Concurrent & Parallel Programming in Python

Kai Anter, Caspar Sachsenmaier

04.07.2023

Structure

- Python Overview
- Libraries
  - threading
  - asyncio
  - multiprocessing
  - concurrent.futures
- Outlook

Overview: Python

- Interpreted, dynamically typed language
- Implementations:
  - CPython (reference implementation)
  - MicroPython
  - Stackless Python (→ see other Lightning Talk)
  - …

```
def my_funcion():
    print("Hello World")

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

CPython

- Written in C
- Current version: 3.11.4
- User scripts are read → compiled to bytecode → executed
- Uses a single thread to[1]:
  - Run the user's program and
  - the memory garbage collector
- Has the infamous Global Interpreter Lock

[1] Ramalho (2022)

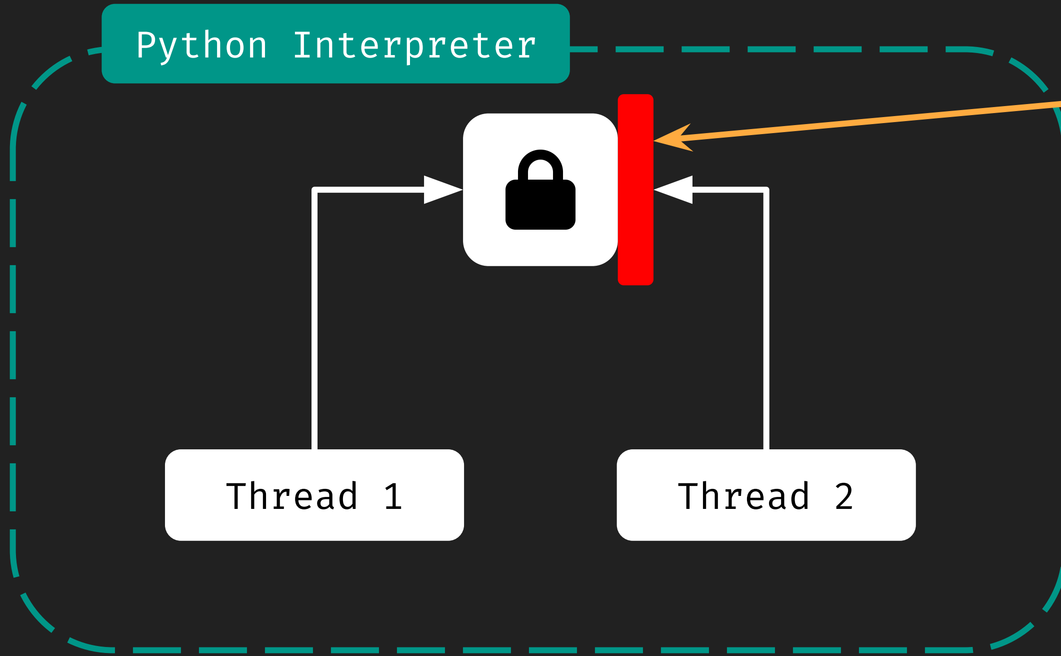
The Global Interpreter Lock (GIL)

- Basically a mutex
  → only 1 Python thread can run bytecode at the same time
- Ensures exclusive access to interpreter internals for current thread[2]
- Mostly not a problem for performance[3]
  - Exception: CPU-heavy workloads implemented in Python
  - Larger issue: blocking IO operations

→ (In general,) A single Python interpreter can run code concurrently, but not in parallel

[2] Beazley (2010)
[3] Reitz & Schlusser (2016)

The Global Interpreter Lock (GIL)

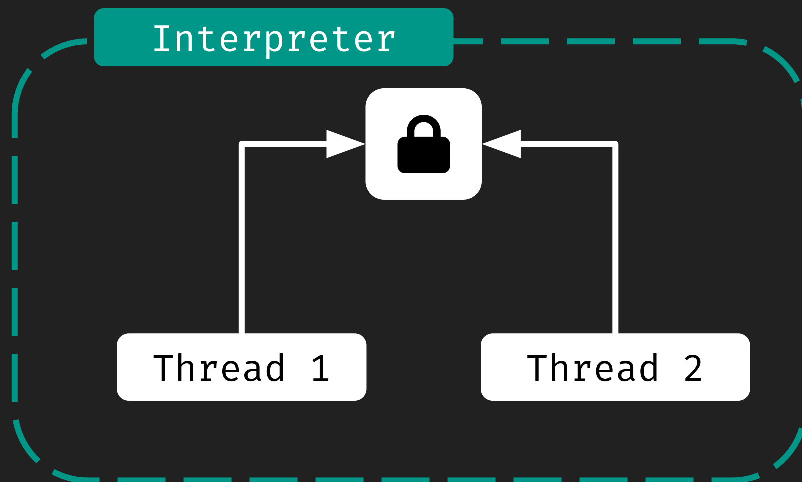


Blocked until Thread 1 releases GIL:

- Timeout reached (5ms)
- Syscall
  - Blocking IO
  - Network
  - time.sleep()
  - …
- Special library function called, e.g. NumPy, SciPy, zlib

Library: Threading

- Thread based, similar to Java Threads
- Creates new OS level threads (not user level)
- Limited parallelism due to GIL
- Use Cases:
  - Running blocking IO operations
  - Background (daemon) services



threading: Simple Example

