An Introduction to Concurrency in Python

Natalia Maniakowska

skygate

16-01-2020

Outline

Outline

1. Introduction
   - Definitions
   - Different Types of Concurrency – More Details
2. Concurrency in Python
   - Example
3. GIL
4. References

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

Disclaimer: Not all programmers agree on the meaning!

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).

Definition

Concurrency is a program-structuring technique in which there are multiple threads of control. The user sees their effects interleaved. Whether they actually execute at the same time or not is an implementation detail.

Concurrency vs. Parallelism

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

Disclaimer: Not all programmers agree on the meaning!

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).

Definition

Concurrency is a program-structuring technique in which there are multiple threads of control. The user sees their effects interleaved. Whether they actually execute at the same time or not is an implementation detail.

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

CPU-bound tasks – continued

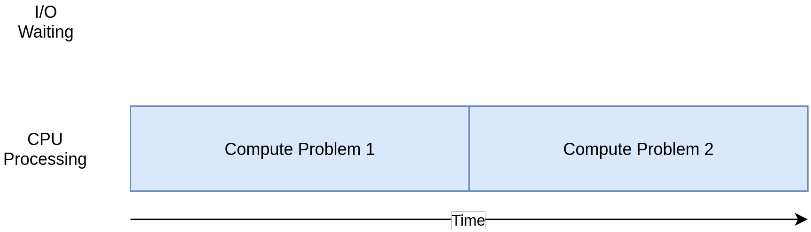

Figure: A diagram for a CPU-intensive program. Blue boxes show time when the program is doing work.

I/O-bound tasks: threading, asynchronicity

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

I/O-bound tasks: threading, asynchronicity

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

I/O-bound tasks: threading, asynchronicity

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

I/O-bound tasks – continued

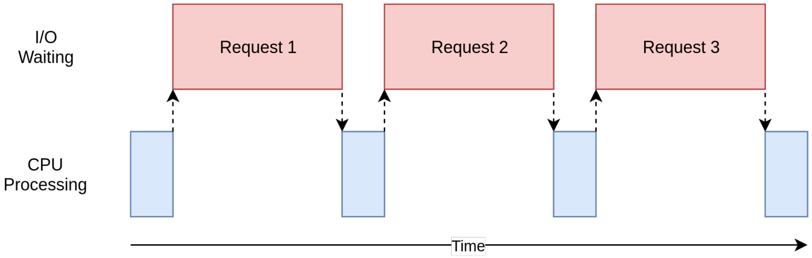

Figure: A diagram for a I/O-intensive program. Blue boxes show time when the program is doing work, the red boxes are time spent waiting for an I/O operation to complete (not to scale!).

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
- Caveats: race conditions $\Rightarrow$ random bugs. Thread scheduling is non-deterministic!

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

/Jason Whittington/

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
- Caveats: race conditions $\Rightarrow$ random bugs. Thread scheduling is non-deterministic!

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

/Jason Whittington/

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
- Caveats: race conditions $\Rightarrow$ random bugs. Thread scheduling is non-deterministic!

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

/Jason Whittington/

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
- Caveats: race conditions $\Rightarrow$ random bugs. Thread scheduling is non-deterministic!

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

/Jason Whittington/

Threading – continued

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Figure: Race condition: when a program depends on the timing of the program's threads. Here, two threads modify a global variable without locking or synchronisation.

Asynchronicity

- **Event loop** – waits for and dispatches events or messages in a program
- **Cooperative multitasking:** tasks have to be programmed to yield control when they don't need system resources ⇒ easier resources sharing
- What if a task does not cooperate? ⇒ blocking operations
- Sometimes more tricky than threading

Asynchronicity

- **Event loop** – waits for and dispatches events or messages in a program
- **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? $\Rightarrow$ blocking operations
- Sometimes more tricky than threading

Asynchronicity

- **Event loop** – waits for and dispatches events or messages in a program
- **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? $\Rightarrow$ blocking operations
- Sometimes more tricky than threading

Asynchronicity

- **Event loop** – waits for and dispatches events or messages in a program
- **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? $\Rightarrow$ blocking operations
- Sometimes more tricky than threading

Concurrent Programming

- **Rust** – focus on safe concurrency (memory safety)
- **Go** – goroutines, channels; parallel processing too
- **Elixir** – shared nothing concurrent programming (Actor model)
- ...

Concurrent Programming

- **Rust** – focus on safe concurrency (memory safety)
- **Go** – goroutines, channels; parallel processing too
- **Elixir** – shared nothing concurrent programming (Actor model)
- ...

Concurrent Programming

- **Rust** – focus on safe concurrency (memory safety)
- **Go** – goroutines, channels; parallel processing too
- **Elixir** – shared nothing concurrent programming (Actor model)
- . . .

Concurrent Programming

- **Rust** – focus on safe concurrency (memory safety)
- **Go** – goroutines, channels; parallel processing too
- **Elixir** – shared nothing concurrent programming (Actor model)
- ...

