操作系统

基本概念

1. 操作系统特点?
    - 并发、共享、虚拟、异步
2. 并行，并发?

并行	**并发**
一起进行	一起发生
几个同时进行	几个间隔进行
边吃饭边打电话	吃饭、打电话、吃饭

3. 同步、异步、阻塞、非阻塞
    - 数据准备、数据读写

    - 同步：调用发出后，等待结果返回
    - 异步：调用发出后，可以去干自己的了，等待处理完通知（callback、状态、通知方式获取结果）

    - 阻塞：调用结果返回前，挂起线程
    - 非阻塞：调用结果返回前，不阻塞线程，每隔一段时间进行检测，没有就绪就可以做其他事情

    - 同步/异步：是否需要内核空间到用户空间的拷贝
    - 阻塞/非阻塞：是非需要等待所需的I/O存在
4. 同步、互斥?
    - 互斥：对资源的竞争；彼此之间不需要知道对方的存在；执行顺序是一个乱序
    - 同步：协调多个相互关联线程**合作完成任务**；彼此之间知道对方存在；执行顺序往往是有序的
5. **用户态、内核态?**
    - 用户态：处于用户态的 CPU 只能受限的访问内存，并且不允许访问外围设备，用户态下的 CPU 不允许独占，也就是说 CPU 能够被其他程序获取。
    - 内核态：处于内核态的CPU可以访问任意数据，处于内核态的 CPU 可以从一个程序切换到另外一个程序，并且占用 CPU 不会发生抢占情况，一般处于特权级 0 的状态我们称之为内核态。
    - 为什么要有这些：计算机中有一些比较危险的操作，极容易导致系统崩坏

	A	B
1	**内核态**	**用户态**
2	运行操作程序、操作硬件	运行用户程序
3	CPU可以访问任意的数据	CPU只能受限的访问内存
4	占用CPU不允许被抢占	不允许独占，可被抢占的

    - 进程由内核管理调度、进程的切换只能发生在内核态
6. **用户态和内核态如何切换?**
    - 用户态->内核态：
        - 系统调用：如fork()
        - 异常：运行的某些程序发生了不可知的异常；如缺页异常
        - 外围设备的中断：如硬盘读写操作完成，切换到硬盘读写的中断处理程序中执行后续操作
    - 内核态->用户态：设置程序状态字PSW
7. 中断?
    - 正常运行程序过程中，由于某些随机内部外部事件，需要机器干预转入要处理的程序，处理完之后再返回被暂停的程序继续运行
    - 作用：支持实时处理功能；并行操作；支持多道程序并发运行；硬件故障报警与处理
8. **硬中断和软中断**
    - 硬中断：**外设**自动产生的，网卡收到数据包等
        - 可屏蔽
    - 软中断：执行中断指令产生的
        - 不可屏蔽
9. 中断保存的东西存在哪里?
    - 栈中
10. 中断的处理过程?
    - 中断请求
    - 中断判优
    - 中断响应
    - 中断服务：
        - 保护现场：将当前程序的相关数据保存在寄存器中，入栈
        - 开中断：允许级别更高的中断请求中断
        - 中断服务
        - 关中断：保证恢复现场时不被新中断打扰
        - 恢复现场：从堆栈中取出程序，返回
    - 中断返回
    - 中断处理
11. 中断和轮询?

- 轮询：让CPU按一定周期按次序查询每一个外设，检查是否有输入输出要求；如果有就进行相应服务，处理完查询下一个外设

中断	轮询
通过特定事件提醒CPU	CPU对特定设备
容易遗漏问题，CPU利用率不高	效率低等待时间长，CPU利用率不高

12. 外中断、异常?
    - 外中断：CPU执行指令的外部事件引起（输入输出、时钟中断）
    - 异常：CPU执行指令的内部事件引起（地址越界、溢出）
13. CPU中断?
    - 定义：系统内发生了急需处理事件；CPU转去执行相应的事件处理程序；处理完毕后返回中断处继续执行
    - 作用：系统及时响应外部事件；多个外设同时工作，提高了CPU利用率；处理硬件故障
14. 一个程序从开始到结束的完整过程?
    - 预编译：主要处理#的内容
    - 编译：词法分析、语法分析、语义分析 x.c->x.i
    - 汇编：汇编指令翻译成机器指令 x.i->x.o
    - 链接：将目标文件进行链接，形成可执行程序 x.o->可执行
15. 静态链接、动态链接?

静态链接	动态链接
编译链接时将代码拷贝到调用处	不直接拷贝代码，需要的时候加载到内存，多个程序调用一个时候可以共享内存
运行速度快	多个程序共享一段代码
浪费空间、更新困难	运行时加载、速度慢：

16. 系统调用、函数调用

系统调用	函数调用
调用系统内核的服务，操作系统提供给用户程序特殊的接口	运行在用户空间，主要通过压栈操作进行函数调用
调用开销大	调用开销小
可以用来控制硬件；读取内核数据；进程管理	

17.

进程管理

- Conception：
  a. 程序
  b. 进程：运行着的程序
  c. 线程：进程当中的一条执行流程
  d. 协程

e. 上下文切换

f. 进程间通信

g. 进程、线程同步（管程）

h. 进程调度算法

i. 互斥锁、读写锁、条件变量、自旋锁、递归锁/非递归锁

j. 守护进程、僵尸进程、孤儿进程

k. 多进程、多线程

l. 线程安全

m. 生产者-消费者问题、哲学家问题、读者-写者问题

n. CAS（compare and swap）

- Question：

a. 程序和进程的区别？

程序	**进程**
静态	动态
永久的	暂时存在的
	运行着的程序

b. 进程、线程、协程？

- 三个概念的提出：先有协程、再有进程、再有线程
- 进程：资源分配的最小单位；私有地址空间，私有栈、堆；上下文切换需要切换虚拟地址空间
- 线程：资源调度的基本单位；公有同一地址空间，公有堆、私有栈；上下文切换只需要切换少量寄存器
- 协程：用户级的轻量线程，上下文切换由开发者决定，不由内核决定，类似于函数执行到一半，一会回来继续执行
- 为什么要有进程：单道批处理系统只能串行，CPU利用率不高；多道批处理系统难以实现共享资源；
- 为什么要有线程：进程上下文切换开销大，线程共享资源节省空间，进程上下文切换需要切换虚拟内存，切换页表TLB失效，线程节省时间
- 为什么要有协程：线程是抢占式任务，由操作系统控制，不知道什么时候被抢走，需要加锁；协程是异步机制，需要代码编写者主动让出控制权，不需要上下文切换

c. 进程、线程、协程区别？

	进程	**线程**	**协程**
单位	资源分配的最小单位	CPU调度的最小单位，程序执行的基本单位	线程内部调度的基本单位
内存空间不同	每个进程拥有独立的地址空间	同一进程的线程共享同一地址空间	

