Extra Notes on Concurrency

A computer program is executed by needing at least one thread to run on.

A **coroutine** lives in a thread.

A **thread** lives in process and is the smallest execution unit

A **process** lives in a core and is the smallest resource management unit.

Different processes share different resources.

A **core** lives in a CPU.

Multiprocessing usually refers to many processes executed in parallel.

Multithreading usually refers to many threads executing concurrently. Threads can use idle cores to run in parallel.

Coroutine usually refers to many routines executed concurrently.

Concurrent and Parallel

Many threads live in a process, and only one thread can own the process.

When they own the process, they can run, and they take turns to own the core.

Concurrent: Thread A and Thread B are in the same process, taking turns to own the process, yield when waiting for resources or scheduled CPU time is used up.

Use case is dealing with I/O intensive tasks.

Parallel: Thread A and Thread B are in different processes, execute at the same.

Use case is dealing with CPU (data) intensive tasks.

CPU (Data) Intensive Tasks

CPU (data) intensive tasks usually use multiprocessing and multithreading, as a normal task won't wait for some external data. It needs the CPU to do the computation for most of the time.

But which one to choose depends on different kinds of computing tasks.

Multithreading vs Multiprocessing

A thread is lighter than a process.

Lighter means easy to create and destroy, occupying less resources.

Threads share the same address space and heap but each thread has its own stack.

- Communication between threads is faster.
- Multiprocessing is more reliable. If one process crashes, it won't affect other processes. In contrast, if one thread crashes, the process it belongs to will also

crash as well as other threads in the same process become dead.

Multithreading is good for **cooperative** tasks.

If the tasks are highly related and need to frequently create/destroy sub-tasks (like running an O(n) parallel BFS algorithm), then multithreading is preferred.

Multiprocessing is good for **isolated** tasks.

This includes computing a page-rank from 3 different sources in parallel.

- Other Examples: Spark, Hadoop, Distributed Computing.

I/O Intensive Tasks

For these, you usually use multithreading and a coroutine since a normal task spends most of its time waiting for I/O (like Disk, Network, RAM). An example would be **web servers**.

Coroutine vs Multithreading

Coroutines are one of the reasons why Python is so popular to write web servers.

They run concurrently in a thread just like threads run concurrently in a process, but:

- Concurrent execution needs context-switching and context-switching by coroutine is extremely light, which means much cheaper, faster than multithreading
- Coroutine is managed by the user, multi-threading is managed by the kernel. Developers have better control of the execution flow by using a coroutine (for example, a coroutine won't be forced to yield).
- Coroutines cannot run in parallel.

Combination

You can also use a coroutine and reimplement key module with C to avoid GIL.

Multi-core CPU vs Multi-CPU

In the same CPU, there can be multiple cores.

Different cores work together and share resources.

Across different CPUs there could also be multiple cores, but usually not a good practice because some resources are not shared, getting resources owned by other CPUs is expensive.

- For the same amount of cores, we usually have a much better performance gain by putting them in the same CPU. Now for commercial CPUs, 16 cores is pretty common. Multi-CPU is usually good for running isolated programs (not sharing any data).