Goutom Roy Follow

Jul 22, 2019 · 5 min read · ✨ · Listen

Parallelism

Implementing in Python.

After releasing Python 3 we are hearing a lot about `async` and `concurrency` which can be achieved by `asyncio` module.But there are other ways to achieve asynchronous capability in python also, these are `Threads` and `Processes` .

Lets discuss basic terms we will use in this article.

Sync

In Synchronous operations, if you start more than one task, the tasks will be executed in sync, one after one.

Async

In asynchronous operations, if you start more than one task, all the tasks will be started immediately but will complete independently of one another. An async task may start and continue running while the execution may move to a new task. First task will wait for completion of second task, after completing second task, first one will be resumed to complete.

Concurrency and Parallelism

Concurrency and parallelism are philosophical words, the ways how tasks will be executed. On the other hand synchronous and asynchronous concepts are programming model.

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously.

Parallelism means executing multiple tasks at the same time simultaneously. Parallelism is hardware dependent. why? In a computer with single core processor, only one task is said to be running at any point of time. So if you want to achieve parallelism, you need multi core processor.

- ***In a single core environment, concurrency happens*** with tasks executing over same time period via context switching i.e at a particular time period, only a single task gets executed.
- ***In a multi-core environmen… …ved via parallelism*** in which multiple tasks are executed simultaneously.

Understanding by Real World Example

Your boss told you to buy a air ticket and confirm him by email.There are two tasks : buy ticket and confirm by email.

`Sync` : You called to airline agency and asked for ticket, agency guy confirms you that ticket is available and he told you to wait for few minutes to book it finally, you keeps waiting. After ticket is finally booked, you write email to your boss.

`Async` : You called to airline agency and asked for ticket, agency guy confirms you that ticket is available and he told you to wait for few minutes to book it finally, at the moment agency guy confirmed you that ticket is available, you start writing email to your boss. You did not wait for book the ticket finally to start writing email. Here both tasks making progress together(concurrently) but by only one person.

`Parallelism` : Previously you were doing both task alone, now you asked for help of one of your colleague. When agency guy confirms that ticket is available, you asked your colleague to start writing email. Here you doing one task and your colleague is doing

other task.This is called `parallelism`. Here both tasks making progress together(concurrently) by two persons.

Threads

Using Python thread you can achieve concurrency but not parallelism because of **Global Interpreter Lock (GIL)** which ensure that only one thread runs at a time. Thread takes advantage of CPU's time-slicing feature of operating system where each task run part of its entire task and then go to waiting state. When first task is in waiting state, second task is assigned to CPU to complete it's part of entire task.

Let's see an example.

```
import time
import random
from threading import Thread

class Worker(Thread):
    def __init__(self, number):
        Thread.__init__(self)
        self._number = number

    def run(self):
        sleep = random.randrange(1, 10)
        time.sleep(sleep)
        print(f"Worker {self._number}, slept for {sleep} seconds")


if __name__ == "__main__":

    for i in range(1, 6):
        task = Worker(i)
        task.start()

    print("Total 5 Threads are queued, let's see when they finish!")

```

thread_test.py hosted with ❤️ by GitHub view raw

Output :

```
Total 5 Threads are queued, let's see when they finish!
Worker 5, slept for 3 seconds
Worker 4, slept for 4 seconds
Worker 2, slept for 5 seconds
Worker 1, slept for 6 seconds
Worker 3, slept for 6 seconds
```

Here 5 threads are making progress together, asynchronously and concurrently.

Processes

To achieve parallelism Python has `multiprocessing` module which is not affected by the `Global Interpreter Lock`. Lets check an example.

```python
import time
import random
from multiprocessing import Process


class Processor(Process):
    def __init__(self, number):
        Process.__init__(self)
        self._number = number

    def run(self):
        sleep = random.randrange(1, 10)
        time.sleep(sleep)
        print(f"Worker {self._number}, slept for {sleep} seconds")


if __name__ == "__main__":

    for i in range(1, 6):
        t = Processor(i)
        t.start()

    print("Total 5 Processes are queued, let's see when they finish!")

```

multiprocessing_test.py hosted with ❤️ by GitHub view raw

Here multiple processes are running on different core of your CPU (assuming you have multiple cores). It's true parallelism!