共有内容	进程ID、进程目录、公共数据、代码段	堆、**地址空间**、全局变量、静态变量等；	
私有内容	堆、栈、地址空间、全局变量、寄存器	栈、寄存器、程序计数器	寄存器和栈
优点		创建时分配资源少；上下文切换开销小；隔离程度更小	无须上下文切换的开销；不高的执行效率；跨平台、跨
切换者	操作系统	操作系统	用户程序（不被操作系统内全由程序控制）
上下文切换系统开销	切换虚拟地址空间，切换内核栈和硬件上下文，CPU高速缓存失效、页表切换，开销大	只切换少量寄存器内容，开销小	能保存上一次调用时的状态先将寄存器上下文和栈保存来的时候再进行恢复
机制	同步机制	同步机制	异步机制
	进程与进程之间是完全隔离的	线程与线程之间是没有隔离的	

d. 上下文切换？进程、线程的上下文切换？

- 上下文切换：从当前执行任务切换到另一个任务执行的过程；同时，为了确保下次能从正确的位置继续执行，在切换之前，会保存上一个任务的状态
- 相同：都是由操作系统内核完成（最显著的性能损耗：寄存器中的内容切换）
- 不同：线程切换虚拟地址空间内存是相同的，进程切换的虚拟地址空间内存是不同的
- 进程：①切换页表以使用新的地址空间 ②切换内核栈和硬件上下文
- 线程：①切换内核栈和硬件上下文

e. 为什么虚拟地址空间切换耗时（为什么线程切换比进程切换快）？

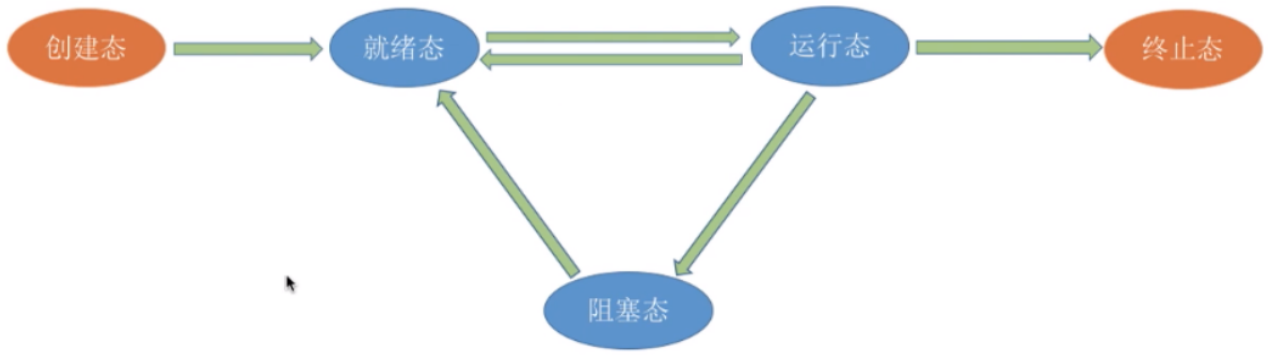
- 把虚拟地址转换成物理地址需要查找页表，页表查找很慢，故用cache（TLB）缓存地址映射，可以加速
- 每个进程有自己的**虚拟地址空间**，每个进程有自己的页表，进程切换时候页表也要切换，页表切换TLB就失效了，cache命中率低，查找慢，程序运行慢

f. 为什么要有进程?

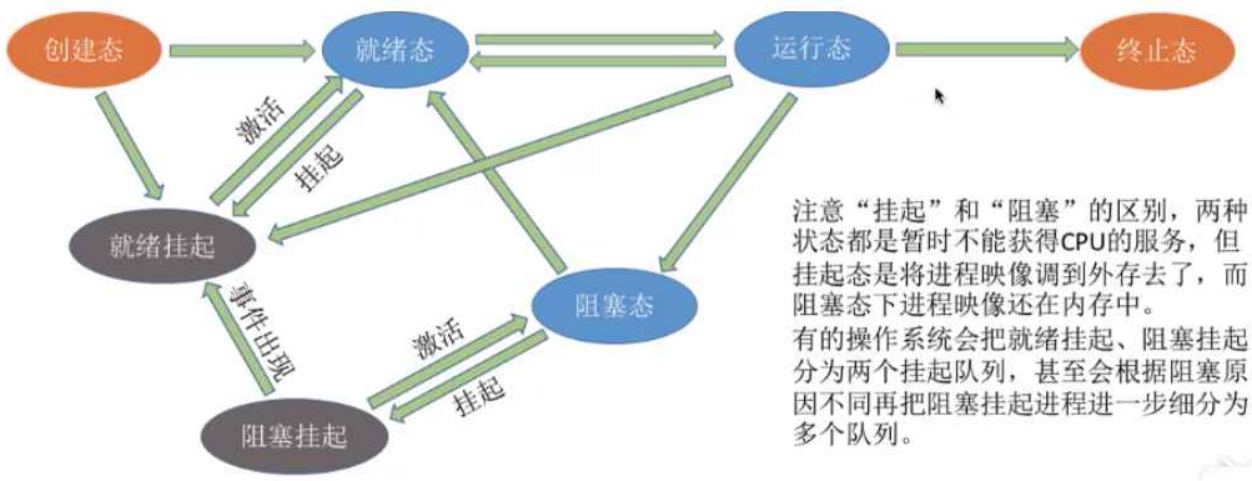
- 单道批处理中CPU效率高，IO低速，cpu要等待；多道批处理，存在共享资源导致程序相互限制
- 进程实现了多个程序并发执行

g. 进程状态切换?

