- 线程和进程
  - 进程和线程的区别
  - 多进程和多线程
  - 协程
  - 进程同步机制
  - 线程同步机制
  - 线程的状态
  - 进程切换
  - 进程调度策略
  - 线程调度策略
  - 解释生产者消费者模型；
- 内存
  - 虚拟内存和物理内存
  - 内存管理
  - 页面置换算法
  - 分页
  - 分段
  - 分段和分页的不同
  - 段页
  - Linux中Buffer和Cache
- IO模型
  - I/O多路复用
    - select
      - seclect实现原理
    - poll
    - epoll
      - epoll底层是什么？epoll在内核级是怎么实现信息回调的？底层怎么实现的？怎么获取epoll里面的信息，那个函数？
    - 区别
  - proactor reactor 是什么，区别在哪里
- 命令
  - LINUX找特定的文本段
  - LINUX如何杀掉特定名称的进程
  - 内核态和用户态区别
  - 主线程写一个buf，子线程去读，怎么做？读写时候的游标更新可能会出什么问题？怎么解决？
  - 操作系统中的锁
  - 只有互斥锁，如何实现读写锁？

线程和进程

进程和线程的区别

进程：指在系统中正在运行的一个应用程序；程序一旦运行就是进程；或者更专业化来说：进程是指程序执行时的一个实例，即它是程序已经执行到课中程度的数据结构的汇集。从内核的观点看，进程的目的就是担当分配系统资源（CPU时间、内存等）的基本单位。

线程：系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元执行流。进程——资源分配的最小单位，线程——程序执行的最小单位。

线程进程的区别体现在4个方面

1、因为进程拥有独立的堆栈空间和数据段，所以每当启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这对于多进程来说十分“奢侈”，系统开销比较大，而线程不一样，线程拥有独立的堆栈空间，但是共享数据段，它们彼此之间使用相同的地址空间，共享大部分数据，比进程更节俭，开销比较小，切换速度也比进程快，效率高，但是正由于进程之间独立的特点，使得进程安全性比较高，也因为进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。一个线程死掉就等于整个进程死掉。

2、体现在通信机制上面，正因为进程之间互不干扰，相互独立，进程的通信机制相对很复杂，譬如管道，信号，消息队列，共享内存，套接字等通信机制，而线程由于共享数据段所以通信机制很方便。。

3、体现在CPU系统上面，线程使得CPU系统更加有效，因为操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

4、体现在程序结构上，举一个简明易懂的列子：当我们使用进程的时候，我们不自主的使用if else嵌套来判断pid，使得程序结构繁琐，但是当我们使用线程的时候，基本上可以甩掉它，当然程序内部执行功能单元需要使用的时候还是要使用，所以线程对程序结构的改善有很大帮助。

多进程和多线程

多进程：

多进程**优点**：

1、每个进程互相独立，不影响主程序的稳定性，子进程崩溃没关系；

2、通过增加CPU，就可以容易扩充性能；

3、可以尽量减少线程加锁/解锁的影响，极大提高性能，就算是线程运行的模块算法效率低也没关系；
4、每个子进程都有2GB地址空间和相关资源，总体能够达到的性能上限非常大。
多进程**缺点**：
1、逻辑控制复杂，需要和主程序交互；
2、需要跨进程边界，如果有大数据量传送，就不太好，适合小数据量传送、密集运算 多进程调度开销比较大；
3、最好是多进程和多线程结合，即根据实际的需要，每个CPU开启一个子进程，这个子进程开启多线程可以为若干同类型的数据进行处理。当然你也可以利用多线程+多CPU+轮询方式来解决问题……
4、方法和手段是多样的，关键是自己看起来实现方便有能够满足要求，代价也合适。
多线程：
多线程的**优点**：
1、无需跨进程边界；
2、程序逻辑和控制方式简单；
3、所有线程可以直接共享内存和变量等；
4、线程方式消耗的总资源比进程方式好。
多线程**缺点**：
1、每个线程与主程序共用地址空间，受限于2GB地址空间；
2、线程之间的同步和加锁控制比较麻烦；
3、一个线程的崩溃可能影响到整个程序的稳定性；
4、到达一定的线程数程度后，即使再增加CPU也无法提高性能，例如Windows Server 2003，大约是1500个左右的线程数就快到极限了（线程堆栈设定为1M），如果设定线程堆栈为2M，还达不到1500个线程总数；
5、线程能够提高的总性能有限，而且线程多了之后，线程本身的调度也是一个麻烦事儿，需要消耗较多的CPU。

协程

一种用户态的轻量级线程，完全由用户调度控制，拥有自己的寄存器上下文和栈，协程调度切换的时候，先将寄存器上下文和栈保存到其他地方，切换回来的时候再恢复之前保存的寄存器上下文和栈。直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

但是同一时间只能执行一个协程，大致来说是一系列互相依赖的协程间依次使用CPU，每次只有一个协程工作，而其他协程处于休眠状态，适合对任务进行分时处理；而线程，一次可以执行多个线程，适合执行多任务处理。

