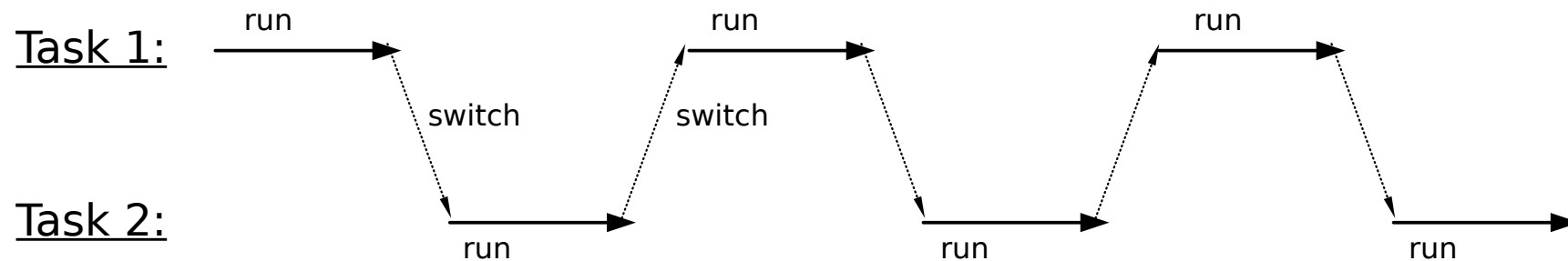# Concurrency

Concurreny

- Basic principles

- The Global Interpreter Lock

- Async and asyncio

- Threading

- Multiprocessing

- The future

Multitasking

- **Multitasking with a single CPU means task switching**

Task 1:   → run →   switch   switch → run →   → run →

Task 2:   run →   run →   run →

- **The cost of task switching means running tasks concurrently is slower than running them consecutively**

True Concurrency

With multiple cores and multiple CPUs, modern computers can do "parallel processing" and run multiple tasks simultaneously without the overhead of task switching.

Task Execution

- Tasks tend to be "I/O bound" (Input/Output) or "CPU bound"

- I/O means a network request, or accessing a file or a database which is much slower than the CPU, so the task spends a lot of time waiting

- CPU bound tasks do a lot of processing and will use as much processing power as you can give them

An I/O bound task → [I/O] → [I/O] → [I/O] → [I/O] → [I/O] → [I/O] →

A CPU bound task ——————→ [I/O] ——————→ [I/O] →
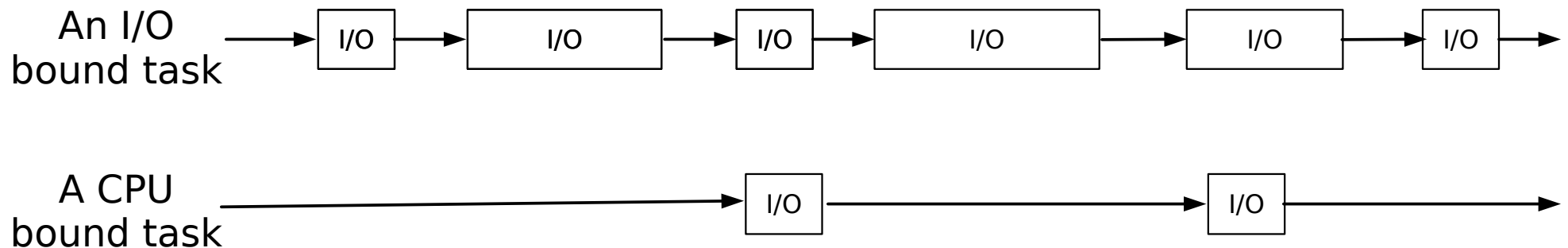
Task Execution

- A web crawler is an example of an I/O bound task
- Image processing is an example of a CPU bound task
- A single processor can run many I/O bound tasks concurrently as they spend most of their time waiting
- The basic unit for concurrent task execution is the "thread", an OS level unit of execution
- Multiple threads can run within a process
- Threads in Python are real system threads, the OS does the scheduling/thread switching
  - POSIX threads (pthreads) for Linux/Mac
  - Windows threads
- An alternative model for running multiple tasks concurrently is to use multiple processes

