# 一、操作系统基础

## Q：操作系统的四个特性

A：**并发**：同一段时间内多个程序执行(注意区别并行和并发，前者是同一时刻的多个事件，后者是同一时间段内的多个事件)；**共享**：系统中的资源可以被内存中多个并发执行的进线程共同使用；**虚拟**：通过时分复用（如分时系统）以及频分复用（如虚拟内存）技术实现把一个物理实体虚拟为多个；**异步**：系统中的进程是以走走停停的方式执行的，且以一种不可预知的速度推进；

## Q：操作系统的主要功能

A： 处理机管理、存储器管理、文件管理、提供用户接口

## Q：动态库、静态库有哪些使用方式？

A： linux中，gcc编译时使用-shared指明编译生成动态库；需要加载其它动态库时，（1）用-L加上动态库（静态库）路径加载动态（静态）链接库。

（2）运行时，通过dlopen（使dll文件可读）、dlsym（获取函数地址）、dlclose等使用动态链接库中的函数。

eg：gcc -o main main.c -L./ -ltest

windows下

1. 使用LoadLibrary（dllName）方式加载动态库
2. 使用#pargma comment(lib, "xxx.lib")加载静态库 编译指令

<http://blog.csdn.net/shado_walker/article/details/53337874>

<http://blog.csdn.net/stpeace/article/details/47030017>

# 二、文件管理

## Q：Linux文件系统

A： a、文件系统从上而下依次分成以下几层：用户层、VFS层、文件系统层、缓存层（页缓存，加速性能）、块设备层、磁盘驱动层、磁盘物理层；

b、Linux中的每个分区包含一个文件系统，各分区通过挂载的方式挂到目录树上的某个位置。常见的分区包括：

/根目录（只能包含目录）；

/swap分区（虚拟内存页面的换出）；

/boot（系统启动文件）；

/etc（存放系统配置信息）；

/bin（存放普通用户可以调用的linux命令）；

/home（各用户工作目录的主目录）；

/usr（用户目录）；

/lib 共享库和内核模块

c、每个分区包括空闲区管理区、i节点区、数据区和目录区。一个文件在磁盘中可能占用多个磁盘块，通过索引块保存数据块的组织信息，Linux采用三级索引的方式，以保证可以记录大文件信息。为了减小磁盘IO的开销，

d．大部分的Linux文件系统（如ext2、ext3）规定，一个文件由目录项、inode和数据块组成：

* 目录项：包括文件名和inode节点号。

Inode：又称**文件索引节点**，包含文件的基础信息以及数据块的指针。inode包含文件的元信息，具体来说有以下内容：文件的字节数。文件拥有者的User ID。文件的Group ID。文件的读、写、执行权限。文件的时间戳。链接数，即有多少文件名指向这个inode。文件数据block的位置。

* 数据块：包含文件的具体内容。

d、在系统内存中，系统维护了一张“系统打开文件表”，整个系统一张，用于保存已打开文件的信息；

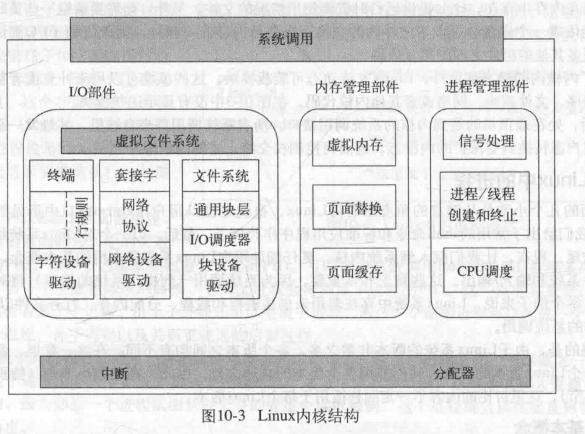
e、每个进程中维护一张“进程打开文件表”，表中记录了进程打开的文件对应在“系统打开文件表”中的索引。

<https://blog.csdn.net/kyang_823/article/details/73302248>

## Q：什么是内核？

内核由一系列程序组成，包括负责响应中断的中断服务程序、负责管理多个进程从而分享处理器时间的调度程序、负责管理地址空间的内存管理程序、网络、进程间通信的系统服务程序等。

内核负责管理系统的硬件设备。



## Q：有哪些类型的文件系统？fat32限制为什么是4GB？

A： NTFS、fat16、fat32、ext2、ext3、ext4、swap

Fat32支持的磁盘大小为2TB，fat16只支持2GB；

Linux中支持的文件系统：

ext2、ext3、swap、NFS、fat；

正常FAT32格式，是以32bit记录文件长度的,所以单一文件最多能记录2的32次方---4G的容量

## Q：常用的磁盘调度算法？

A：**先来先服务、最短寻道时间优先算法、扫描算法（SCAN）电梯调度、循环扫描算法**

**最短寻道时间优先算法：**其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短

**扫描算法（SCAN）电梯调度：**当磁头正在自里向外移动时，扫描算法所选择的下一个访问对象应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直到再无更外的磁道需要访问才将磁臂换向，自外向里移动。类似电梯

