Multiprocessing VS Threading VS AsyncIO in Python

07-11-2020 BLOG 11 MINUTES READ (ABOUT 1661 WORDS) 2787 VISITS

Introduction

In modern computer programming, concurrency is often required to accelerate solving a problem. In Python programming, we usually have the three library options to achieve concurrency, `multiprocessing`, `threading`, and `asyncio`. Recently, I was aware that as a scripting language Python's behavior of concurrency has subtle differences compared to the conventional compiled languages such as C/C++.

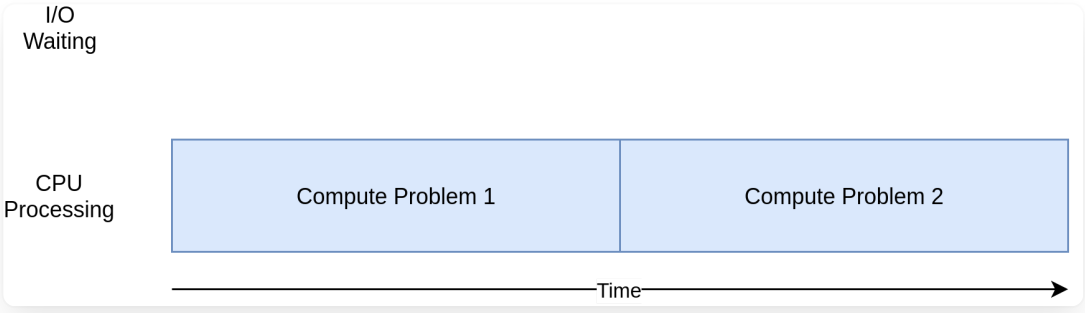
Real Python has already given a good tutorial with code examples on Python concurrency. In this blog post, I would like to discuss the `multiprocessing`, `threading`, and `asyncio` in Python from a high-level, with some additional caveats that the Real Python tutorial has not mentioned. I have also borrowed some good diagrams from their tutorial and the readers should give the credits to them on those illustrations.

CPU-Bound VS I/O-Bound

The problems that our modern computers are trying to solve could be generally categorized as CPU-bound or I/O-bound. Whether the problem is CPU-bound or I/O-bound affects our selection from the concurrency libraries `multiprocessing`, `threading`, and `asyncio`. Of course, in some scenarios, the algorithm design for solving the problem might change the problem from CPU-bound to I/O-bound or vice versa. The concept of CPU-bound and I/O-bound are universal for all programming languages.

CPU-Bound

CPU-bound refers to a condition when the time for it to complete the task is determined principally by the speed of the central processor. The faster clock-rate CPU we have, the higher performance of our program will have.

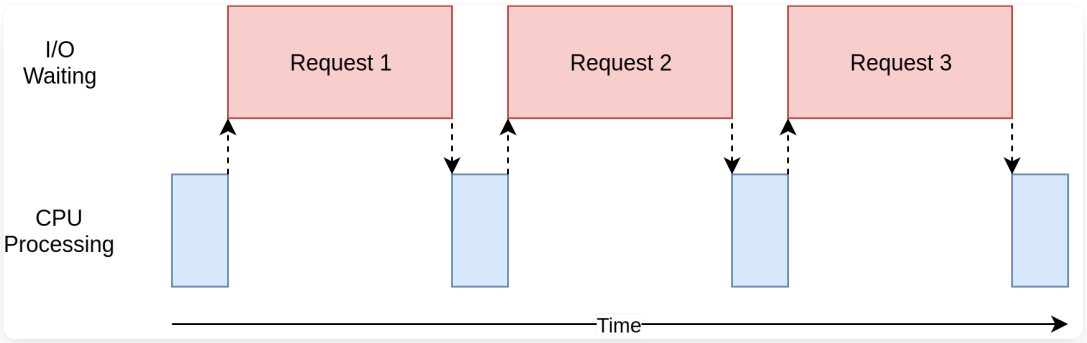


Single-Process Single-Thread Synchronous for CPU-Bound

Most of single computer programs are CPU-bound. For example, given a list of numbers, computing the sum of all the numbers in the list.

