# An Introduction to Concurrency in Python

Natalia Maniakowska

skygate

16-01-2020

# Outline

# What is concurrency, actually?

- Medieval Latin *concurrentia* "a running together" (cf. English *current*, *concur*)
- so: simultaneous occurrence. . .
- . . . but is it **really** simultaneous? What does is mean, anyway?

# What is concurrency, actually?

- Medieval Latin *concurrentia* "a running together" (cf. English *current*, *concur*)
- so: simultaneous occurrence. . .
- . . . but is it **really** simultaneous? What does is mean, anyway?

# What is concurrency, actually?

- Medieval Latin *concurrentia* "a running together"
  (cf. English *current*, *concur*)
- so: simultaneous occurrence...
- ...but is it **really** simultaneous? What does is mean, anyway?

# Concurrency vs. Parallelism

https://wiki.haskell.org/Parallelism_vs._Concurrency

- Not all programmers agree on the meaning of the terms 'parallelism' and 'concurrency'. They may define them in different ways or do not distinguish them at all.

**Definition**

A parallel program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more quickly. Different parts of the computation are delegated to different processors that execute at the same time (in parallel), so that results may be delivered earlier than if the computation had been performed sequentially.

**Definition**

In contrast, concurrency is a program-structuring technique in which there are multiple threads of control. Notionally the threads

# Concurrency vs. Parallelism

https://wiki.haskell.org/Parallelism_vs._Concurrency

- Not all programmers agree on the meaning of the terms 'parallelism' and 'concurrency'. They may define them in different ways or do not distinguish them at all.

**Definition**

A parallel program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more quickly. Different parts of the computation are delegated to different processors that execute at the same time (in parallel), so that results may be delivered earlier than if the computation had been performed sequentially.

**Definition**

In contrast, concurrency is a program-structuring technique in which there are multiple threads of control. Notionally the threads

# Concurrency vs. Parallelism

https://wiki.haskell.org/Parallelism_vs._Concurrency

- Not all programmers agree on the meaning of the terms 'parallelism' and 'concurrency'. They may define them in different ways or do not distinguish them at all.

**Definition**

A parallel program is one that uses a multiplicity of computational hardware (e.g. multiple processor cores) in order to perform computation more quickly. Different parts of the computation are delegated to different processors that execute at the same time (in parallel), so that results may be delivered earlier than if the computation had been performed sequentially.

**Definition**

In contrast, concurrency is a program-structuring technique in which there are multiple threads of control. Notionally the threads

# CPU-bound tasks: multiprocessing/parallel processing

- Simultaneous. Like, really simultaneous!
- Multiple processes, multiple CPU cores
- Good for: matrix multiplications, searching, image processing, etc.
- Problems: computation parallelization, sending data between processes

# CPU-bound tasks: multiprocessing/parallel processing

- Simultaneous. Like, really simultaneous!
- Multiple processes, multiple CPU cores
- Good for: matrix multiplications, searching, image processing, etc.
- Problems: computation parallelization, sending data between processes

# CPU-bound tasks: multiprocessing/parallel processing

- Simultaneous. Like, really simultaneous!
- Multiple processes, multiple CPU cores
- Good for: matrix multiplications, searching, image processing, etc.
- Problems: computation parallelization, sending data between processes

# CPU-bound tasks: multiprocessing/parallel processing

- Simultaneous. Like, really simultaneous!
- Multiple processes, multiple CPU cores
- Good for: matrix multiplications, searching, image processing, etc.
- Problems: computation parallelization, sending data between processes

# I/O-bound tasks: threading, asynchronicity

- Only one process
- Not *really* simultaneous: but can sometimes fake it quite well
- Good for: sending requests (to DB, to APIs), waiting for user input, etc.
- Problems:

# I/O-bound tasks: threading, asynchronicity

- Only one process
- Not *really* simultaneous: but can sometimes fake it quite well
- Good for: sending requests (to DB, to APIs), waiting for user input, etc.
- Problems:

# I/O-bound tasks: threading, asynchronicity

- Only one process
- Not *really* simultaneous: but can sometimes fake it quite well
- Good for: sending requests (to DB, to APIs), waiting for user input, etc.
- Problems:

# I/O-bound tasks: threading, asynchronicity

- Only one process
- Not *really* simultaneous: but can sometimes fake it quite well
- Good for: sending requests (to DB, to APIs), waiting for user input, etc.
- Problems:

# Threading

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Pre-emptive multitasking: the system (scheduler) interrupts threads at arbitrary moments, switches to another thread, and later resumes the stopped tasks
- No task cooperation is necessary
- Caveats: Race conditions! – random bugs. Thread-safety is important!

# Threading

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Pre-emptive multitasking: the system (scheduler) interrupts threads at arbitrary moments, switches to another thread, and later resumes the stopped tasks
- No task cooperation is necessary
- Caveats: Race conditions! – random bugs. Thread-safety is important!

