Multiprocessing vs Multithreading vs Async IO

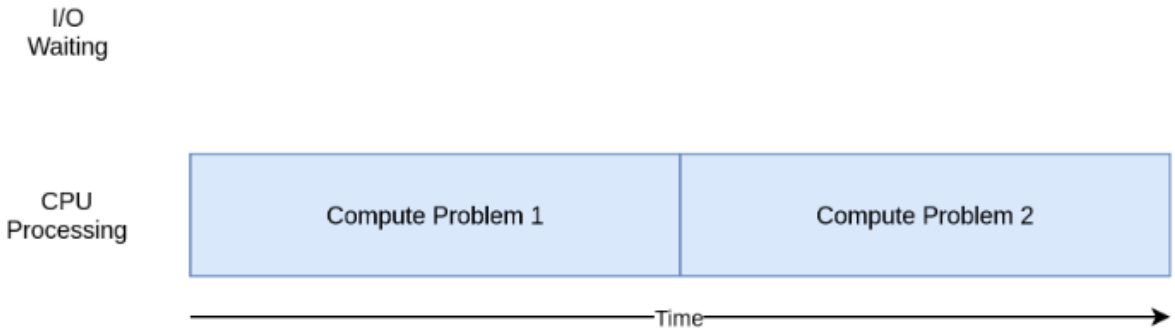
Parallelism

1. It means performing multiple operations at the same time or simultaneously.
2. It requires the use of multiple cores. What it means is that multiple operations will be running at the same point in time on different cores.

Multiprocessing

1. It is a means to parallelism.
2. It means spreading tasks over CPUs, or cores.
3. When is it useful? - It is well-suited for CPU-bound tasks. Tightly bound for loops and mathematical computations usually fall into this category.

Here's a corresponding diagram for a CPU-bound program:



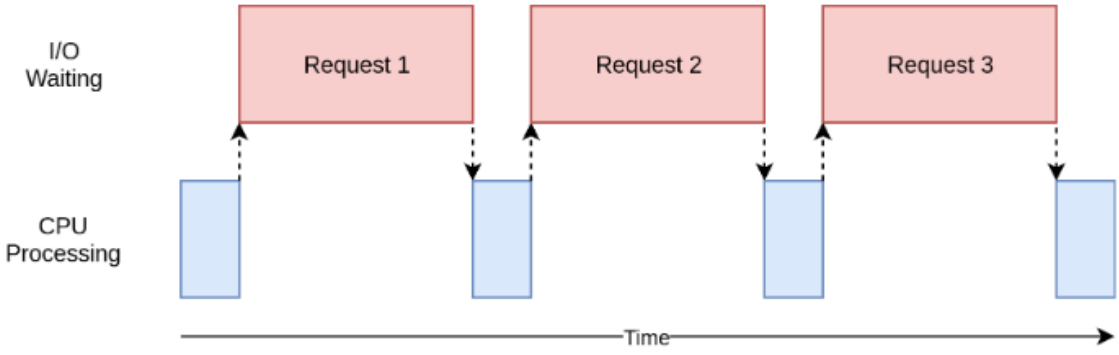
Concurrency

1. It is a slightly broader term than parallelism. It suggests that multiple tasks have the ability to run in an overlapping manner.
2. Concurrency does not imply parallelism.
3. We can say that parallelism is a special type of concurrency where tasks are executed simultaneously.

Multithreading

1. It is a means to concurrency.
2. In this multiple threads take turns executing tasks.
3. Multithreading runs on a single processor.
4. It's better for IO-bound tasks. An IO-bound job is dominated by a lot of waiting on input/output to complete. E.g interactions with the file system and network connections.

Let's see what that looks like:



5. In multithreading, 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.
6. Python has a complicated relationship with threading thanks to its GIL.
    a. GIL is a lock that allows one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time.
    b. The GIL does not have much impact on the performance of I/O-bound multi-threaded programs as the lock is released by a thread while it's waiting for I/O.
7. Pre-emptive multitasking
    a. Pro - Code doesn’t need to do anything to make the switch.
    b. Con - This switch can happen anytime even in the middle of a single Python statement, even a trivial one like `x = x + 1`.

