

需要为每个共享结点设置一个共享计数器，用于记录此时有多少个地方在共享该结点。用户提出删除结点的请求时，只是删除该用户的FCB、并使共享计数器减1，并不会直接删除共享结点。

只有共享计数器减为0时，才删除结点。 注意不是赋值，用户指向的是同一个文件！

索引节点：改进FCB

原来目录项有文件名，类型，权限，等等信息，可以把除了文件名以外的信息封装为一个索引节点 优化内存中的FCB 变为：文件名+索引节点指针

当找到文件名对应的目录项时，才需要将索引结点调入内存，索引结点中记录了文件的各种信息，包括文件在外存中的存放位置，根据“存放位置”即可找到文件

文件分配（物理结构）：连续分配，链接分配，索引分配

连续分配：物理块号=起始块号＋逻辑块号

链接分配——隐式链接

用户给出要访问的逻辑块号i，操作系统找到该文件对应的目录项(FCB)

从目录项中找到起始块号（即0号块），将0号逻辑块读入内存，由此知道1号逻辑块存放的物理块号，于是读入1号逻辑块，再找到2号逻辑块的存放位置......以此类推。

因此，读入i号逻辑块，总共需要i+1次磁盘l/O。

**链接分配——显式链接**

把用于链接文件各物理块的指针显式地存放在一张表中。**即文件分配表（FAT，File Allocation Table)**。

一个磁盘仅设置一张FAT。开机时，将FAT读入内存，并常驻内存

FAT的各个表项在物理上连续存储，且每一个表项长度相同，因此“物理块号”字段可以是隐含的。

**索引分配：**

索引分配允许文件离散地分配在各个磁盘块中，系统会**为每个文件建立一张索引表，索引表中记录了文件的各个逻辑块对应的物理块**（索引表的功能类似于内存管理中的页表――建立逻辑页面到物理页之间的映射关系)。**索引表存放的磁盘块称为索引块。文件数据存放的磁盘块称为数据块**

流程：

用户给出要访问的逻辑块号i，操作系统找到该文件对应的目录项（FCB)

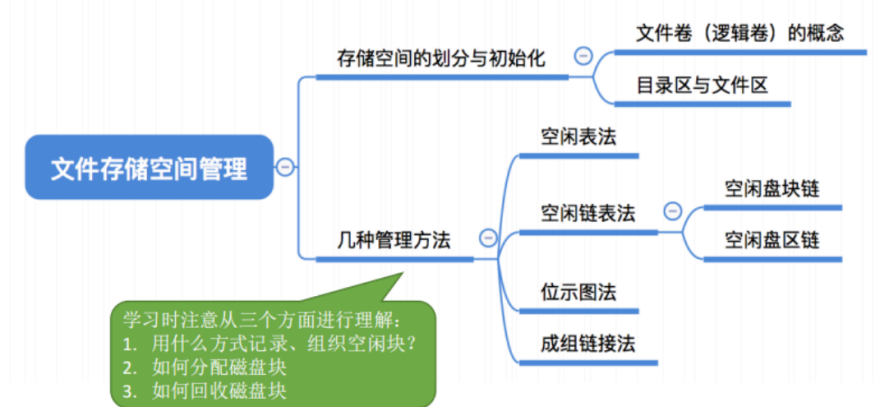
从目录项中可知索引表存放位置，将索引表从外存读入内存，并查找索引表即可知道i号逻辑块在外存中的存放位置。