# Threading

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Pre-emptive multitasking: the system (scheduler) interrupts threads at arbitrary moments, switches to another thread, and later resumes the stopped tasks
- No task cooperation is necessary
- Caveats: Race conditions! – random bugs. Thread-safety is important!

# Threading

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- Pre-emptive multitasking: the system (scheduler) interrupts threads at arbitrary moments, switches to another thread, and later resumes the stopped tasks
- No task cooperation is necessary
- Caveats: Race conditions! – random bugs. Thread-safety is important!

# Asynchronicity

- Event loop – definition
- Cooperative multitasking: tasks have to be programmed to yield control when they don't need system resources $\Rightarrow$ easier resources sharing
- What if a task does not cooperate? – blocking operations

# Asynchronicity

- Event loop – definition
- Cooperative multitasking: tasks have to be programmed to yield control when they don't need system resources $\Rightarrow$ easier resources sharing
- What if a task does not cooperate? – blocking operations

# Asynchronicity

- Event loop – definition
- Cooperative multitasking: tasks have to be programmed to yield control when they don't need system resources $\Rightarrow$ easier resources sharing
- What if a task does not cooperate? – blocking operations

# Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – . . . ?
- `asyncio` – from Python 3.5 on

# Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – ...?
- `asyncio` – from Python 3.5 on

# Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – ...?
- `asyncio` – from Python 3.5 on

# Example!

# Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection; (more about it in the next slide)
- The GIL is always released when doing I/O

# Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection; (more about it in the next slide)
- The GIL is always released when doing I/O

# Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection; (more about it in the next slide)
- The GIL is always released when doing I/O

# Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection; (more about it in the next slide)
- The GIL is always released when doing I/O

# Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection; (more about it in the next slide)
- The GIL is always released when doing I/O

# Global Interpreter Lock – continued

Reference counting: any reference to an object modifies it! (or at least its refcount)

```
> import sys
> a = []
> sys.getrefcount(a)
2
> b = a
> sys.getrefcount(a)
3
```

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

- the count needs protection against race conditions!
- Otherwise: memory leaks (never released) or incorrectly released memory, while a reference to the object still exist
- A solution? Add locks to all the objects that are shared between threads
- Consequences: decreased performance, and deadlocks!
- A better solution? A single lock: execution of any Python code requires acquiring the lock on the interpreter
- Consequences: easier, thread-safe, but any CPU-bound program becomes, effectively, single-threaded.
- Back then, when Python was a young language, it made it easy to add C extensions (didn't have to be thread-safe) – this helped make Python more popular

# Global Interpreter Lock – continued

*[O]nly the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (...). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.*

# Concurrency in Python – bibliography & further reading


- Jim Anderson, *"Speed Up Your Python Program With Concurrency"*, `https://realpython.com/python-concurrency/`
- David Beazley, *"An Introduction to Python Concurrency"*, presented at USENIX Technical ConferenceSan Diego, June, 2009. Slides available at `https://speakerd.s3.amazonaws.com/presentations/37707132332549 08b259542c4361e976/Concurrent.pdf`

# Concurrency in Python – bibliography & further reading


- Jim Anderson, *"Speed Up Your Python Program With Concurrency"*, `https://realpython.com/python-concurrency/`
- David Beazley, *"An Introduction to Python Concurrency"*, presented at USENIX Technical ConferenceSan Diego, June, 2009. Slides available at `https://speakerd.s3.amazonaws.com/presentations/3770713233254908b259542c4361e976/Concurrent.pdf`

# GIL – bibliography & further reading


- Abhinav Ajitsaria, *"What is the Python Global Interpreter Lock (GIL)?"* `https://realpython.com/python-gil/`
- Python Wiki, *"Global Interpreter Lock"*, `https://wiki.python.org/moin/GlobalInterpreterLock`
  Discussion of problems with GIL and why it has not been eliminated
- *"Thread State and the Global Interpreter Lock"*, `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`

# GIL – bibliography & further reading

- Abhinav Ajitsaria, *"What is the Python Global Interpreter Lock (GIL)?"* `https://realpython.com/python-gil/`
- Python Wiki, *"Global Interpreter Lock"*, `https://wiki.python.org/moin/GlobalInterpreterLock`
  Discussion of problems with GIL and why it has not been eliminated
- *"Thread State and the Global Interpreter Lock"*, `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`

# GIL – bibliography & further reading

- Abhinav Ajitsaria, *"What is the Python Global Interpreter Lock (GIL)?"* `https://realpython.com/python-gil/`
- Python Wiki, *"Global Interpreter Lock"*, `https://wiki.python.org/moin/GlobalInterpreterLock`
  Discussion of problems with GIL and why it has not been eliminated
- *"Thread State and the Global Interpreter Lock"*, `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`