Processes

- Every program runs as a separate process

- Processes have "memory isolation"

- A process inherits an environment from the OS

- With multiple CPUs multiple processes can run concurrently

- Programs may launch "sub-processes", which inherit the same (or a modified) environment

- Sharing information between processes means "inter-process communication" of some kind (or shared memory)

- Both the subprocess and multiprocessing modules enable you to work with processes

- "Use processes not threads" is a common Python mantra

The Global Interpreter Lock

- Python has a Global Interpreter Lock, the GIL
- Only one thread of Python code executes at a time
- The GIL simplifies the interpreter and means fewer locks internally (faster single threaded code, the common case)
- The GIL makes threads suitable for I/O bound tasks only
- CPU bound tasks will run slower!
- C extensions may release the GIL and run concurrently
- For parallel processing use processes
- For I/O bound tasks, event loops and coroutines (green threading) are another alternative (asyncio)

Async

- **Not true concurrency**, tasks are run sequentially with an "event loop", asyncio by default
- Tasks are non-blocking coroutines
- Defined with async, called with await
- await adds a new task to the loop
- Any blocking calls block the whole event loop
- Under the hood there are special non-blocking system calls used by the event loop (*see methods on loops*)
- Also called green threading
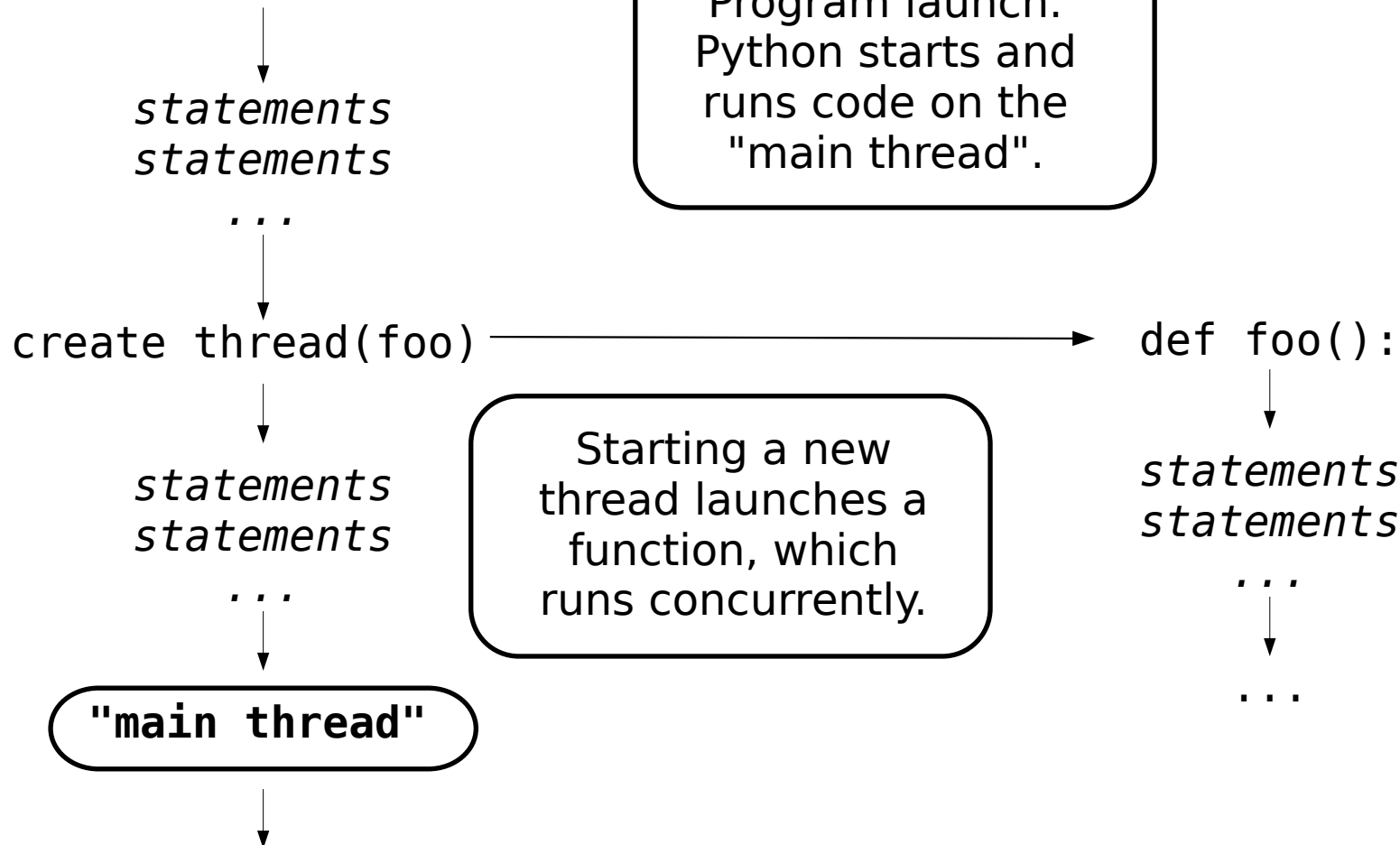- Suitable for I/O bound concurrency

