多进程复习

并行 并发

并行（parallel）

• 在同一时刻，有多条指令在多个处理器上同时执行

并发（concurrency）

• 在同一时刻，只能有一条指令执行，但多个指令被快速轮换执行，使得在宏观上表现多个指令同时执行的效果

进程和线程

– 进程（process）

• 操作系统分配资源的基本单位

– 线程（thread）

• CPU调度和分派的基本单位

– 应用程序至少有一个进程

– 一个进程至少有一个线程

– 同一进程的多个线程可以并发执行

– 进程在执行过程中拥有独立的内存单元，而线程共享内存

– 多进程编程需要考虑进程间的通信 (IPC)

– 进程切换时，耗费资源较大

进程与线程比较

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	共享进程数据，数据共享简单，但导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程出错将导致整个进程中止	进程占优
分布式	适用于多核、多机分布式	适用于多核分布式	进程占优

进程概念

- 进程（process）
  - 进程号
    - os.getpid()
  - 孤儿进程
    - 子进程在父进程退出后仍在运行，会被init进程接管，init以父进程的身份处理子进程运行完毕后遗留的状态信息
  - 僵尸进程
    - 父进程创建子进程后，如果子进程退出，但父进程并没有调用wait或waitoid获取子进程的状态信息，那么子进程的进程描述符将一直保存于系统，对应的子进程称为僵尸进程
    - 僵尸进程无法通过kill命令来清除

进程创建

创建进程类

```
Process([group [, target [, name [, args [, kwargs]]]]])
# 实例对象表示一个子进程（尚未启动）
# 使用关键字的方式来指定参数
# target表示调用对象，即子进程要执行的任务
# args指定传给target调用对象的位置参数，元组形式，仅有一个参数时要有逗号
# kwargs表示调用对象的字典参数
# name为子进程的名称
# 注：在Windows中Process()必须放到if __name__ == '__main__'中
p.start()
# 启动进程，并调用该子进程中的p.run()
p.run()
# 进程启动时运行的方法，会调用target，自定义类定要实现该方法
p.terminate()
'''
• 强制终止进程p，但不进行任何清理操作
• 如果p创建了子进程，该子进程将成为僵尸进程
• p创建的锁也将不会释放，可能导致死锁
• 一般不建议使用
'''
p.is_alive()
# 如果p仍然运行，返回True
p.join([timeout])
# 主进程等待p终止（主进程处于等状态，而p处于运行状态）
# timeout是可选的超时时间
# 只能join住start开启的进程，而不能join住run开启的进程
p.daemon
'''
• 默认值为False
• 可以设置为True，但必须在p.start()之前设置，成为后台
运行的守护进程，当p的父进程终止时，p也随之终止，且p不
能创建新的子进程
'''
p.name
# 进程的名称
p.pid
# 进程的pid
p.exitcode
# 进程在运行时为None
# 如果为–N，表示被信号N结束
p.authkey
# 进程的身份验证键
```

创建实例

```
#在windows上会有问题
from multiprocessing import Process
print('main process start')
def run():
    pass
p=Process(target=run)#recursive
p.start()
if __name__=='__main__':
  pass
```

```
import time
import random
from multiprocessing import Process

n=0

def task(name):
  print("task of {} starts".format(name))
  time.sleep(random.randrange(50,100))
  print("task of {} ends".format(name))

class Task(Process):
  def __init__(self,name):
    super().__init__()#不能忘了
    self._name=name

  @property
  def name(self):
    return self._name

  def run(self):#不能忘了
    print('task {} starts with pid {}...'.format(self.name,self.pid))
    global n
    n=random.random()
    print("n={} in process {}".format(n,self.name))
    #mem
    l=list(range(int(n*100000000)))#观察内存的不同
    time.sleep(random.randrange(20,50))
    print('task {} ends with pid {}...'.format(self.name,self.pid))


if __name__=='__main__':
  '''for i in range(0,4):
    p=Process(target=task,args=('task {}'.format(i),))
    p.start()
    print('pid:{}'.format(p.pid))'''
```

```
  plist=[]
  for i in range(0,4):
    p=Task('process-{}'.format(i+1))
    plist.append(p)
  for p in plist:
    p.start()
  for p in plist:
    #注意一定是先子进程都启动后，再一一join，否则启动后马上join会变成“串行”
    #1. 是的，这样写join仍然会卡着等p1运行结束，但其他进程如p2, p3等仍在运行，等p1运行结束后，循环继续，p2,p3等可能也运行结束了，会迅速完成join的检验
    #2. join花费的总时间仍然是耗费时间最长的那个进程运行的时间，这样跟我们的目的是一致的。
    pass
    p.join()
  print('main')
  print("n={} in main".format(n))
```

