Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Concurrence

Introduction
CPU-bound and I/O-bound
Event loop
asyncio

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Objectives

- CPU-Bound vs I/O-bound concept
- Basic understanding about the problems of blocking actions
- Python GIL
- Event-loop introduction
- Asyncio in Python

References

[1] Python Concurrency With Asyncio, Matthew Fowler, 2022 Manning Publications, ISBN 978-1617298660

https://books.google.ch/books?id=YqljzgEACAAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false

[2] Previous Concurp Slides – The Global Interpreter Lock (GIL)

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Concurrence

Introduction

CPU-bound and I/O-bound
Event loop
asyncio

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Concurrent work

- **There 2 diffenrent ways of dealing with concurrent programming:**

  1. CPU-bound tasks
  2. I/O-bound tasks

  – We will go through the GIL recap

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Concurrence

Introduction
CPU-bound and I/O-bound
Event loop
asyncio

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Concurrent work

- **CPU-bound tasks:**
  - These spend most of their time using the processor (doing calculations).
  - Examples:
    - mathematical computations
    - image processing
    - simulations
- **I/O-bound tasks**
  - These spend most of their time *waiting* for something external:
    - network requests
    - reading/writing files
    - database queries
- The key difference:
  - CPU-bound = “working hard”
  - I/O-bound = “waiting a lot”

Why concurrency matters differently

- **CPU-bound tasks:**
  - If you run multiple CPU-heavy tasks:
    - They **compete for CPU time**
    - On a single core, they don't really run in parallel
    - Even on multiple cores, Python has a limitation (explained next)
  - Best approach: **multiple processes**

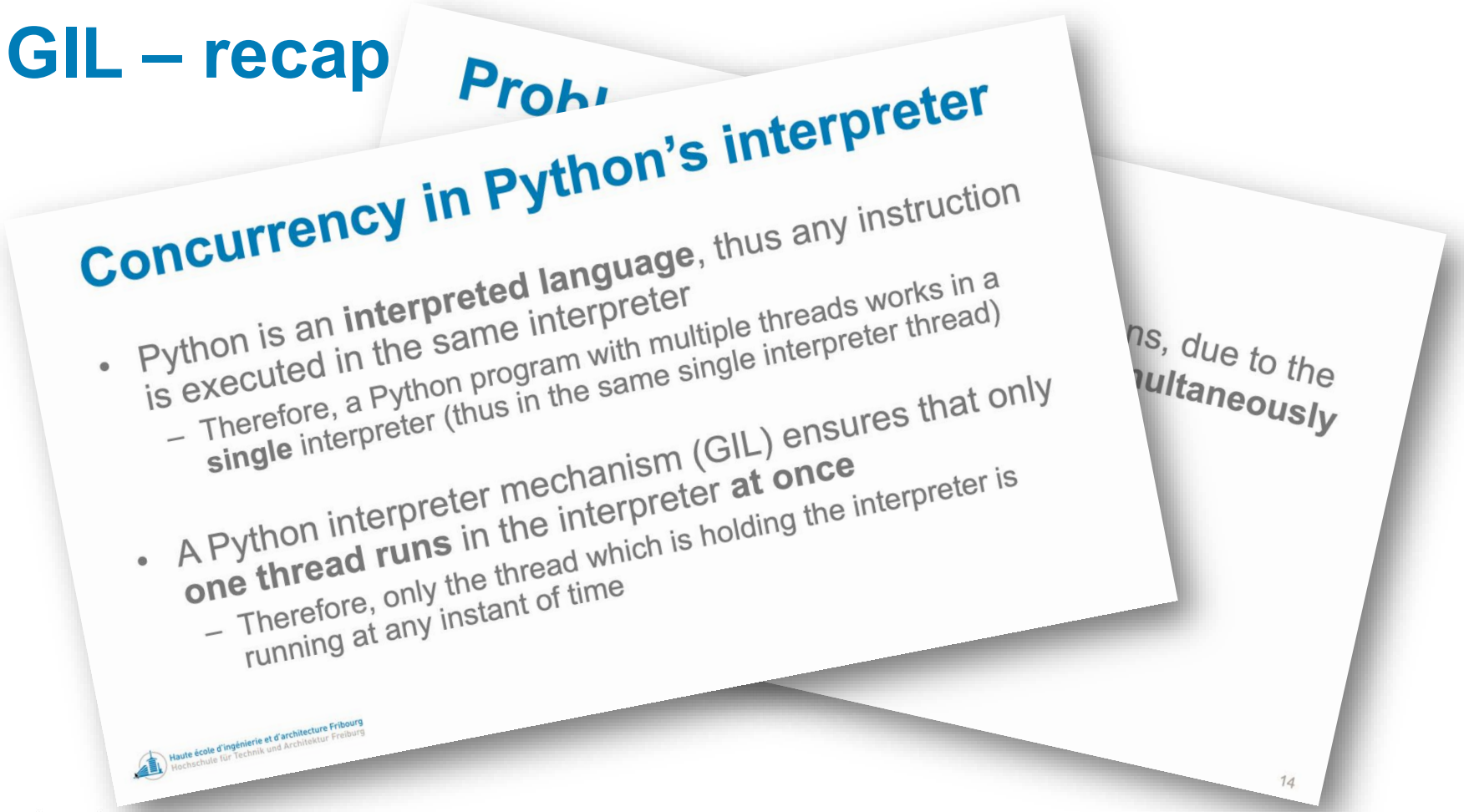
- **I/O-bound tasks**
  - If tasks spend time waiting:
    - While one task waits, another can run
    - So concurrency gives a big speedup
  - Best approach:
    - **threads**
    - or **async / processes**