- 三状态、五状态、七状态
- 创建态
- 就绪态、运行态、阻塞态
- 终止态
- 挂起态



- 就绪->运行：根据调度算法选择进入
- 运行->就绪：时间片用完
- 运行->阻塞：发生等待事件进入阻塞
- 阻塞->就绪：等待的事件已经发生



h. 进程切换场景
   - 进程调度，时间片被耗尽，被系统挂起
   - 系统资源不足，等到资源满足后再运行
   - 通过睡眠函数 sleep()主动挂起
   - 有优先级更高的进程需要运行
   - 发生硬件中断，当前进程被挂起

i. 进程间通信?
   - 进程的用户地址独立，不能互相访问，内核空间是共享的，进程之间通信必须通过内核

名字	相关内容	缺点	优点
管道/匿名管道	管道（内核里面的一串缓存）；单向、半双工；只能在存在父子关系的进程	效率低，容量有限	
有名管道（FIFO）	半双工；不相关的进程间也能相互通信		
共享内存	不同进程拿出自己的一段地址空间，映射到相同的物理内存	同时修改，存在冲突（线程安全）	速度快，因为是直接对内存进行存取**（理论上效率最高）**

消息队列	消息链表	容量有限、不适合大数据传输	
信号量	一个整型计数器，实现进程间互斥和同步，P（-）V（+）操作，是原子操作；信号量+共享内存通常组合使用	不能传递复杂消息，只能同步	
信号	通知某个事件已经发生；唯一的异步通信机制；常用信号（SIGCLD子进程退出、SIGKILL用户终止进程、SIGPIPE管道破裂信号常用信号）		
套接字 Socket	不同主机上的进程间的通信	速度慢	任何进程都能通讯

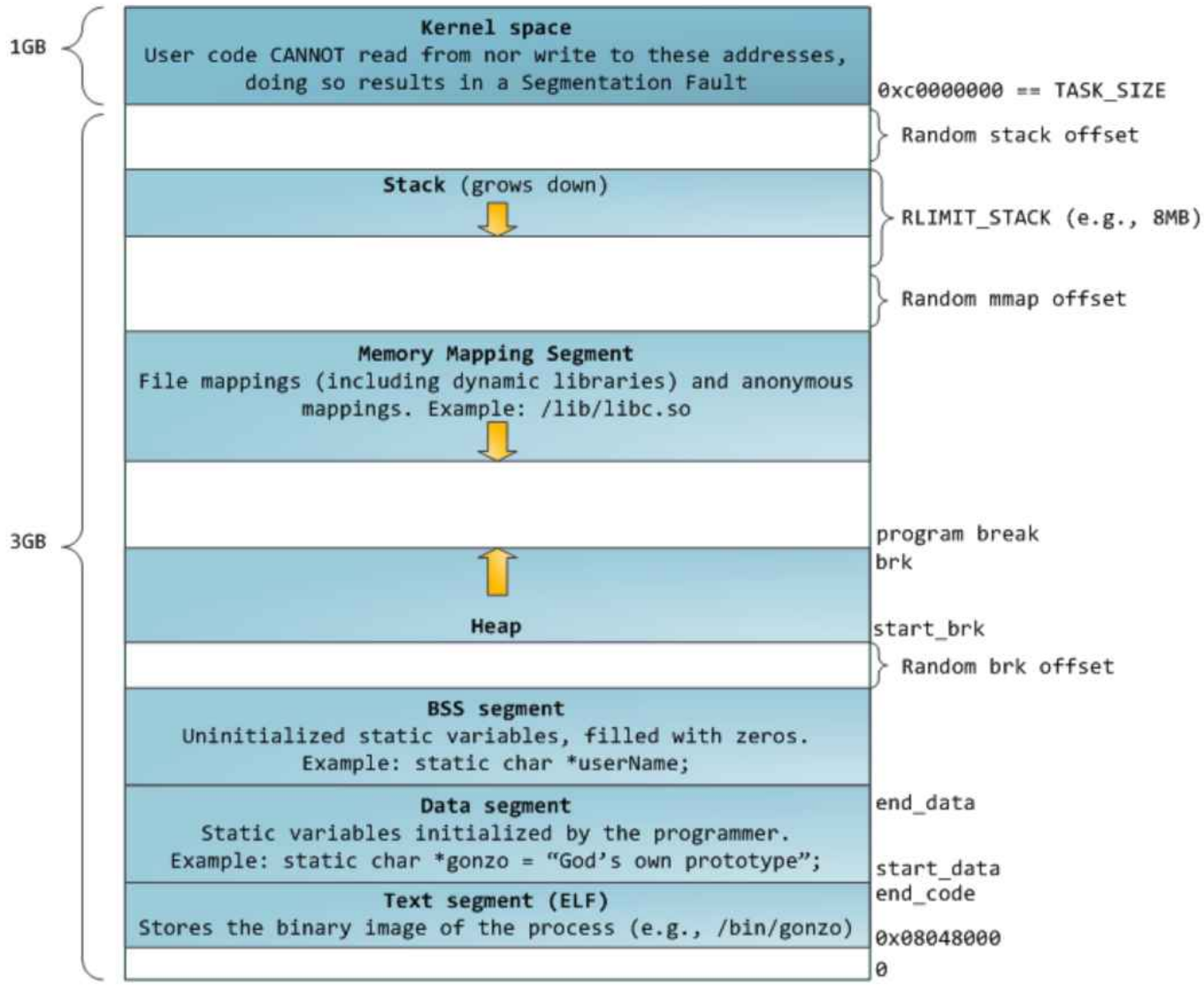
j. 进程同步？
   - 互斥量
   - 信号量：P、V操作，整型变量
   - 管程：
     - 一个或多个过程，一个初始化序列和局部数据组成
       - 局部数据变量只能被管程的过程访问，任何外部过程都不能访问、
       - 一个进程通过调用管城的一个过程进入管程
       - 任何时候，只能有一个进程进去管程，其他的都被堵塞
     - 管程使用条件变量实现对同步的支持，条件变量只能在管程中被访问
   - 事件/信号
   - 消息传递
   - 文章

k. 进程互斥？
   - 临界区：对临界资源进行访问的那段程序代码
   - 临界区值允许一个进程进入，其他进程阻塞，等待临界资源释放

l. 如何解决临界区冲突/临界区的调度规则？
   - ①一次仅允许一个进程进入（加锁）
   - ②其他试图进入的进程必须等待
   - ③已经进入临界区的进程要在有限时间诶退出
   - ④如果不能进入临界区则应让出CPU，避免出现“忙等”现象（忙等：不断的测试循环代码中变量的值，占用处理机不释放）

m. 进程内存布局？
   - 高地址向低地址：
   - 栈：编译器自动分配
   - 映射区：动态链接库等文件映射、mmap申请大内存
   - 堆：程序员使用new、malloc申请，程序员不释放会由os释放
   - BSS段：未初始化的全局变量
   - 数据段：初始化的全局变量、静态数据