**循环扫描算法：**只自里向外移动，当磁头移到最外的被访问磁道时，磁头立即返回到最里的欲访磁道，即将最小磁道号紧接着最大磁道号构成循环，进行扫描。

## Q：linux分区

A： a、磁盘分区分为基本分区和扩充分区，总和为4，用占用hda1-hda4。基本分区可以马上被使用。扩充分区必须分区为逻辑分区后才能被使用。

b、linux采用“载入”的方式，只有一个根目录，整个文件系统包含了一整套的文件和目录。一个分区和一个目录联系在一起。

c、每个系统必须有一个/swap分区，一个/分区。swap分区用于实现虚拟内存时的页面置换，/用于放置各种系统、启动、程序、配置相关的数据。

<https://www.linuxidc.com/Linux/2017-10/147456.htm>

# 三、存储器管理

## Q：操作系统动态内存分配

A：**首次适应(First  Fit)算法**：空闲分区按地址排序，顺序查找第一个满足条件的区块。最简单的，而且通常也是最好和最快的分配策略，首次适应算法会使得内存的低地址部分出现很多小的空闲分区，而每次分配查找时，都要经过这些分区，因此也增加了查找的开销。

**邻近适应(Next  Fit)算法**：又称循环首次适应算法，由首次适应算法演变而成。不同之处是分配内存时从上次查找结束的位置开始继续查找。常常会导致在内存的末尾分配空间，分裂成小碎片。它通常比首次适应算法的结果要差。

**最佳适应(Best  Fit)算法：**空闲分区按容量递增形成分区链，找到第一个能满足要求的空闲分区。性能通常很差。因为每次最佳的分配会留下很小的难以利用的内存块，它会产生最多的外部碎片。

**最坏适应(Worst  Fit)算法**：又称最大适应(Largest Fit)算法，空闲分区以容量递减的次序链接。找到第一个能满足要求的空闲分区，也就是挑选出最大的分区。很快导致没有可用的大的内存块，因此性能也非常差

首次适应算法可能比最佳适应法效果好，而它们两者一定比最大适应法效果好。另外，在算法实现时，分配操作中最佳适应法和最大适应法需要对可用块进行排序或遍历查找，而首次适应法和邻近适应法只需要简单查找；回收操作中，当回收的块与原来的空闲块相邻时，需要将这些块合并。在算法实现时，使用数组或链表进行管理。

## Q：操作系统内存管理（分页式、分段式、段页式、虚拟内存）：

<https://www.cnblogs.com/peterYong/p/6556619.html>

在内存管理中，内部碎片是已经被分配出去的的内存空间大于请求所需的内存空间。

外部碎片是指还没有分配出去，但是由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。

A： 在操作系统中，内存是宝贵紧俏的资源，为了方便用户使用，并提高存储器的利用率，需要对计算机内存进行管理。常见的内存管理技术包括：简单连续分配、页式存储管理、请求分页式存储管理、段式存储管理、段页式存储管理。

**简单连续分配**

分为单一连续分配、固定式分区和动态分区。单一连续分配有内部碎片、无外部碎片；固定式分区有内部碎片、无外部碎片；动态分区无内部碎片，有外部碎片。物理块。

**基本分页式**

a、将程序逻辑地址空间划分为固定大小的页(page)，将物理内存划分为同样大小的页框(page frame)。每一个进程维护一个页表，记录各页在内存上页框的对应关系。平均每个进程会产生0.5页的内部碎片

b、地址结构包含两个部分：页号和页内偏移量。进行地址变换时，存取一个数据或一条指令至少需要访问两次内存，一次获取页表对应的页起始地址，将起始地址和偏移量组合成数据实际地址，第二次根据实际地址读取数据或指令。在具有快表的CPU中，可以存放当前访问的若干项页表项，根据局部性原理，大多数情况下，可以直接从快表中读取到页表的起始地址。快表是一种高速存储缓冲器，存储了页表的一部分。

页表：进程分片为**页**，内存分片为**叶框**，**页表**为页与叶框对应表

**请求分页式**

a、将程序逻辑地址空间划分为固定大小的页(page)，将物理内存划分为同样大小的页框(page frame)。每一个进程维护一个页表，记录各页在内存上页框的对应关系。平均每个进程会产生0.5页的内部碎片。

b、实际运行时，只将部分页面载入内存，发生缺页中断时，再将页面换入内存，同时还可以通过置换功能将暂时不用的页面换出内存。

**分段式**

a、根据进程中数据的组成分段、比如栈、堆、代码段、bss、data等每个分一段，每段分配一段连续内存空间，不同段之间不用保证连续。

1.目的：页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说，分页是出于系统管理的需要而不是用户需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地满足用户的需要。

2.长度：页的大小固定而且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面。段的长度不固定，决定于用户所编写的程序，通常由编译程序在对程序进行编译时，根据信息的性质来划分。

4.碎片：分页有内部碎片无外部碎片，分段有外部碎片无内部碎片

5.绝对地址：处理器使用页号和偏移量计算绝对地址，处理器使用段号和偏移量计算绝对地址

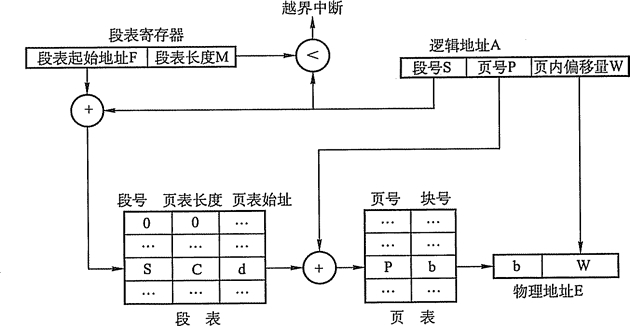
**段页式**

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。如果将这两种存储管理方法结合起来，就形成了段页式存储管理方式。

在段页式系统中，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，然后再将每一段分成若干个大小固定的页。对内存空间的管理仍然和分页存储管理一样，将其分成若干个和页面大小相同的存储块，对内存的分配以存储块为单位