The Python-specific issue: the GIL

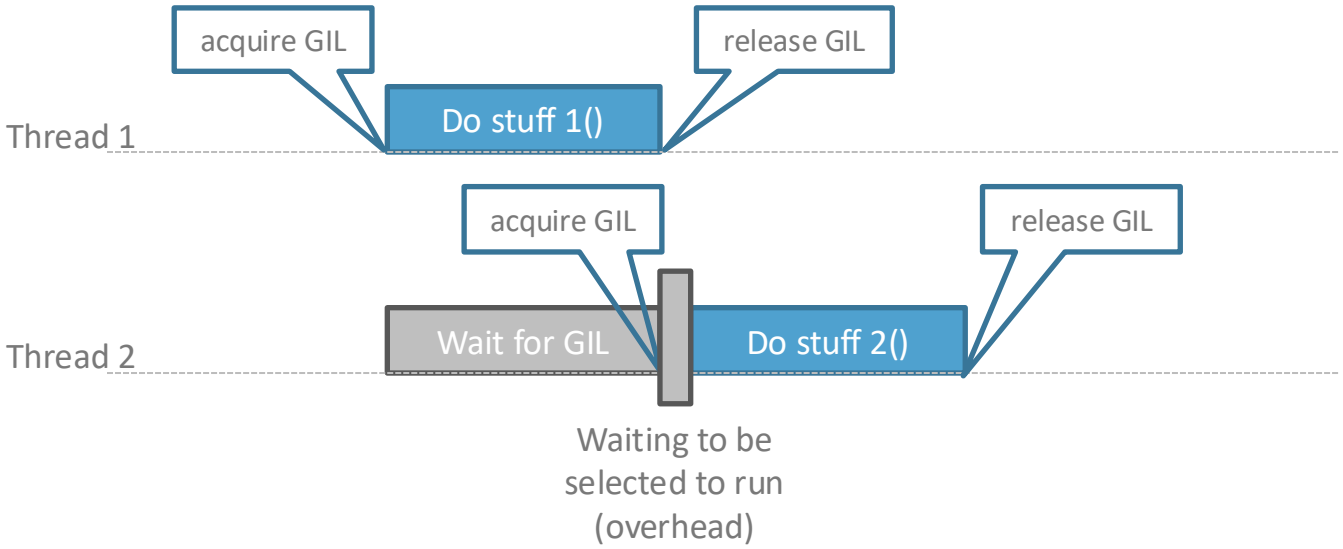
Python (specifically the standard implementation, **CPython**) has something called the **Global Interpreter Lock (GIL)**.

- **What is the GIL?**
  - It ensures that **only one thread executes Python bytecode at a time**
  - Even on a multi-core CPU !!