I/O-Bound

I/O bound refers to a condition when the time it takes to complete a computation is determined principally by the period spent waiting for input/output operations to be completed. This is the opposite of a task being CPU bound. Increasing CPU clock-rate will not increase the performance of our program significantly. On the contrary, if we have faster I/O, such as faster memory I/O, hard drive I/O, network I/O, etc, the performance of our program will boost.



Single-Process Single-Thread Synchronous for I/O-Bound

Most of the web service related programs are I/O-bound. For example, given a list of restaurant names, finding out their ratings on Yelp.

Process VS Thread in Python

Process in Python

A global interpreter lock (GIL) is a mechanism used in computer-language interpreters to synchronize the execution of threads so that only one native thread can execute at a time. An interpreter that uses GIL always allows exactly one native thread to execute at a time, even if run on a multi-core processor. Note that the native thread here is the number of threads in the physical CPU core, instead of the thread concept in the programming languages.

Because Python interpreter uses GIL, a single-process Python program could only use one native thread during execution. That means single-process Python program could not utilize CPU more than 100% (we define the full utilization of a native thread to be 100%) regardless whether it is single-process single-thread or single-process multi-thread. Conventional compiled programming languages, such as C/C++, do not have interpreter, not even mention GIL. Therefore, for a single-process multi-thread C/C++ program, it could utilize many CPU cores and many native threads, and the CPU utilization could be greater than 100%.

Therefore, for a CPU-bound task in Python, we would have to write multi-process Python program to maximize its performance.

Thread in Python

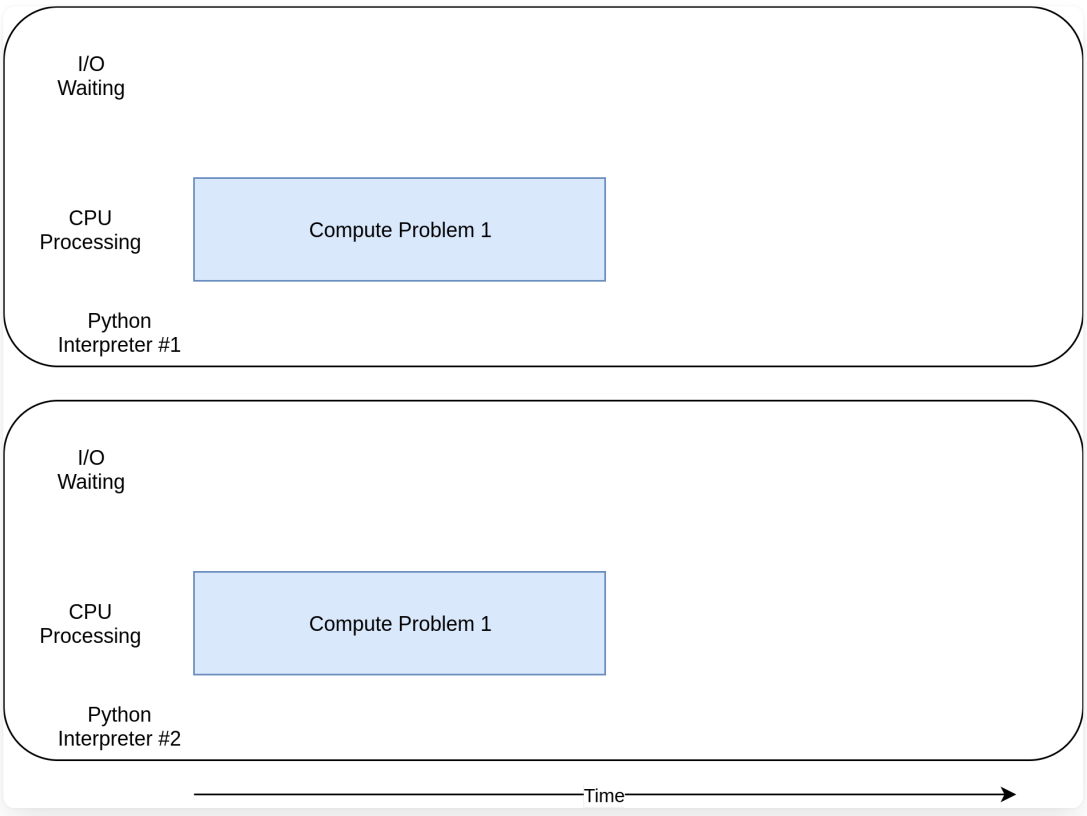
Because a single-process Python could only use one CPU native thread. No matter how many threads were used in a single-process Python program, a single-process multi-thread Python program could only achieve at most 100% CPU utilization.