Async Demo

```
>>> loop = asyncio.new_event_loop()
>>> async def first():
...  print("one")
...  print(await other())
...  print("three")
...
>>> async def other():
...  return "two"
...
>>> loop.run_until_complete(first())
one
two
three
```

Threads

- With threads we have multiple tasks running together within a program, each thread is independent but with access to shared resources

```
$ python program.py
```

↓

statements
statements

...

↓

create thread(foo) ──────────────→ def foo():

↓   ↓

statements   *statements*
statements   *statements*

...   *...*

↓   ↓

"main thread"   *...*

↓

> Program launch. Python starts and runs code on the "main thread".

> Starting a new thread launches a function, which runs concurrently.

Threads

- Threads run independently until they terminate on return or program exit

- Many "thread primitives" (locks, events, queues, etc) allow them to communicate and synchronise

- No shared resources (the "actor model"/CSP) is a way to minimise locking and maximise sanity

- The GIL ensures only one thread is actually running Python code at any time

- The GIL is released/acquired every hundred interpreter "ticks" or so (around bytecode boundaries)

- We use the threading module to access threads

Functions as Threads

- How to launch a function in a thread

```python
import threading
import time

def countdown(count):
    while count > 0:
        print("Counting down", count)
        count -= 1
        time.sleep(5)

>>> t1 = threading.Thread(target=countdown, args=(10,))
>>> t1.start()
Counting down 10
>>> Counting down 9
Counting down 8
Counting down 7
```

- countdown runs in a thread. We can still execute code in the interactive interpreter ("main thread") whilst it runs

Joining a Thread

- Once you start a thread it runs independently until it returns or the program exits

- Threads don't return a value, use a queue

- Use t.join() to wait for a thread to exit

```
t.start()        # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()         # blocks until t exits
```

- This only works from *another* thread

- A thread can't join itself!

Thread Termination

- Python has no built in support for thread cancellation

- Sometimes programs with threads can only be force killed! (kill -9)

- The normal solution is periodic polling and a terminate flag

- You can cancel threads at the OS level using the ctypes library and this solution (but it isn't recommended):

https://stackoverflow.com/questions/323972/is-there-any-way-to-kill-a-thread-in-python

Race Conditions

- Threads can introduce great complexity
- Race conditions deadlocks, livelocks, resource contention, etc...
- You will need locks to access shared resources to avoid race conditions

Executing concurrently

```
x = 9


statements

statements

...

x += 1
```

race condition

```
Thread 1

x is 9

...

x is 9
+=1
x is 10
```

```
Thread 2

x is 9

...

x is 9
+=1
x is 10
```