GIL – recap


Concurrency in Python's interpreter
• Python is an **interpreted language**, thus any instruction is executed in the same interpreter
– Therefore, a Python program with multiple threads works in a **single** interpreter (thus in the same single interpreter thread)
• A Python interpreter mechanism (GIL) ensures that only **one thread runs** in the interpreter **at once**
– Therefore, only the thread which is holding the interpreter is running at any instant of time
Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg
14

GIL – Recap


acquire GIL
release GIL
Do stuff 1()
Thread 1
acquire GIL
release GIL
Wait for GIL
Do stuff 2()
Thread 2
Waiting to be selected to run (overhead)

Why CPU-bound tasks need processes in Python

Because of the GIL:

- Multiple threads **cannot truly run CPU code in parallel**
- So threads won't speed up CPU-heavy work

That's why:

- You use **multiprocessing** (separate processes)
- Each process has its own Python interpreter and its own **GIL !**
- So they can run on different CPU cores

✔ Result: real parallelism

Why I/O-bound tasks work well with threads

Here's the important nuance:

When a thread performs I/O (like downloading data):

- Python **releases the GIL**
- Other threads can run while waiting

So even with the GIL:

- Threads can overlap waiting time efficiently

✔ Result: big performance gains for I/O-heavy workloads

Simple intuition

Think of it like this:

- **CPU-bound**
  - chefs cooking intensely → only one can use the stove → need more kitchens (processes)
- **I/O-bound**
  - chefs waiting for ingredients → others can cook meanwhile → threads work well

Concurrency comparison : Python vs Java

Concurrency model	Python	Java
CPU-bound	• Threads are limited by the GIL • Use of Processes • “Use more kitchens (processes)”	• Threads map to real OS threads, run **truly in parallel**, • fully use multi-core CPUs. Use of essentially Thread. • “Use more chefs in the same kitchen (threads)”
I/O-bound	• GIL is **released during I/O** • Threads work well • Also has asyncio (event loop, non-blocking I/O) • Threads and Async is **very efficient and lightweight** • “Use async or threads to avoid waiting”	• Traditional model: 1 **thread per request** • Also supports async (e.g. NIO) • Threads are heavier than Python async tasks • “Use threads (or async if scaling massively)”
Existing differences	GIL simplifies: • memory management • interpreter design Tradeoff: • poor CPU thread scaling	Designed from the start for: • multi-threading • parallel execution JVM handles: • thread scheduling • memory safety

Key take-away messages

- CPU-bound tasks:
  - Limited by CPU
  - Python threads blocked by GIL
  - ✅ Use **processes**
- I/O-bound tasks:
  - Limited by waiting time
  - GIL released during I/O
  - ✅ Use **threads or async**

What actually happens under the hood

… when Python does I/O?

- It's a really important mental model, especially for **asyncio**.

Userland → Kernel

- Your Python code runs in **user space** (also called *userland*).
  The **operating system kernel** is the part that actually controls:
  - network
  - disk
  - hardware

What actually happens under the hood

So when you do something like:

```
data = socket.recv(1024)
```

(covered in slide 22)

Python **cannot fetch the data itself**.

It must ask the OS:

"Hey, please read from the network for me."

This is called a **system call** :

crossing from *userland* → *kernel*

Why the GIL is released during I/O

When Python hands work to the OS:

- The thread is now mostly **waiting**
- The OS is doing the real work

So Python says: “No point holding the GIL while I wait.”

→It **releases the GIL**

This allows:

- another Python thread to run
- better concurrency

✓ This is why threads work well for I/O-bound tasks

What the OS does (preemptive multitasking)

Modern OSes (Linux, macOS, Windows) use **preemptive multitasking**:

(recap from Systèmes Concurrents course)

- The OS decides which task runs and when
- It can pause/resume tasks at any time

So while your Python thread is waiting for I/O:

- The OS can run other processes or threads
- Your program is not blocking the whole system

When does Python resume?

Once the OS finishes the I/O:

- It sends a signal/event: “data is ready”
- Python wakes up the waiting thread
- The thread **reacquires the GIL**
- Execution continues

Blocking vs non-blocking sockets

Default: blocking sockets

```
data = sock.recv(1024)
```

- Python asks OS for data
- ❌ Thread waits (blocked)
- Nothing else happens in that thread

Even though:

- The GIL is **released**
- Other threads *can* run

Blocking vs non-blocking sockets

OS-level non-blocking sockets

At the OS level, you can say:

"Start the operation and tell me later when it's ready"

This is **non-blocking I/O**

- The call returns immediately
- You don't wait
- The OS notifies you later via an **event system**

Examples of OS event systems:

- select
- poll
- epoll (Linux)
- kqueue (macOS)

“Fire and forget” idea

With non-blocking I/O:

- You ask: “start reading”
- The OS says: “OK, come back later”
- Your program does other work
- OS notifies: “data is ready”

