1. **什么是进程？什么是线程？**

（1）线程是CPU独立运行和独立调度的基本单位；  
 （2）进程是资源分配的基本单位；  
 两者的联系：进程和线程都是操作系统所运行的程序运行的基本单元。  
 区别：  
 （1）进程具有独立的空间地址，一个进程崩溃后，在保护模式下不会对其它进程 产生影响。  
 （2）线程只是一个进程的不同执行路径，线程有自己的堆栈和局部变量，但线程 之间没有单独的地址空间，一个线程死掉就等于整个进程死掉。

1. **解释一下，什么是IO密集型程序，什么是CPU密集型程序**

CPU密集型（CPU-bound）

CPU密集型也叫计算密集型，指的是系统的硬盘、内存性能相对CPU要好很多，此时，系统运作大部分的状况是CPU Loading 100%，CPU要读/写I/O(硬盘/内存)，I/O在很短的时间就可以完成，而CPU还有许多运算要处理，CPU Loading很高。

在多重程序系统中，大部份时间用来做计算、逻辑判断等CPU动作的程序称之CPU bound。例如一个计算圆周率至小数点一千位以下的程序，在执行的过程当中绝大部份时间用在三角函数和开根号的计算，便是属于CPU bound的程序。

CPU bound的程序一般而言CPU占用率相当高。这可能是因为任务本身不太需要访问I/O设备，也可能是因为程序是多线程实现因此屏蔽掉了等待I/O的时间。

IO密集型（I/O bound）

IO密集型指的是系统的CPU性能相对硬盘、内存要好很多，此时，系统运作，大部分的状况是CPU在等I/O (硬盘/内存) 的读/写操作，此时CPU Loading并不高。

I/O bound的程序一般在达到性能极限时，CPU占用率仍然较低。这可能是因为任务本身需要大量I/O操作，而pipeline做得不是很好，没有充分利用处理器能力。

CPU密集型 vs IO密集型

我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

总之，计算密集型程序适合C语言多线程，I/O密集型适合脚本语言开发的多线程。

1. **进程三态分为哪三态？详细解释**

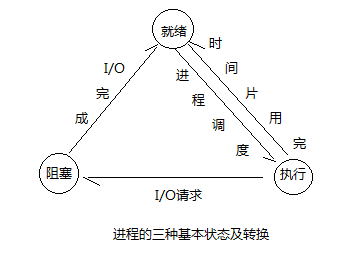
(1)就绪状态：进程已获得除CPU外的所有必要资源，只等待CPU时的状态。一个系统会将多个处于就绪状态的进程排成一个就绪队列。

(2)执行状态：进程已获CPU，正在执行。单处理机系统中，处于执行状态的进程只一个；多处理机系统中，有多个处于执行状态的进程。

(3)阻塞状态：正在执行的进程由于某种原因而暂时无法继续执行，便放弃处理机而处于暂停状态，即进程执行受阻。（这种状态又称等待状态或封锁状态）

通常导致进程阻塞的典型事件有：请求I/O，申请缓冲空间等。

一般，将处于阻塞状态的进程排成一个队列，有的系统还根据阻塞原因不同把这些阻塞集成排成多个队列。



在一些系统中，进程还有一种很重要的状态是：挂起状态（是该进程暂时不接受调度）。

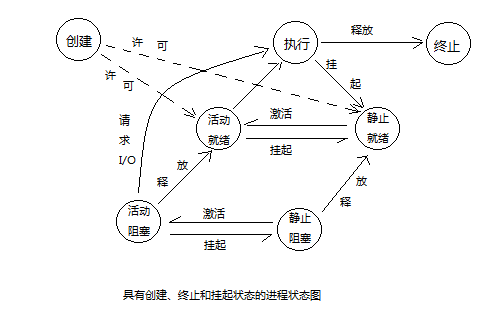
另外，在实际系统中，为管理需要，还存在着两种比较常见的状态：创建状态和终止状态。

创建状态：此时，进程已经拥有了字节的PCB，但该进程所必需的资源或其它信息（如主存资源）尚未分配，进程自身还未进入主存，即创建工作尚未完成，进程还不能够被调度运行。

（创建进程的两个步骤： 为一个新进程创建PCB，并填写必要管理信息；把该进程转入就绪状态并插入就绪队列。）

终止状态：进程的终止首先要等待操作系统进行善后处理，然后将其PCB清零，并将PCB空间返还系统。

（当一个进程到达自然结束点或出现了无法克服的错误，或是被操作系统或其它有终止权的进程所终结，它将进入终止状态。进入终止状态的进程不能再执行，但在操作系统中依然保留一个记录，其中保存状态码和一些计时统计数据，供其它进程收集。一旦其它进程完成了对终止状态进程的信息提取之后，操作系统将删除该进程。



1. **进程三态优化之后又新增了哪两个状态？**

答案见3题.