进程的同步

```
from multiprocessing import Process
from multiprocessing import Lock
import json
import random
import time

tf='ticks.json'

def info():
  ticks=json.load(open(tf))
  print("ticks from {} to {} left
{}".format(ticks['origin'],ticks['dest'],ticks['count']))

def buy(pname):
  ticks=json.load(open(tf))
  time.sleep(random.random())
  if ticks['count']>0:
    ticks['count']-=1
    time.sleep(random.random())
    json.dump(ticks,open(tf,'w'))
    print('{} buy one tick from {} to {}!'.format(pname,ticks['origin'],ticks['dest']))
  else:
    print('oh....no ticks :(')

def task(pname):
  info()
  buy(pname)

#加锁
def lock_task(name,lock):
```

```
  info()
  lock.acquire()
  try:
    buy(name)
  except:
    raise
  finally:
    lock.release()

if __name__=='__main__':
  lock=Lock()
  clients=[]
  for i in range(20):
    name='client-{}'.format(i+1)
    #p=Process(target=task,name=name,args=(name,))
    p=Process(target=lock_task,name=name,args=(name,lock))
    clients.append(p)
  for p in clients:
    p.start()
  for p in clients:
    p.join()
  print("all clients finished...")
```

```
from multiprocessing import Process
from multiprocessing import Semaphore
from multiprocessing import current_process
import time
import random

def get_connections(s):
  s.acquire()
  try:
    print(current_process().name+' acqiure a connection')
    time.sleep(random.randint(1,2))
    print(current_process().name+' finishes its job and return the connection')
  except:
    raise
  finally:
    s.release()

if __name__=='__main__':
  connections=Semaphore(5)
  workers=[]
  for i in range(20):
    p=Process(target=get_connections,args=(connections,),name='worker:'+str(i+1))
    workers.append(p)
  for p in workers:
    p.start()
```

```
  for p in workers:
    p.join()
  print("all workers exit")
```

```
from multiprocessing import Process
from multiprocessing import Event
import time
import random

def car(event,name):
  while True:
    if event.is_set():
      time.sleep(random.random())
      print("car {} passes...".format(name))
      break
    else:
      print("car {} waits...".format(name))
      event.wait()#阻塞直至事件状态发生变化

def light(event):
  while True:
    if event.is_set():
      event.clear()
      print("红灯")
      time.sleep(random.random())
    else:
      event.set()
      print("绿灯")
      time.sleep(random.random())

if __name__=='__main__':
  event=Event()
  event.clear()
  l=Process(target=light,args=(event,))
  l.daemon=True
  l.start()
  cars=[]
  for i in range(10):
    c=Process(target=car,args=(event,'c_'+str(i+1)))
    cars.append(c)
  for c in cars:
    c.start()
  for c in cars:
    c.join()
  print("all cars passed...")
```

队列

Queue([maxsize])
q.put()
• 插入数据到队列，参数blocked默认为True
• 可能抛出Queue.Full异常
q.get()
• 从队列读取并删除一个元素
• 可能抛出Queue.Empty异常

JoinableQueue([maxsize])