This is the foundation of **async programming**

Covered in next chapter

Putting it all together

Blocking I/O (threads)

1. Python → OS (start I/O)
2. GIL released
3. Thread sleeps
4. OS finishes
5. Thread wakes up, GIL reacquired

Non-blocking I/O (asyncio)

1. Python → OS (start I/O)
2. Returns immediately
3. Event loop continues other tasks
4. OS signals readiness
5. Event loop resumes the task

Key take-away messages

- Key intuition
  - Threads + I/O = “wait, but others can run”
  - Async I/O = “don’t wait at all — come back later”
- The GIL is **not the enemy for I/O**
- It’s released during I/O because:
  - the OS is doing the work
- Async takes it further:
  - avoids blocking entirely using OS event notifications

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

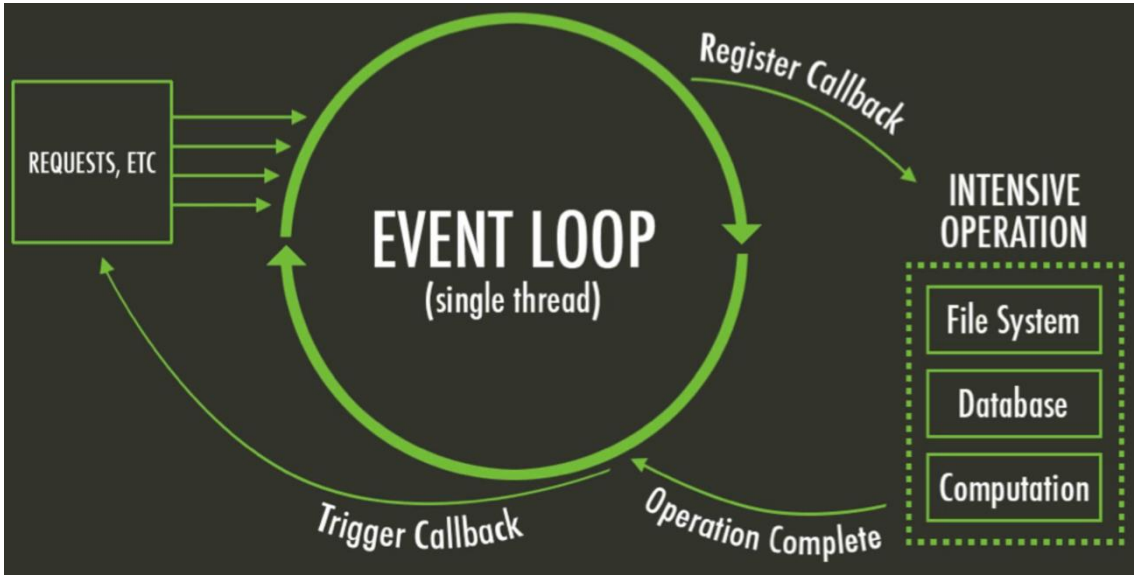
Concurrence

Introduction
CPU-bound and I/O-bound
Event loop
asyncio

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Event loop

An event loop is a program that continuously waits for events and runs the code that should react to them.



Source: https://stackoverflow.com

Event loop – Analogy

📞 **Call center operator**

- The operator has a list of clients (tasks)
- Some are:
  - ready to talk
    (ready to run)
  - waiting for a callback
    (I/O)

The operator:

1. Checks who is ready
2. Talks to them
3. If someone says:
   - "Call me back when data arrives"
     → puts them on hold
1. Moves to the next person
2. When a callback comes in
   → resumes that client

👉 That operator **is the event loop**

Event loop (cont.)