可见，索引分配方式可以支持随机访问。文件拓展也很容易实现

（又有链接索引 多级索引 混合索引）

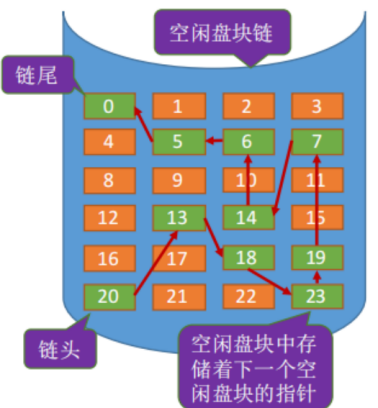
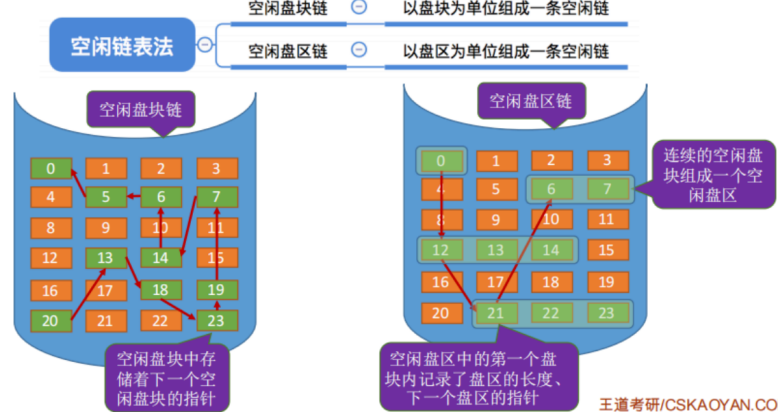
混合索引:多种索引分配方式的结合。例如，一个文件的顶级索引表中，既包含直接地址索引（直接指向数据块），又包含一级间接索引（指向单层索引表)、还包含两级间接索引（指向两层索引表)

空闲磁盘块的管理：



空闲表：注意合并

空闲链表：每个盘都有一个指向下一个空闲盘的指针



空闲盘块链：不再是每个盘，而是盘的集合（上右图）

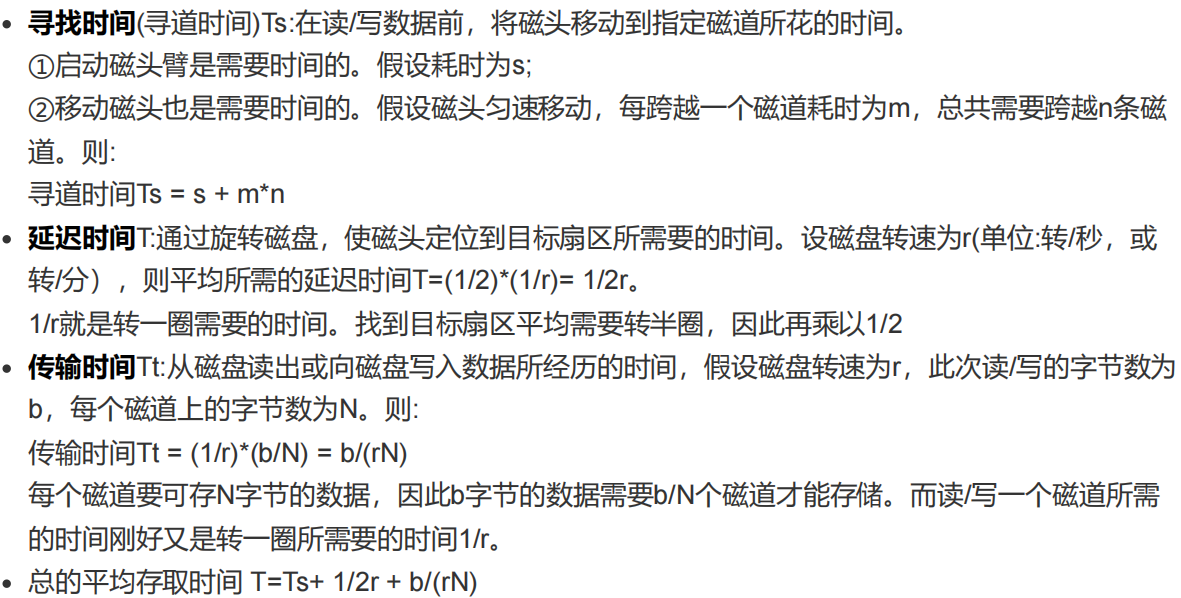
文件共享：

基于索引节点的共享方式（硬链接） 基于符号链的共享方式（软链接） 文件保护……

磁盘结构：

最内侧磁道上的扇区面积最小，因此数据密度最大

所有盘面中相对位置相同的磁道组成柱面。



磁盘调度算法：  
先来先服务，最短寻找时间优先，扫描算法（电梯算法：只有磁头移动到最外侧磁道的时候才能往内移动，移动到最内侧磁道的时候才能往外移动），look算法（如果在磁头移动方

