1. 进程和线程的区别
   1. 进程是资源调度的最小单位，线程是执行任务的最小单位。
   2. 进程有自己独立的地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段，栈堆和数据段，这种操作非常昂贵。而线程是进程中的数据，使用相同的地址空间，因此切换线程的花费远比进程小。同时创建一个线程的开销也比进程小。
   3. 线程之间的通信更加方便，同一个进程之间的线程可以共享全局变量，静态变量等数据，而进程的通信需要通过IPC等方式，较麻烦。
   4. 不过多进程的程序更见健壮，多线程的程序处理不好会造成阻塞甚至死锁，从而造成整个系统崩溃，而多进程的程序一个进程死掉也不会影响另外一个进程，因为他们有独立的地址空间。
   5. 线程之间的处理较为复杂，容易出现死锁，如何处理好线程之间的同步是一个难点。
2. 进程程间的通信机制
   1. 管道：用于具有亲缘关系的父子进程间或者兄弟进程之间的通信。比如fork出来的进程和父亲进程的通信。
   2. 信号：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生；
   3. 消息队列：消息队列是消息的链表，管道和消息队列的通信数据都是先进先出的原则。例子：rabbitMq,rocketMq
   4. 共享内存：使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。
   5. 信号量：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。
   6. 套接字：此方法主要用于在客户端和服务器之间通过网络进行通信。
3. 进程调度算法
4. FCFS:先来先服务，按照作业进入就绪队列的次序进行调度
5. SJF：短作业优先，运行时间最短的进入
6. 时间片轮转调度算法：给每个队首的进程分配一个时间片，用完了就等待下一次时间片再进行。
7. 高响应比优先：响应比：（已等待时间+作业时间）/要求运行时间。
8. 优先权调度：根据进程的优先权来进行调度。
9. 多级队列调度算法：根据作业的性质不同，分成多个就绪队列，不同的队列使用不同的算法。
10. 线程同步(通信)的方式
    1. 互斥量（锁机制）：只有拥有锁的线程才能访问资源，一个资源只能有一个线程进行访问。
    2. 信号量：（AQS）同一时刻有多个线程进行访问，但是有信号量控制着最大的访问数量，通常用作同步。
    3. 事件（wait/notify）:等待通知机制。
    4. CAS
11. 僵尸进程和孤儿进程

僵尸进程：即子进程先于父进程退出后，子进程的PCB需要其父进程释放，但是父进程并没有释放子进程的PCB。僵尸进程实际上是一个已经死掉的进程，但是内存并没有完全被释放。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

危害：孤儿进程有init(1号进程)为他善后，因此是没有太大危害的。但是僵尸进程虽然实体已经没有了，但是还是在系统中占用着记录，这是对资源的一个浪费。

僵尸进程处理：杀死父进程，让他变为孤儿进程；重启系统；通过信号机制，让子进程退出时通知父进程，然后让父进程调用wait()方法清理子进程的信息。

1. 进程有哪几种状态

就绪状态：进程已经获得CPU以外的资源，等待CPU分配资源

运行状态：占用CPU资源运行

阻塞状态：进程等待某种条件，在满足条件之前无法运行

1. [协程的概念？](https://leetcode-cn.com/circle/discuss/XXGdoF/#27-%E5%8D%8F%E7%A8%8B%E7%9A%84%E6%A6%82%E5%BF%B5%EF%BC%9F)

协程是一种比【线程】更加【**轻量级**】的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。最重要的是，【协程】不是被操作系统【内核】所管理的，而完全是由【程序】所控制（也就是在【用户态】执行）这样带来的好处就是：性能得到了很大的提升，不会像线程切换那样消耗资源。不过java并没有对协程的支持。

1. **什么是虚拟内存？**

**首先，我们知道每一个进程都有它自己独立的地址空间，然后这些空间呢，通过分页技术又被划分成多个大小相等的页，但是呢，并不是所有的页都要加载到物理内存中去这个进程才能够运行程序，另一方面也是由于物理内存的不足的原因，现在物理内存普遍才4G,8G,全部加载进去肯定是不够用的，因此通过虚拟内存技术，让不需要的页暂时存放在硬盘时。当程序请求对应的地址空间的时候，会由操作系统进行映射，如果映射的物理内存不存在，就会去硬盘中读取并重新执行失败的指令。这样，在逻辑上程序似乎有很大的逻辑空间，实际上只对应了一部分的物理内存，还有一部分在硬盘上。**