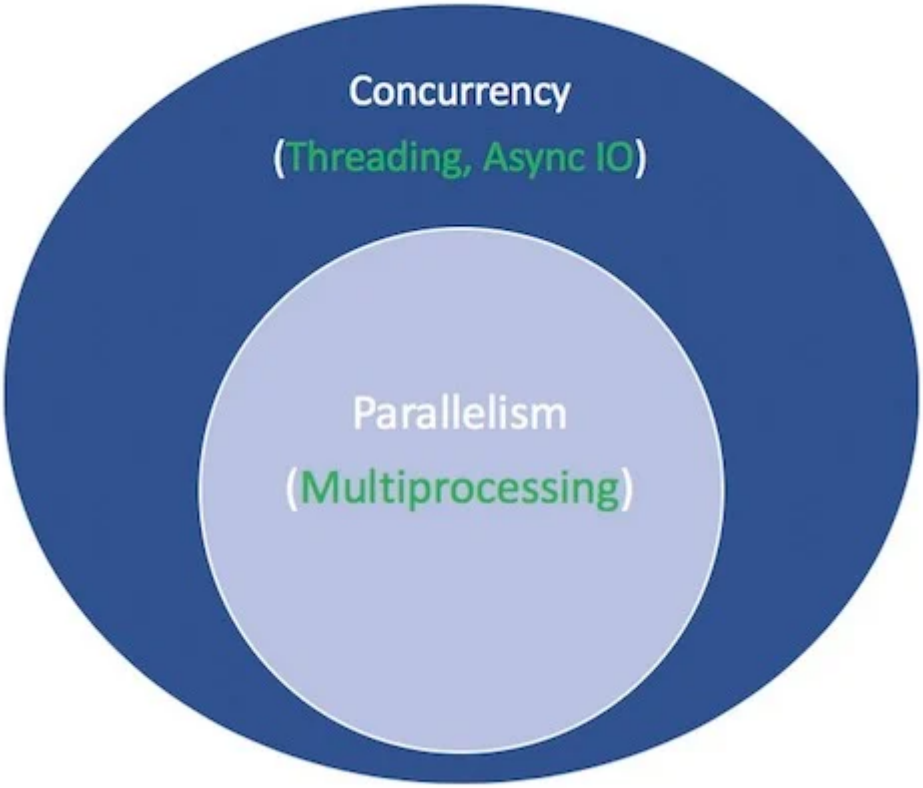
Async IO

1. It is a means to concurrency.
2. It is a language-agnostic programming paradigm. Async IO is not a newly invented concept, and it has existed or is being built into other languages and runtime environments, such as Go, C#, or Scala, Js.
3. Async IO is not multithreading, nor is it multiprocessing. It is not built on top of either of these.
4. It is a single-threaded, single-process design. Intuitively async IO gives a feeling of concurrency despite using a single thread in a single process. It just cleverly find ways to take turns to speed up the overall processing.
5. It uses **cooperative multitasking** - The tasks must cooperate by announcing when they are ready to be switched out. Your code decides when to give up control.
    a. Pro - You always know where your task will be swapped out. It will not be swapped out in the middle of a Python statement. You don’t have to worry about making your code thread-safe.
    b. Con - The code has to change slightly to make this happen.

6. It’s also better for IO-bound tasks. It takes long waiting periods in which functions would otherwise be blocking and allows other functions to run during that downtime.

