Programação Paralela e Distribuída

Processos

Prof. Hugo Alberto Perlin

Modelo Multi-Processos

- O paralelismo pode ser obtido tanto por meio de threads quanto por meio de processos
- No caso do Python, o uso de processos-filhos, permite executar código em múltiplos núcleos de processamento ao mesmo tempo
- Apesar de ser uma forma que possua um custo computacional maior

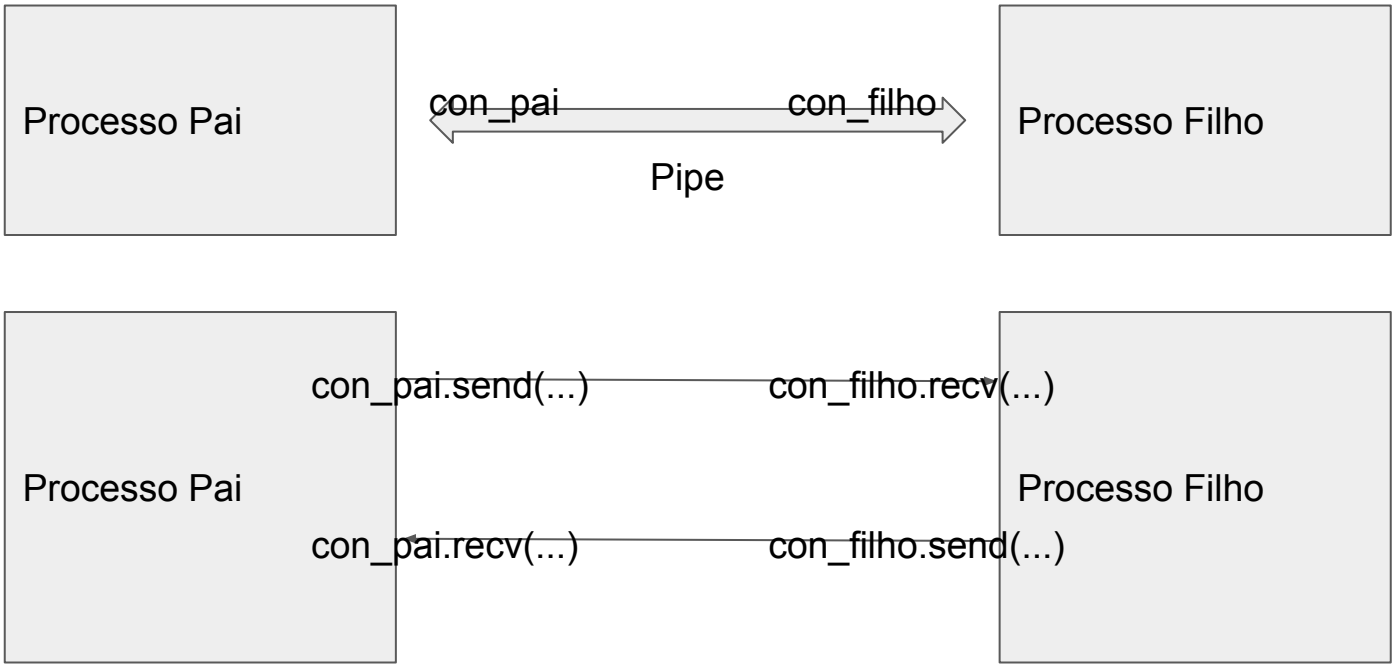
Comunicação entre Processos

- Para trocar objetos entre processos existem duas opções:
- Queue: é muito semelhante a Queue utilizada em threads

```
from multiprocessing import Process, Queue, current_process

def funcao(q):
    nomeProcesso=current_process().name
    q.put([nomeProcesso,42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    ...
        p = Process(target=funcao,args=(q,))
    ...

    while(not q.empty()):
        print(q.get())
```

Comunicação entre Processos

- Pipe: é uma função que cria um par de objetos de conexão, que por padrão são duplex (bidirecional)



Comunicação entre Processos

- Pipe: é uma função que cria um par de objetos de conexão, que por padrão são duplex (bidirecional)

```
from multiprocessing  import  Process,Pipe

def  funcao(con):
    lista=[]
    lista=con.recv()

    print("Recebido...{}".format(lista))

    res=[]
    for  n  in  lista:
        res.append(n*n)

    con.send(res)
    con.close()
```

```
if  __name__  ==  "__main__":
    con_pai,con_filho  =  Pipe()
    p  =  Process(target=funcao,args=(con_filho,))
    p.start()
    con_pai.send([1,2,3,4,5])
    print(con_pai.recv())
    p.join()
```