在一个进程中，段表只有一个，而页表可以有多个。页表实质上是程序中虚拟页面到内存叶框号的映射表。

https://blog.csdn.net/zhongyangtony/article/details/80879425



## Q：操作系统页面置换算法

A：**最佳置换算法(OPT)**

最佳(Optimal, OPT)置换算法所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面。OPT是最理想的算法，但是无法实现，可以用来评价其他算法。

**先进先出(FIFO)页面置换算法**

实现简单，但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。

**最近最久未使用(LRU)置换算法**

该算法为每个页面设置一个访问字段，来记录页面自上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的予以淘汰。LRU**性能较好，但需要寄存器和栈的硬件支持，开销较大**，属于堆栈类的算法。

**时钟(CLOCK)置换算法（又称最久未使用算法）**

给每个帧关联一个使用位和修改位，CLOCK算法循环地检查各页面两个位的情况，选择页面进行替换。每个帧处于以下四种情况之一：

1、最近未被访问，也未被修改(u=0, m=0)。

2、最近被访问，但未被修改(u=1, m=0)。

3、最近未被访问，但被修改(u=0, m=1)。

4、最近被访问，被修改(u=1, m=1)。

算法执行如下操作步骤：

1、从指针的当前位置开始，扫描帧缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个帧(u=0, m=0)用于替换。

2、如果第1)步失败，则重新扫描，查找(u=0, m=1)的帧。选择遇到的第一个这样的帧用于替换。在这个扫描过程中，对每个跳过的帧，把它的使用位设置成0。

3、如果第2)步失败，指针将回到它的最初位置，并且集合中所有帧的使用位均为0。重复第1步，并且如果有必要，重复第2步。这样将可以找到供替换的帧。

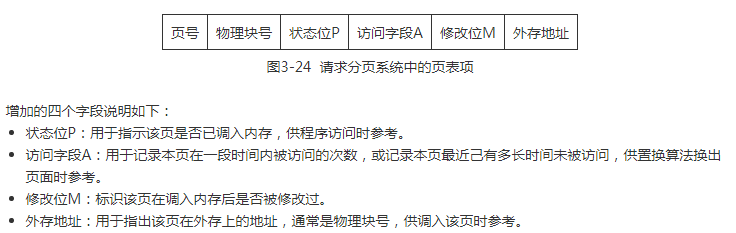
**对比：**

LRU算法的性能接近于OPT,但是实现起来比较困难，且开销大；FIFO算法实现简单，但性能差。CLOCK是对两者的综合，以较小的开销接近LRU性能。

## Q：虚拟存储

A： 操作系统中的物理内存是紧俏资源。如果将进程运行的所有资源全部加载到内存中，则大大限制了系统的并发量。考虑到程序运行的时间局部性和空间局部性特征，程序运行时，短时间内，只有部分内容会被访问到。因此，程序启动时，只需要加载部分块到内存中，运行过程中，当所访问的信息不在内存中时，再将需要的部分调入内存。并且，系统可以将暂时不用的内容换出到外存上，从而腾出内存空间。这样，操作系统就像给程序分配了足够大存储器，将程序需要用到内容全部都加载到内存，这就是虚拟存储器。

在虚拟页式存储方式中，以32位linux系统为例，系统给每个进程分配了4GB的虚拟存储空间，其中高地址1GB分配给内核地址空间，低地址3GB分配给用户地址空间。这4GB地址空间被划分为大小相同的块，即页，每个页一般4KB。同时将内存空间划分为同样大小的块，称为页框。每个进程还维护有一个页表，页表中记录每个页的分配情况，页表中的页分为三种：未分配页（不对应内存和磁盘空间）、未缓冲页（对应有磁盘空间，但未分配内存空间）、缓存页（已经载入内存的页面）。当发生缺页异常时，根据页表项中记录的磁盘地址，去磁盘中加载该页面到内存。当已缓冲页数量达到驻留集大小时，根据页面置换算法，将页面换出内存，换出时，检查修改状态位，如果该页面被修改，还需要写回到磁盘。

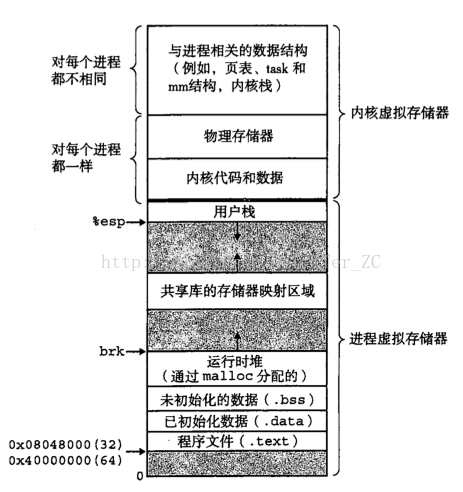


## Q：进程的内存分布

A： a、在32位操作系统中，每个进程都有4GB的虚拟地址空间。其中，Linux操作系统默认高位的1GB空间为内核空间，windows默认高位的2GB空间为内核空间。

b、内核空间分为两个区域，进程共享区和进程独有区。内核中，有很多代码和数据是所有进程公用的，该区域内的内容相同，且该虚拟内存映射到同一内存区域；另外，对于不同进程，其页表、内核栈等不同，这部分内核空间对每个进程都不相同。

c、从高地址往低地址，用户地址空间分为栈、共享库映射区、堆、未初始化数据区、已初始化数据区、程序代码区。其中，用户栈从高地址空间往低地址空间扩展；堆地址空间从低地址往高地址扩展。（bss和data段即静态存储区，全局变量和静态变量都放在该区域，如果未初始化，放在bss段，已初始化，放在data段）。