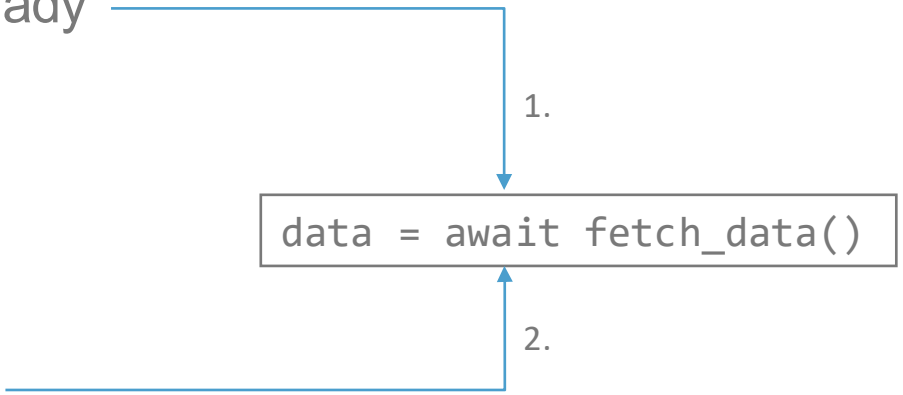
- EL is at the heart of every **asynchronous** I/O application
- It is a common design pattern
  - JavaScript in Node.js heavily use EL (https://nodejs.org/en/about)
  - Recap course “systèmes d’informations”
- Example of a basic EL:

```
from collections import deque
messages = deque()
while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

In Python’s **asyncio,** the EL keeps a queue of **tasks** instead of messages

Event loop (cont.) and I/O

- A task starts an I/O operation (network, file, etc.)
- It says:
  - “Pause me until data is ready”
- The event loop:
  - registers this with the OS
  - moves on to another task
- Later:
  - OS says: “data is ready”
  - event loop resumes the task



Event loop and Threads

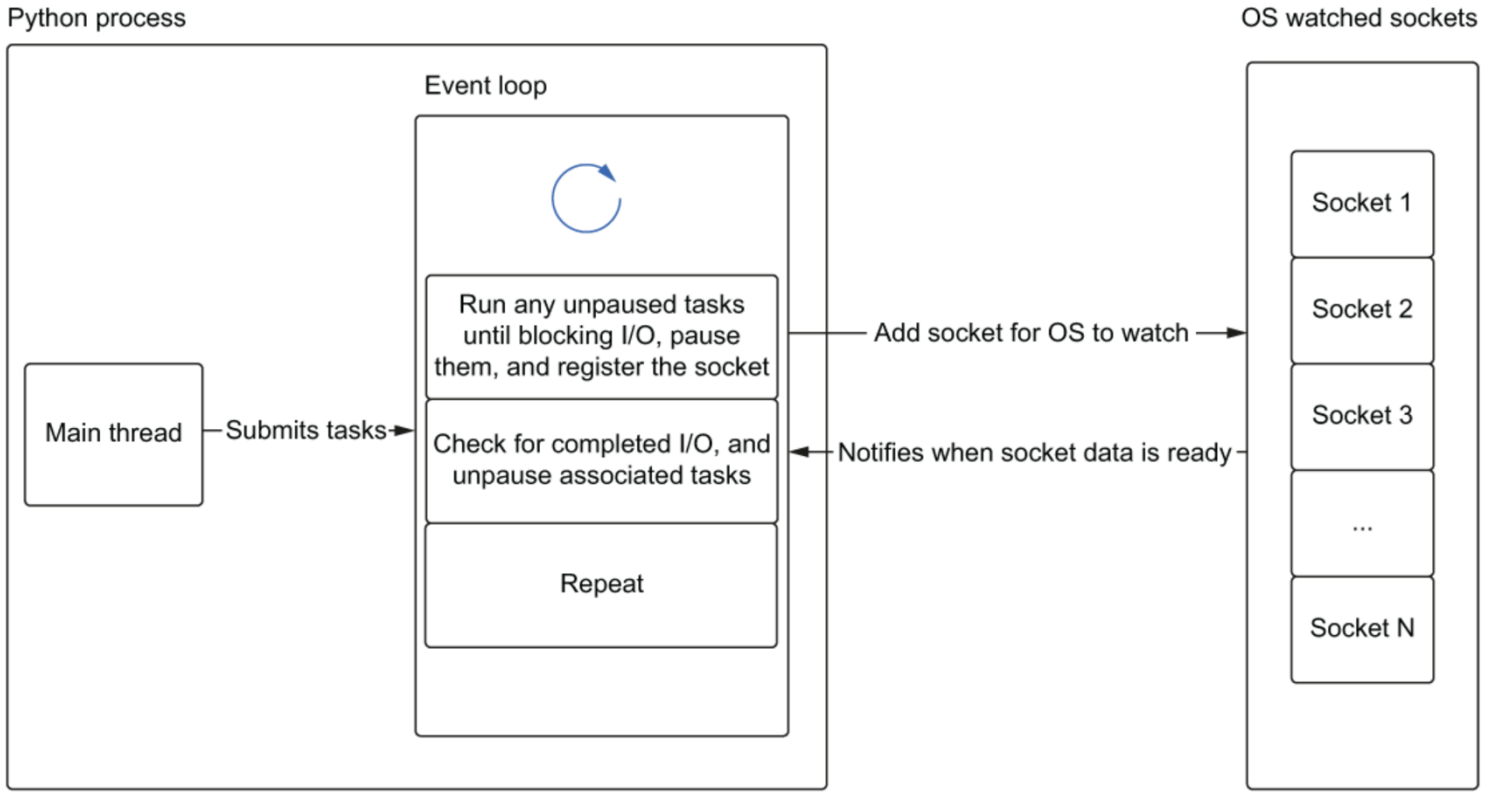
Model	How it works
Thread	• OS switches between threads • Preemptive multitasking
Event Loop	• Code voluntarily pauses (await) • Cooperative multitasking

Event Loop:

- Task A → needs data → pauses
- Task B → runs
- Task C → pauses
- OS → “A is ready”
- Event loop → resumes A

👉 No waiting, no blocking

Thread submitting tasks to an EL



Source: [1]

Coroutines

- Tasks are wrappers around a **coroutine**
- Coroutine can pause execution when it hits an I/O-bound operation (or other operation that might yield the execution)
  - This will let the event loop run other tasks (coroutines) that are not waiting for I/O operations
  - **Yield** the execution to another coroutine
  - Cooperative scheduling (vs. preemptive scheduling)

Interaction between coroutine and event loop

- At creation, the queue of tasks is empty (empty event loop)
- During execution of the program, tasks can be added to the queue
- Each iteration of the event loop checks for tasks need to be run
  - Eventually these task will be run
- Tasks hitting an I/O operation will be *paused* and the EL looks for another task to execute
- On every iteration of the EL, it checks to see if any of our I/O operation has complete
  - These tasks then can be brought back to the EL and be woken up

Key take-away messages