Summary

Here’s a diagram to put it all together. The white terms represent concepts, and the green terms represent ways in which they are implemented or effected:



Coroutines

1. A coroutine is a specialized version of a Python generators.
2. A coroutine is a function that can suspend its execution before reaching return, and it can indirectly pass control to another coroutine for some time.

```
def jumping_range(up_to):
    index = 0
    while index < up_to:
        jump = yield index
        if jump is None:
            jump = 1
```

```
        index += jump


if __name__ == '__main__':
    iterator = jumping_range(5)
    print(next(iterator))
    print(iterator.send(2))
    print(next(iterator))
    print(iterator.send(-1))
    for x in iterator:
        print(x)
```

```
0
2
3
2
3
4
```

3. Coroutines are not asynchronous/concurrent on their own. Coroutines are suspendable, and the possibility to suspend and resume computation allows them to be effectively used at concurrent and finally asynchronous programs.

Event Loop

1. You can think of an event loop as something like a while True loop that monitors coroutines, taking feedback on what’s idle, and looking around for things that can be executed in the meantime. It is able to wake up an idle coroutine when whatever that coroutine is waiting on becomes available.
2. Coroutines don’t do much on their own until they are tied to the event loop.
3. By default, an async IO event loop runs in a single thread and on a single CPU core. It is also possible to run event loops across multiple cores.
4. Event loops are pluggable. That is, you could, if you really wanted, write your own event loop implementation and have it run tasks just the same.The asyncio package itself ships with two different event loop implementations.
5. Basically an event loop lets you go, "when A happens, do B". Probably the easiest example to explain this is that of the JavaScript event loop that's in every browser. Whenever you click something ("when A happens"), the click is given to the JavaScript event loop which checks if any onclick callback was registered to

handle that click ("do B"). If any callbacks were registered then the callback is called with the details of the click. The event loop is considered a loop because it is constantly collecting events and loops over them to find what to do with the event.

Asyncio (The Python library)

1. The asyncio package is a python library to write Async IO/concurrent code.
2. The general concept of asyncio is that a single Python object, called the event loop, controls how and when each task gets run. The event loop is aware of each task and knows what state it’s in.
3. An important point of asyncio is that the tasks never give up control without intentionally doing so. They never get interrupted in the middle of an operation. This allows us to share resources a bit more easily in asyncio than in threading. You don’t have to worry about making your code thread-safe.
4. There are other python framework's also that implements Async IO. Some other libs that provides Async IO APIs are
    a. https://github.com/dabeaz/curio
    b. https://github.com/python-trio/trio
5. Also there’s a common argument that having to add async and await in the proper locations is an extra complication. To a small extent, that is true. The flip side of this argument is that it forces you to think about when a given task will get swapped out, which can help you create a better, faster, design.

```
import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    count()
    count()
    count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
```

```
$ python3 countasync.py
One
Two
One
Two
One
Two
executed in 3.0s.
```

```
    elapsed = time.perf_counter() - s
    print(f"executed in {elapsed:0.2f}s.")
```

```
import asyncio
import time

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"executed in {elapsed:0.2f}s.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
executed in 1.01s.
```

The order of this output is the heart of async IO. Talking to each of the calls to count() is a single event loop, or coordinator. When each task reaches await asyncio.sleep(1), the function yells up to the event loop and gives control back to it, saying, “I’m going to be sleeping for 1 second. Go ahead and let something else meaningful be done in the meantime.”

Resources

1. https://realpython.com/async-io-python/
2. https://realpython.com/python-concurrency/
3. https://realpython.com/python-gil/
4. https://towardsdatascience.com/cpython-internals-how-do-generators-work-ba1c4405b4bc
5. https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/
6. https://sookocheff.com/post/concurrency/concurrency-a-primer/
7. https://gist.github.com/maxfischer2781/27d68e69c017d7c2605074a59ada04e5
8. https://stackoverflow.com/questions/49005651/how-does-asyncio-actually-work

9. https://stackoverflow.com/questions/9708902/in-practice-what-are-the-main-uses-for-the-new-yield-from-syntax-in-python-3
10. https://www.python.org/dev/peps/pep-0380/#formal-semantics
11. http://www.dabeaz.com/coroutines/Coroutines.pdf
12. https://gist.github.com/maxfischer2781/27d68e69c017d7c2605074a59ada04e5