```python
from threading import Thread
import time

def my_func(line: str):
    time.sleep(5)
    print(f"Output: {line}")

t = Thread(target=my_func, args=("test",))
t.start()

# Alternative: Create Subclass of Thread
class MyThread(Thread):
    def run():
        # ..
        pass
```

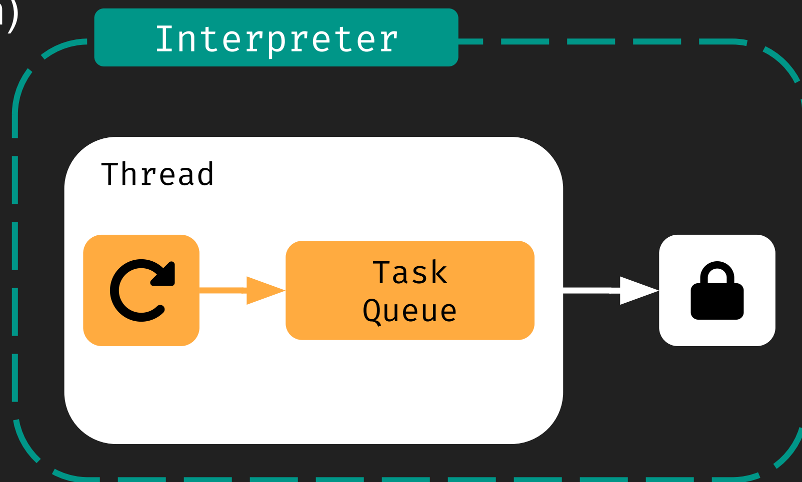
args as tuple or array

non-blocking, wait for finish with `t.join()`

pass `daemon=True` for background service

Library: `asyncio`

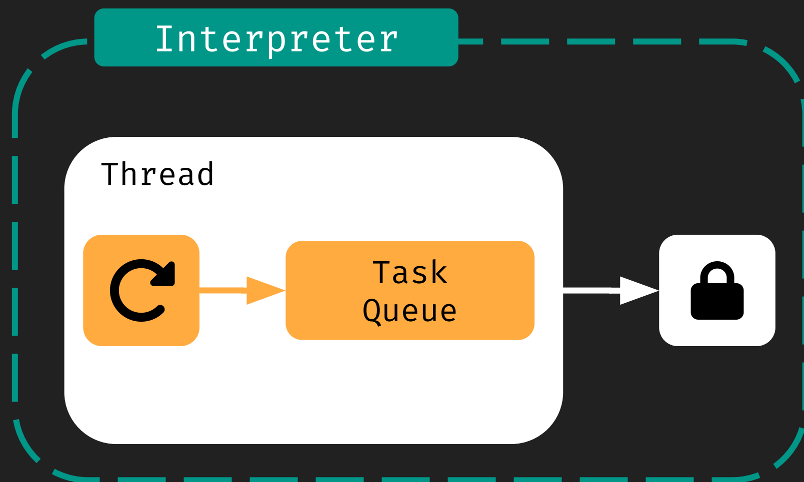
- Based on Coroutines + Event Loop
- Introduces new syntax: `def async` / `await` (Similar to JS Promises)
- Use cases:
  - Asynchronous IO (without creating new OS threads per blocking operation)



[4] Ramalho (2022)

Library: `asyncio`

- Use `async def` to define a coroutine function → return a coroutine object
- Three ways to run coroutine objects:
  - `asyncio.run( … )`
  - `await` *`awaitable_obj`*
  - `asyncio.create_task( … )`
- `await` can be used on *awaitables*[4]:
  - Coroutine objects
  - Tasks



[4] Ramalho (2022)

asyncio: Run Coroutine with await