# 四、CPU管理（进程、线程）

## Q：操作系统进程调度

A：调度性能的衡量指标：

等待时间：每个进程在就绪队列中等待的时间；

周转时间：每个进程从提出请求到运行完成时间；

响应时间：从提出请求到第一次回应的时间；

CPU利用率：CPU做有效工作的时间比例；

优先权：批处理系统、分时系统、实时系统都可以优先执行优先级别更高的作业。

调度算法：

**先来先服务调度算法（FCFS，first come first served）：**非抢占式

算法优点：易于理解且实现简单，只需要一个队列(FIFO)，且相当公平；

算法缺点：有利于长进程，不利于短进程，有利于CPU 繁忙的进程，不利于I/O 繁忙的进程；

**短作业优先算法（SJF, Shortest Job First）**：非抢占式

算法原理：对预计执行时间短的进程优先分派处理机。通常后来的短进程不抢先正在执行的进程。

算法优点：相比FCFS 算法，该算法可改善平均周转时间和平均带权周转时间，缩短进程的等待时间，提高系统的吞吐量。

算法缺点：对长进程非常不利，可能长时间得不到执行，产生饥饿问题，且未能依据进程的紧迫程度来划分执行的优先级，以及难以准确估计进程的执行时间，从而影响调度性能。

**最高响应比优先法(HRRN，Highest Response Ratio Next)**

算法原理：响应比R定义如下： R =(W+T)/T = 1+W/T

其中T为该作业估计需要的执行时间，W为作业在后备状态队列中的等待时间。每当要进行作业调度时，系统计算每个作业的响应比，选择其中R最大者投入执行。

算法优点：由于长作业也有机会投入运行，在同一时间内处理的作业数显然要少于SJF法，从而采用HRRN方式时其吞吐量将小于采用SJF 法时的吞吐量。

算法缺点：由于每次调度前要计算响应比，系统开销也要相应增加。

**时间片轮转算法（RR，Round-Robin）**

算法原理：让就绪进程以FCFS 的方式按时间片轮流使用CPU 的调度方式，即将系统中所有的就绪进程按照FCFS 原则，排成一个队列，每次调度时将CPU 分派给队首进程，让其执行一个时间片，时间片的长度从几个ms 到几百ms。在一个时间片结束时，发生时钟中断，调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程，进程可以未使用完一个时间片，就出让CPU（如阻塞）。

算法优点：时间片轮转调度算法的特点是简单易行、平均响应时间短。

算法缺点：不利于处理紧急作业。在时间片轮转算法中，时间片的大小对系统性能的影响很大，时间片太大，降级为先来先服务算法，时间片太小会导致吞吐量很低，进程切换开销大，因此时间片的大小应选择恰当。

**多级反馈队列(Multilevel Feedback Queue)**

多级反馈队列调度算法是一种CPU处理机调度算法，UNIX操作系统采取的便是这种调度算法。

多级反馈队列调度算法描述：

1、进程在进入待调度的队列等待时，首先进入优先级最高的Q1等待。

2、首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3三个队列，只有在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。

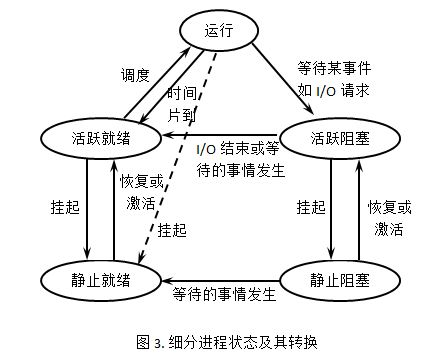
3、对于同一个队列中的各个进程，按照时间片轮转法调度。比如Q1队列的时间片为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。

4、在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配给新到达的作业（抢占式）。

在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能够较好的满足各种类型用户的需要。

## Q：进程的状态与转换

A：



静止就绪和活动就绪：

活动就绪是指就绪进程当前仍放在主存，并且随时可以被调度；

静止就绪是指当系统内存不够用时，将进程换到辅存时的就绪状态。当系统内存有较多空闲，或者静止就绪进程有较高优先级时，系统会将其激活，调回内存，进入活动就绪；

静止阻塞和活动阻塞同理。

## Q：用户态切换到内核态？

A：用户态和内核态对应不同的特权级别，Inter的X86中CPU提供了0-3四个特权级别。Linux采用了0和3两个级别，分别对应内核态和用户态。

**导致切换的原因**：（read、write、open等库函数，进一步进行系统调用）

a、系统调用（软中断）；例如：打开文件，open，write等，通过库函数，调用软中断，查找中断向量表，进行系统调用。

b、产生异常（软中断）；例如缺页异常

c、外设中断（硬中断）；例如：IO操作等

**切换过程：**

[1] 从当前进程的描述符中提取其内核栈的ss0及esp0信息。

[2] 使用ss0和esp0指向的内核栈将当前进程的eip,ss,esp信息保存起来，这个过程也完成了由用户栈到内核栈的切换过程，同时保存了被暂停执行的程序的下一条指令。

[3] 将先前由中断向量检索得到的中断处理程序的cs,eip信息装入相应的寄存器，开始执行中断处理程序，这时就转到了内核态的程序执行了。

SS——堆栈段寄存器(Stack Segment Register)，其值为堆栈段的段值；

1.EIP寄存器里存储的是CPU下次要执行的指令的地址。

 也就是调用完fun函数后，让CPU知道应该执行main函数中的printf（"函数调用结束"）语句了。

2.EBP寄存器里存储的是是栈的栈底指针，通常叫栈基址，这个是一开始进行fun()函数调用之前，由ESP传递给EBP的。（在函数调用前你可以这么理解：ESP存储的是栈顶地址，也是栈底地址。）