Therefore, for a CPU-bound task in Python, single-process multi-thread Python program would not improve the performance. However, this does not mean multi-thread is useless in Python. For a I/O-bound task in Python, multi-thread could be used to improve the program performance.

Multiprocessing VS Threading VS AsyncIO in Python

Multiprocessing

Using Python `multiprocessing`, we are able to run a Python using multiple processes. In principle, a multi-process Python program could fully utilize all the CPU cores and native threads available, by creating multiple Python interpreters on many native threads. Because all the processes are independent to each other, and they don't share memory. To do collaborative tasks in Python using `multiprocessing`, it requires to use the API provided the operating system. Therefore, there will be slightly large overhead.

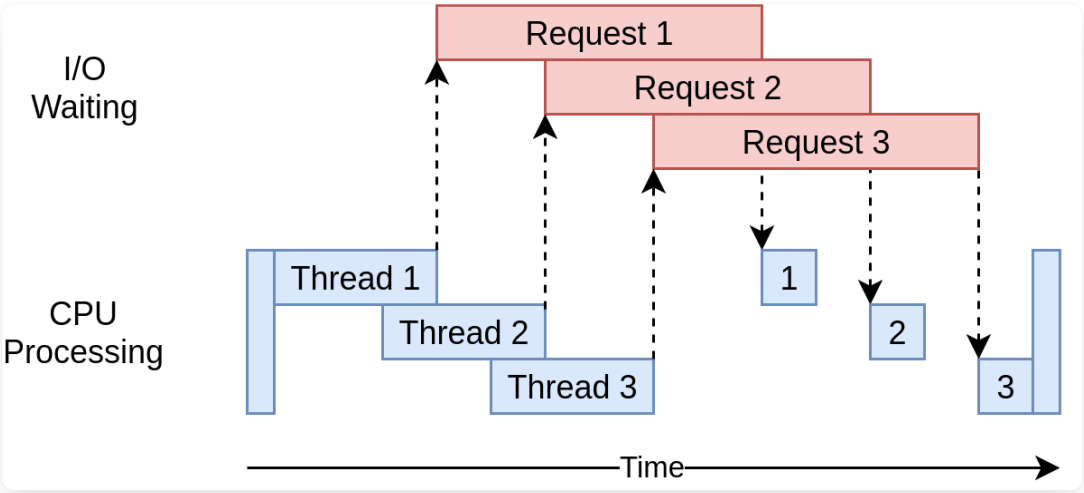


Multi-Process for CPU-Bound

Therefore, for a CPU-bound task in Python, `multiprocessing` would be a perfect library to use to maximize the performance.

Threading

Using Python `threading`, we are able to make better use of the CPU sitting idle when waiting for the I/O. By overlapping the waiting time for requests, we are able to improve the performance. In addition, because all the threads share the same memory, to do collaborative tasks in Python using `threading`, we would have to be careful and use locks when necessary. Lock and unlock make sure that only one thread could write to memory at one time, but this will also introduce some overhead. Note that the threads we discussed here are different to the native threads in CPU core. The number of native threads in CPU core is usually 2 nowadays, but the number of threads in a single-process Python program could be much larger than 2.



Single-Process Multi-Thread for I/O-Bound

Therefore, for a I/O-bound task in Python, `threading` could be a good library candidate to use to maximize the performance.