- 代码段：常量、二进制代码



n. 有了进程，为什么要有线程？

- 进程切换系统开销大，线程切换仅需保存和设置少量寄存器内容
- 适合高并发环境，线程成为并发执行的最小单元
- 节省空间：线程之间共享内存空间和资源，无序拷贝（eg：b去完成a的一个任务，进程b需要拷贝a很大一部分资源）
- 节省时间：进程切换需要切换虚拟内存，线程不需要；进程切换虚拟内存导致TLB被清空从而失效，线程节省时间

o. 线程通信？

- 进程资源独立
- 线程资源共享：信号量、锁、原子操作

2、线程通信的方式

线程通信主要可以分为三种方式，分别为**共享内存**、**消息传递**和**管道流**。每种方式有不同的方法来实现

- 共享内存：线程之间共享程序的公共状态，线程之间通过读-写内存中的公共状态来隐式通信。

> volatile共享内存

- 消息传递：线程之间没有公共的状态，线程之间必须通过明确的发送信息来显示的进行通信。

> wait/notify等待通知方式
> join方式

- 管道流

> 管道输入/输出流的形式

- 锁机制：
  - 互斥锁：在同一时间只能有一个线程访问临界资源
  - 读写锁：读模式下共享，写模式下互斥
  - 自旋锁：上锁受阻的时候线程不阻塞，轮询查看是否获得了该锁
  - 条件变量：以原子的方式则阻塞，直到某个条件为真。对条件的测试始终在互斥锁的保护下，线程解开互斥锁并阻塞线程等待条件变化，条件为真时进行通知；条件变量和互斥锁始终一起使用，
- 信号量机制：整数变量
- 信号机制：

p. 线程同步？

- 互斥量：互斥对象只有一个，可以保证公共资源不被多个进程访问
- 信号量：整型变量，允许多个线程同一时刻访问同一资源，但是需要限制访问的最大数目（最大资源数 = 1就是互斥量）
- 临界区：对临界资源进行访问的那段程序代码；多个线程访问一个独占性共享资源，拥有临界区的对象被保护起来，其他想访问临界区的线程被挂起
- 事件/信号：通知/唤醒线程有时间发生

	优点	**缺点**
临界区	保证某一时刻只有一个线程访问的简便方法，速度快	只能用来同步本进程内的线程
互斥量	可以在不同进程的线程之间实现资源的安全共享；和临界区很相似，但是互斥量是可以命名的	创建耗费资源更多
信号量	适用于Socket程序中的同步	必须有**公共内存**，不适合分布式操作系统，不易管理可控制、易出错
事件/信号	可以时间不同进程中线程的同步	

q. 线程分类？

- 用户级线程：有关线程管理的所有工作都由应用程序完成；不依赖操作系统，操作系统内核意识不到用户级线程的存在
- 内核级线程/轻量级线程：有关线程的所有工作都是由内核完成的；应用程序没有进程线程管理的代码，只有对接操作系统的API

r. 多线程如何实现？

- CPU通过给每个线程分配CPU时间片来实现多线程，时间片很短，让我们感觉到多个线程同时进行

s. 线程安全？

- 多个线程同一时刻对同一个全局变量做写操作，如果和预期结果一样，就称为线程安全

t. **线程崩溃会怎么样？**（一个进程有10个线程，如果down 掉一个线程会不会对其他的有影响？）

- 线程崩溃触发了 segment fault ，触发SIGSEGV信号
- 不屏蔽信号，所有都会崩溃

- 屏蔽信号
  - 线程私有栈，**崩溃位置是栈**，屏蔽后不影响其他线程
  - 线程共有堆、全局变量，崩溃位置是堆、全局变量，屏蔽后崩溃影响其他线程
- 参考1

u. **线程资源怎么回收，线程怎么退出？**

- 线程可以简单地从启动历程中返回，返回值是线程的退出码
- 线程可以被同一进程中的其他线程所取消
- 线程调用pthread_exit

v. 协程？

- 含义：用户级的轻量线程，上下文切换由开发者决定，不由内核决定
- 通俗一点的解释：是函数或者一段程序能够被挂起/暂停，待会儿再恢复

```
void func(){
    printf("a");
    暂停再回来
    printf("b");
    暂停再回来
    printf("c");
}
```

- 缺点：
  - 无法使用CPU的多核（**协程本质是单线程**，需要和进程配合才能运行在多核CPU上）
  - 处处要使用非阻塞代码

w. 有了线程，为什么要有协程？

- 先有协程，后有进程，再有线程
- 线程是抢占式任务，操作系统控制，不知道什么时候被抢走，所有需要加锁；
- 协程需要编写代码者主动让出控制权，无需操作系统内核的上下文切换，不需要加锁
- 线程是同步机制
- 协程是异步机制

x. 进程的调度算法？

- 先来先服务：先来的先
- 短作业优先：短的先
- 时间片轮转：运行相同时间后切换
- 最短剩余时间优先：剩的少优先
- 高响应比优先：（等待时间 + 要求服务时间）/ 要求服务时间
- 优先级：选优先级高的
- CFS（Completely Fair Scheduler 完全公平调度）调度算法：
  - 红黑树，vruntime虚拟时间越小，越靠左，每次选最左边节点作为下一个任务O(1)
  - vruntime += 实际运行时间* 1024 / 进程权重(权重根据nice值进行计算，任务分配的优先级)

- vruntime并不是无限小为0，每个CPU运行队列维护一个min_vruntime，新进程的vruntime以此为准
- CFS缺点：休眠进程在唤醒时候会获得vruntime的补偿；因此主动休眠进程并不要求快速相应，交互式进程要求（鼠标键盘啥的），主动休眠进程因为补偿而抢占了更重要的进程

y. 进程终止的方式?

- 正常退出（5种）：main函数return自然返回、exit()系统调用
- 异常退出（3种）：
- 博客-《UNIX环境高级编程》的说明
- 正常退出（自愿）：完成了工作
- 出错退出（自愿）：严重的错误，编译文件却不存在
- 严重错误（非自愿）：程序中的错误
- 被其他进程杀死（非自愿）：杀死某个进程
- 《现代操作系统》的说明P51

z. 经典的锁?

- 互斥锁：确保一个同一时间只有一个线程访问临界资源
- 读写锁：读读不互斥，读写、写写互斥，写优先于读；共享版读写锁
- 自旋锁：一直循环尝试获取锁；适用于加锁时间短的场景
- 条件变量：等待一个条件为真，必须和互斥锁联合使用，利用共享的全局变量进行同步
- 递归锁：

	互斥锁	**条件变量**
	用来上锁	用来等待
作用	线程互斥	**线程同步**
	缺点：只有两种状态锁定和非锁定	

- **通常互斥锁和条件变量同时使用**：条件不足，线程解开互斥锁并阻塞线程等待条件变化；一旦有一个线程改变了变量，改变条件唤醒被阻塞的线程

递归锁	**非递归锁**	
同一线程中，想要多次获得一个锁，只能用递归锁		

aa. 锁的底层实现原理?