- If two threads read x as 9 at the same time and both add 1 then x will be 10 when it should be 11

Mutex Locks

- Thread safe concurrent access to resources with locks

```python
import threading

m = threading.Lock()
with m:             # Acquires the lock (or blocks)
    statements
    statements
    ...
                    # Lock is released

statements
```

- Only one thread can acquire the lock at a time

- Others will block waiting for it to be released

- with automatically acquires and releases the lock

- All access to shared resources must be protected with locks

Coordinating Threads

- Synchronising threads or communicating between them can be done with an **Event**
  - Events have an internal flag that can be **set/clear** and other threads can **wait** on it being set
- Other useful thread specific primitives:
  - Re-entrant locks
  - Queues (useful for implementing the actor model)
  - Conditions
  - Semaphores
  - Thread local variables

See also: `concurrent.futures.ThreadPoolExecutor` is a high level interface to push tasks to a background thread.

Processes

- For concurrent task execution with processes we use the multiprocessing module

- Provides a similar API and primitives to the threading module

- Uses fork on Linux

- On Windows tasks and their environment are pickled and sent to the new process (so higher startup cost)

- Using processes bypasses the problem of the GIL (one GIL per process)

Multiprocessing

- Creating a process and sending it a task is easy
- The task must be "importable" by the new process
- Processes have useful methods like join/kill/is_alive, etc

```python
import multiprocessing

def worker(i):
    print(f'Worker {i}')

if __name__ == '__main__':
    for i in range(3):
        p = multiprocessing.Process(target=worker, args=(i,))
        p.start()
```

```
$ python workerprocesses.py
Worker 0
Worker 1
Worker 2
```

Higher Level API: ProcessPool

- A process based worker pool

```
p = multiprocessing.Pool([num_processes])
```

- It executes functions in a subprocess
- It maintains a pool of workers to divide tasks between
- A high level API, no need to worry about the details
- Core pool operations:

```
p.apply(func [, args [, kwargs]])

p.apply_async(func [, args [, kwargs [,callback]]])
```

- apply blocks and waits for a result
- apply_async returns a result object you can poll for the result or wait on

Pool: apply()

- Running a function in another process

```python
import multiprocessing

def add(x, y):
    return x + y

if __name__ == '__main__':
    p = multiprocessing.Pool()
    r = p.apply(add, (3, 4))
    print(r)

$ python processingpool.py
7
```

- apply() runs the function in one of the subprocesses and waits for the result

Pool: apply_async()

- Running a function in another process without blocking
- Provides a handle (an ApplyResult) to fetch the result

```python
import multiprocessing
import time

def add(x, y):
    return x + y

if __name__ == '__main__':
    p = multiprocessing.Pool()
    r = p.apply_async(add, (3, 4)) # r is an ApplyResult object

    # We could call r.wait([timeout])
    while not r.ready():
        time.sleep(0.1)

    print(r.ready())
    print(r.successful())
    print(r.get())          # get has an optional timeout param
```

The Future: PEP 703

- Python 3.13: The GIL is optional.
- It's a compile time choice, extensions must be compiled against this version `--disable-gil`
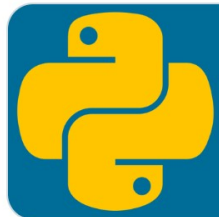- 5-8% slower for single threaded code

Soumith Chintala ✔
@soumithchintala · **Follow**

No More GIL!
the Python team has officially accepted the proposal.

Congrats @colesbury on his multi-year brilliant effort to remove the GIL, and a heartfelt thanks to the Python Steering Council and Core team for a thoughtful plan to make this a reality.

discuss.python.org
A Steering Council notice about PEP 703 (Making t…
Posting for the whole Steering Council, on the subject of @colesbury's PEP 703 (Making the Global …

6:34 AM · Jul 30, 2023

❤️ **4.7K**   💬 **Reply**   🔗 **Copy link**