- I/O-bound operations
  - Using in a non-blocking manner (thanks to the OS)
  - Events showing completion of I/O operations
- Event loop
  - Queue full of tasks that can be paused and resumed
  - Tasks are wrappers for coroutines
  - The EL loops forever, looking for tasks with CPU-bound work to run while also pausing tasks that are waiting for I/O

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Concurrence

Introduction
CPU-bound and I/O-bound
Event loop
asyncio

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

References

[1] Python Concurrency With Asyncio, Matthew Fowler, 2022 Manning Publications, ISBN 978-1617298660
https://books.google.ch/books?id=YqljzgEACAAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false

[2] Awesome async https://github.com/timofurrer/awesome-asyncio

[3] aiodebug – A debugging library for asyncio applications
https://gitlab.com/quantlane/libs/aiodebug

[4] https://www.youtube.com/watch?v=t5Bo1Je9EmE

Python asyncio – current versions ;-)

- **asyncio** is quite a new construct in Python and was first introduced with Python version 3.5
  - https://peps.python.org/pep-0492/
- It is still under heavy development and gains in functionalities and API methods with each new version
- The examples work all with Python 3.10 and higher
  - They also should work with version 3.8 and 3.9, but are not tested
  - Don’t use older versions!

Python's coroutines – Introduction

- Coroutine ➔ regular Python **function** that it can pause its execution when it encounters an operation that could take a while to complete
- While a coroutine is paused, the system can run other code
- Finally, asyncio uses **cooperative multitasking** to achieve concurrency
  - Coroutines voluntarily yield control when idle or logically blocked

Python’s coroutines – Key words

- Two new Python keywords are introduced:
- **`async`**: definition of a coroutine
- **`await`**: suspends the execution of a coroutine on an awaitable object

Creating coroutines

- The creation of a coroutine is like creating a normal Python function, except:
  - `def` will be replaced by **`async def`**