1. **解释一下：PCB,pid**

PCB(process control block)，进程控制块，是我们学习操作系统后遇到的第一个数据结构描述，它是对系统的进程进行管理的重要依据，和进程管理相关的操作无一不用到PCB中的内容。一般情况下，PCB中包含以下内容：

（1）进程标识符（内部，外部）  
 （2）处理机的信息（通用寄存器，指令计数器，PSW，用户的栈指针）。  
 （3）进程调度信息（进程状态，进程的优先级，进程调度所需的其它信息，事件）  
 （4）进程控制信息（程序的数据的地址，资源清单，[进程同步](https://www.baidu.com/s?wd=%E8%BF%9B%E7%A8%8B%E5%90%8C%E6%AD%A5&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "/home/anwc/文档\\x/_blank)和通信机制，链接指针）

PID是程序被操作系统加载到内存成为进程后动态分配的资源。

每次程序执行的时候，操作系统都会重新加载，PID在每次加载的时候都是不同的。

1. **列举常见的进程状态码并说明其含义**

D            不可中断睡眠（通常进程在进行I/O）

R            运行中或者可运行状态（在运行队列中）

S            可中断睡眠（等待event，进程idle中）

T            进程停止（进程被trace或者进程被信号停止）

X            进程死亡（应该永不可见）

Z            僵尸进程（结束，但是父进程没有wait该子进程）

对于BSD格式，状态码可以附加额外的字符（如S+）：

<            高优先级进程

N            低优先级进程

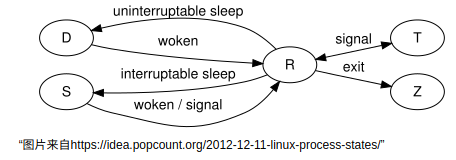
L             进程进行了内存锁定 (避免swap)

s             进程是session leader

l              多线程（using CLONE\_THREAD, like NPTL pthreads do）

+            属于前台进程组

简单的进程状态切换图：



1. **创建进程都有哪些方式？**

在Python中创建进程有两种方式，第一种是：

from multiprocessing import Process  
import time  
  
  
def test():  
 while True:  
 print(**'---test---'**)  
 time.sleep(1)  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 p = Process(target=test)  
 p.start()  
 while True:  
 print(**'---main---'**)  
 time.sleep(1)

上面这段代码是在windows下跑的，通过Process类可以创建一个进程对象，然后p.start()即可开启进程，test函数是你想进程实现的功能。

第二种方式是：

from multiprocessing import Process  
import time  
  
  
class MyNewProcess(Process):  
 def run(self):  
 while True:  
 print(**'---1---'**)  
 time.sleep(1)  
  
  
if \_\_name\_\_ == **'\_\_mian\_\_'**:  
 p = MyNewProcess()  
 # 调用p.start()方法，p会先去父类中寻找start()，然后在Process的start方法中调用run方法  
 p.start()  
  
 while True:  
 print(**'---Main---'**)  
 time.sleep(1)

这里是第二种创建进程的方式，通过子类继承Process类，子类中必须有run方法，里面实现进程功能，创建子类对象之后，调用对象的start方法。

下面来说一下进程池，就是实现已经创建好的进程:

# 进程池  
from multiprocessing import Pool  
import os  
import random  
import time  
  
  
def worker(num):  
 for i in range(5):  
 print(**'===pid=%d==num=%d='** % (os.getpid(), num))  
 time.sleep(1)  
  
  
# 3表示进程池中最多有三个进程一起执行  
pool = Pool(3)  
  
for i in range(10):  
 print(**'---%d---'** % i)  
 # 向进程中添加任务  
 # 注意：如果添加的任务数量超过了进程池中进程的个数的话，那么就不会接着往进程池中添加，如果还没有执行的话，他会等待前面的进程结束，然后在往  
 # 进程池中添加新进程  
 pool.apply\_async(worker, (i,))  
  
pool.close() # 关闭进程池  
pool.join() # 主进程在这里等待，只有子进程全部结束之后，在会开启主线程

大家可以看到在Pool方法就是创建了一个进程池，3表示创建进程数，通过pool.apply\_async( )将子进程添加到进程池中，pool.close( )表示关闭进程池，一定要添加join函数，否则主进程直接崩了，看不到进程池中子进程的现象。

1. **如何产生僵尸进程？如何产生孤儿进程？**

在 unix 或 unix-like 的系统中，当一个子进程退出后，它就会变成一个僵尸进程，如果父进程没有通过 wait 系统调用来读取这个子进程的退出状态的话，这个子进程就会一直维持僵尸进程状态。

Zombie process - Wikipedia 中是这样描述的：

On Unix and Unix-like computer operating systems, a zombie process or defunct process is a process that has completed execution (via the exit system call) but still has an entry in the process table: it is a process in the "Terminated state". This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the wait system call, the zombie's entry is removed from the process table and it is said to be "reaped". A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a resource leak.