- lock()、unlock()
- 控制中断、测试并设置指令、比较并交换指令、链接的加载和条件式存储指令、获取并增加指令

- 讲解

ab. 孤儿进程、僵尸进程、守护进程

- 原因：一般情况下，子进程由父进程创建，但是父子进程的退出是无顺序的
- 孤儿进程：父进程先退出，子进程还没有退出，托孤给init进程
- 僵尸进程：子进程已经终止，父进程还没有调用wait、waitpid获取到其状态，子进程残留的状态信息变成僵尸进程
  - 如何避免僵尸进程：
    - fork()两次
    - 调用wait/waitpid获取子进程退出状态
    - 忽略SIGCHLD信号（子进程终止、收到SIGSTOP（19）信号后会产生）
    - 捕获SIGCHLD信号并在捕获程序中调用wait/waitpid函数
    - 讲解
- 守护进程：运行在后台，不与任何终端关联的进程；通常情况系统启动时就在运行；通常是周期性的执行某些任务
  - 如何创建守护进程：
    - 调用fork创建子进程，父进程调用exit退出
    - 调用setsid()使进程成为一个会话组长
    - 调用chdir()将当前目录更改为根目录
    - 调用umask()重新设置文件权限掩码umask(0)
    - 关闭不需要的文件描述符

ac. 多进程、多线程？

	多进程	**多线程**
数据共享、同步	数据共享复杂，需要IPC；同步简单	数据共享简单；同步复杂
内存	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高
创建销毁	创建、销毁、切换复杂	创建、销毁、切换简单
编程调试	编程简单，调试简单	编程复杂，调试复杂
自身多个影响	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉
分布式	适应于多核、多机分布式	适应于多核分布式

ad. 什么时候用多进程？什么时候用多线程？进程和线程和使用场景

- 需要频繁创建/销毁：多线程
- 需要进行数据共享的：多线程
- CPU/计算密集型：多进程；IO密集型：多线程讲解
- 弱相关的处理：多进程；强相关的处理：多线程
- 多机分布：多进程；多核分区：多线程
- Nginx、Chrome是多进程方式

ae. 接收一个文件来说，为什么多线程接收感觉快一点

- 文章
- 单线程：
  - 场景一：从头接收到尾，每次5MB
  - 场景二：接受5MB之后需要其他操作，单线程
- 多线程：
  - 场景一：一个从头开始，一个从尾开始，每次可以两个5MB
  - 场景二：从头接受5MB之后进行其他操作，其他线程还可以从尾部读取5MB然后进行处理
- 线程越多越好？
  - 线程调度开销变大，性能降低
  -

af. 线程安全？

- 线程安全：多个线程运行结果和单线程运行的结果是一样的，其他变量的值和预期也是一样的
- 线程安全问题都是由全局变量和静态变量引起（若每个线程对全局、静态变量只有写，没有读操作，那么是线程安全的）
- 若多个线程同时执行写操作，一般需要考虑线程同步，否则可能有线程安全问题
- 线程安全实现方式：加锁、非阻塞同步、线程本地化

ag. **乐观锁、悲观锁**

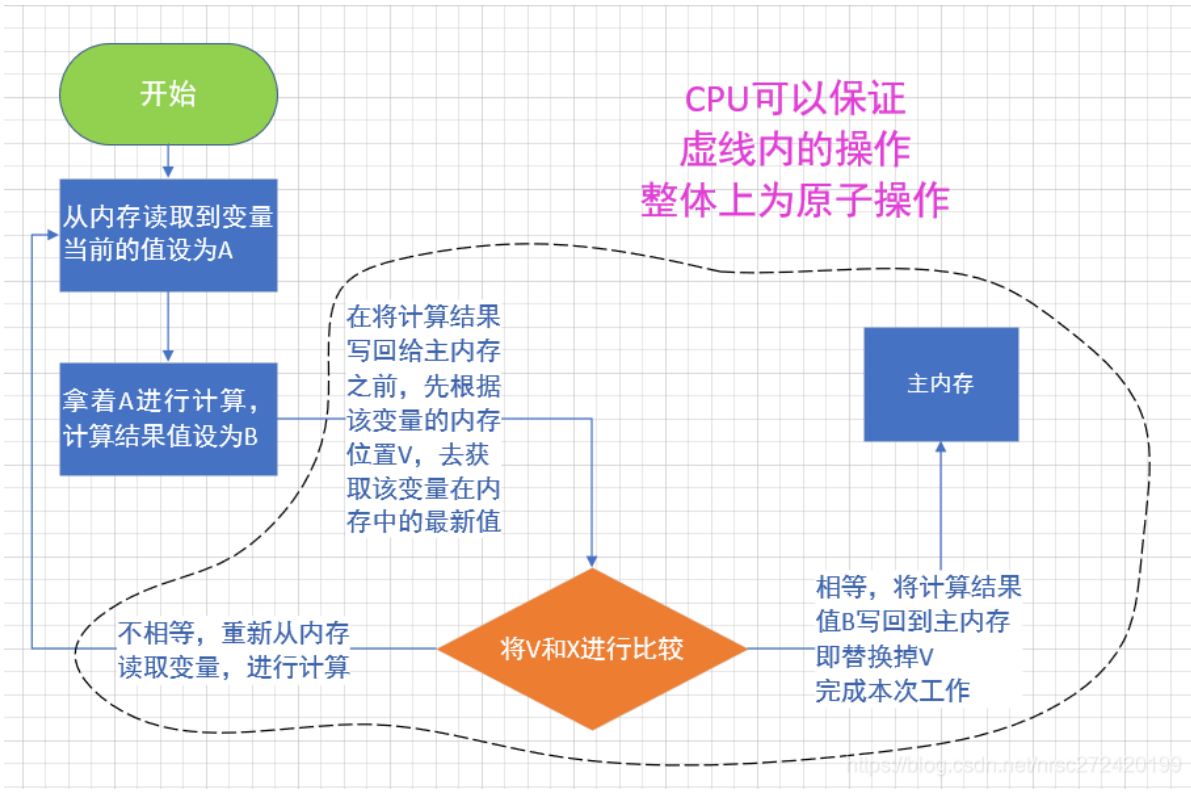
- 乐观锁实现：记录数据版本（version）或者是时间戳、CAS

ah. 生产者、消费者

- 生产、消费、队列
- 全局互斥锁，同一时间时间只能有一个角色访问队列
- 两个条件变量：阻塞队列为空或者队列已经满
- 文章

ai. CAS？

- CAS是一条CPU并发原语，整个操作是一个**原子操作**
- 解决多线程并行情况下使用锁造成性能损耗的一种机制
- CAS如何保证原子性：
  - 总线锁定：
    - 使用总线锁开销较大，会导致大量阻塞
  - 缓存锁定：
    - 在同一时刻，我们只需保证对某个内存地址的操作是原子性即可
    - 缓存锁定是基于**缓存一致性（先更新数据库，再删除缓存最为广泛应用）**机制来实现的，缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，现代CPU基本都支持和使用缓存锁定机制
- 具体实现：
  - 三个操作数：内存地址（V，V中存的是A）、进行比较的值（A）、拟写入的新值（B）
  - 当且仅当V上的值==A时，CAS用B来更新V；否则不更新，告诉一个新的A值（一般是个自旋操作）



