1.

要让Python程序实现多进程（multiprocessing， 对应python中的multiprocess模块），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个fork () 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是fork () 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

**子进程永远返回0，而父进程返回子进程的ID**。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。

Python的os模块封装了常见的系统调用，其中就包括fork，可以在Python程序中轻松创建子进程：

# multiprocessing.py

import os

print 'Process (%s) start...' % os.getpid()

pid = os.fork()

if pid==0:

print 'I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid())

else:

print 'I (%s) just created a child process (%s).' % (os.getpid(), pid)

运行结果如下：

Process (876) start...

I (876) just created a child process (877).

I am child process (877) and my parent is 876.

由于Windows没有fork调用，上面的代码在Windows上无法运行。由于Mac系统是基于BSD（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了fork调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的Apache服务器就是由父进程监听端口，每当有新的http请求时，就**fork出子进程**来处理新的http请求。

2. Python多线程 (multithreading, 对应python中的threading模块) 的缺陷

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。

3.Python多线程和多进程直接的区别：

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了， 所以多线程需要有锁的机制。

4. 多任务模型

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭” ，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式。

5.

ps -ef 输出具体含义

UID        PID PPID  C STIME TTY          TIME CMD

各相关信息的意义：

UID 程序被该 UID 所拥有

PID 就是这个程序的 ID

PPID 则是其上级父程序的ID

C CPU 使用的资源百分比

STIME 系统启动时间

TTY 登入者的终端机位置

TIME 使用掉的 CPU 时间。

CMD 所下达的指令为

Python的Multiprocessing模块。 (基于进程的模块， 可以充分发挥出多核处理器的功用, 同时也是弥补使用多线程threading（以前的multithreading模块的劣势)

1. Process类

p = Process (target=f, args=('bob',)) 🡪 target指明此进程所要执行的函数，args 表明输入的参数

p.start() 🡪 调用对象的start方法开始执行

p.join()

🡪在进程中，以os.getppid()获取父进程id, 以getpid()获取自己进程的id

2. 进程之间数据的交流。Python给予了两种方式，提供进程之间的交流。

队列(Queue)

Queue的功能是将每个核或线程的运算结果放在队里中， 等到每个线程或核运行完毕后再从队列中取出结果。我们可以在一个进程中使用Queue对象的put() 方法像队列中放入数据，然后可以在任何进程中通过Queue对象的get()方法取出队尾的一个数据。

管道(Pipes)

Manager

3. 效率上，多进程比多线程快许多，多线程和普通执行过程相比差别不大，甚至还会比普通执行过程更长。

4.

Multiprocessing模块中的Pool类可以提供指定数量的并发进程，当有新的请求提交到池中时，如果池子的限额没有满，则会加入池中，否则请求等待。

---调用Pool实例的join方法，将会等待池中所有的子进程完毕。

---调用join方之前必须调用close方法

---当close以后，池子不能再加入新的请求。

---Pool的apply\_async方法是向池中加入异步请求，池中的请求可以并发执行。

---Pool的apply方法是向池中加入同步请求，池中的请求完成后才可以加入下一个请求。该方法只能允许一个进程进入池子，在一个进程结束之后，另外一个进程才可以进入池子。

---Pool的apply\_async方法执行后将返回异步结果对象，将对象get方法将获得原方法的结果。

---Pool对象的map方法集合了基础函数map的功能以及Pool的apply\_async()函数的功能。

**基础Python函数map的功能是：**

第一个参数 function 以参数序列中的每一个元素作为参数传入 function 函数，并直接function函数，map函数的返回为每次 function 函数执行的返回值所形成的新列表。

map (function, iterable, ...)

**基于同样原理Pool中的map函数**的功能是根据给定的iterable，每次将一个元素作为job的参数（所以map函数只允许job接受一个参数，如果job需要多个参数使用Process循环start,循环join这样有更高自由度），放入pool池中并且异步执行。将各进程返回的结果放入队列中，作为map函数的返回值。

map (job, iterable, ...)

