# 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
- É uma forma de contornar uma limitação de projeto do Python chamada GIL (Global Interpreter Lock)

# GIL (Global Interpreter Lock)

- O efeito que o GIL afeta o funcionamento das threads em Python é relativamente simples:
- “Uma thread é executada em Python, enquanto N outras threads dormem ou aguardam I/O”;
- Isso é chamado de *Cooperative Multitasking*

# Pacote multiprocessing

- Para criar multiprocessos utilizamos o pacote multiprocessing

```
import multiprocessing
```

- A criação do processo é similar as threads, é feita através da chamada

```
multiprocessing.Process(target=funcao,args=(i,))
```

- Nesse caso será criado um processo filho, que irá executar a função **funcao**, com o parâmetro **i**

# Exemplo 1/2

```
import multiprocessing
import time



def funcao(i):
    print('função chamada no processo:%s'%i)
    time.sleep(5)
    return
```

# Exemplo 2/2

```
if __name__ == "__main__":
    print("Inciando meu trabalho...!!!")
    processos = []
    for i in range(5):
        p = multiprocessing.Process(target=funcao,args=(i,))
        processos.append(p)
        p.start()

    for p in processos:
        p.join()

    print("Finalizando meu trabalho...!!!")
```

# Explicando

- *Esta função é responsável por determinar a carga de trabalho do processo*

```
def funcao(i):
```

- *Esta instrução cria um novo processo, indicando qual será a função a ser executada e quais os parâmetros*

```
p = multiprocessing.Process(target=funcao,args=(i,))
```

- *O processo inicia a sua execução com a chamada ao método start*

```
p.start()
```

- *O processo pai deve aguardar todos os filhos terminarem sua execução*

```
p.join()
```

# Visualizando os processos

1. Abra um segundo terminal
2. Aumente o tempo de sleep no código de exemplo para 30 segundos
3. Execute o código de exemplo
4. No outro terminal execute o seguinte comando, para mostrar todos os processos rodando no computador que tenham a string python

```
ps aux | grep python
```

```
haperlin 26730  0.8  0.1  38388 11212 pts/3    S+   23:26   0:00 python3 multiprocessos.py
haperlin 26731  0.0  0.1  38388  9144 pts/3    S+   23:26   0:00 python3 multiprocessos.py
haperlin 26732  0.0  0.1  38388  9208 pts/3    S+   23:26   0:00 python3 multiprocessos.py
haperlin 26733  0.0  0.1  38388  9144 pts/3    S+   23:26   0:00 python3 multiprocessos.py
haperlin 26734  0.0  0.1  38388  9144 pts/3    S+   23:26   0:00 python3 multiprocessos.py
haperlin 26735  0.0  0.1  38388  9152 pts/3    S+   23:26   0:00 python3 multiprocessos.py
```

# Nomeando processos

- É possível nomear um processo no momento de sua criação chamada

```
multiprocessing.Process(name=nome,target=funcao,args=(i,))
```

- Nesse caso será criado um processo filho, com o **nome,** que irá executar a função **funcao**, com o parâmetro **i**
- Para saber o nome do processo, pode-se utilizar

```
nomeProcesso = multiprocessing.current_process().name
```

# Extendendo a classe Process

- Outra forma de criar processos é extender a classe Process

```
from multiprocessing import Process

class MeuProcesso(Process):

    def __init__(self):
        Process.__init__(self)

    def run(self):
        print("Eu sou um processo!! Meu nome é {}".format(self.name))

if __name__ == "__main__":
    p = MeuProcesso()
    p.start()
    p.join()
```

# Compartilhamento!!!

- Diferente de threads, os processos não compartilham memória “por padrão”
- Os dados são passados por cópia e não por referência.

```
from multiprocessing import Process
from threading import Thread

class MeuProcesso(Process):

    def __init__(self,lista):
        Process.__init__(self)
        self.lista = lista

    def run(self):
        for a in range(5,10,1):
            self.lista.append(a)
        print("Processo...{}".format(lista))
```

```
class MinhaThread(Thread):

    def __init__(self,lista):
        Thread.__init__(self)
        self.lista = lista

    def run(self):
        for a in range(5,10,1):
            self.lista.append(a)

        print("Thread...{}".format(lista))
```

# Compartilhamento!!!

```
if __name__ == "__main__":
    lista=[1,2,3,4]
    p = MeuProcesso(lista)
    p.start()
    p.join()
    print("Pai...{}".format(lista))

    lista=[1,2,3,4]
    t = MinhaThread(lista)
    t.start()
    t.join()
    print("Pai...{}".format(lista))
```

Saída da execução:

```
Processo...[1, 2, 3, 4, 5, 6, 7, 8, 9]
Pai...[1, 2, 3, 4]
Thread...[1, 2, 3, 4, 5, 6, 7, 8, 9]
Pai...[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# 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())
```

# Exercícios

1. Desenvolva um programa em Python, utilizando o modelo Produtor-Consumidor, onde o produtor lê um arquivo texto, separa as palavras e envia para o consumidor. O consumidor, por sua vez, deverá contar o número de ocorrências de cada palavra e salvar o resultado em um arquivo texto. Utilize paralelismo através de processos.
2. Escreva o código do exercício 1 utilizando Threads.
3. Apresente a média e o desvio padrão do tempo de execução, com 10 repetições, para cada versão da solução.