Tópicos de programação não síncrona

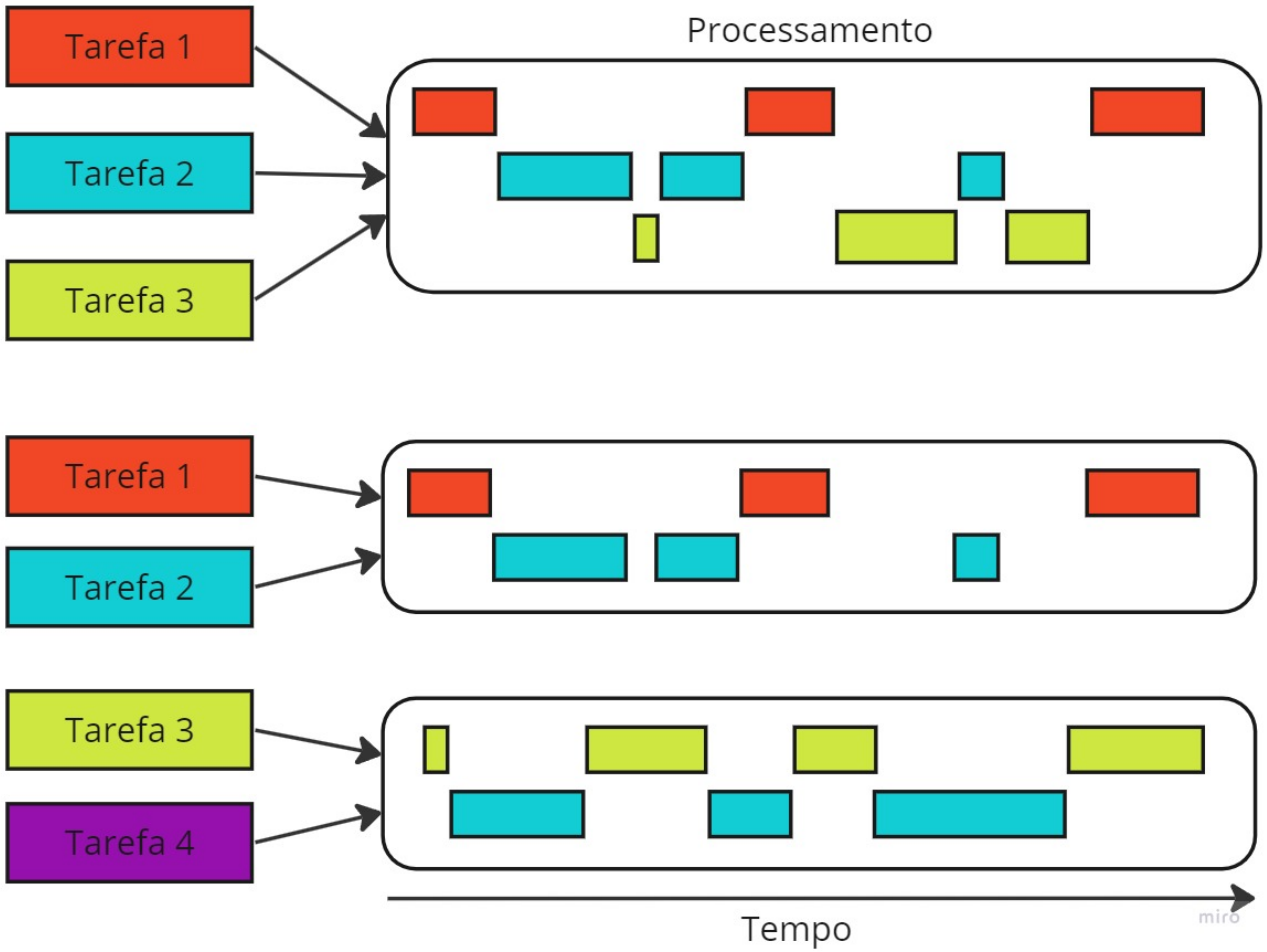
Evandro Paulo Folletto

Volpi.tech

2023

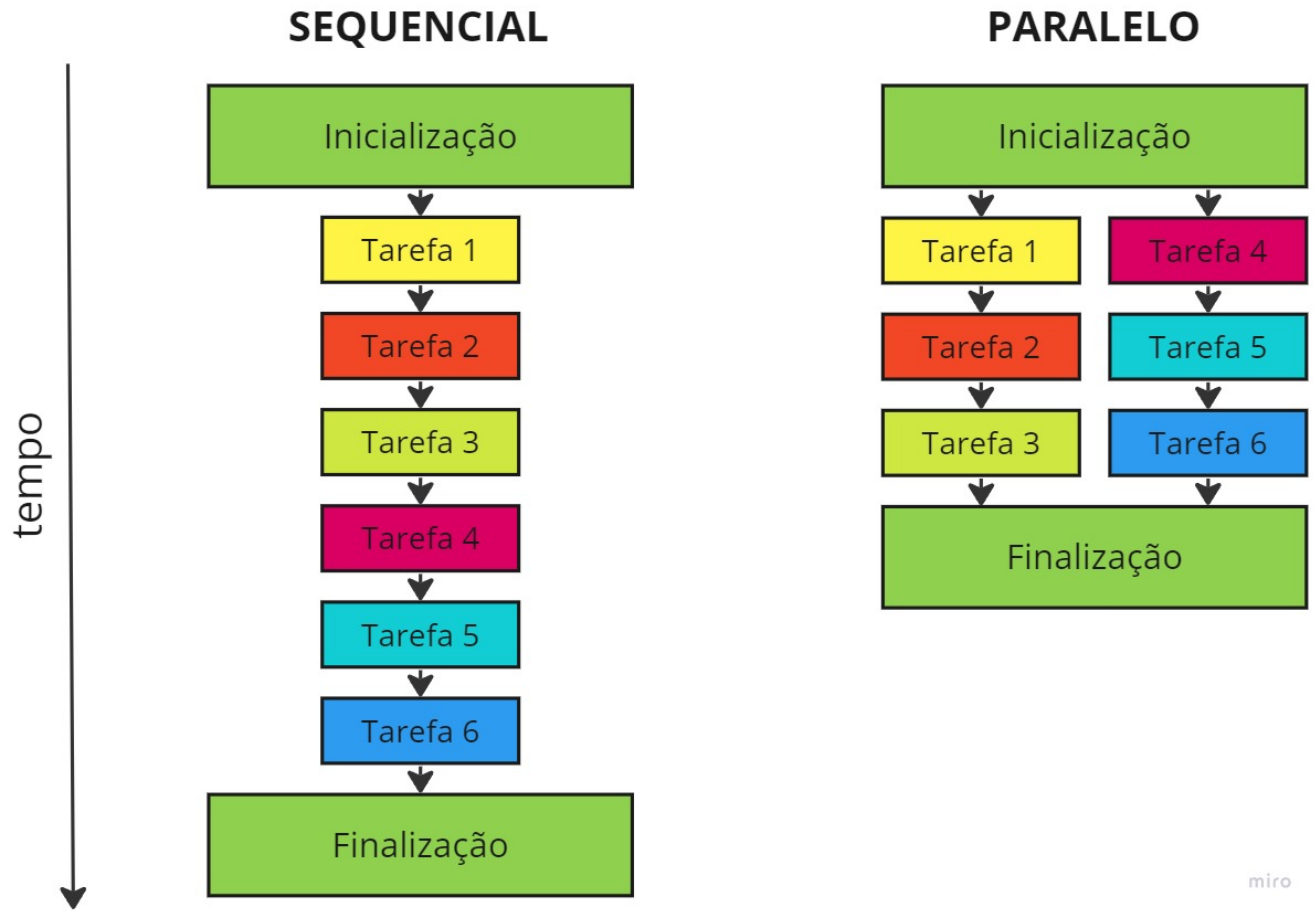
Concorrência: capacidade de executar sequencialmente um conjunto de tarefas independentes.

Paralelismo: execução de mais de uma tarefa por vez (de forma simultânea), a depender da quantidade de núcleos (cores) do processador. Quanto mais núcleos, mais tarefas paralelas podem ser executadas.



Sequencial: as instruções são executadas uma após a outra, em uma ordem predefinida. Cada instrução é executada completamente antes que a próxima seja iniciada, de forma linear.

Paralelismo: as instruções são executadas simultaneamente em múltiplos processadores ou núcleos de processamento, permitindo que várias instruções sejam processadas simultaneamente.



A utilização dos conceitos de multithreading e multiprocessamento para computação paralela e consequente ganho de desempenho é bastante difundida. Mas em Python existem particularidades:

Multithreading: útil para operações de E/S ou tarefas vinculadas à rede, como a execução de scripts, por exemplo, no caso de web scraping. As *threads* não podem alcançar o paralelismo completo aproveitando vários núcleos de CPU devido a restrições GIL em Python.

GIL (Global Lock Interpreter): só deixa uma thread por vez executar, isso acontece pois o python não é *Thread Safe* o que significa que diferentes threads podem ter acesso a memória de forma indevida e causar quebras no programa. Isso pode resultar em pior performance.