- ABA问题：A->B->A，并不能察觉被改变过（解决：**加一个版本号version**）
- CAS好文

死锁

- Conception：

  a. 死锁

  b. 银行家算法

- Question：

  a. **死锁?**

  - 多个进程在运行过程中因争夺资源而造成的一种僵局，占有自身资源并请求对方资源

  b. **产生死锁的原因?**

  - 资源分配不当，系统资源不足
  - 程序推进的顺序不合适

  c. **产生死锁的必要条件?**

  - 互斥条件：一段时间内某资源仅为一个进程占用
  - 请求和保持条件：**对已获得的资源保持不放**
  - 不剥夺条件：获得之后不能被剥夺
  - 环路等待条件：存在环形链

  d. **解决死锁的方法?**

  - 预防死锁
    - 破坏互斥条件没有实用价值
    - 一次性分配所有资源（破坏请求条件）
    - 有其中一个资源得不到，其他的也不给分配（破坏请保持条件）
    - 已经得到部分，但得不到其他，释放拥有的（破坏不可剥夺条件）
    - 给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源（破坏环路等待条件）
  - 避免死锁

        - 在进行资源分配之前预先计算资源分配的安全性，如果会导致系统进入不安全的状态，则分配给进程
        - 解决：银行家算法
    - 检测死锁
        - 首先为每个进程和每个资源指定一个唯一的号码
        - 然后建立资源分配图
        - 判断是否有环路
    - 解除死锁
        - 剥夺资源/终止进程/撤销进程
        - 进程回退/回滚
    - 鸵鸟策略
        - 原因：解决死锁的代价很高
        - 方法：影响不大，发生概率很低时候，假装根本没发生问题
        - Unix、Linux、Windos解决死锁的办法仅仅是忽略

内存管理

- Conception：
    a. 栈、堆
    b. 静态存储区
    c. 大端、小端
    d. 缓冲区溢出
    e. 分页、分段
    f. 逻辑地址（有效地址）、线性地址（虚拟地址）、物理地址
    g. 页面置换算法
    h. 动态分区分配算法
    i. 交换空间
    j. 虚拟内存
    k. 外存（磁盘）：分为文件区（离散分配）和对换（连续分配）区
    l. 抖动/颠簸
    m. RAM
    n. 操作系统局部性原理
    o. 写时复制COW
    p. MESI协议
- Question：
    a. **栈、堆**
        - 栈：系统自动分配；高地址向低地址；小且快；栈比较死
        - 堆：程序员分配；低地址向高地址；大且慢；堆灵活
        - 详解文章
        - **为什么有了堆，还要有栈？/为什么有了栈，还要有堆？**

- 栈：**一条指令（上下移动esp）就可以实现**，方便、快速；有效避免内存碎片，不用手动free，但是不利于管理大内存，不利于动态管理内存资源
- 堆：栈需要先进后出，无法自由的控制生命周期，但是管理和申请相对复杂

b. **栈溢出的情况?**

- 局部数组过大
- 递归调用层次太多。压栈次数太多
  - 函数中的变量都是在栈空间，
  - 当输入的变量值太大，超过给变量赋予的大小，这时候就会把这个栈空间的一些指令给覆盖掉，比如覆盖到返回地址。
  - 然后覆盖返回地址的为jmp之类的跳转指令，然后jmp跳到后面你写的恶意代码指令，就造成了缓冲区溢出攻击

c. **函数调用过程中栈的变化?**

- ESP：栈顶指针
- EBP：栈底指针（在最上面）
- 栈（栈顶）分配开始前有一定空余，和数据对齐有关
- 栈从**高地址向低地址**延伸
- 参数**从右到左**压入栈
-
- 栈顶空余、压入参数、保存main函数基指针（ebp、esp重合）、esp下移创建新的栈帧、拷贝参数、执行函数操作、函数返回/ebp回到起始位置，main函数返回
- 参考1
- 参考2

d. **静态存储区?**

- 一定会存在且不会消失的数据
- 常量（const）、静态变量、全局变量
- 程序编译完就分配好，直到程序结束才释放

e. **大端、小端**

- 网络字节序：大端
- 主机字节序：小端
- 小端：低地址放低位 0x12345678　78 56 34 12
- x86_64是小端
- 如何判断：

```
#include<iostream>

using namespace std;


int main(){
    union{
        int n;
        char ch;
    } data;
```

```
11      data.n = 0x00000001;
12      //union下公用一段四字节地址空间，data.ch只占一个字节，小端情况下为01，大端情况
13      cout << data.n << endl;
14      cout << data.ch << endl;;
15      if(data.ch == 1){
16          cout << "little" << endl;
17      }
18      else{
19          cout << "big" << endl;
20      }
21      return 0;
22  }
```

f. **缓冲区溢出？危害？**

- 缓冲区：暂时放置输入、输出资料的内存
- 缓冲区溢出：向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖了合法数据
- 危害：程序崩溃、跳转执行一段恶意代码

g. **分页、分段？**

	段	**页**
定义	提高内存利用率，每个段是连续分配的，段和段之间是离散分配的	把内存分成大小相等且固定的块，主存的基本单位
单位	信息的逻辑单位、根据用户需要划分	信息的**物理单位**，为了管理主存方便划分
大小	大小不固定，可以动态改变	**大小固定**，系统决定
地址空间	向用户提供二维地址空间（段名+段地址）	向用户提供一维地址空间
作用	分段系统能反映程序的**逻辑结构**，便于段的**共享与保护**	**提高内存的利用率**，主要用于实现**虚拟内存**，获得更大的地址空间