例如：

def multicore():

pool = mp.Pool()

res = pool.map(job, range(10)) 🡪直接用pool.map函数发起任务并且获取结果，结果为map函数将各进程执行job函数的的返回值放入队列中，将队列作为map的返回结果

print(res)

res = pool.apply\_async(job, (2,)) 🡪也可以不用map函数获取结果，使用apply\_async,然后从返回对象中get()方法获取这个进程的返回结果。

# 用get获得结果

print(res.get())

5.

from multiprocessing import Pool 🡪 这个pool是基于进程的Pool，一般使用这个

from multiprocessing.pool import ThreadPool🡪 这个pool是基于线程的Pool

6.

def generate\_data(data):  
 for i in range(1, 4):  
 data.append(i)  
 return data  
  
  
data = []  
pool = Pool(processes=cpu\_count()) # 使用进程时候无法改变父进程的值  
for i in range(3):  
 async\_result = pool.apply\_async(generate\_data, args=(data,))  
pool.close()  
pool.join()  
print("All Processes End")  
print("Data is:" + str(data))  
  
data = []  
pool = ThreadPool(processes=cpu\_count()) # 使用线程时候可以改变父进程的值  
for i in range(3):  
 async\_result = pool.apply\_async(generate\_data, args=(data,))  
pool.close()  
pool.join()  
print("All threading End")  
print("Data is:" + str(data))

从以上例子我们可以看出，如果我们使用线程Threading那么最后data中会有数据，此数据由线程中提供，所以主线程和子线程共用同一份数据并且都可以访问修改。如果我们使用多进程，那么一般情况下变量将依据进程分割，父进程和子进程的变量在默认下不能互相修改，会有隔离，最后的data中没有任何数据。并且在多进程的环境下我们也不能期望子进程可以在默认情况下，从父进程拷贝一份值一样的全局变量。

7.

对于进程之间的变量共享，我们可以使用multiprocessing.Manger(). Manager支持普通数值Value, list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Queue, 和Array的共享。并且manager已经默认实现锁的机制。

**-🡪 Manager Dict的深层匹配赋值（超过2层的赋值将不生效）（2.7情况下）**

mng\_dict = Manager().dict()  
  
mng\_dict['a'] = {"b": 1}  
  
# 可以生效因为只有[‘a’]一层  
# mng\_dict['a'] = 1  
# 可以生效因为只有[‘a’]一层  
# mng\_dict['a'] = [1, 2, 3, 4]  
# 可以生效因为只有[‘a’]一层  
# mng\_dict['a'] = {"1": "2", "3": "4"}  
# 无法因为有[‘a’][‘b’]两层， 多于一层将无效，文档：proxy不知道dict更新。  
# mng\_dict['a']["b"] = "1"

# 可以生效，因为我们一层一层指定  
tmp = mng\_dict['a']  
tmp["b"] = 2  
mng\_dict['a'] = tmp  
  
print(str(mng\_dict))

8.

我们也可以使用Queue作为进程之间的变量共享：

例如：

def f(q, hostname):  
 test\_dict = q.get()  
 test\_dict[hostname] = "Filled"  
 q.put(test\_dict)  
  
  
result=[]  
test\_dict = {"A": {"B": "C"}}  
hostNames = ["apple", "banana"]  
for hostname in hostNames:  
 q = Queue()  
 q.put(test\_dict)  
 p = Process(target=f, args=(q, hostname))  
 p.start()  
 result.append((p, q))  
for each\_process, each\_queue in result:  
 each\_process.join()  
 each\_test\_dict = each\_queue.get()  
 for key in each\_test\_dict.keys():  
 test\_dict[key] = each\_test\_dict[key]  
print(str(test\_dict))

--需要注意的事，put方法对于queue是生产者对象，get方法对于queue是消费者对象。Queue.get()方法如果在队列中没有可以取出的内容，将会一直等待，除非设置timeout参数。

--Queue中不能放入大于32763的数据，否则将会出错无法放入Queue, 无法放入Queue以后，queue.get将获取不到数据，从而卡在那里。