允许数据的消费者通知生成者数据已经被成功处理
• 通过共享信号和条件变量来实现通知
– maxsize
• 队列中允许最大数据项数，省略则无大小限制
– q.task_done()
• 消费者使用此方法发出信号，表示q.get()的返回数据已经被处理，但如果调用此方法的次数大于从队列中删除数据的数量，将引发ValueError异常
– q.join()
• 生产者调用此方法进行阻塞，直到队列中所有的项目均被处理，阻塞将持续到队列中的每个数据均调用q.task_done()方法为止

```
from multiprocessing import Process, Queue
import time
import random
import os

def produce(q):
  time.sleep(random.random())
  q.put('car_by_{}'.format(os.getpid()))
  print("{} produces a car...".format(os.getpid()))

def buy(q):
  car=q.get()
  if car is None:
    print("no car. {} ends.".format(os.getpid()))
    return
  else:
    time.sleep(random.random())
    print("{} buy the car {}".format(os.getpid(),car))

if __name__=='__main__':
  q=Queue()
  procucers=[]
  consumers=[]
  for i in range(0,2):
```

```
    p=Process(target=produce,args=(q,))
    procucers.append(p)
  for i in range(0,100):
    c=Process(target=buy,args=(q,))
    consumers.append(c)
  for p in procucers:
    p.start()
  for p in procucers:
    p.join()
  for c in consumers:
    c.start()
  for c in consumers:
    q.put(None)#主进程发信号结束，但要给每一个consumer准备
  print('main')
```

```
from multiprocessing import Process
from multiprocessing import JoinableQueue
import time
import random
import os

class Product:
  def __init__(self,name,volume):
    self._name=name
    self._volume=volume
  def __str__(self):
    return "Name:{}\tVolume:{}".format(self._name,self._volume)

def consume(q):
  print(f"consumer {os.getpid()} started...")
  prod=q.get()
  time.sleep(random.randint(1,5))
  print("{} consumes {}".format(os.getpid(),prod))
  q.task_done()#发送信号，表明数据获取成功

def produce(q):
  print(f"producer {os.getpid()} started...")
  prod=Product(str(os.getpid())+"_pro",random.randint(10,100))
  time.sleep(random.randint(1,5))
  q.put(prod)
  print("{} produces {}".format(os.getpid(),prod))
  q.join()

if __name__=='__main__':
  q=JoinableQueue()
  producers=[]
  consumers=[]
  for i in range(0,2):
    p=Process(target=produce,args=(q,))
```

```
    producers.append(p)
  for i in range(0,100):
    c=Process(target=consume,args=(q,))
    c.daemon=True #必要，否则部分消费者进程并不退出
    consumers.append(c)
  for c in consumers:
    c.start()
  for p in producers:
    p.start()
  for p in producers:
    p.join()
  print('main')
```

管道

```
# 只需了解

from multiprocessing import Process, Pipe
import time
import os
import random

def receiver(conn,name):
  #recv_c, send_c=conn
  #send_c.close()#只用读端，关闭写端，注意windows有可能不能提前关闭
  while True:
    try:
      message=conn.recv()
      print("{} receives a message: {}".format(name,message))
    except EOFError as eof:
      conn.close()
      break

def send(conn,name,messages):
  #recv_c,send_c=conn
  #recv_c.close() #只用写端，关闭读端，注意windows有可能不能提前关闭
  for message in messages:
    conn.send(message)
    time.sleep(random.randint(1,3))
  conn.close()

if __name__=='__main__':
  messages=[]
  with open('po.txt') as f:
    for line in f:
      messages.append(line.strip())
```

```
    recv_con,send_con=Pipe()#在主进程中建立管道

    printer=Process(target=receiver,args=(recv_con,'printer'))
    printer.start()

    sender=Process(target=send,args=(send_con,'sender',messages))
    sender.start()

    #注意windows建议仅在此处关闭
    recv_con.close()
    send_con.close()

    sender.join()
    print("所有消息发送完成...")
    printer.join()
    print('所有消息接收完成...')
```