Asynchronous Programming

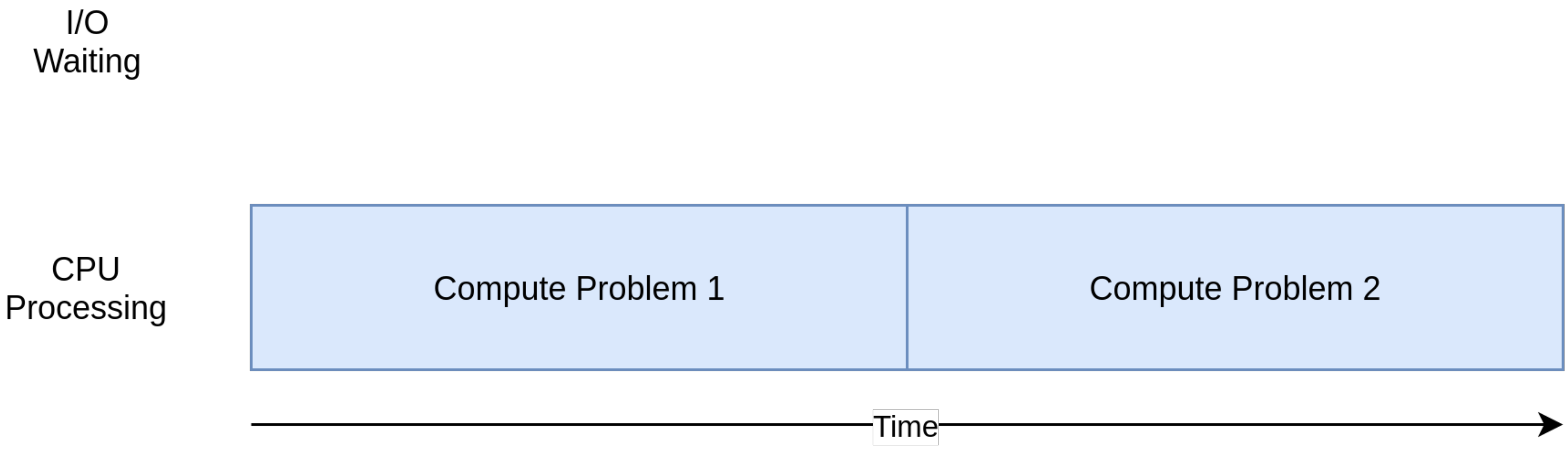
Asynchronous Programing Development
Python

- Threading
- Multiprocessing
- Asynchronous
  - Event Loop
  - Task

CPU-Bound

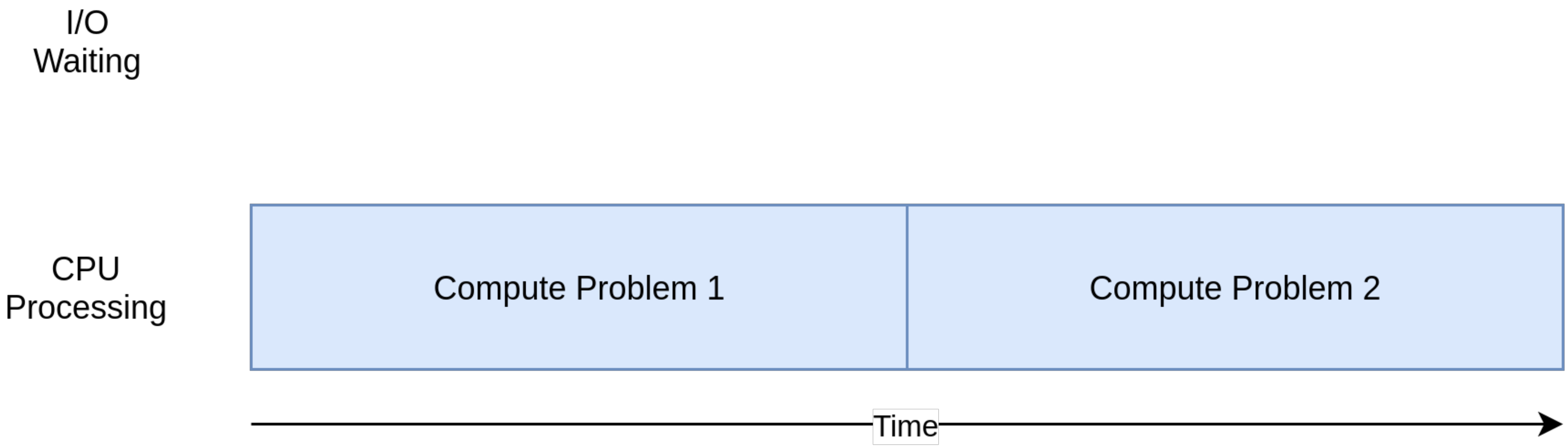
Single-Process Single-Thread Synchronous for CPU-Bound

- 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.