With the `Pool` class, we can also distribute one function execution across multiple processes for different input values. If we take the example from the official docs:

```python
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

multiprocessing.pool_test.py hosted with ❤️ by GitHub view raw

Output :

```
[1, 4, 9]
```

Here same function is executing with different values in different processes and finally the results are being aggregated in a list. This would allow us to break down heavy computations into smaller parts and run them in parallel for faster calculation.

`concurrent.futures`

The `concurrent.futures` module has `ThreadPoolExecutor` and `ProcessPoolExecutor` classes for achieving async capability. These classes maintain a pool of threads or processes. We submit our tasks to the pool and it runs the tasks in available thread/process. A `Future` object is returned which we can use to query and get the result when the task has completed.

Lets check an example of `ThreadPoolExecutor` :

```python
import concurrent.futures
import urllib.request
from time import sleep

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        if url == 'http://www.cnn.com/':
            sleep(10)
        return conn.read()

if __name__ == "__main__":

    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
        for future in concurrent.futures.as_completed(future_to_url):
            url = future_to_url[future]
            try:
                data = future.result()
            except Exception as exc:
                print(f'{url} generated an exception: {exc}\n')
            else:
                print(f'{url} page size is {len(data)} bytes')

```

thread_pool_executor.py hosted with ❤️ by GitHub view raw

Output :

```
http://some-made-up-domain.com/ generated an exception: <urlopen error
[Errno -2] Name or service not known>

http://www.foxnews.com/ page size is 246819 bytes
http://europe.wsj.com/ page size is 973090 bytes
```

```
http://www.bbc.co.uk/ page size is 349527 bytes
http://www.cnn.com/ page size is 1130261 bytes
```

At the moment any task is being finished, it returns and we are printing result. Check that for `cnn.com` we are sleeping for 10 seconds. For `ProcessPoolExecutor` just replace `ThreadPoolExecutor` with `ProcessPoolExecutor(5)`. Remember that the `ProcessPoolExecutor` uses the `multiprocessing` module and is not affected by the Global Interpreter Lock. I suggest you to run this code to understand properly.

Multiprocessing allocates separate memory and resources for each process/program whereas, in multithreading threads belonging to the same process shares the same memory and resources as that of the process.

Why We Need Asyncio

We have threads and processes to achieve concurrency, then why we need **Asyncio?** Lets identify the problem with an example.

Guess we have three threads **T1**, **T2**, **T3** , each one has I/O operations and other few lines of code to execute. Our operating system gives very small amount of time to each task to use CPU and switches between them until they finishes. Assume **T1** finishes its I/O operation first and without executing other codes interpreter switches to **T2**, which is still waiting for I/O, then interpreter switches to **T3**, its also still waiting, then interpreter moves to **T1** and executes other remaining codes. Did you noticed the problem?

T1 was ready to execute other codes but the interpreter switched between **T2** and **T3**. Wouldn't it been better if interpreter would have been switched to **T1** again to execute the other codes? then switch to **T2** and **T3**.

`asyncio` maintains an event loop and that event loop tracks different I/O events and switches to tasks which are ready and pauses the ones which are waiting on I/O. Thus we don't waste time on tasks which are not ready to run right now. In Thread we don't have control to pause/resume task but `asyncio` gives us pause/resume capacity.

The first advantage compared to multiple threads is that you decide where the scheduler will switch from one task to another, which means that sharing data between

tasks it's safer and easier.

When to Use Which One

- CPU Bound : mathematical computations >Multi Processing
- I/O Bound, Fast I/O, Limited Number of Connections : network get request > Multi Threading
- I/O Bound, Slow I/O, Many connections : lots of frequent File r/w, network file download, DB query > Asyncio

Further Readings:

https://www.youtube.com/watch?v=2h3eWaPx8SA

https://realpython.com/python-concurrency/
https://zetcode.com/python/multiprocessing/

https://realpython.com/intro-to-python-threading/
https://pymotw.com/3/threading/index.html
https://pymotw.com/3/multiprocessing/index.html
https://docs.python.org/3/library/multiprocessing.html
https://pymotw.com/3/concurrent.futures/http://www.dabeaz.com/GIL/

https://callhub.io/understanding-python-gil/

Programming Python Asyncio Threads Process