Multi-threading vs multi-processing

Do you know how many CPU(s) are there in your computer? What is the model name?

```
>>> import platform
>>> platform.processor()
'arm'

>>> platform.system()
'Darwin'

>>> platform.architecture()
('64bit', '')

>>> import multiprocessing
>>> multiprocessing.cpu_count()
8
```

Process

- A process is basically a program in execution.
- For example, we write a program to make an entry into database table. When we execute this program CPU allocates a processor to execute this program .
- The process keeps running until program has finished off with the task.
- Processes do NOT share memory.
- Processes are not lightweight and take bit of more time to start and terminate (as compared to threads).
- **Process life cycle:** Start → Ready → Running → Waiting → Exit

Thread

- A thread is an execution unit that is part of a process.
- Every thread is part of a process and NOT vice versa.
- A process can have multiple threads.
- Threads can share memory.
- Threads are lightweight and takes less time to create and terminate.

IO operations

- There are three main jobs of a computer - input, output and processing.
- Most of the times it’s input and output and processing is simply incidental.
- Data-in and data-out operations are termed as `IO operations`.
- Following are the examples of IO operations -
  - Reading and writing data from and to the file
  - Reading and writing data into database
  - Network communication (accessing third party APIs)

How does my computer do several things at once?

- On a single core machine, it doesn’t.
- Computers can only do one task (or *process*) at a time. But a computer can change tasks very rapidly, and fool slow human beings into thinking it's doing several things at once. This is called *timesharing*.
- One of the kernel's jobs is to manage timesharing. It has a part called the *scheduler* which keeps information inside itself about all the other (non-kernel) processes.

`Concurrency` VS `Parallelism`

- `Concurrency` is when two or more tasks can start, run, and complete in overlapping time **periods.** It does NOT necessarily mean they'll ever both be running **at the same instant.** For example, *multi-tasking* on a single-core machine.
- `Parallelism` is when tasks *literally* run at the same time, e.g., on a multicore processor.

GIL 🔒

(Global Interpreter Lock)

- GIL is a limitation in Python.
- Due to GIL, Python threads are restricted to an execution model that only one thread can execute in the interpreter at any given time. This blocks us from taking advantage of multiple CPU cores.
- For this reason, Python threads are generally not used for compute intensive tasks.
- Python threads are much suited for IO operations.
- It is mutex that protects to access Python objects, preventing multiple threads from executing Python byte code at once.
- This lock is necessary because underlying `CPython` implementation is not thread-safe.
- It acts as a gate keeper on the object allowing one thread in and blocking access to all others.

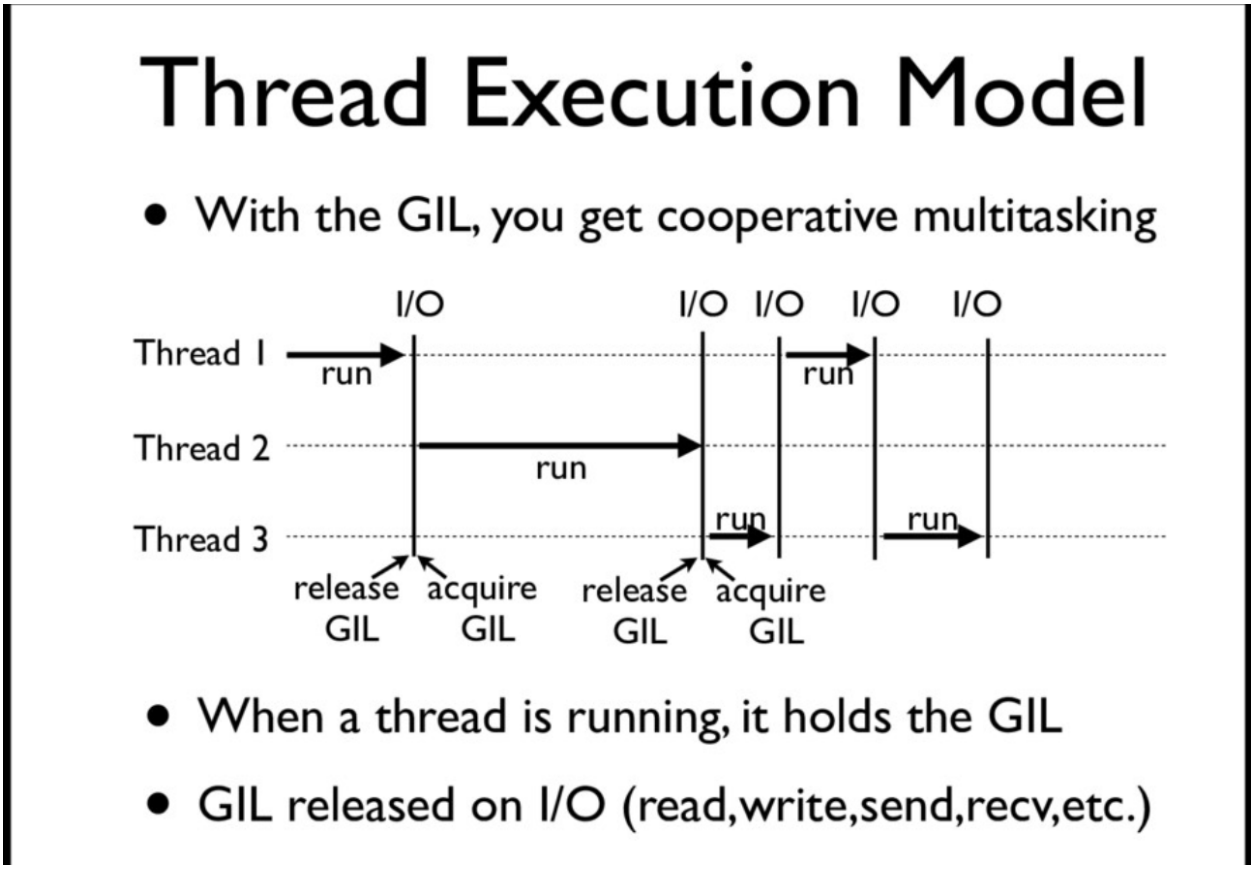
Simple analogy -

```
Suppose you're part of a big hitted discussion.

You establish a rule with a "ball" being passed around.

RULE - "whoever is holding the ball can speak. Other should shut their mouth."

This rule ensures people do not speak over each other.

Here ball is a "mutex" and person holding the ball is a "thread"
```

```
Word "mutex" means "Mutually Exclusive Object"
```

