#PYTHONTIPS

THREAD VS PROCESS

CHOOSING THE PERFECT CONCURRENCY WEAPON IN PYTHON

@VEDANTSOLANKI

WHAT IS THREAD ?

- A thread is like a separate worker that performs a specific task within a larger process.
- Threads allow multiple tasks to be executed concurrently within a process, sharing the same memory space.

ANALOGY

- Let's understand thread better with the help of an analogy , Team Members in a Project
- Imagine you're working on a project with a team. Each team member focuses on a specific task, and all team members work together simultaneously to accomplish the project goals.
- This parallel execution resembles how threads operate within a process.

WHAT IS PROCESS ?

- A process is like an independent instance of a program that runs on the operating system.
- It has its own memory space and resources. Processes enable multiple programs or tasks to run concurrently and independently of each other.

ANALOGY

- Let's understand process better with the help of an analogy , Separate Companies
- Think of processes as separate companies working on different projects. Each company has its own employees, resources, and workspace. They can work independently without directly affecting each other.
- This represents how processes operate independently in the operating system.

#PYTHONTIPS

INTERNAL WORKING OF THREADS IN PYTHON

- Threads in Python are managed by the interpreter's thread module. The Global Interpreter Lock (GIL) allows only one thread to execute Python bytecode at a time.
- This means that although threads can run concurrently, they do not achieve true parallelism for CPU-bound tasks.
- However, they can provide concurrency for I/O-bound tasks since the GIL is released during I/O operations.

- Threads share the same memory space (heap) within a process, which can lead to potential data race conditions if proper synchronization mechanisms are not used.

#PYTHONTIPS

INTERNAL WORKING OF PROCESSES IN PYTHON

- Processes in Python are managed by the multiprocessing module.
- Unlike threads, processes run in separate memory spaces.
- This allows them to bypass the GIL and achieve true parallelism for CPU-bound tasks.
- Each process has its own interpreter, memory, and resources. Communication between processes can be done through inter-process communication (IPC) mechanisms like pipes, queues, and shared memory.

WHEN TO USE THREADS ?

- Threads are suitable for scenarios where tasks are I/O-bound, such as downloading files, making HTTP requests, or performing database operations.
- Since threads can release the GIL during I/O operations, they can achieve concurrency and improve performance.
- They are also beneficial for tasks that involve a large number of relatively small and independent operations.

WHEN TO USE PROCESSES ?

- Processes are ideal for CPU-bound tasks that require intensive computational work, such as data processing, scientific calculations, or simulations.
- Since processes operate in separate memory spaces and bypass the GIL, they can achieve true parallelism and utilize multiple CPU cores effectively.
- Processes are also useful when data needs to be isolated, ensuring that one process doesn't affect the execution of another.

#PYTHONTIPS

THREAD EXAMPLE

threading.py

```python
import threading

def fibonacci(n):
    fib_sequence = [0, 1]
    for i in range(2, n):
        fib_sequence.append(fib_sequence[i-1] + fib_sequence[i-2])
    print(f"Fibonacci sequence: {fib_sequence}")

# Create and start the thread
thread = threading.Thread(target=fibonacci, args=(10,))
thread.start()

print("Main thread continues executing...")

# Wait for the thread to finish
thread.join()

print("Main thread exiting.")


'''

output :

Main thread continues executing...
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Main thread exiting.

'''
```

@VEDANTSOLANKI

- In above example, the main thread starts the Fibonacci calculation in the background using a separate thread.
- While the Fibonacci sequence is being calculated, the main thread continues executing and prints the message "Main thread continues executing...".
- Once the Fibonacci calculation is complete, the sequence is printed, and the main thread exits, printing the message "Main thread exiting.".

#PYTHONTIPS

PROCESS EXAMPLE

processing.py

```python
import multiprocessing
from PIL import Image

def process_image(image_path):
    image = Image.open(image_path)
    # Perform image processing operations
    # ...

    processed_image_path = f"processed_{image_path}"
    image.save(processed_image_path)
    print(f"Image processed: {processed_image_path}")

if __name__ == '__main__':
    # List of image file paths
    image_files = ['image1.jpg', 'image2.jpg', 'image3.jpg', 'image4.jpg']

    # Create and start the processes
    processes = []
    for image_file in image_files:
        process = multiprocessing.Process(target=process_image, args=(image_file,))
        process.start()
        processes.append(process)

    print("Main process continues executing...")

    # Wait for all processes to finish
    for process in processes:
        process.join()

    print("Main process exiting.")


'''

output :

Main process continues executing...
Image processed: processed_image1.jpg
Image processed: processed_image2.jpg
Image processed: processed_image3.jpg
Image processed: processed_image4.jpg
Main process exiting.

'''
```

@VEDANTSOLANKI

- In above example, the main process creates multiple processes to perform image processing on different image files concurrently.
- While the image processing is happening in the background, the main process continues executing and prints the message "Main process continues executing...".
- Once all image processing is completed, the processed image paths are printed, and the main process exits, printing the message "Main process exiting.".

SUMMARY

- Threads represent sequences of instructions within a process.
- Threads share the same memory space within a process.
- Python threads are affected by the Global Interpreter Lock (GIL), limiting true parallelism for CPU-bound tasks but allowing concurrency for I/O-bound tasks.
- Threads are lightweight and have low overhead, making them suitable for a large number of small tasks.
- Threads can be used for I/O-bound tasks and scenarios with a large number of small and independent operations.

SUMMARY

- Processes are independent instances of programs running concurrently.
- Processes have separate memory spaces.
- Python processes, managed by the multiprocessing module, bypass the GIL , have higher overhead due to memory and resource isolation but enable true parallelism.
- Processes are recommended for CPU-bound tasks and situations where data isolation is required.

@TAG

SOMEONE WHO WILL FIND THIS HELPFUL

FOLLOW FOR MORE !

Vedant Solanki

@vedantsolanki