```
import asyncio

async def calc_coro():
    print("calculating...")
    await asyncio.sleep(2) # some asynchronous operation
    return "foo"

async def main1():
    coroutine_object = calc_coro()
    calcresult =  await coroutine_object
    print(f"Result of calculation: {calcresult}")

if __name__ == "__main__":
    asyncio.run(main1())
```

(coroutine function)

(2) call coroutine function to get coroutine object

(3) run obtained coroutine object with await

(1) run coroutine from “normal” function

asyncio: Run Coroutine as Task

```python
import asyncio

async def calc_coro():
    print("calculating...")
    await asyncio.sleep(2) # some asynchronous operation
    return "foo"

async def main2():
    task = asyncio.create_task(calc_coro())
    print("do other stuff")
    await task
    # If without await: throws InvalidStateError result is not set
    calcresult = task.result()
    print(f"Result of calculation: {calcresult}")

if __name__ == "__main__":
    asyncio.run(main2())
```

Important: Store reference of task somewhere
→ Prevent Garbage Collector freeing task
before it is executed

can be skipped if completion /
result of coroutine is not
relevant here

asyncio: Run Coroutines with TaskGroups

```python
import asyncio

async def main():
    # New in Python 3.11
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(coro_func1())
        task2 = tg.create_task(coro_func2())
    print("all tasks completed")
```

Alternative to storing Task reference: asyncio.TaskGroup

blocks until all Tasks in TaskGroup are finished

From: Python asyncio documentation

Library: `multiprocessing`

- Uses processes instead of threads
- Creates new subprocesses of the Python interpreter
  - spawn (Unix & Windows)
  - fork (Unix only)
  - forkserver (some Unix platforms)
- Bypassing GIL by using one for every process
- Use Cases:
  - CPU intensive programs without IO operations

`multiprocessing`: Communication between processes

- Processes can communicate by Queues or Pipes
- Queues are process safe
- Pipes can get corrupted by multiple processes accessing the same end at the same time → only one sending & one receiving process per pipe

```python
from multiprocessing import Process, Queue

def f(q):
    q.put(['python', 'is', 'gr' + str(8)])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "['python', 'is', 'gr8']"
    p.join()
```

```python
from multiprocessing import Process, Pipe

def f(conn):
    conn.send(['python', 'is', 'gr' + str(8)])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "['python', 'is', 'gr8']"
    p.join()
```

From: Python multiprocessing documentation

`multiprocessing`: Shared memory

- use `multiprocessing.sharedctypes`
- `multiprocessing.manager` can be used to hold Python objects
- A manager can also be shared on different computers (forkserver)

```
from multiprocessing import Process, Manager

def f(d, l):
    d['python'] = 'good'
    d['java'] = 'not so much'
    d[1] = '😉'

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
        p = Process(target=f, args=(d, l))
        p.start()
        p.join()
        print(d)
        print(l)
```

From: Python multiprocessing documentation

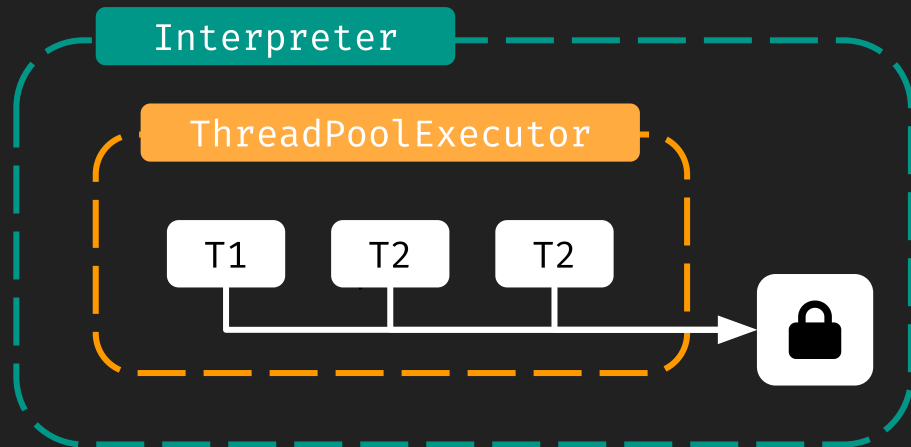
multiprocessing: Process Pools