3.ESP寄存器里存储的是在调用函数fun()之后，栈的栈顶。并且始终指向栈顶。

## Q：进程、线程、协程的区别，各自的适用场合

A：

**进程：**资源分配的基本单位；引入进程是为了使多个程序间可以并发的执行，以提高系统的资源利用率和吞吐量。上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

**线程：**线程是CPU调度的基本单位。引入目的是为了减少程序在并发执行过程中的开销，使系统在进程内也可以并发。只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈)

进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。进程所维护的是程序所包含的资源（静态资源）， 如：地址空间，打开的文件句柄集，文件系统状态，信号处理handler等；线程所维护的运行相关的资源（动态资源），如：运行栈，调度相关的控制信息，待处理的信号集等；

系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。进程切换时，耗费资源较大，效率要差一些。

**协程：**用户态的轻量级线程，协程拥有自己的寄存器上下文和栈，允许协程在特定的地方挂起恢复。

协程，是一种比线程更加轻量级的存在，协程不是被操作系统内核所管理，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。

   协程在**子程序(函数)**内部是可中断的，然后转而执行别的子程序，在适当的时候再返回来接着执行。

**极高的执行效率：**因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显；

**不需要多线程的锁机制：**因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

<https://blog.csdn.net/daaikuaichuan/article/details/82951084>

## Q: 多线程和多进程对比：

1、调度方面：在引入线程的OS中，线程是独立的调度和分派单位，而进程作为资源的拥有单位。

2、安全性方面，多进程比多线程更安全。进程中可以创建多个线程，进程结束，所有线程都会被销毁，任意一个线程崩溃也会导致进程崩溃。

3、进程拥有PCB（进程控制块，记录进程状态）、页表、内核堆栈等资源，进程的创建、撤销和切换需要保存PCB和内核堆栈等信息，开销比线程大。

4、进程间互相独立，所以协作时主要考虑通信问题；同一进程的线程共享进程空间，通信容易，主要考虑同步问题

<http://blog.csdn.net/zhou753099943/article/details/51771220>

## Q: 多线程（包含多个线程的进程）和单线程（包含一个线程的多个进程）适用场合：

首先，所有使用多线程能解决问题的场合，都可以采用单线程替代。对于纯粹的IO密集型（一个线程阻塞了，IO也就被占用了，就算切换到其它线程，也没有设备可用）和CPU密集型程序，采用多线程没有意义。只有在IO和“计算”相互重叠的情况下，采用多线程可以降低等待时间，提高效率。

单线程：

a、**对程序容错性要求很高时**；

b、**程序可能会fork()**如集群中在计算节点上负责启动任务的守护进程（即“看门狗”程序）。

c、**限制程序CPU占用率**；例如一个8核的服务器上，一个单线程的程序即便发生busy-wait，占满一个核心，其CPU使用率也只有12.5%。最坏情况下仍有87.5%的计算资源可用。

多线程：

a、有界面交互的应用，输入的时候界面不会卡住；

b、不同任务间需要大量数据共享或频繁通信时；

c、多线程可以保证服务的优先级（使用单线程，所有任务都只能串行运行，不存在优先级，短任务可能要等待很久才能运行）；

d、对于需要频繁切换，或者频繁创建销毁时（比如web服务），可以采用线程池；

d、在多核CPU上运行计算密集型软件，采用多线程并行方式可以大大提高速度。

## Q: 线程共享进程的哪些内容？

a、线程共享进程的堆区，进程代码段、进程打开文件描述符、全局变量、进程目录、进程用户ID和进程组ID；

b、线程有各自独立的栈，包括线程ID、线程寄存器、线程栈；

**协程与线程对比：**

a、一个线程可以多个协程，一个进程也可以单独拥有多个协程，一个CPU同时只能有一个协程运行。

b、线程进程都是同步机制，而协程则是异步

c、协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态

## Q：每个进程都至少有一个线程吗？没有线程的进程叫什么？

A： 是的；每个进程至少有一个main函数入口，即主线程。

## Q：线程池对线程的管理方式，包括初始化线程的方法、线程创建后的管理、指派任务的方式

A：**线程池的组成**

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

2、工作线程（PoolWorker）：线程池中线程

3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

**线程池的作用**

a、重用线程池中的线程，减少进程的创建、销毁过程；

b、能有效的控制线程池的最大并发数，避免以为其共同抢占资源而造成的阻塞；

c、能够对线程进行简单的管理，提供制定间隔和定时执行。

**线程池的适用场景**

a、单位时间内处理任务频繁，且任务处理时间短；

b、对实时性要求高；

c、需要处理突然的高并发事件。

<https://www.cnblogs.com/cpper-kaixuan/p/3640485.html>

## Q：线程是不是越多越好，一般设置几个？

A： 不是。

a、使用多线程是为了进程内并发，这种并发只有对IO密集型程序才是有意义的，因为在IO阻塞阶段CPU可以切换其它进程处理其它事情。对于CPU密集型程序，进程运行期间不会阻塞，此时使用多线程还会增加线程切换开销，这样反而会使性能降低。

b、线程是操作系统的资源，操作系统对线程数有限制。一个进程中创建了太多无意义的进程，可能会使其它程序无法创建线程。

**数量：**

N核服务器，通过执行业务的单线程分析出本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N\*(x+y)/x，能让CPU的利用率最大化。所以，对于CPU密集型进程来说，设置线程数和CPU数量一致就行。