```
async def coroutine_add_one(number: int) -> int:
    return number + 1
```

Creating coroutines (cont.)

- Coroutines **can not be called directly** like normal routines!
  - The `def async` creates a coroutine. So, we get a *coroutine object* back

- The created coroutine object can be run later
  - **It must be run in an event loop!**

  - How to create an event loop?

Async Coroutine vs. sync Function

```python
async def example_coroutine_function(a, b):
    # Asynchronous code goes here
    ...

def example_function(a, b):
    # Synchronous code goes here
    ...
```

Running coroutine in an event loop

```python
async def coroutine_add_one(number: int) -> int:
    return number + 1


if __name__ == '__main__':
    coroutine_object = coroutine_add_one(1)
    coroutine_result = asyncio.run(coroutine_add_one(1))
```

Main entry point of the asyncio app

- `asyncio.run()` creates a new event loop and places the coroutine in the loop
  - In this **main** event loop other coroutines can be created and started
  - It also does cleanup stuff at the end once the ”main” coroutine finished
  - This function **cannot be called** when another asyncio event loop is running in the same thread! → Only one event loop in a single thread!

https://docs.python.org/3/library/asyncio-task.html#running-an-asyncio-program

`await()` - Suspending execution of a coroutine

- The benefit of asyncio is being able to **suspend** execution to let the event loop run other tasks during a "**long-running**" operation
- `await coro` will run the given coroutine (`coro`) and pauses the parent (main) coroutine in the event loop

```
async def main():      # main coroutine
    …
    # pause the 'main' coroutine, start foo coroutine
    # wait until foo finished, return possible results
    result = await foo('cool')
    …
```

Example – async and await

```python
import asyncio


async def main():
    print("Hello asyncio")          #2
    ret = await foo('cool')         #3, 6
    print("finished")               #6


async def foo(text):                #4
    print(text)                     #4, 5
    await asyncio.sleep(1)          #5



if __name__ == '__main__':
    asyncio.run(main())             #1
```

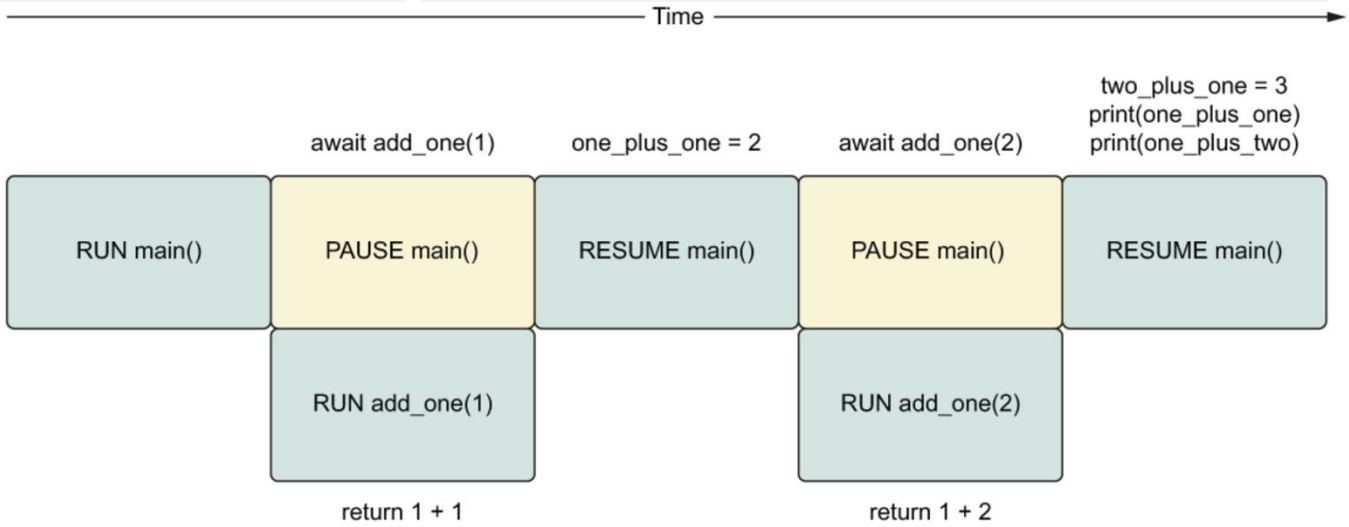
1. The main eventloop (EL) will be created (asyncio.run)
2. Code is sequentially executed until the `await` statement
3. The “main” coroutine in the EL will suspend and *await* until the coroutine `foo` finishes
4. The coroutine `foo` will be executed as soon as possible (eventually, other coroutines in the EL)
5. `foo` executes its code and can, if desired, launch other coroutines (`asyncio.sleep`)
6. As soon `foo` has finished, the main coroutine (gets an eventual return value) continues with the line after the `await` statement

Example – async and await