例：在主线程中生产数据，协程中消费数据
yield是python中的语法，当协程执行到yield关键字时，会暂停在那一行，等到主线程调用send方法发送了数据，协程才会接到数据继续执行。
yield让协程暂停，和线程的阻塞有本质区别。协程的暂停完全由程序控制，线程的阻塞状态是由操作系统内核来切换。**协程的开销远小于线程的开销**

进程同步机制

为什么要引入进程间的同步？由于操作系统中的进程是并发的，因此当协同进程对共享数据进行访问时，可能会造成数据的不一致性问题。为了保证数据的一致性，那么我们就需要一种有效地机制，这就是被我们称之为的进程同步机制。

进程间资源访问冲突的类型

进程间资源访问的冲突主要有两种类型，分别为

- 共享变量的访问冲突
- 访问顺序的冲突

对于共享变量的访问冲突，我们解决的方式就是**进程的互斥**，而对于进程间访问顺序的冲突，我们解决的方式就是**进程的同步**。

1) 信号量机制

一个信号量只能置一次初值，以后只能对之进行p操作或v操作。
由此也可以看到，信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点。

2) 自旋锁

旋锁是为了保护共享资源提出的一种锁机制。
调用者申请的资源如果被占用，即自旋锁被已经被别的执行单元保持，则调用者一直循环在那里看是否该自旋锁的保持着已经释放了锁
自旋锁是一种比较低级的保护数据结构和代码片段的原始方式，可能会引起以下两个问题;
（1）死锁
（2）过多地占用CPU资源

3) 管程

信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。因此后来又提出了一种集中式同步进程——管程。其基本思想是将共享变量和对它们的操作集中在一个模块中，操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰，便于维护和修改，易于保证正确性。

4) 会合

进程直接进行相互作用

5) 分布式系统

由于在分布式操作系统中没有公共内存，因此参数全为值参，而且不可为指针。

线程同步机制

同步与互斥的区别与联系

互斥：是指散布在不同进程（线程）之间的若干程序片断，当某个进程（线程）运行其中一个程序片段时，其它进程（线程）就不能运行它们之中的任一程序片段，只能等到该进程（线程）运行完这个程序片段后才可以运行。

同步：是指散布在不同进程（线程）之间的若干程序片断，它们的运行必须严格按照规定的 某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。

同步是一种更为复杂的互斥，而互斥是一种特殊的同步。也就是说互斥是两个线程之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步大部分情况下也是不能同时运行，但他是必须要安照某种次序来运行相应的线程，而且少数时候同步可以允许多个线程同时运行！

1）临界区

通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。它并不是核心对象，不是属于操作系统维护的，而是属于进程维护的。

2）互斥对象

互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。

3）信号量

信号量也是内核对象。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。

在用CreateSemaphore()创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最 大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1 ，只要当前可用资源计数是大于0 的，就可以发出信号量信号。但是当前可用计数减小 到0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离 开的同时通过ReleaseSemaphore （）函数将当前可用资源计数加1 。在任何时候当前可用资源计数决不可能大于最大资源计数。

4）事件对象

通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。即事件机制允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务。

线程的状态

创建、就绪、运行、阻塞、死亡

进程切换

进程调度策略

1. **先来先服务调度算法**：先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。
2. **短作业(进程)优先调度算法**：短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。
3. **高优先权优先调度算法**：为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

- 3.1) **非抢占式优先权算法**：在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。
- 3.2) **抢占式优先权调度算法**：在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

4. **高响应比优先调度算法**：在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

   在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

5. **时间片轮转法**：在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

线程调度策略

解释生产者消费者模型；

内存

虚拟内存和物理内存

内存管理

页面置换算法

分页

1. 把主存空间划分为大小相等且固定的块，块相对较小，作为主存的基本单位
2. 页面：
   1)页面大小应是2的幂，通常为512B-8KB
   2)页面小内存利用率高，但页表长占用内存多，页面换进换出效率低 3)页面大页表占用内存少，页面换进换出效率高，但是内存碎片大
3. 地址变换机构
   1)页面映射表的作用就是用于实现从页号到物理块号的变换，地址变换任务是借助于页表来完成的
   2)基本的地址变换机构：1)访问内存中的页表，从中找到指定页的物理块号，以形成物理地址 2)从第一次所得地址中获得所需数据
   3)具有快表的地址变换机构：1)增加高速缓存 2)页号存在高速缓存中，直接从快表中读出该页所对应的物理块号，并送到物理地址寄存器中
4. 页表
   1)记录页面在内存中对应的物理块号
   2)两级和多级页表

分段

1. 作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息
2. 引入分段存储管理方式的目的：
   1)方便编程
   2)信息共享：1、分段是以信息的逻辑单位为基础的 2、“页”只是存放信息的物理单位(块)，并无完整的意义
   3)信息保护
   4)动态增长：数据段在使用过程中会不断地增长，而事先又无法确切地知道数据段会增长到多大
   5)动态链接：运行过程中又需要调用某段时，才将该段(目标程序)调入内存并进行链接
3. 段表：
   1)每个进程都有一张逻辑空间和内存空间映射的段表
   2)段表是用于实现从逻辑段到物理内存区的映射
4. 地址变换机构