```python
import multiprocessing

def worker_function(x, y):
    return x * y

def main():
    x_values = (1, 2, 3)
    y_values = (9, -5, -9)

    with multiprocessing.Pool() as pool:
        results = pool.starmap(worker_function, zip(x_values, y_values))
        print(results)

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

Like map, but returns iterator

Library: `concurrent.futures`

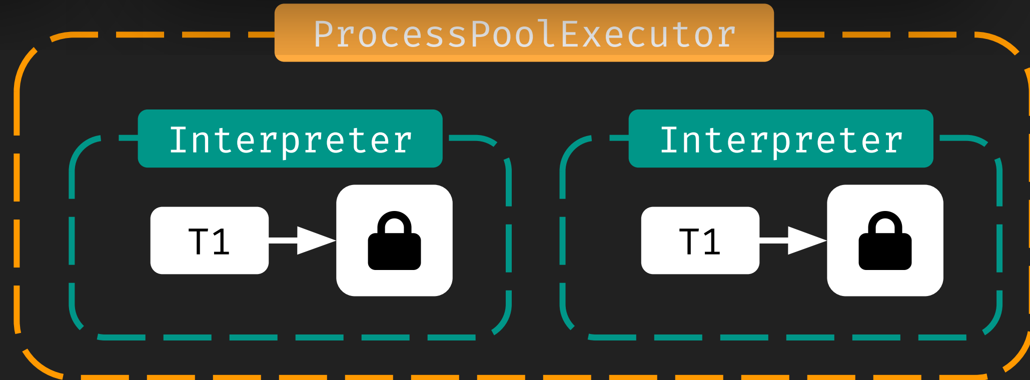
- `ThreadPoolExecutor` or `ProcessPoolExecutor`
- Use Threads for I/O intensive programs, Processes for CPU intensive ones
- Beware of deadlocks by waiting on another future



concurrent.futures: ProcessPoolExecutor

```python
import concurrent.futures

def worker_function(x, y):
    return x * y

def main():
    x_values = (1, 2, 3)
    y_values = (9, -5, -9)

    with concurrent.futures.ProcessPoolExecutor() as executor:
        results = list(executor.map(worker_function, x_values, y_values))
        print(results)

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

```python
import multiprocessing

def worker_function(x, y):
    return x * y

def main():
    x_values = (1, 2, 3)
    y_values = (9, -5, -9)

    with multiprocessing.Pool() as pool:
        results = pool.starmap(worker_function, zip(x_values, y_values))
        print(results)

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



Outlook: A Per-Interpreter GIL

- PEP 684 – A Per-Interpreter GIL
  - Currently: Interpreters in the same process share GIL → sharing of global states…
  - Proposal: Stop sharing GIL between Interpreters
  - Proposal was accepted, feature will be released with 3.12

Outlook: Making the Global Interpreter Lock Optional

- PEP 703 – Making the Global Interpreter Lock Optional in CPython
  - Currently: No parallelism possible in threads because of the GIL
  - Proposal: Making it possible to disable the GIL
  - Proposal just a Draft

References

- Reitz, K., & Schlusser, T. (2016). The hitchhiker's guide to python. O'Reilly Media.
- Beazley, D. (2010). Understanding the Python GIL. https://www.dabeaz.com/python/GIL.pdf
- Ramalho, L. (2022). Fluent Python: Clear, Concise, and Effective Programming (2nd ed.). O'Reilly Media.
- https://docs.python.org/3/glossary.html#term-bytecode (Retrieved: 30.06.2023)
- https://docs.python.org/3/library/threading.html (Retrieved: 30.06.2023)
- https://docs.python.org/3/library/asyncio.html (Retrieved: 30.06.2023)
- https://docs.python.org/3/library/multiprocessing.html (Retrieved: 30.06.2023)
- https://docs.python.org/3/library/concurrent.futures.html (Retrieved: 30.06.2023)