向上已经没有别的请求，就可以立即改变磁头移动方向。(边移动边观察，因此叫LOOK)）

1. SCAN（只有磁头朝某个特定方向移动时才处理磁道访问请求，而返回时直接快速移动至起始端而不处理任何请求），C-look（如果磁头移动的方向上已经没有磁道访问请求了，就可以立即让磁头返回，并且磁头只需要返回到有磁道访问请求的位置即可）

磁盘管理：

磁盘初始化:

Step 1:进行低级格式化**（物理格式化），将磁盘的各个磁道划分为扇区。**一个扇区通常可分为头、数据区域（如512B大小）、尾三个部分组成。管理扇区所需要的各种数据结构一般存放在头、尾两个部分，包括扇区校验码（如奇偶校验、CRC循环冗余校验码等，校验码用于校验扇区中的数据是否发生错误)

Step 2:将磁盘分区，每个分区由若干柱面（磁道）组成（即分为我们熟悉的C盘、D盘、E盘)

step 3:进行**逻辑格式化，创建文件系统**。包括创建文件系统的根目录、初始化存储空间管理所用的数据结构（如位示图、空闲分区表)

引导块：

计算机开机时需要进行一系列初始化的工作，这些初始化工作是通过执行初始化程序（自举程序）完成的。 自举程序boosttrap也称引导加载程序

初始化程序可以放在ROM(只读存储器）中。ROM中的数据在出厂时就写入了，并且以后不能再修改（问题：万一要更新自举程序？）

解决：

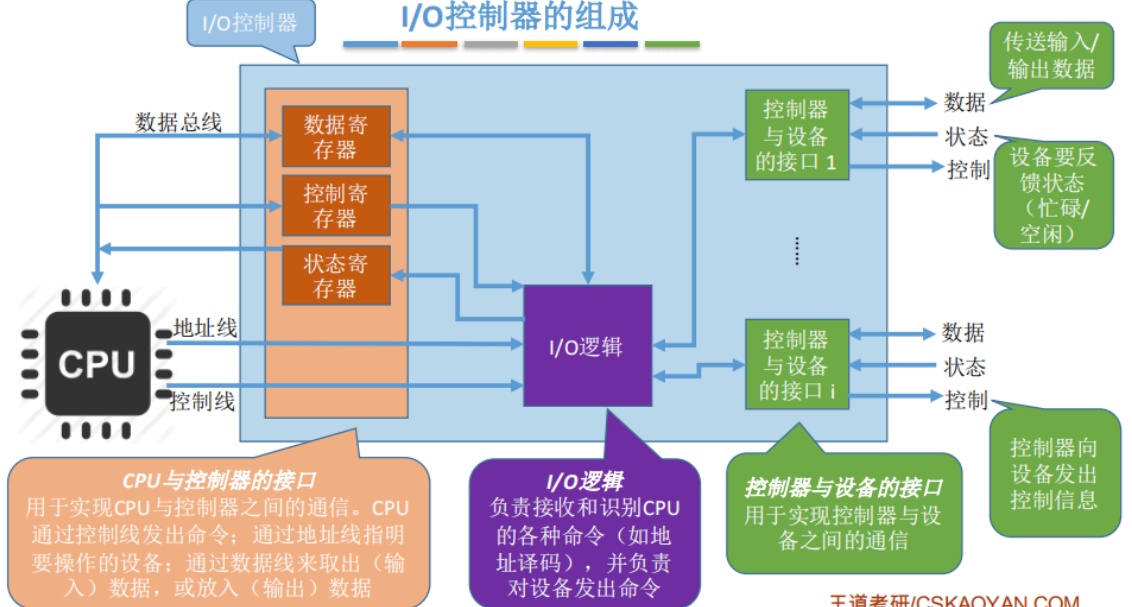
ROM中只存放很小的“自举装入程序”。开机时计算机先运行“自举装入程序”，通过执行该程序就可找到引导块，并将完整的“自举程序”读入内存，完成初始化

完整的自举程序放在磁盘的启动块(即引导块/启动分区)上，启动块位于磁盘的固定位置

IO管理：

IO控制器：CPU无法直接控制l/O设备的机械部件，因此I/O设备还要有一个电子部件作为CPU和I/O设备机械部件之间的“中介”，用于实现CPU对设备的控制

注意那几个寄存器和IO逻辑



IO控制方式：

