# concurrency

March 28, 2021

## 1 Python Concurrency:

### 1.1 Threads, Processes & AsyncIO

#### 1.1.1 Theee Dreaded *Global Interpreter Lock (GIL)!*

CPython's limits the intretation of Python source to a single thread.

GIL released when not interpreting Python source: - Sleep - I/O - Preemptively in 15 milliseconds in Python 3 - *That* bold C/C++ extensions author *can* release and reacquire the GIL.

It is not uncommmon for other, even "high-level" languages, where - *IF* code processing is CPU-intensive AND - *IF* processing algorithm is distributable

… threading can leverage available CPU cores to divide and conqure.

Running distributable CPU-bound algoritms using Python threads will run slower than running to completion on a single thread.

**Thanks GIL!**

### 1.2 CPU-Bound Serial vs Threaded Execution

[1]:
```
import os
from datetime import datetime

cpus = os.cpu_count()
start = datetime.now()
print("Finished in: ")
for n in range(cpus):
    pow(42, 1_250_000)
    print(f"  [{n}]: {datetime.now() - start}")
```

```
Finished in:
  [0]: 0:00:00.804947
  [1]: 0:00:01.587342
  [2]: 0:00:02.386957
  [3]: 0:00:03.150548
  [4]: 0:00:03.927536
  [5]: 0:00:04.681961
  [6]: 0:00:05.449826
  [7]: 0:00:06.217420
```

### 1.2.1 Adding threads are very easy to introduce

[2]:
```
from concurrent.futures import ThreadPoolExecutor, as_completed
start = datetime.now()

print("Finished in: ")
with ThreadPoolExecutor(max_workers=cpus) as executor:
    futures = {executor.submit(pow, 42, 1_250_000): n for n in range(cpus)}
    for n, fut in enumerate(as_completed(futures)):
        print(f"  [{n}]: {datetime.now() - start}")
```

```
Finished in:
  [0]: 0:00:06.192482
  [1]: 0:00:06.193013
  [2]: 0:00:06.193047
  [3]: 0:00:06.193068
  [4]: 0:00:06.193088
  [5]: 0:00:06.193107
  [6]: 0:00:06.193322
  [7]: 0:00:06.193359
```

### 1.2.2 … not so gud.

Yet, threads are: - difficult to guard shared state along all execution paths - difficult to reason: read, maintain, grok - brittle to change - fuzzing can discover thread-safety weaknesses but labor intensive and still incomplete an complete

Failures that smell like incorrect thread-safety are hard to reproduce

Fuzzing can ferret out thread-safety weaknesses but that is very labor intensive and really, it wouldn't to test for complete thread-safety

Would Python have the popularity it enjoys w/o The GIL?

What significant community hates the GIL?

Threads in Python are primarily relegated to optimizing sets of independent I/O tasks that can benefit from being performed in parallel.

## 1.3 I/O-bound Threaded Execution

[3]:
```
import requests
urls = [ "f1040.pdf","f1040sb.pdf","p785.pdf","p487.pdf","p5384.pdf" ]
start = datetime.now()

def requests_url(url):
    response = requests.get("https://www.irs.gov/pub/irs-pdf/" + url)
    num_str = f'{len(response.content):,}'
    return f'{num_str:>10}'
```

```
print("Finished in:")
with ThreadPoolExecutor(max_workers=cpus) as executor:
    futures = {executor.submit(requests_url, url): url for url in urls }
    for n, fut in enumerate(as_completed(futures)):
        print(f"  [{n}]: {futures[fut]:>11}[{fut.result()}] {datetime.now() -␣
↪start}")
```

```
Finished in:
  [0]:    p785.pdf[    19,111] 0:00:01.203659
  [1]: f1040sb.pdf[    71,970] 0:00:01.260036
  [2]:    p487.pdf[    22,893] 0:00:01.273348
  [3]:   f1040.pdf[   150,184] 0:00:01.346713
  [4]:   p5384.pdf[52,350,859] 0:00:06.592887
```

### 1.3.1 Converting Threading to Processes? Wow. So, Easy.

[4]:
```
from concurrent.futures import ProcessPoolExecutor
start = datetime.now()

print("Finished in: ")
with ProcessPoolExecutor(max_workers=cpus) as executor:
    futures = {executor.submit(pow, 42, 1_250_000): n for n in range(cpus)}
    for n, fut in enumerate(as_completed(futures)):
        print(f"  [{n}]: {datetime.now() - start}")
```

```
Finished in:
  [0]: 0:00:01.606965
  [1]: 0:00:01.615849
  [2]: 0:00:01.619947
  [3]: 0:00:01.624372
  [4]: 0:00:01.628455
  [5]: 0:00:01.632202
  [6]: 0:00:01.635341
  [7]: 0:00:01.639127
```