分段和分页的不同

1. 分页是系统管理的需要：减少内存碎片；分段是应用程序的需要：提供一组完整的信息
2. 一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处
3. 页大小固定且由系统决定；段的长度不固定
4. 页式系统地址空间是一维的；分段的作业地址空间是二维的，程序换在标识一个地址时，既需要给出段名，又需要给出段内地址：1)页号直接加上业内地址 2)二维：段名和段内地址，先找到某段，再找到段内地址

段页

1. 分段和分页原理的结合
2. 页式存储管理能有效地提高内存利用率，而分段存储管理能反映程序的逻辑结构并有利于段的共享
3. 作业的地址空间首先被分成若干个逻辑段，每段都有自己的段号，然后再讲每一段分成若干个大小固定的页
4. 在一个进程中，段表只有一个，而页表可能有多个
5. 在段页式系统中，为了获得一条指令或者数据，需三次访问内存：
   1)第一次访问是访问内存中的段表，从中取得页表始址
   2)第二次访问是访问内存中的页表，从中取出该页所在的物理块号，并将该块号与业内地址一起形成指令或者数据的物理地址
   3)从第二次访问所得的地址中，取出指令或者数据

Linux中Buffer和Cache

1、Buffer（缓冲区）是系统两端处理**速度平衡**（从长时间尺度上看）时使用的。它的引入是为了减小短期内突发I/O的影响，起到**流量整形**的作用。比如生产者——消费者问题，他们产生和消耗资源的速度大体接近，加一个buffer可以抵消掉资源刚产生/消耗时的突然变化。

2、Cache（缓存）则是系统两端处理**速度不匹配**时的一种**折衷策略**。因为CPU和memory之间的速度差异越来越大，所以人们充分利用数据的局部性（locality）特征，通过使用存储系统分级（memory hierarchy）的策略来减小这种差异带来的影响。
3、假定以后存储器访问变得跟CPU做计算一样快，cache就可以消失，但是buffer依然存在。比如从网络上下载东西，瞬时速率可能会有较大变化，但从长期来看却是稳定的，这样就能通过引入一个buffer使得OS接收数据的速率更稳定，进一步减少对磁盘的伤害。

IO模型

I/O多路复用

通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

在不使用线程，独立处理文件的情况下，进程无法在多个文件描述符上阻塞。

同步阻塞IO：假如其中某个文件描述符没有准备好，进程就会阻塞，不能再处理其它文件，直到该文件描述符准备好。

同步非阻塞IO：假如其中某个文件描述符没有准备好，向进程返回一个错误信息，从而避免阻塞。

若是采用非阻塞IO,进程需要以某种不确定的方式不断发起IO操作，直到某个打开的文件描述符已经就绪。

IO多路复用允许应用在多个文件描述符上同时阻塞，直到有一个或者更多的文件描述符处于就绪状态，然后处理已经就绪的文件描述符。

select

可以观察多个流的IO事件，如果所有流都没有IO事件，则将线程进入阻塞状态，如果有一个或多个发生了IO事件，则唤醒线程去处理。但是还是得遍历所有的流，才能找出哪些流需要处理。如果流个数为N，则时间复杂度为O（N）。

1. 每次调用select，都需要把fd_set集合从用户态拷贝到内核态，如果fd_set集合很大时，那这个开销也很大
2. 同时每次调用select都需要在内核遍历传递进来的所有fd_set，如果fd_set集合很大时，那这个开销也很大
3. 为了减少数据拷贝带来的性能损坏，内核对被监控的fd_set集合大小做了限制，并且这个是通过宏控制的，大小不可改变(限制为1024：在内核中写死了)

seclect实现原理

poll

poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。也就是说，poll只解决了上面的问题3，并没有解决问题1，2的性能开销问题。

epoll

相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

在 select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll_ctl()来注册一 个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait() 时便得到通知。

- **水平触发（LT）**：默认工作模式，即当epoll_wait检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用epoll_wait时，会再次通知此事件
- **边缘触发（ET）**： 当epoll_wait检测到某描述符事件就绪并通知应用程序时，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次通知此事件。（直到你做了某些操作导致该描述符变成未就绪状态了，也就是说边缘触发只在状态由未就绪变为就绪时只通知一次）。

epoll底层是什么？epoll在内核级是怎么实现信息回调的？底层怎么实现的？怎么获取epoll里面的信息，那个函数？

区别

epoll底层应该是红黑树

proactor reactor 是什么，区别在哪里

命令

LINUX找特定的文本段

LINUX如何杀掉特定名称的进程

内核态和用户态区别

主线程写一个**buf**，子线程去读，怎么做？读写时候的游标更新可能会出什么问题？怎么解决？

操作系统中的锁

只有互斥锁，如何实现读写锁？

linux地址空间

管道分类

可执行文件的内容分布