```python
async def add_one(number: int) -> int:
    return number + 1

async def main() -> None:
    one_plus_one = await add_one(1)
    two_plus_one = await add_one(2)
    print(one_plus_one)
    print(two_plus_one)
```

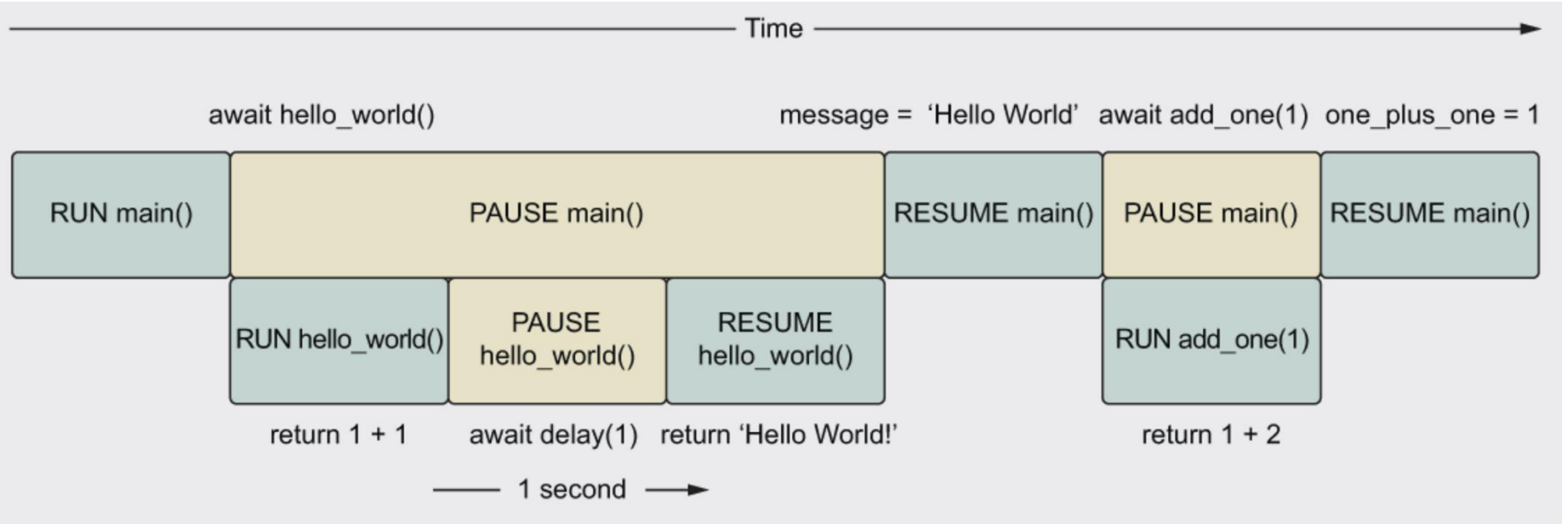
```python
asyncio.run(main())
```



Sleeping in a coroutine

- `asyncio.sleep()` makes a coroutine “sleep” for a given number of seconds
  - Used to simulate long-running calls to web API or databases
  - Yields the processor’s possession and allow to another coroutine to run
- `asyncio.sleep()` itself is a coroutine and must therefore be called with the help of the `await()` statement
  - When a coroutine awaits it, other code (coroutine) will be able to run

Example of simulated long-running code



Still quite synchro !

Concurrent coroutines → Tasks

- Asynchronous tasks can be placed in the event loop and therefore can run **concurrently**
  - Task is a wrapper around a **coroutine** that schedules a coroutine to run on the event loop **as soon as possible**

- Coroutines run as tasks → **non-blocking** execution
- Coroutines run with `await` → **blocking** execution

- Scheduling of **multiple** task on the event loop allows concurrency

Creation of tasks in the event loop

- Creation of a task in the event loop: `asyncio.create_task(coro)`
  - It returns a coroutine object

```python
import asyncio
from util.delay_function import delay

async def main(sleep_time):
    result = None
    sleep_for = asyncio.create_task(delay(sleep_time)) # create and launch task on the EL
    print(type(sleep_for))
    result = await sleep_for                    # wait until task finishes and return value
    print(result)

if __name__=='__main__':
    SLEEP_TIME = 1
    asyncio.run(main(SLEEP_TIME))
```

Key take-away messages

- Main event loop is launched with **`asyncio.run(coro)`**
- Coroutines are defined with **`async`** keyword
- Only coroutines can be awaited → **`await`** statement
- Tasks can be placed on event loop
  - **Concurrent** execution
  - `await` for waiting of the end of the coroutine's execution