并且僵尸进程无法通过 kill 命令来清除。

本文将探讨如何手动制造一个僵尸进程以及清除僵尸进程的办法。

手动制造一个僵尸进程

为了便于后面讲解清除僵尸进程的方法，我们使用日常开发中经常使用的 multiprocessing 模块来制造僵尸进程（准确的来说是制造一个长时间维持僵尸进程状态的子进程）：   
from multiprocessing import Process, current\_process  
import logging  
import os  
import time  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'%(asctime)-15s - %(levelname)s - %(message)s'**)  
  
  
def run():  
 logging.info(**'exit child process %s'**, current\_process().pid)  
 os.\_exit(3)  
  
  
p = Process(target=run)  
p.start()  
time.sleep(100)

所谓孤儿进程，故名思义，和现实生活中的孤儿有点类似，当一个进程的父进程结束时，但是它自己还没有结束，那么这个进程将会成为孤儿进程。最后孤儿进程会被init进程（init进程的pid为1）的进程收养，当然在子进程结束时也会由init进程完成对它的状态收集工作，因此一般来说，孤儿进程并不会有什么危害.

1. **什么是进程池？完整的进程池使用步骤是什么？**

答案见7题

1. **进程间通信的方式都有哪些？详细解释**

1. 管道pipe：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。  
 2. 命名管道FIFO：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。  
 4. 消息队列MessageQueue：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。  
 5. 共享存储SharedMemory：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。  
 6. 信号量Semaphore：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。  
 7. 套接字Socket：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。  
 8. 信号 ( sinal ) ： 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。

1. **解释一下同步与互斥。**

进程同步

定义：指系统中多个进程中发生的事件存在某种时序的关系，需要相互合作，共同完成一项任务。如如果进程A产生数据，而进程B打印数据，则B在打印之前必须等A产生数据。

进程互斥

定义：由于操作系统各进程需要使用共享资源，而这些资源需要排他性使用，各进程之间竞争使用这些资源，这些关系称为进程互斥。如：一个文件的读写问题。一个文件在被读的时候，不能同时进行写的操作。

临界资源：系统中某些资源一次只允许一个进程使用，这样的资源称为临界资源或互斥资源或共享资源

临界区：各个进程中对某个临界资源实施操作的程序片段。

临界区使用原则：

没有进程在临界区时，想进入临界区的进程可进入。

不允许两个进程同时在临界区。

临界区运行的进程不得阻塞其他进程进入临界区

不得使进程无限等待进入临界区

1. **线程有哪些特点？**

1)轻型实体  
 线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证独立运行的 资源。  
 线程的实体包括程序、数据和TCB。线程是动态概念，它的动态特性由线程控制 块TCB(Thread Control Block)描述。TCB包括以下信息：  
 (1)线程状态。  
 (2)当线程不运行时，被保存的现场资源。  
 (3)一组执行堆栈。  
 (4)存放每个线程的局部变量主存区。  
 (5)访问同一个进程中的主存和其它资源。  
 用于指示被执行指令序列的程序计数器、保留局部变量、少数状态参数和返回地址 等的一组寄存器和堆栈。  
 2)独立调度和分派的基本单位。  
 在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的 基 本单位。由于线程很“轻”，故线程的切换非常迅速且开销小(在同一进程中的)。  
 3)可并发执行。  
 在一个进程中的多个线程之间，可以并发执行，甚至允许在一个进程中所有线程都 能并发执行;同样，不同进程中的线程也能并发执行，充分利用和发挥了处理机与外围 设备并行工作的能力。  
 4)共享进程资源。  
 在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有 线程都具有相同的地址空间(进程的地址空间)，这意味着，线程可以访问该地址空间的 每一个虚地址;此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。 由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。