**单核CPU设置多线程有意义吗**

a、是用多线程可以让代码逻辑结构更清晰。

b、对于CPU密集型程序来说，单核CPU上使用多线程并无意义，还会增加CPU切换开销。

c、对于IO密集型程序来说，在进程调用IO阻塞时，使用多线程的话，CPU可以切换其它线程处理其它任务，可以很好的提高性能。

## Q：进程间通信方式

A： 进程间通信（IPC，InterProcess Communication）的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket等。其中 Socket支持不同主机上的两个进程IPC。

**进程间通信方式：**管道（FIFO）/共享内存/消息队列/信号/socket

**管道**：分为无名管道和有名管道（FIFO），对于无名管道，只能用于具有亲缘关系的进程间通信；有名管道可以用于任意两个进程间通信；无名管道必须依附于进程而临时存在，有名管道可以独立于进程存在。

管道的实现：通过在两个进程的已打开文件表中增加一个记录，指向一个inode（索引节点），inode指向一个内存中的物理页面。

eg：父进程调用pipe()函数，然后fock一个子进程，父进程关闭读端，子进程关闭写端；父进程write、子进程read。

**消息队列**：消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。消息具有特定的格式和优先级，其独立于发送和接收进程，由内核维护，进程终止时，消息队列及其内容不会被删除。

eg：用函数msget创建消息队列，调用msgsnd函数，把输入的字符串添加到消息队列中，然后调用msgrcv函数，读取消息队列中的消息并打印输出，最后再调用msgctl函数，删除系统内核中的消息队列。

**信号**：在计算机中，信号就是一个内核对象或者是一个内核数据结构。发送方将该数据结构的内容填好，并指明该信号的目标进程后，发出特定的软件中断（这就是一个发电报的操作）。OS接收到特定的中断请求后，知道是有进程要发送信号，于是到特定的内核数据结构里查找信号接收方，并进行通知。接到通知的进程则对信号进行相应处理。

eg：父进程调用kill函数，给子进程发送SIGKILL信号，终止子进程。

**共享内存**：指两个或多个进程共享一个给定的存储区，因为有多个进程同时进行操作，所以需要进行同步，通常共享内存和信号量结合起来使用。因为消息本身就存在各自共享的内存中，不需要拷贝数据，所以共享内存是最快的一种进程通信方式，可以共享大量数据。

eg：父进程调用mmap映射内存，然后fork一个子进程。子进程继承父进程匿名映射后的地址空间，也继承了mmap函数的返回地址，这样父子进程就可以通过映射区进行通信。

**信号量：**计数器对象，通过值大于小于零，来实现进程间的互斥与同步。

**socket**：套接字（Socket）的功能非常强大，可以支持不同层面、不同应用、跨网络的通信。

<https://www.cnblogs.com/LUO77/p/5816326.html>

## Q: 进程间同步方式

临界区、互斥锁、信号量、事件四种方式

互斥量、条件变量、信号量、读写锁、文件锁、屏障  
临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）  
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。  
2、互斥量：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访问。互斥量比临界区复杂。因为使用互斥不仅仅能够在同一应用程序不同线程中实现资源的安全共享，而且可以在不同应用程序的线程之间实现对资源的安全共享。  
3、信号量：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 .信号量对象对线程的同步方式与前面几种方法不同，信号允许多个线程同时使用共享资源，这与操作系统中的PV操作相同。它指出了同时访问共享资源的线程最大数目。

4、事 件：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。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 .

**进程/线程同步机制与进程间通信机制比较**

同步主要是临界区、互斥锁、信号量、事件

进程间通信是管道、FIFO（命名管道）、内存共享、消息队列、信号量、socket

<https://www.cnblogs.com/virusolf/p/5331946.html>

## Q：共享内存方式的缺点

A： a、使用共享内存机制通信的两个进程必须在同一台物理机上；

b、安全性较弱，假如一个进程有病毒，会很容易传给另一个进程；

c、共用一片内存，必须实现同步，同步消耗资源。

## Q：线程间通信、同步方式？

A：**线程间通信方式：**

同一进程的线程可以通过共享内存和全局变量通信。另外，由于同一进程的线程共享进程内存，所以线程同步是很重要的问题。

**线程间同步的方式：**

（主要四种）Windows：临界区、互斥锁、信号量、事件

Linux：通过互斥量（Mutex）、信号量（Sem）、条件变量（Cond）、屏障、读写锁、自旋锁（pthread\_join()就是一种屏障）等实现同步。

<https://www.cnblogs.com/vincently/p/4749359.html>

<https://www.cnblogs.com/raichen/p/5768752.html>（写得好）

## Q：自旋锁及其优缺点

自旋锁（spinlock）：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

获取锁的线程一直处于活跃状态，但是并没有执行任何有效的任务，使用这种锁会造成[busy-waiting](https://en.wikipedia.org/wiki/Busy_waiting" \t "_blank)。它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，”自旋”一词就是因此而得名。

**优点：**

自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快

非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。 （线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能）

实现原理：

linux内核提供了位和整形变量的原子操作函数，这些函数的实现也是以来cpu的原子操作，与cpu底层架构有关。原子操作可以保证对位和整形数据的操作是排他性的，所以为临界区代码的互斥访问奠定了基础。自旋锁就是通过执行一个原子操作，测试并设置某个内存变量，由于它是原子操作，所以对该变量的访问是排他性的，如何测试到锁为空闲，则获取该自旋锁，否则重复执行该原子操作，直至该锁可用为止。