Yet, processes need RPC and data serialization to share: - input - output - errors - state

### 1.3.2 Possible to realize concurrent I/O without threads?

### 1.3.3 Possible to realize concurrent I/O without threads?

**Yes. Twisted python package pulled this off, but Guido wanted to build AsyncIO into language.**

**Oooh... new keywords: `async` and `await`**

### 1.3.4 I/O-Bound Execution Threaded vs AsyncIO

```
[5]: def threaded_io():
         start = datetime.now()
         print("Threaded:")
         with ThreadPoolExecutor(max_workers=cpus) as executor:
             futures = {executor.submit(requests_url, url): url for url in urls }
             for n, fut in enumerate(as_completed(futures)):
                 print(f"  [{n}]: {futures[fut]:>11}[{fut.result()}] {datetime.now()␣
     ↪- start}")
```

```
[6]: threaded_io()

     import aiohttp; import asyncio; import nest_asyncio; from datetime import␣
     ↪datetime
     nest_asyncio.apply()

     async def load_urls():
         n = 0
         async with aiohttp.ClientSession() as session:
             print("AsyncIO:"); start = datetime.now()
             for url in urls:
                 async with session.get("https://www.irs.gov/pub/irs-pdf/" + url) as␣
     ↪resp:
                     content = await resp.read()
                     file_size = f"{len(content):,}"
                     print(f"  [{n}]: {url:>11}[{file_size:>10}] {datetime.now() -␣
     ↪start}"); n += 1

     loop = asyncio.get_event_loop()
     loop.run_until_complete(load_urls())
```

```
Threaded:
  [0]:    p785.pdf[    19,111] 0:00:00.365855
  [1]:    p487.pdf[    22,893] 0:00:00.366042
  [2]: f1040sb.pdf[    71,970] 0:00:00.376147
  [3]:   f1040.pdf[   150,184] 0:00:00.422244
  [4]:   p5384.pdf[52,350,859] 0:00:05.785993
AsyncIO:
  [0]:   f1040.pdf[   150,184] 0:00:00.350478
  [1]: f1040sb.pdf[    71,970] 0:00:00.400695
  [2]:    p785.pdf[    19,111] 0:00:00.425121
  [3]:    p487.pdf[    22,893] 0:00:00.453385
  [4]:   p5384.pdf[52,350,859] 0:00:05.759719
```

Q: What be this async magic? A: Coopertative mult-tasking

What is old is new.

Q: How? A: At the heart of any `await()` is a `yeild` of `generator` fame.

#### 1.3.5 AsyncIO's Attractions

- No locks! So, much easier to get your code correct.
- Switching from tasks cooperatively using AsyncIO is *faster* than calling a function.

#### 1.3.6 AsyncIO's Detractions

- Breaks abstractions, meaning: hard to reason `goto` behavior
- Breaks `with` statements
- Breaks exceptions
- Any blocking (I/O, time.sleep(), etc.) compromises AysncIO's benefits; async versions of blocking methods must be leveraged
- Enormous learning curve to understand/debug the Async loop and numerous replacements for blocking I/O

### 1.4 The Next Level: Co-routines + Threads

#### 1.4.1 Nathanial Smith's Trio Package

Makes asyncio easier to implement, reason

### 1.5 Alex Martelli Model of Scalability

- 1 core: single thread and single process
- 2-8 cores: multiple threads and multiple processes
  - GPUs for limited computations and low-level languages
- 9+ cores: distributed computing

Martelli was part of early Google and made the following observations: - Single threaded scaling due to the constant improvements of the speed of single core CPU (Moore's Law) MANY problems can be solved with the massive CPU power that can be purchased. - 2-N cores (augemented w/ cache trashing hyper-threading) can combine with threaded programming to perform parallel operations on separate data.
If your problem can be solved with N cores, great. But, your problem is pracariously close to reaching the hard limits for massively parallel operations.
Even when considering GPU, there are limits to what kind of computations and data sizes that can be addressed using low-level CUDA.

So, there are good reasons to consider the diminishing return of leveraging parallelization through threading and skip right to distributed computing; essentially the Kubernetes' focus to leverage CPU cores at a higher level further managing failures, state and data using distribution designs.

## 2 Recommendations

Raymond Hettinger's Concurrency Keynote PyBay 2017

Nathaniel J. Smith'a Trio: Async concurrency for mere mortals - PyCon 2018