**因此呢，虚拟内存技术主要适用于大的对象和数据，好处就是解决了内存不足，减少了内存的占用。坏处就是频繁的硬盘读取，可能会降低系统的速度。**

1. 分段和分页的区别？

段式存储管理：将程序分为若干段，比如说数据段，代码段，堆栈段等。他的好处就是没有内碎片，因为**段的大小的可以变化**的，你需要多少就分配多少。坏处就是有外碎片。

页式存储管理：将程序的逻辑地址分为大**小相等的页**，物理地址划分为大小相等的块，由**页表建立起页号到物理块号的映射**。在程序加载的时候，可以将任一页放入任一帧中，这些帧不必连续，从而实现了离散分离。优点就是没有外碎片，但是有内碎片，因为分配的大小不变，但是程序可能用不完。

段页式管理机制结合了段式管理和页式管理的优点。段页式管理机制就是把主存先分成若干段，每个段又分成若干页。

不同点：

1. 内存碎片不同，段式产生外碎片，页式产生内碎片
2. 分段是满足用户的需要，由用户进行划分，分页是满足系统的需要，由操作系统划分，更加方便管理
3. 大小不同，段是根据数据动态增长的，而页的大小只有一种，是不会变化的。
4. 地址不同，页的地址空间是一维的，而段的地址空间是二维的（段号，段内偏移）。
5. 页面置换算法

在虚拟内存技术中，当硬盘上的数据加载到内存中时，如果物理内存不足，则需要进行页面置换，将不需要的页换出来，从而能将数据加载到内存中去。常见的页面置换算法有：

* + - 1. FIFO（先进先出）:把在页面中停留最久的置换出去。(队列)
      2. LRU（最近最久未使用）：把在最近且最久没使用的置换出去。（双向链表+hashMap）
      3. LFU （Least Frequently Used）页面置换算法（最少使用页面置换算法）：该置换算法选择在之前时期使用最少的页面作为淘汰页。被使用次数最少的数据优先淘汰。每个数据块都有一个引用计数，按照引用计数排序，具有相同计数的数据块按时间排序。
      4. OPT（最佳置换算法）：选择未来长时间不被访问或者永远不会被访问的置换出去。是一种理想的算法，无法实现，因为无法进行预知。

1. 逻辑地址，物理地址，虚拟地址

逻辑地址：是程序中的段地址。由两部份组成，段标识符和段内[偏移量](https://baike.baidu.com/item/%E5%81%8F%E7%A7%BB%E9%87%8F)。

物理地址：实际的内存的地址。

虚拟地址。虚拟内存管理下的一个地址。

1. Windows下的内存是如何管理的

**1.虚拟内存：**  
最适合用来管理大型对象或者结构数组  
**2.内存映射文件：**  
最适合用来管理大型数据流（通常来自文件）以及在单个计算机上运行多个进程之间共享数据  
**3.内存堆栈：**  
最适合用来管理大量的小对象

1. 内碎片和外碎片

内碎片是已经被分配出去，但是没有被充分利用的空间，就是分配多了。

外碎片是没有被分配出去，但是由于太小了，无法分配给申请内存空间的进程的空闲内存。

1. 什么是临界区、如何解决冲突？

临界区：**每个进程访问临界资源的那段代码**。每次只能有一个进程进入临界区。如果已经有进程进去临界区，其他尝试进入临界区的进程必须等待。已经进入的进程要在有限的时间内退出，已让其他进程能够进入自己的临界区。

临界资源：**一次只允许一个进程访问的资源**。比如变量，数组，缓冲区等。

1. 用户态，内核态

在进程在执行运行自己代码的时候，就处于用户态，进程在大部分的情况下都是处于用户态的，但当需要操作系统帮助的时候，就需要切换到内核态。他们之间最大的区别就是权限不一样，用户态特权等级为3级，最低，内核态用户等级为0级，最高。有3种方式会使用户态切换到内核态：

1. 系统调用：比如文件操作，网络数据的发送等。
2. 异常：当CPU执行时发生了比如缺页异常等会切换到异常的内核相关进程种。
3. 外围设备中断：
4. 常用的IO模型

阻塞IO模型：应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

非阻塞IO模型：进程发起IO系统调用后，内核返回一个错误码而不会被阻塞；应用进程可以继续执行，但是需要不断的执行系统调用来获知 I/O 是否完成。如果内核缓冲区有数据，内核就会把数据返回进程。

IO复用模型：使用 select 或者 poll 等待数据，可以等待多个套接字中的任何一个变为可读。这一过程会被阻塞，当某一个套接字可读时返回，之后把数据从内核复制到进程中。（在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用。）

信号驱动IO模型：当进程发起一个IO操作，会向内核注册一个信号处理函数，然后进程返回不阻塞；当内核数据就绪时会发送一个信号给进程，进程便在信号处理函数中调用IO读取数据。