h. **物理地址、逻辑地址、有效地址、线性地址、虚拟地址？**

物理地址	**逻辑地址**
物理内存真正的地址	计算机用户看到的地址

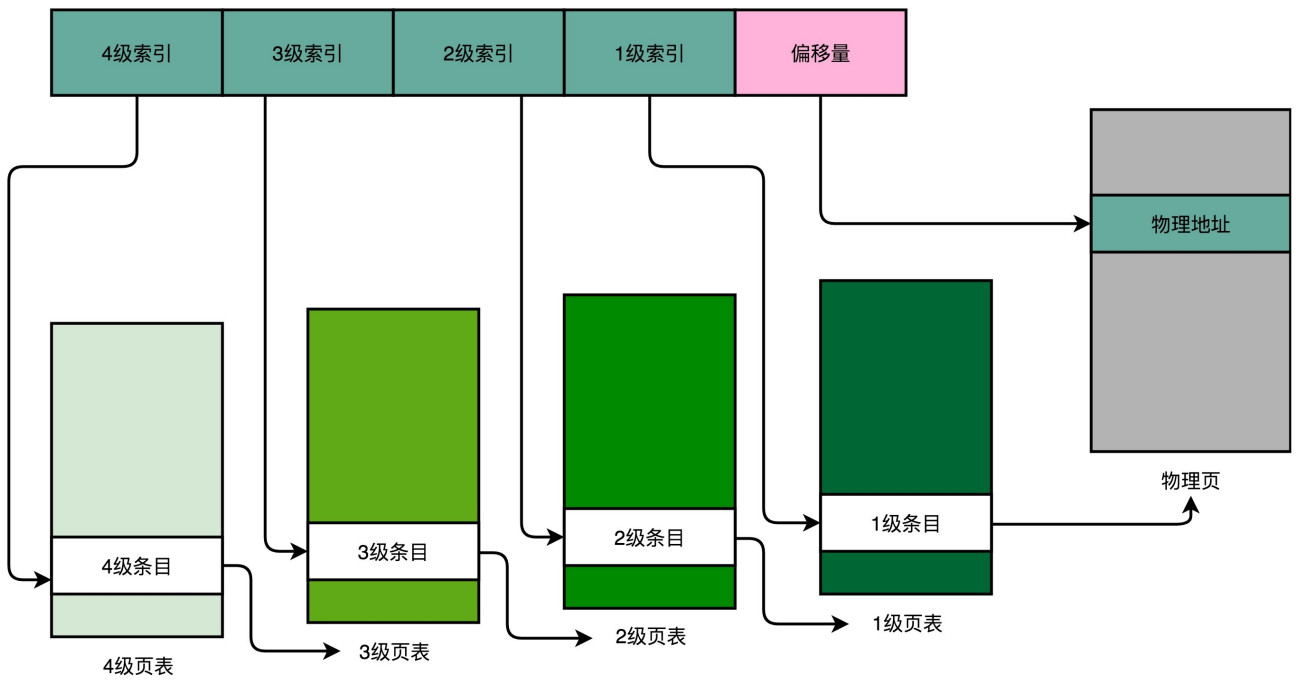
地址名称	**变换**
逻辑地址 = 有效地址	
线性地址（虚拟地址）	逻辑地址分段后变为线性地址；不启用分页功能，线性地址即物理地址；开启的话线性地址又多了个名字虚拟地址
物理地址	线性地址分页后变为分段地址

- 无论哪种地址，最后都映射成物理地址

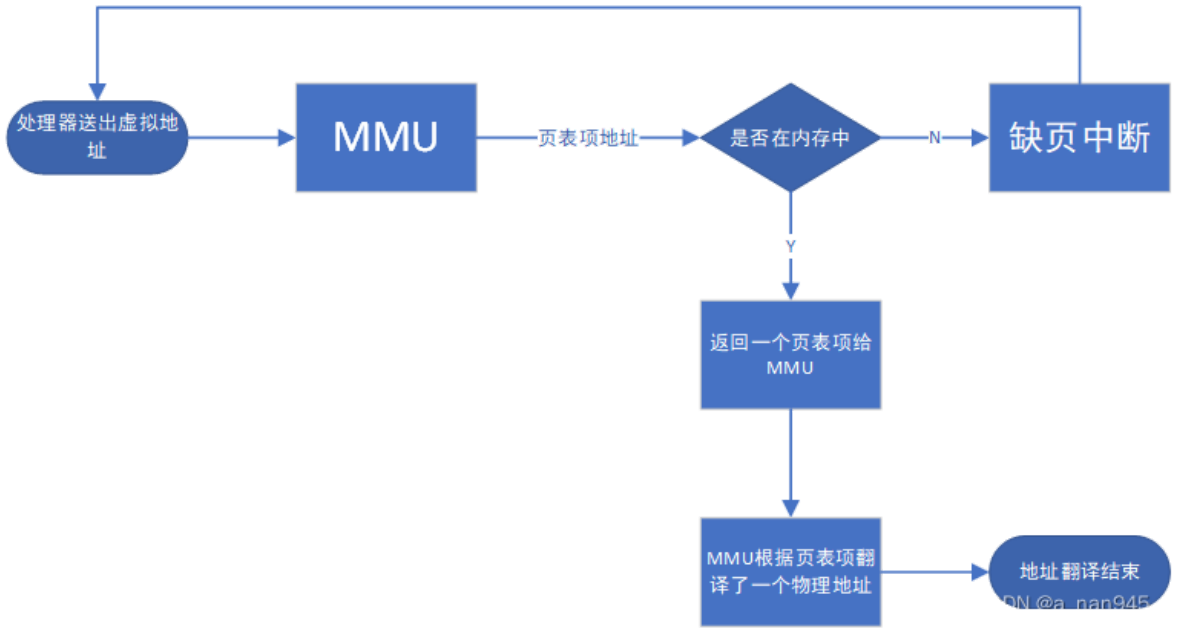
- CPU工作方式：实模式、保护模式参考

i. **虚拟地址如何映射成物理地址？**

- 单极页表：页号+偏移量
- 多级页表：多级索引+偏移量



- 段表：段号+偏移量
- 段页式：段号、段内页号、页内位移
- MMU（内存管理单元 Memory Management Unit）：完成地址变换，和TLB的访问和交互



- TLB（地址变换高速缓冲 Translation-Lookaside Buffer）：指令的内存是连续的，通常在一个虚拟页中，把之前的内存转换地址缓存下来，不需要重复访问（LRU）
- 参考

j. **页面置换/替换算法？**

- 先进先出
- 第二次机会算法：存在一个R位检查是否仍在使用，R位为0直接淘汰，为1置0放在链表末尾
- 最佳置换算法：选未来最远要使用的淘汰，并不可行
- 最近最久未使用（LRU：Least Recently Used）
- 最近最少未使用（LFU：Least Fruquently Used）：淘汰访问次数最少的