为什么自旋锁不能睡眠：

注意到“preempt\_disable()”，这个调用的功能是“关抢占”（在spin\_unlock中会重新开启抢占功能）。从中可以看出，使用自旋锁保护的区域是工作在非抢占的状态；即使获取不到锁，在“自旋”状态也是禁止抢占的。了解到这，我想咱们应该能够理解为何自旋锁保护的代码不能睡眠了。试想一下，如果在自旋锁保护的代码中间睡眠，此时发生进程调度，则可能另外一个进程会再次调用spinlock保护的这段代码。而我们现在知道了即使在获取不到锁的“自旋”状态，也是禁止抢占的，而“自旋”又是动态的，不会再睡眠了，也就是说在这个处理器上不会再有进程调度发生了，那么死锁自然就发生了。

咱们可以总结下自旋锁的特点：

● 单处理器非抢占内核下：自旋锁会在编译时被忽略；

● 单处理器抢占内核下：自旋锁仅仅当作一个设置内核抢占的开关；

● 多处理器下：此时才能完全发挥出自旋锁的作用，自旋锁在内核中主要用来防止多处理器中并发访问临界区，防止内核抢占造成的竞争。

A如果那到锁以后，内核的抢占暂时被禁止。然后它休眠了，切换到另一个进程B（注意，这不是抢占，是进程自己放弃CPU）。等到进程B想要获得这个锁时发生了死锁，尽管B的时间片会被用完，但由于内核抢占被禁止了，所以B不会被调度出去。  
更糟的情况是，如果A用irq save方式来得到这个spin lock，那中断是被禁止的，时钟中断不会被响应，B的时间片根本不会被跟新。

## Q：屏障

barrier(屏障)与互斥量，读写锁，自旋锁不同，它不是用来保护临界区的。相反，它跟条件变量一样，是用来协同多线程一起工作！！！

条件变量是多线程间传递状态的改变来达到协同工作的效果。屏障是多线程各自做自己的工作，如果某一线程完成了工作，就等待在屏障那里，直到其他线程的工作都完成了，再一起做别的事。举个通俗的例子：

1.对于条件变量。在接力赛跑里，1号队员开始跑的时候，2，3，4号队员都站着不动，直到1号队员跑完一圈，把接力棒给2号队员，2号队员收到接力棒后就可以跑了，跑完再给3号队员。这里这个接力棒就相当于条件变量，条件满足后就可以由下一个队员(线程)跑。

2.对于屏障。在百米赛跑里，比赛没开始之前，每个运动员都在赛场上自由活动，有的热身，有的喝水，有的跟教练谈论。比赛快开始时，准备完毕的运动员就预备在起跑线上，如果有个运动员还没准备完(除去特殊情况)，他们就一直等，直到运动员都在起跑线上，裁判喊口号后再开始跑。这里的起跑线就是屏障，做完准备工作的运动员都等在起跑线，直到其他运动员也把准备工作做完！

## Q：互斥量、条件变量

A： 互斥锁提供互斥机制、条件变量提供信号机制。互斥锁和条件变量结合可以实现让主线程暂时休眠，直到某线程通知它才醒过来。

条件变量的作用是在等待某个条件达成时自身要进行睡眠或阻塞，避免忙等待带来的不必要消耗，所以条件变量的作用在于同步。条件变量这个变量其实本身不包含条件信息，条件的判断不在pthread\_cond\_wait函数功能中，而需要外面进行条件判断。这个条件通常是多个线程或进程的共享变量，这样就很清楚了，对于共享变量很可能产生竞争条件尤其还对共享变量加了条件限制，所以从这个角度看，必须对共享变量加上互斥锁。

<https://blog.csdn.net/run32875094/article/details/78272240>

**互斥锁：**

int pthread\_mutex\_lock(pthread\_mutex\_t \*[mutex](http://en.wikipedia.org/wiki/Mutual_exclusion));

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

**条件变量：**

int pthread\_cond\_wait(pthread\_cond\_t \*cond,   pthread\_mutex\_t \*mutex);

int pthread\_cond\_signal(pthread\_cond\_t \*cptr);

## Q：互斥锁和读写锁的比较；开销、实现

读写锁特点：

1）多个读者可以同时进行读

2）写者必须互斥（只允许一个写者写，也不能读者写者同时进行）

3）写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

互斥锁特点： 一次只能一个线程拥有互斥锁，其他线程只有等待

读写锁比互斥锁提供了跟好的并行性。读写锁可以用信号量和条件变量替代。

## Q：同步与互斥的区别

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

## Q：如何保证某些线程是单例执行的？

A： 使用文件锁。创建一个和线程绑定的文件，在执行线程前必须先对该文件加文件锁，执行完成后释放。

## Q：有哪些线程模型？

A： 从线程和内核的关系分：

a、用户级线程：内核管理的还是进程，不知道线程的存在；线程切换不需要内核态特权；同一进程内的线程无法并发执行；（例如UNIX）

b、核心级线程：内核管理所有线程，维护进程和线程的上下文，线程的切换需要内核支持。

c、混合两级模型

从抽象模型上看：

a、生产者消费者模型；

b、线程池模型；

## Q：什么是死锁，如何预防死锁？

A： 死锁是指由于两个或者多个进程互相持有对方所需要的资源，导致这些进程全部处于阻塞状态。死锁的发生必须同时具备以下条件：

资源互斥且不可抢占，进程占有资源且还能申请，循环等待

a、互斥条件（某资源一段时间只能由一个进程占有）；

b、不可抢占条件（进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放）；

c、占有且申请条件（进程至少已经占有一个资源，但又申请新的资源）；

d、循环等待条件（存在一个循环等待环）。

**预防死锁：**

a、对于临界资源只能互斥使用，破坏互斥条件不可行；

b、破坏不可抢占条件，对进程增加一个优先级，或者按时间分片；

c、破坏占有且申请条件，让进程在运行前一次性申请完所有需要的资源。银行家算法

d、破坏循环等待条件，通过信号量限制并发数量。

## Q：生产者/消费者模型

A： 当仓库处于满状态时，生产者必须等待消费者取出 1 个或多个产品后才能继续生产；同理，当仓库处于空状态时，消费者必须等待生产者将产品放入仓库后才能消费（取出）产品。