CPU-Bound

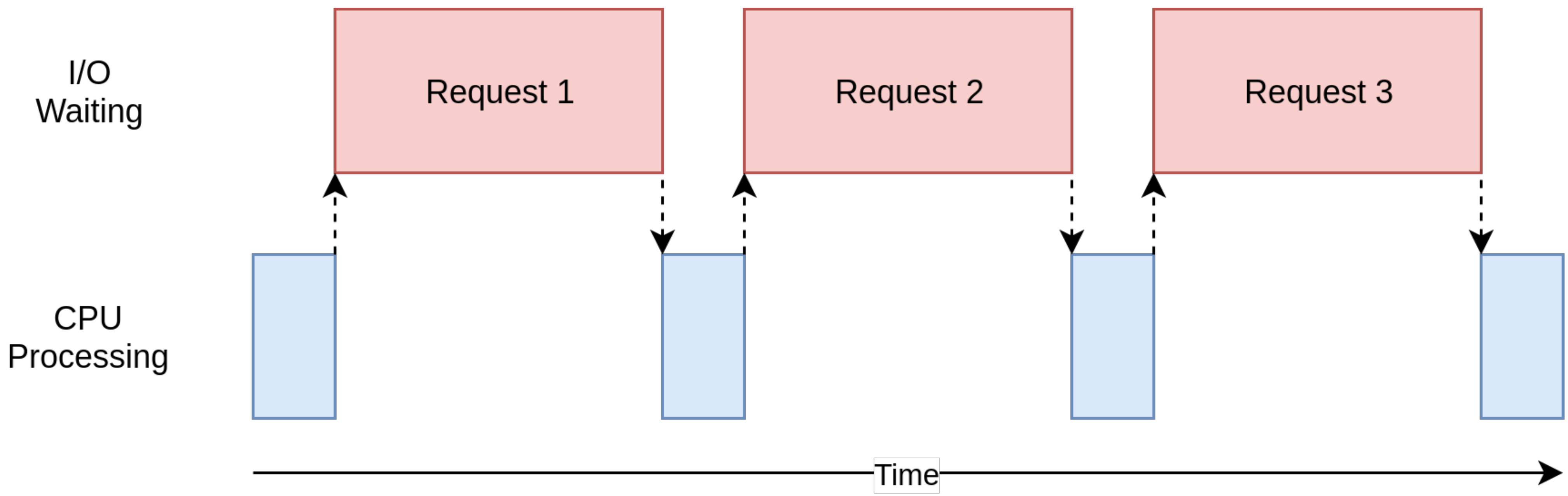
- Most of single computer programs are CPU-bound. For example, given a list of numbers, computing the sum of all the numbers in the lis



IO-Bound

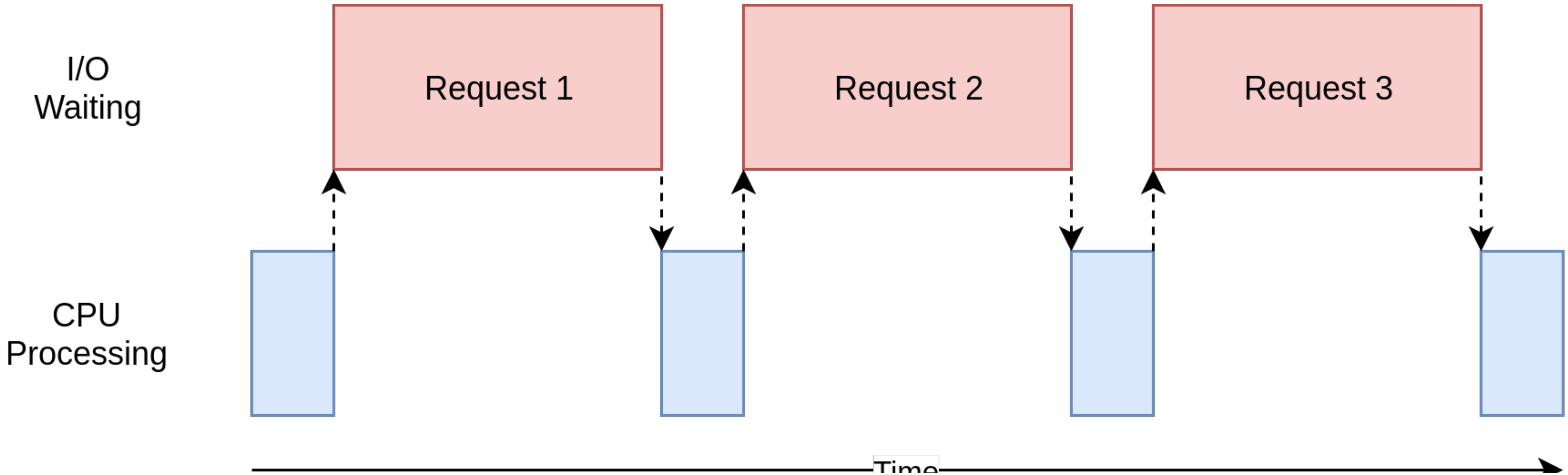
Single-Process Single-Thread Synchronous for I/O Bound

