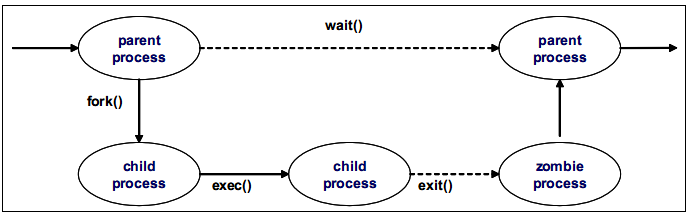
一、多进程编程

**1、forking工作原理**

**forking**：父进程将自身资源拷贝一份，fork出一个子进程

**进程生命周期**：



-父进程fork出子进程并挂起

-子进程运行完毕，释放资源成为僵尸进程，通知父进程

-父进程获知子进程结束，释放子进程所有资源

**僵尸进程**：

僵尸进程会占用PID，导致系统不能产生新的进程；

可以通过杀死其父进程来消除僵尸进程（孤儿进程由systemd接管，发现其为僵尸进程，执行清除代码）

1. **forking编程**

**1）os.fork()**

"扫描存活主机"

import os

import subprocess

def ping(ip):

rc = subprocess.call('ping -c2 %s &> /dev/null' % ip, shell=True)

if rc == 0: #rc=ping的退出码$?

print('%s:up' % ip)

else:

print('%s:down' % ip)

if \_\_name\_\_ == '\_\_main\_\_':

ips = ('176.121.202.%s' % i for i in range(1,255))

for ip in ips:

ret\_val = os.fork()

#对fork()的调用，父进程返回子进程的PID；子进程返回0

#父进程生成子进程，子进程执行功能代码

if not ret\_val:

ping(ip)

exit()

#退出子进程，若不退出将继续多进程历遍ip

**2）查看僵尸进程**

#watch -n1 ps a 观察僵尸进程的出现(Z状态)

#kill父进程可以把僵尸进程杀死

**3）使用waitpid()处理僵尸进程**

import os

import time

ret\_val = os.fork()

if ret\_val:

time.sleep(20)

result = os.waitpid(-1, 0)

#参数一：-1表示与函数wait()相同

#参数二：0表示挂起父进程，直到子结束再继续；1表示不挂起，往下执行

#返回值result是元组；若子进程已结束返回(子pid，0)，若未结束返回(0, 0)

#一个waitpid只能处理一个僵尸进程

time.sleep(20)

else:

time.sleep(40)

**4）案例：多进程TCP服务器**

- 父进程负责接收客户端的连接请求

- 子进程负责与客户端通信

import os

import socket

from time import strftime

class TcpServ:

def \_\_init\_\_(self, host='', port=1234):

... ...

def chat(self, cli\_sock):

while True:

cli\_sock.recv(1024)

... ...

cli\_sock.send(data.encode())

def forkmain(self):

while True:

cli\_sock, cli\_addr = self.serv.accept() #接受客户端连接

ret\_val = os.fork()

if not ret\_val:

self.serv.close() #子进程关闭服务器套接字

self.chat(cli\_sock) #与客户端通信

cli\_sock.close() #关闭客户端连接

exit()

cli\_sock.close() #父进程关闭客户端套接字

while True:

result = os.waitpid(-1, 1)

#处理僵尸进程

if result[0] == 0:

break

self.serv.close()

if \_\_name\_\_ == '\_\_main\_\_':

serv = TcpServ()

serv.forkmain()

二、多线程编程

所有的线程运行在同一个进程中，共享相同的运行环境；

在单CPU 的系统中，每个线程会被安排成每次只运行一小会，然后就把CPU 让出来，让其它的线程去运行，从而实现”并发“。

1. **案例：多线程扫描存活主机**

import threading

import subprocess

def ping(host):

rc = subprocess.call('ping -c2 %s &> /dev/null' % host, shell=True)

if not rc:

print('%s:up' % host)

else:

print('%s:down' % host)

if \_\_name\_\_ == '\_\_main\_\_':

ips = ('192.168.1.%s' % i for i in range(1,255))

for host in ips: #主线程用于产生工作线程

t = threading.Thread(target=ping, args=(host,))

t.start() #调用target(args)，相当于执行ping(host)

###方法2###

class Ping:

def \_\_init\_\_(self, host):

self.host = host

def \_\_call\_\_(self):

rc = subprocess.call('ping -c2...' % self.host, shell=True)

if not rc:

print('%s:up' % self.host)

else:

print('%s:down' % self.host)

if \_\_name\_\_ == '\_\_main\_\_':

ips = ('192.168.1.%s' % i for i in range(1,255))

tlist = []

for host in ips:

t = threading.Thread(target=Ping(host)) #直接创建实例

tlist.append(t) #将线程放到列表

t.start() #调用target()

for tr in tlist:

tr.join() #相当于多进程的waitpid，等待线程执行完毕再往下

**2、案例：多线程TCP服务器**

import socket

from time import strftime

import threading

class TcpServ:

def \_\_init\_\_(self, host='', port=1234):

... ...

def chat(self, cli\_sock):

while True:

cli\_sock.recv(1024)

... ...

cli\_sock.send(data.encode())

cli\_sock.close()

def mainloop(self):

while True:

cli\_sock, cli\_addr = self.serv.accept()

t = threading.Thread(target=self.chat, args=(cli\_sock, ))

t.start()

self.serv.close()

if \_\_name\_\_ == '\_\_main\_\_':

serv = TcpServ()

serv.mainloop()

Tips：查看线程的命令：ps -T -p <pid> top -H -p <pid>

**3、多进程、多线程的对比**

GIL：全局解释器锁，它限制某个时刻只能有一个线程的代码交给python解释器处理，导致计算密集型应用，多线程效率不高。

计算密集型应用：多线程不适用，不能利用多CPU的优势

IO密集型应用：多线程使用