**信号量配合互斥锁：**

采用两个信号量记录空槽和填充槽的数量，满足条件时竞争互斥锁，使用完更新空槽和填充槽。需要轮询去判断条件是否满足。（必须先调用wait判断信号量是否为0，大于0之后再竞争互斥锁）

**互斥锁配合条件变量：**

需要使用生产消费互斥锁、仓库非满条件变量、仓库非空条件变量。先竞争互斥锁，如果条件变量不满足释放互斥锁。

<http://blog.csdn.net/cgxrit/article/details/38331147>

## Q：解释哲学家进餐问题，实现一种算法解决哲学家进餐问题。

A： 解决办法：

a、破坏占有且申请条件，一次获得个筷子：通过采用信号量，保证任意时刻最多有n - 1个进程执行：

Semaphore chopstick[5]={1,1,1,1,1};//分别表示5支筷子

Semaphore footman=4;//初始值为4最多允许4个哲学家进程同时进行

Philosopher(inti) {

while(true)

{

wait(footman);

Think();

Wait(chopstick[i]);//申请左筷子

Wait(chopstick[(i+1)%5]);//申请右筷子

Eat();

Signal(chopstick[i]);//释放左筷子

Signal(chopstick[(i+1)%5]);//释放右筷子

Signal(footman);

}

}

b、破坏不可抢占条件：让每名哲学家先去申请他左手边的筷子，然后再申请他右手边的筷子，如果右手边的筷子不空闲，则比较当前哲学家 i 和他右手边的哲学家( i +1)%5，看谁取得各自左手边筷子的时间更晚，令其回退( 即释放左手筷子, 再重新申请左手筷子)，直到此哲学家右手边的筷子空闲为止。

c、破坏环路等待条件：添加附加法则，所有哲学家先竞争奇数号筷子，获得后才能去竞争偶数号筷子（由于5号哲学家左右都是奇数号筷子，在本文中规定他先竞争5号筷子）。这样的话就总会有一名哲学家可以顺利获得两支筷子开始进餐。

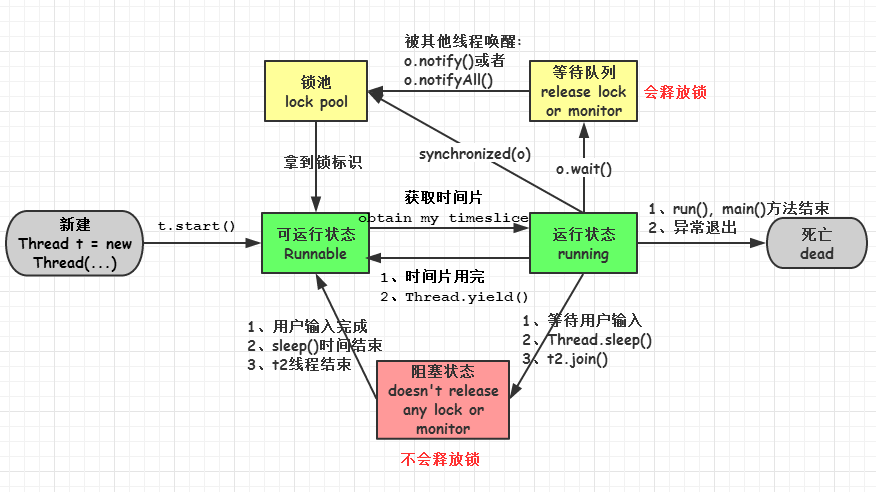
## Q：银行家算法（避免死锁）

A： a、当进程首次申请资源时，测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。

b、当进程在执行中继续申请资源时，先测试该进程已占用的资源数与本次申请的资源数之和是否超过了该进程对资源的最大需求量。若超过则拒绝分配资源，若没有超过则再测试系统现存的资源能否满足该进程尚需的最大资源量，若能满足则按当前的申请量分配资源，否则也要推迟分配。

## Q：线程有几种状态，之间如何转换？

A： 线程分为五种状态：新建状态、就绪状态、运行状态、阻塞状态及死亡状态。



a、新建状态

当用new操作符创建一个线程时。此时程序还没有开始运行线程中的代码。线程进入新建状态。

b、就绪状态

当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入就绪状态。在线程运行之后或者从阻塞回来后，也返回到就绪状态。

c、运行状态

线程运行start后运行run方法，与其它线程竞争CPU时间，当线程获得CPU时间后，它进入运行状态，真正开始执行run()方法。

d、阻塞状态

正在运行的线程暂时让出CPU。原因见“Q：线程从运行态进入阻塞态的原因”；

e、死亡状态

当线程的run()方法完成时就认为它死去。

## Q：什么是线程安全

A： 一个类或者程序所提供的接口对于线程来说是原子操作的，多个线程的切换不会导致该接口执行结果的二义性。

## Q：fork调用，创建子进程之后父子进程的pcb是否一样，不一样内存如何拷贝

A： 不一样；采用写时复制。

**/////////////////////////////////////////////////////////待整理////////////////////////////////////////////////////////**

## Q：可执行文件中包含哪些参数？

A：

## Q：SystemV进程

A：

## Q：用户态切换到内核态的过程

A：

//以上为整理过的面经