异步IO模型：当进程发起一个IO操作，进程返回不阻塞，但也不能返回结果；内核把整个IO处理完后，会通知进程结果。如果IO操作成功则进程直接获取到数据。

1. [fork 函数的作用？](https://leetcode-cn.com/circle/discuss/XXGdoF/#26-fork-%E5%87%BD%E6%95%B0%E7%9A%84%E4%BD%9C%E7%94%A8%EF%BC%9F)

它的作用是从已经存在的进程中创建一个子进程，而原进程称为父进程。

调用 fork(),当控制转移到内核中的 fork 代码后，内核开始做：

分配新的内存块和内核数据结构给子进程。

将父进程部分数据结构内容拷贝至子进程。

将子进程添加到系统进程列表。

fork返回开始调度器，调度。

**fork 调用失败的原因：**

1)系统中有太多进程。

2)实际用户的进程数超过限制。

1. [通过进程id查看占用的端口，通过端口号查看占用的进程 id？](https://leetcode-cn.com/circle/discuss/XXGdoF/#29-%E9%80%9A%E8%BF%87%E8%BF%9B%E7%A8%8Bid%E6%9F%A5%E7%9C%8B%E5%8D%A0%E7%94%A8%E7%9A%84%E7%AB%AF%E5%8F%A3%EF%BC%8C%E9%80%9A%E8%BF%87%E7%AB%AF%E5%8F%A3%E5%8F%B7%E6%9F%A5%E7%9C%8B%E5%8D%A0%E7%94%A8%E7%9A%84%E8%BF%9B%E7%A8%8B-id%EF%BC%9F)

通过id查看占用的端口：netstat -nap|grep 进程id

通过端口号查看正在占用的进程id,netstat -nap|grep 端口号

1. [如何查看占用内存比较多的进程？](https://leetcode-cn.com/circle/discuss/XXGdoF/#210-%E5%A6%82%E4%BD%95%E6%9F%A5%E7%9C%8B%E5%8D%A0%E7%94%A8%E5%86%85%E5%AD%98%E6%AF%94%E8%BE%83%E5%A4%9A%E7%9A%84%E8%BF%9B%E7%A8%8B%EF%BC%9F)

ps aux | sort -k4nr | head -N

1. [平常用什么 linux 命令比较多？如何打开文件并进行查找某个单词？怎么在某个目录下找到包含 txt 的文件？](https://leetcode-cn.com/circle/discuss/XXGdoF/#217-%E5%B9%B3%E5%B8%B8%E7%94%A8%E4%BB%80%E4%B9%88-linux-%E5%91%BD%E4%BB%A4%E6%AF%94%E8%BE%83%E5%A4%9A%EF%BC%9F%E5%A6%82%E4%BD%95%E6%89%93%E5%BC%80%E6%96%87%E4%BB%B6%E5%B9%B6%E8%BF%9B%E8%A1%8C%E6%9F%A5%E6%89%BE%E6%9F%90%E4%B8%AA%E5%8D%95%E8%AF%8D%EF%BC%9F%E6%80%8E%E4%B9%88%E5%9C%A8%E6%9F%90%E4%B8%AA%E7%9B%AE%E5%BD%95%E4%B8%8B%E6%89%BE%E5%88%B0%E5%8C%85%E5%90%AB-txt-%E7%9A%84%E6%96%87%E4%BB%B6%EF%BC%9F)
2. [用过 ping 命令么？简单介绍一下。TTL 是什么意思？](https://leetcode-cn.com/circle/discuss/XXGdoF/#218-%E7%94%A8%E8%BF%87-ping-%E5%91%BD%E4%BB%A4%E4%B9%88%EF%BC%9F%E7%AE%80%E5%8D%95%E4%BB%8B%E7%BB%8D%E4%B8%80%E4%B8%8B%E3%80%82ttl-%E6%98%AF%E4%BB%80%E4%B9%88%E6%84%8F%E6%80%9D%EF%BC%9F)

ping命令后面紧跟ip号，可以查看和某台主机的连接情况。

1. [怎么判断一个主机是不是开放某个端口？](https://leetcode-cn.com/circle/discuss/XXGdoF/#219-%E6%80%8E%E4%B9%88%E5%88%A4%E6%96%AD%E4%B8%80%E4%B8%AA%E4%B8%BB%E6%9C%BA%E6%98%AF%E4%B8%8D%E6%98%AF%E5%BC%80%E6%94%BE%E6%9F%90%E4%B8%AA%E7%AB%AF%E5%8F%A3%EF%BC%9F)

talnet ip号 端口号