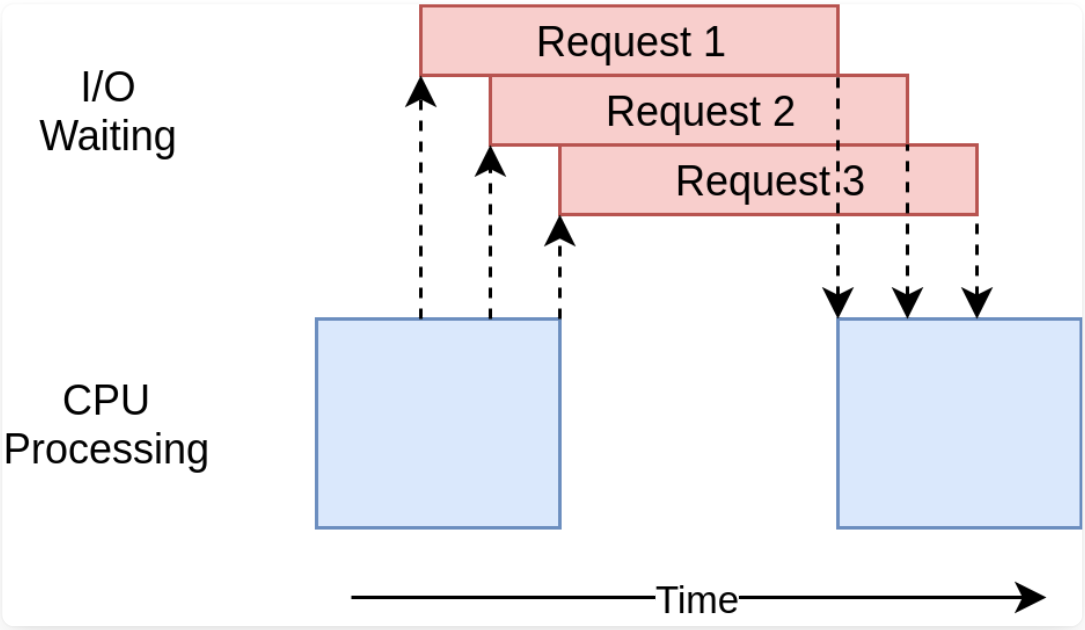
It should also be noted that all the threads are in a pool and there is an executer from the operating system managing the threads deciding who to run and when to run. This can be a short-coming of `threading` because the operating system actually knows about each thread and can interrupt it at any time to start running a different thread. This is called pre-emptive multitasking since the operating system can pre-empt your thread to make the switch.

AsyncIO

Given `threading` is using multi-thread to maximize the performance of a I/O-bound task in Python, we wonder if using multi-thread is necessary. The answer is no, if you know when to switch the tasks. For example, for each thread in a Python program using `threading`, it will really stay idle between the request is sent and the result is returned. If somehow a thread could know the time I/O request has been sent, it could switch to do another task, without staying idle, and one thread should be sufficient

to manage all these tasks. Without the thread management overhead, the execution should be faster for a I/O-bound task. Obviously, `threading` could not do it, but we have `asyncio`.

Using Python `asyncio`, we are also able to make better use of the CPU sitting idle when waiting for the I/O. What's different to `threading` is that, `asyncio` is single-process and single-thread. There is an event loop in `asyncio` which routinely measure the progress of the tasks. If the event loop has measured any progress, it would schedule another task for execution, therefore, minimizing the time spent on waiting I/O. This is also called cooperative multitasking. The tasks must cooperate by announcing when they are ready to be switched out.



Single-Process Single-Thread Asynchronous for I/O-Bound

The short-coming of `asyncio` is that the even loop would not know what are the progresses if we don't tell it. This requires some additional effort when we write the programs using `asyncio`.

Summary

Concurrency Type	Features	Use Criteria	Metaphor
Multiprocessing	Multiple processes, high CPU utilization.	CPU-bound	We have ten kitchens, ten chefs, ten dishes to cook.

Concurrency Type	Features	Use Criteria	Metaphor
Threading	Single process, multiple threads, pre-emptive multitasking, OS decides task switching.	Fast I/O-bound	We have one kitchen, ten chefs, ten dishes to cook. The kitchen is crowded when the ten chefs are present together.
AsyncIO	Single process, single thread, cooperative multitasking, tasks cooperatively decide switching.	Slow I/O-bound	We have one kitchen, one chef, ten dishes to cook.

Caveats

HTOP vs TOP

`htop` would sometimes misinterpret multi-thread Python programs as multi-process programs, as it would show multiple `PID`s for the Python program. `top` does not have this problem. On StackOverflow, there is also such a observation.

References

- Speed Up Your Python Program With Concurrency
- Async Python: The Different Forms of Concurrency