Python is a linear language

To being with, Python was designed to run on single core machine (that gets to tell you how old Python is!).



Workaround for GIL

The most common strategy to work around `GIL` is to use `multiprocessing` module to create `process pool`.

Another strategy could be using `C` extension programming to move compute intensive tasks to `C` programming).

`Multi-processing` library

```
import time


def timeit(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time() - start
        print(f"total time to execute - {end}")
        return result
    return wrapper


# function to perform large calculations (CPU bound)
def some_heavy_work(range_):
    # some complex logic with large data goes here
    result = [i**2 for i in range(range_)]
    return result


@timeit
def main():
    import multiprocessing
    pool = multiprocessing.Pool(3)
    ranges = [10000001, 10000002, 10000003, 10000004, 10000005, 10000006]

    pool.map(some_heavy_work, ranges)


# initialize pool
if __name__ == "__main__":
    main()
```

`Multi-theading` library

```
import time
import random
import requests
from multiprocessing.pool import ThreadPool


def timeit(func):
    def wrapper(*args, **kwargs):
```

```
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time() - start
        print(f"total time to execute - {end}")
        return result
    return wrapper


def fetch_data(url):
    response = requests.get(url)
    return response.status_code


def get_urls():
    urls_ = []
    for id_ in range(10):
        magic = f"https://swapi.dev/api/people/{id_}/"
        magic.format(id_=random.randrange(1, 82))
        urls_.append(magic)
    return urls_


@timeit
def main():
    pool_size = 10
    pool = ThreadPool(pool_size)
    urls = get_urls()
    results = pool.map(fetch_data, urls)
    print(results)


if __name__ == "__main__":
    main()
```

`multi-processing` VS `multi-threading`

Multiprocessing

- `multiprocessing` can be used when there are CPU-bound operations (compute intensive operations). For example, complex mathematical functions, number crunching, image processing etc.
- Scenarios when we can split data into multiple chunks and distribute chunks to a pool of sub-processes so that overall computation will be faster.

- For an example, in `multiprocessing` library we can create process pool `multiprocessing.Pool(8)` based on number of CPU cores available on given machine.
- We can use this process pool to map data chunk to each process. For example - `pool.map(fetch_data, urls)` (Refer code above)

Multi-threading

- `multi-threading` can be used for IO bound task (IO intensive task)
- `multi-threading` can be used when there are multiple IO operations like network call, database queries etc.

Related interview questions

1. **What is GIL in python?**
    - It is mutex that protects to access Python objects, preventing multiple threads from executing Python byte code at once. This lock is necessary because underlying `CPython` implementation is not thread-safe.

2. **Does GIL affect multiprocessing?**
    - If you're using in-built `multiprocessing` library in Python, it actually does side-step GIL limitations.
    - Since it spawns multiple child processes from once parent process and each process getting handled separately by underlying OS.
    - Which is why you'll be able to leverage multiple processors on multi-core machine. Prime example of this would be `Pool` object which does let you do parallel data processing.

3. **What is work around to overcome GIL in Python?**

- One of the ways is to use standard `multiprocessing` library for all multi-tasking related activities.
- Other way could be to create non-Python threads by using callback APIs written other languages (for example C or other third party library that has it's own thread management) and register those threads to Python using Python/C API.

4. **What Is Multiprocessing? How Is It Different than multi-threading in Python?**
    - `Multiprocessing` is helpful in truly parallel processing of data. While `multi-threading` gives an impression of parallelism but in reality, threads do *NOT* run in parallel manner at all due to GIL. However, they run in **time-sliced** manner and interpreter makes very quick switches between threads giving an **impression of parallelism**.
    - In `multi-threading`, threads run concurrently not in parallel.

5. **When does GIL come into play?**
    - **Thumb Rule goes like this**: When thread is running it holds GIL. GIL is released on IO (read, write, send, receive etc.)
    - Until the data is not moved into Python process, there is no reason to acquire the GIL.