- 时钟置换（实际使用）：环形链表；第一个次加载页面的时候，页面标记被设置为0，调用内存中驻留的页面时，标记该页面为1；出现缺页时，指针从当前位置循环查找环形链表，如果遇到标记为1的，标记为0，如果遇到标记为0的，置换它
- 最近未使用（NRU）

k. **动态分区分配算法?**

- 首次适应算法
- 最佳适应算法
- 最坏适应算法：每次使用大的连续块，避免碎片
- 临近适应算法：循环链表，从上次查找结束的地方继续查找

l. **交换空间?**

- 内存资源不足时，Linux把某些页的内容转移到硬盘的一块空间上，以释放内存空间，硬盘上的这块空间叫做交换空间
- 内存 = 桌面，空间有限，交换空间 = 地面，不用的东西先放在地上，用的时候再拿上来
- 物理内存 + 交换空间 = 虚拟内存可用量

a. **虚拟内存?**

- 使得应用程序认为拥有连续的可用的内存，实际上，是多个物理内存碎片
- 虚拟内存使用部分加载的技术，让一个进程的某些页面加入内存，从而能够加载更多的进程
- **优点**:
  - 较小的可用内存中执行较大的用户程序
  - **在内存中容纳更多程序并发执行**
  - 与覆盖技术相比，不必影响编程时的程序结构
- 缺点：占用了一定的物理硬盘空间
- 虚拟内存实现方式：请求分页存储管理、分段、段页式

a. **常见内存分配错误?**

- 内存没有分配成功，确使用了：if(进程检查)
- 内存分配成功，但未初始化
- 内存分配成功且初始化，但操作越界
- 忘记释放内存，内存泄露
- 释放内存依旧使用：释放后没有置为NULL，产生野指针

b. **外存？内存交换中，被换出的进程保存在哪里?**

- 保存在磁盘也就是外存中，磁盘 = 文件区 + 对换区，进程存放在对换区

文件区	对换区
存放文件	存放进程数据
离散分配	连续分配
追求空间利用率	提高对换速度

c. **抖动？颠簸?**

- 抖动/颠簸：刚刚换出的页面马上换入内存，刚刚换入的页面马上换出外存，频繁的页面调度
- 原因：进程频繁访问的页面数目比分配的物理快多
- 进程工作集：抖动影响效率，虚拟内存管理器将一定量的内存页驻留在内存中，根据进程工作的指标，动态调整这个页面数量

d. **RAM?**

- Random Access Memory，随机存取存储器
- 随机存储器。就是我们平时所说的“内存”，也就是手机/电脑等电子设备的运行空间。这地方可以类比“菜鸟驿站”，数据被调用，暂时存储此地，然而一定会被使用的。当然要是遇到爆仓，您的手机就很卡了
- 增加计算机的RAM为什么会提高性能？ C
- A. 增加了虚拟内存
- B. 更大的RAM本身速度更快
- C. 减少了缺页中断
- D. 减少了段错误

e. **操作系统局部性原理**

- 程序常常会使用集中在一起的局部数据，
- 时间局部性：被引用过一次的存储器位置会在未来被多次引用（通常在循环中）
- 空间局部性：一个存储器的位置被引用，那么将来他附近的位置也会被引用

f. 写时复制COW

-

g. MESI协议

-

h. **一些词**

- 页表：记录映射关系、页号到物理块号的映射（原因：页面离散的分配在内存中）
- 段表

文件管理

- Conception：

a. 硬链接、软链接（符号链接）

b. 磁盘调度算法

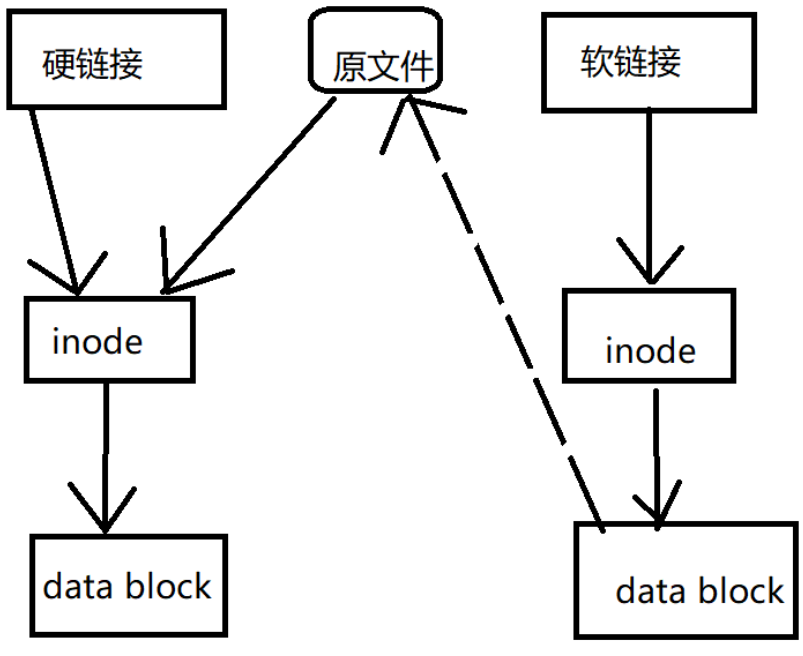
- Question：

a. 硬链接、软链接：

硬链接	**软链接（快捷方式）**
以文件副本的形式存在，指向同一个索引节点	数据块中存放的内容是另一文件的路径名的指向
索引节点中有链接计数count，==0时真正删除文件和索引节点	根据路径查找共享文件，及时源文件已经被删除，还是会去查找但失败

	查询多级目录，多次I/O操作，软链接更慢

- 共同点：**都不会将原本的档案复制一份，只会占用非常少量的磁盘空间**



- 硬链接：
  - 指向同一个文件/inode
  - 跨文件系统不可以实现，因为都指向一个
  - 删除所有文件的硬链接才会彻底删除文件
- 软链接：
  - 有独立的inode，再指向原文件
  - 可以跨文件系统
  - 感知不到文件被删除，只不过找不到而已

b. **空闲空间管理**

- 空闲链表法
- 位图法

c. **一次I/O的量级在什么量级？**

- ms

d. **磁盘调度算法？**

- 先来先服务（FIFO）
- 最短寻道时间优先（SSTF）
- 电梯扫描算法（SCAN）：向一个方向，到头后掉头
- 循环扫描算法（C-SCAN）：向一个方向，到头后再从头开始
- LOOK：一个方向，一个方向没请求后掉头
- C-LOOK：一个方向，一个方向没请求后从头开始

e.

各种算法

- Conception：
  a. 进程调度算法
  b. 页面置换/替换算法（发生缺页异常时调换页面）
  c. 动态分区分配算法（给一个进程分配一段内存）
  d. 磁盘调度算法（收到一个请求序列，在磁盘上的寻找顺序）
- Question：

汇编相关

- Concetion：
- Question：
  a. 汇编阶段将汇编代码转换为计算机可执行的语言，不同操作系统过程否相同？
     - **同一种汇编语言，翻译成的二进制代码完全相同**，汇编语言的定义就是用助记符代替机器指令，注意是代替，纯粹的替换操作，所以，如果是同一种语言，在同一个硬件平台上，那么同样的汇编指令对应的二进制编码是完全相同的。