- 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



IO-Bound

- 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.



Process VX Thread

Process in Python

- Global interpreter Lock (GIL)
  a mechanism used in computer-language **interpreters** to synchronize the execution of threads so that only one native thread can execute at a time

- Python 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%

- C/C++
  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%

Process in Python

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

Thread in Python

- 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
- 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

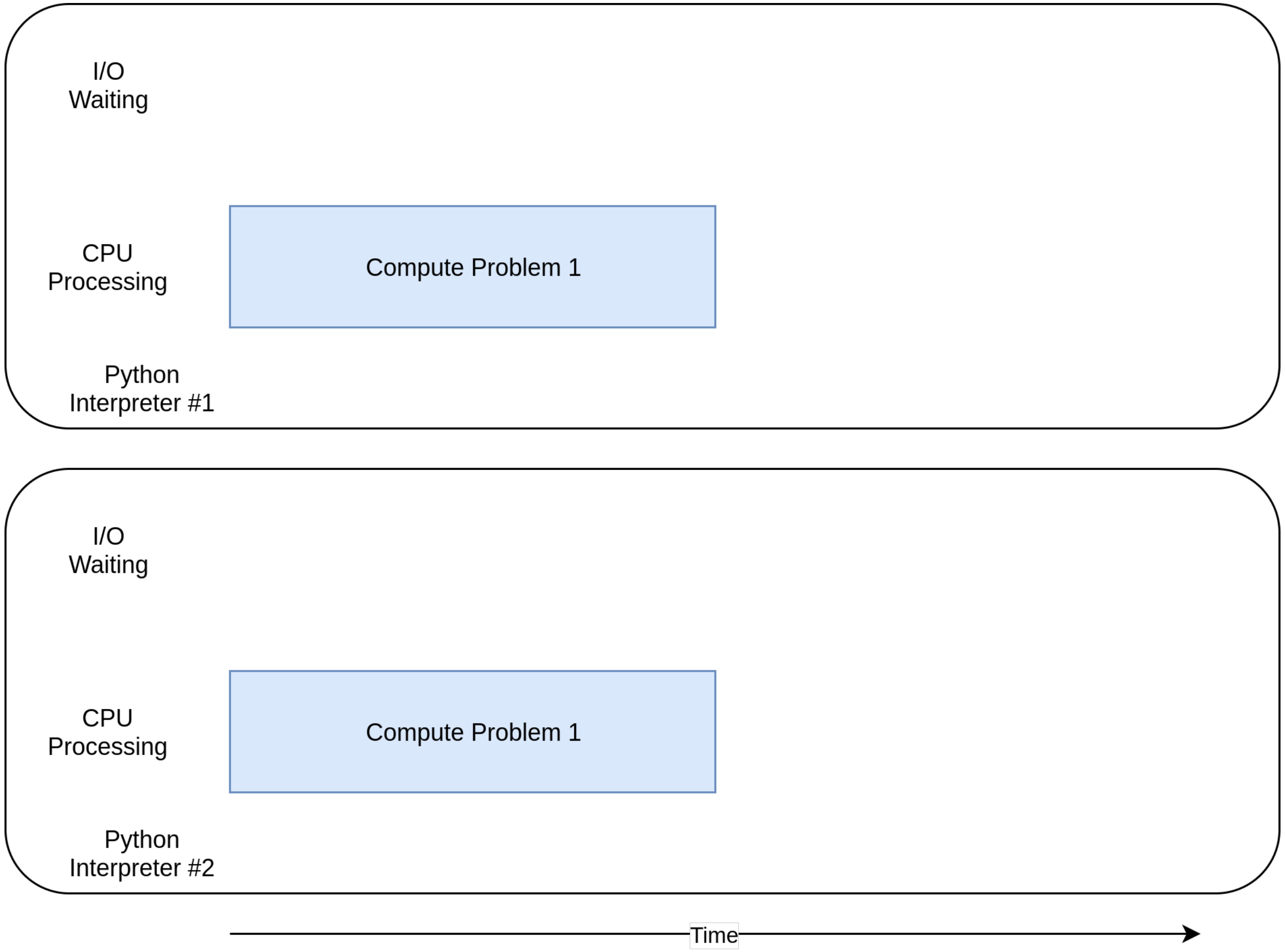
Multiprocessing VS Threading VS AsyncIO

Multiprocessing

- Using Python **multiprocessing**, we are able to run a Python using multiple processes
- 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
- 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

Multiprocessing

Multi-Process for CPU-Bound



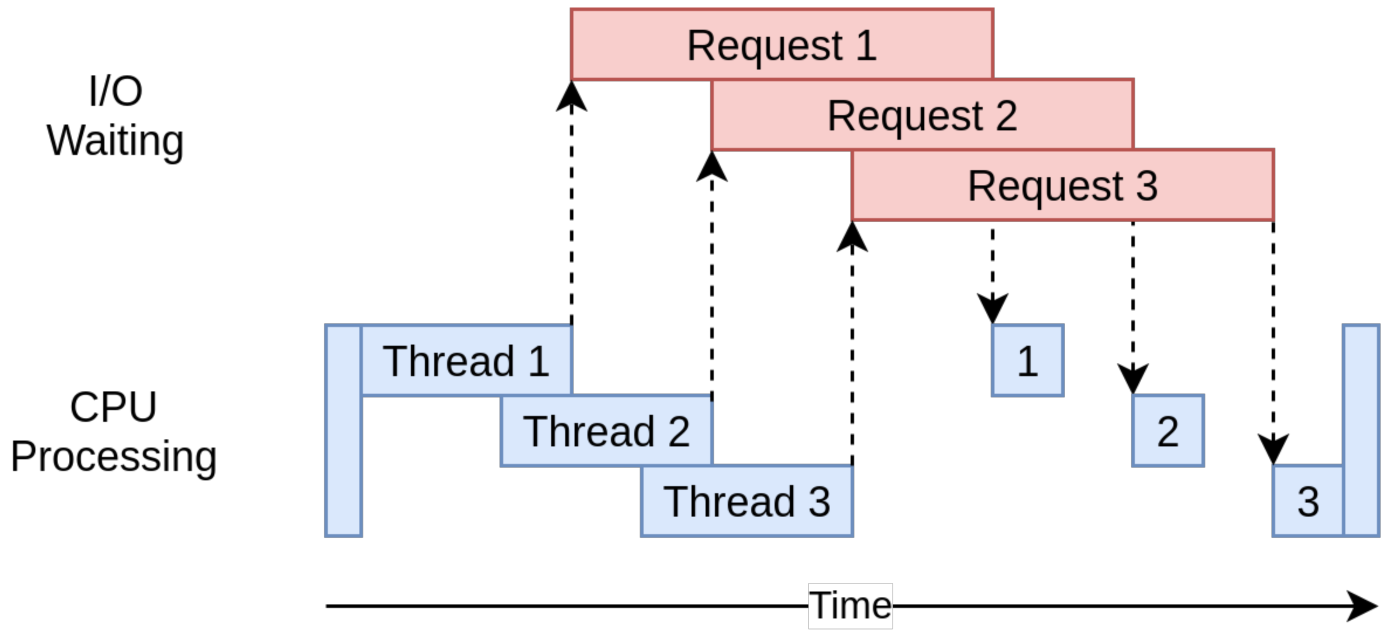
Threading

- 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
- All the threads **share** the same memory
- 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
- 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

Threading

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

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



Threading

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

- 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
- 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

Asynchronous IO

- If you know when to switch the tasks.
- 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**

Asynchronous IO

- Using Python asyncio, we are also able to make better use of the CPU sitting **idle** when **waiting** for the I/O, 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
- it would schedule another task for execution, therefore, minimizing the time spent on waiting I/O.
- called **cooperative** multitasking, The tasks must cooperate by announcing when they are ready to be **switched** out

Asynchronous IO

- 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.

Sumary

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

Labs

- https://krondo.com/an-introduction-to-asynchronous-programming-and-twisted/
- https://realpython.com/python-concurrency/