Sincronização de Processos

- O pacote multiprocessing conta com os mesmos tipos de objetos para realizar a sincronização de processos que o pacote threading
- Geralmente não são tão utilizadas quanto em threads
  - Lock = objeto que permite que somente um processo acesse a região crítica
  - RLock = objeto que permite que somente um processo acesse a região crítica, porém permite que o mesmo processo adquira o lock múltiplas vezes em sequência
  - Condition = permite que um ou mais processos aguardem por uma condição emitida por outra thread
  - Event = gerencia uma flag booleana, onde outros processos podem aguardar pela modificação do seu valor
  - Semáforo = objeto que permite o acesso a um recurso limitado
  - **Barrier =** objeto que especifica um ponto onde todos os processos devem chegar para que o processamento continue

Barrier 1/2

```
from multiprocessing import Process,Barrier
from time import time,sleep
from datetime import datetime


class MeuProcesso(Process):

    def __init__(self,barreira,tempo):
        Process.__init__(self)
        self.barreira = barreira
        self.tempo = tempo

    def run(self):
        print("{} iniciando!!".format(self.name))
        soneca = sleep(self.tempo)
        print("{} acordei!!".format(self.name))
        agora = time()
        self.barreira.wait()
        print("{} - {} continuando processamento!!!"format(self.name,datetime.fromtimestamp(agora)))

if __name__ == "__main__":
```

Barrier 2/2

```
if __name__ == "__main__":


    barreira = Barrier(2)

    p1 = MeuProcesso(barreira,5)
    p1.start()
    p2 = MeuProcesso(barreira,10)
    p2.start()

    p1.join()
    p2.join()
```

Compartilhamento de Dados

- Além de fila, é possível compartilhar objetos primitivos em python
- Deve ser utilizado com cuidado, pois geralmente as operações não são atômicas
- Isso significa que mecanismos de sincronização e acesso devem ser utilizados para garantir a consistência
- Tipos Array e Value

Value

```
from multiprocessing import Process, Value

def soma(n):
    for a in range(100000):
        with n.get_lock():
            n.value += 1.0

def subtracao(n):
    for a in range(100000):
        with n.get_lock():
            n.value -= 1

if __name__ == '__main__':
    #a classe Value aceita como parâmetro o tipo de dado (i=inteiro,d=double) e também
    #se um objeto lock será inicializado para auxiliar no controle de acesso do
    #dado compartilhado
    num = Value('d', 0.0,lock=True)
    pSoma = Process(target=soma, args=(num,))
    pSoma.start()
    pSubtracao = Process(target=subtracao, args=(num,))
    pSubtracao.start()
    print num.value
```

Array

```
from multiprocessing import Process, Array

def soma(vet):
    for a in range(len(vet)):
        for b in range(5000):
            with vet.get_lock():
                vet[a] +=1

def subtracao(n):
    for a in range(len(vet)):
        for b in range(5000):
            with vet.get_lock():
                vet[a] -=1

if __name__ == '__main__':
    vet = Array('i', [0 for a in range(10)],lock=True)
    pSoma = Process(target=soma, args=(vet,))
    pSoma.start()
    pSubtracao = Process(target=subtracao, args=(vet,))
    pSubtracao.start()
    pSoma.join()
    pSubtracao.join()
```

Managers

- O pacote multiprocessing possibilita a criação de managers para facilitar o acesso a objetos compartilhados
- É possível compartilhar dados pela rede
- Um manager controla um processo servidor que possui os objetos compartilhados e permite que outros processos acessem tais objetos
- `multiprocessing.`**`Manager`**`()`
- `multiprocessing.managers.`**`BaseManager()`**

Manager

```
import multiprocessing

def worker(list, item):
    x=0
    for a in range(10000):
        x+=1
    list.append(item)
    print("{} {}".format(item,list))

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    list = mgr.list()
    jobs = [ multiprocessing.Process(target=worker, args=(list, i))
             for i in range(10)
           ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print ('Results:', list)
```

BaseManager

```
##SERVIDOR
from multiprocessing.managers import BaseManager
from multiprocessing import Queue

queue = Queue()
class QueueManager(BaseManager):pass

QueueManager.register('get_queue',callable=lambda:queue)
m = QueueManager(address=('',50000),authkey=b'abc')
m.get_server().serve_forever()
```

BaseManager

```
##CLIENTES
from multiprocessing.managers import BaseManager
import sys

class QueueManager(BaseManager):
    pass

QueueManager.register('get_queue',callable=lambda:queue)
m = QueueManager(address=('localhost',50000),authkey=b'abc')
m.connect()
if(sys.argv[1]=='1'):
    queue = m.get_queue()
    queue.put('hello')
elif(sys.argv[1]=='2'):
    queue = m.get_queue()
    msg = queue.get()
    print(msg)



print("Fim...")
```

Exercícios

1. Utilizando a classe BaseManager, implemente um chat entre dois processos. Para facilitar, assuma que a conversa será sempre alternada, ou seja, escreve/envia/recebe.