$\rightarrow$ Para utilizar Multithreading (pacote python): import threading.

Multiprocessamento: é uma escolha adequada caso as tarefas sejam extensas da CPU e não tenham nenhuma operação de E/S ou interações do usuário. Exemplos: algoritmos de *machine learning* e *deep learning*.

$\rightarrow$ Para utilizar Multiprocessamento (pacote python): import multiprocessing.

Exemplos:

- Exemplo 1
  - ./exemplo_1/main_A.py: exemplo utilizando sincronicidade
  - ./exemplo_1/main_B.py: exemplo utilizando multithreading
  - ./exemplo_1/main_B2.py: exemplo utilizando multithreading e podendo receber uma lista de tarefas
  - ./exemplo_1/main_C.py: exemplo utilizando multiprocessamento
- Exemplo 2
  - ./exemplo_2/main_A.py: exemplo utilizando sincronicidade
  - ./exemplo_2/main_B.py: exemplo utilizando multithreading
  - ./exemplo_2/main_C.py: exemplo utilizando multiprocessamento
  - ./exemplo_2/main_D.py: exemplo utilizando multiprocessamento e podendo receber uma lista de tarefas

Celery é uma ferramenta que possibilita a criação de filas de tarefas (*task queues*), que é um mecanismo que permite distribuir tais tarefas entre *threads* ou processos diferentes. Ou seja, a execução dessas tarefas ocorre de forma independente do nosso programa e, dessa forma, podemos mandar o computador executar uma operação custosa pelo Celery em, por exemplo, um banco de dados, enquanto o nosso programa prossegue com o roteiro.

→ Instalação Celery: pip install celery

→ Iniciar o Celery: celery -A <nome_arquivo> worker -l info –pool=solo

Broker: o Celery precisa de pelo menos um mediador (*broker*) onde informações sobre as tarefas serão enfileiradas. Exemplos: RabbitMQ, Redis, Amazon SQS e Zookeeper.

→ Instalação Redis: pip install redis

→ Iniciar o Redis no docker: docker run -d -p 6379:6379 redis

Flower: ferramenta web para monitorar e administrar clusters Celery.

→ Instalação Flower: pip install flower

→ Iniciar o Flower: celery -A <nome_arquivo> flower –address=127.0.0.6 –port=5566

Exemplos:

- ./exemplo_3/A:
  - main.py: exemplo sem utilização do Celery.
  - tasks.py: exemplo sem utilização do Celery.
- ./exemplo_3/B:
  - main.py: exemplo com utilização do Celery.
  - tasks.py: exemplo com utilização do Celery.
- ./exemplo_3/C:
  - main.py: exemplo utilizando método Chain.
  - tasks.py: exemplo utilizando método Chain.

Introduzida em 2012, a biblioteca *asyncio* fornece à linguagem recursos para a criação de código concorrente sem precisar recorrer ao uso de múltiplas *threadings*, utilizando *event loops* em seu lugar. O *asyncio* é hoje o módulo padrão em Python para atingir concorrência através de uma única *thread*.
$\rightarrow$ Instalação AsyncIO: pip install asyncio
A sintaxe do Python utiliza duas palavras reservadas:
async: indica que uma função deve ser executada de forma assíncrona.
await: significa que a corrotina será paralisada naquele ponto aguardando um resultado futuro.
Em outras palavras, o controle de execução será dado à outra corrotina e só será retomado quando o resultado ficar pronto.

Exemplos:

- ./exemplo_4/main_A.py: exemplo utilizando sincronidade
- ./exemplo_4/main_B.py: exemplo utilizando asyncio
- ./exemplo_4/main_C.py: exemplo utilizando asyncio, mas reescrito para receber uma lista de tarefas

A biblioteca aiohttp é utilizada para fazer solicitações HTTP assíncronas, que é um dos casos de uso mais comuns do código sem bloqueio.

→ Instalação aiohttp: pip install aiohttp

Exemplo documentação (`https://pypi.org/project/aiohttp`):

```
1  import aiohttp
2  import asyncio
3
4  async def main():
5
6      async with aiohttp.ClientSession() as session:
7          async with session.get('http://python.org') as response:
8
9              print("Status:", response.status)
10             print("Content-type:", response.headers['content-type'])
11
12             html = await response.text()
13             print("Body:", html[:15], "...")
14
15 asyncio.run(main())
16
```

Exemplos:

- ./exemplo_5/main_A.py: exemplo utilizando solicitações síncronas
- ./exemplo_5/main_B.py: exemplo utilizando aiohttp
- ./exemplo_5/main_C.py: exemplo utilizando aiohttp e tarefas do asyncio