1. **如何创建线程**

创建线程有3种方法：

1、thread模块的start\_new\_thread函数

2、继承自threading.Thread模块

3、用theading.Thread直接返回一个thread对象，然后运行它的start方法

1. **线程间如何进行通信？**

Queue

使用线程队列有一个要注意的问题是，向队列中添加数据项时并不会复制此数据项，线程间通信实际上是在线程间传递对象引用。如果你担心对象的共享状态，那你最好只传递不可修改的数据结构（如：整型、字符串或者元组）或者一个对象的深拷贝。

Queue 对象提供一些在当前上下文很有用的附加特性。比如在创建 Queue 对象时提供可选的 size 参数来限制可以添加到队列中的元素数量。对于“生产者”与“消费者”速度有差异的情况，为队列中的元素数量添加上限是有意义的。比如，一个“生产者”产生项目的速度比“消费者”“消费”的速度快，那么使用固定大小的队列就可以在队列已满的时候阻塞队列，以免未预期的连锁效应扩散整个程序造成死锁或者程序运行失常。在通信的线程之间进行“流量控制”是一个看起来容易实现起来困难的问题。如果你发现自己曾经试图通过摆弄队列大小来解决一个问题，这也许就标志着你的程序可能存在脆弱设计或者固有的可伸缩问题。 get() 和 put() 方法都支持非阻塞方式和设定超时。

1. **解释名词：GIL线程全局锁**

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个GIL锁：Global Interpreter Lock，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，Python虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。

1. **GIL线程全局锁所造成的问题该如何解决？**

见上题

1. **你知道程序设计的基本原则有哪些吗？**

1.单一职责

简单来说单一职责就是一个类只负责一个功能。更加具体的说就是对一个类而言，应该是一组相关性很高的函数、数据的封装，是高内聚低耦合的，对外界而言应该仅有一个引起它变化的原因。

单一职责在项目中的使用:

1.项目中的新手引导变量的管理可以统一在各自的Modle中用单独的类来管理

2.MVP模式P层生命周期与V层生命周期的同步可以用单独的包装类来实现，

3.各种基础框架功能的定义，例如：图片的加载、缓存、显示等都应该在各自的类中去做。

2.开闭原则

开闭原则的英文全称是Open Close Principle缩写即OCP。开闭原则的定义是：软件中的对象(类、模块、函数等)应该对于扩展是开放的，但是对于修改是封闭的。在软件的生命周期内，因为变化、升级和维护等原因需要对软件的原有代码进行修改时，可能会将错误的代码引入，从而破坏原有系统。因此当软件需求发生变化时，我们应该尽量通过扩展的方式 来实现变化，而不是通过修改已有的代码。

开闭原则在项目中的使用:

1.基类与子类，子类可以继承父类并扩展父类的功能

2.接口与实现类，接口定义功能，实现类按照各自的需求实现

3.里氏替换原则

里氏替换原则的定义:如果对每一个类型为S的对象O1，都有类型为T的对象O2，程序P在所有的对象O1都带换成O2时，程序P的行为没有发生变化，那么类型S是类型T的子类型换言之就是所有引用基类的地方必须能透明的使用其子类的对象。更通俗的讲就是只要父类出现的地方子类就可以出现，而且替换为子类也不会产生任何的错误或者异常。

里氏替换原则的核心是抽象，而抽象又依赖于继承这个特性，在OOP当中，继承的优缺点都相当明显。

优点：

1.代码重用，减少创建类的成本，每个子类都拥有父类的方法和属性

2.子类与父类基本相似，但又与父类有所区别

3.提高代码的可扩展性

缺点:

1.继承是侵入性的，只要继承就必须拥有父类的方法和属性

2.可能造成子类代码冗余，灵活性降低，因为子类必须拥有父类的属性和方法

4.依赖倒置原则

依赖倒置原则指定了一种特定的解耦形式，使得高层次的模块不依赖与低层次模块的实现细节的目的，依赖模块被颠倒了。依赖倒置原则有以下几个关键点:

1.高层模块不应该依赖于低层模块，两者都应该依赖其抽象

2.抽象不应该依赖于细节

3.细节应该依赖于抽象

5.接口隔离原则

接口隔离原则的定义是:客户端不应该依赖于他不需要的接口。另一种定义是:类之间的依赖关系应该建立在最小的接口上。接口隔离原则将非常庞大，臃肿的接口拆分成更小的和更具体的接口，这样客户端将会值需要知道它们感兴趣的方法。接口隔离原则的目的是系统解开耦合，从而容易重构、更改和部署。

6.迪米特原则

迪米特原则：一个对象应该对其他对象有最少的了解，通俗的讲，一个类应该对自己需要耦合或调用的类知道的最少，类的内部如何实现与调用者或者依赖者没有关系，调用者或者依赖者只需要知道他需要的方法即可，其他的一概不管。类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。