Outline

Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – well. . . ?
- `asyncio` – from Python 3.4 on; for I/O bound tasks

Pycharm: concurrency diagrams (reveals locking issues, but omits GIL)

Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – well. . . ?
- `asyncio` – from Python 3.4 on; for I/O bound tasks

Pycharm: concurrency diagrams (reveals locking issues, but omits GIL)

Built-ins

- `multiprocessing` – for CPU-bound tasks
- `threading` – well. . . ?
- `asyncio` – from Python 3.4 on; for I/O bound tasks

Pycharm: concurrency diagrams (reveals locking issues, but omits GIL)

Example

multiprocessing

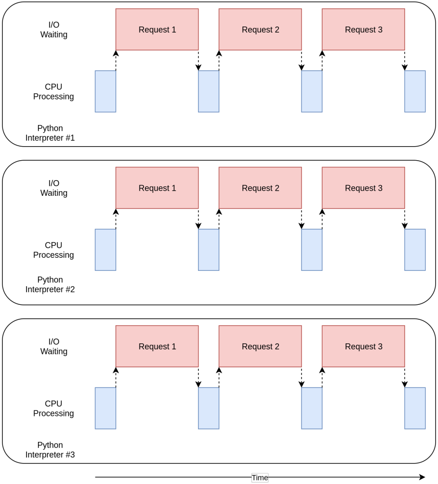

Figure: A multiprocessing version of an I/O-intensive program.

threading

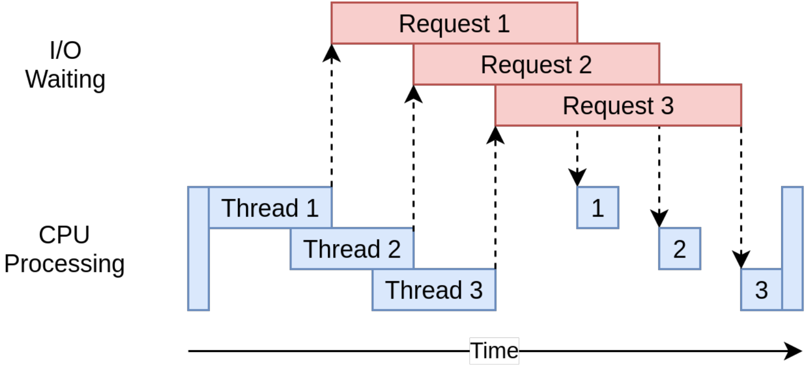

Figure: A `threading` version of an I/O-intensive program.

asyncio

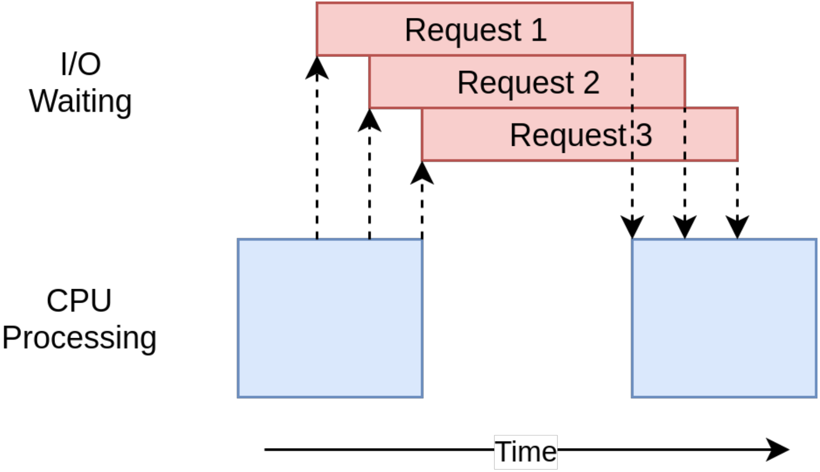

Figure: An `asyncio` version of an I/O-intensive program. All the I/O

Examples!

Let's see some code.

Outline

Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Design choice: Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection

Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Design choice: Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection

Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Design choice: Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection

Global Interpreter Lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time
- But why?!
- Design choice: Makes the object model implicitly safe against concurrent access
- Reference counting – instead of garbage collection

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 reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

- The reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

- The reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

- The reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

- The reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

- The reference 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, deadlocks, very difficult to develop and maintain
- **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

Global Interpreter Lock – continued

Is it really so bad?

- The GIL is always released when doing I/O.
- Threads can still be run in separate processes, sometimes.
- NumPy, TensorFlow, PyTorch – written to release the GIL when possible (or not use Python interpreter)
- "Premature optimization is the root of all evil." – Donald Knuth

Global Interpreter Lock – continued

Is it really so bad?

- The GIL is always released when doing I/O.
- Threads can still be run in separate processes, sometimes.
- NumPy, TensorFlow, PyTorch – written to release the GIL when possible (or not use Python interpreter)
- "Premature optimization is the root of all evil." – Donald Knuth

Global Interpreter Lock – continued

Is it really so bad?

- The GIL is always released when doing I/O.
- Threads can still be run in separate processes, sometimes.
- NumPy, TensorFlow, PyTorch – written to release the GIL when possible (or not use Python interpreter)
- "Premature optimization is the root of all evil." – Donald Knuth

Global Interpreter Lock – continued

Is it really so bad?

- The GIL is always released when doing I/O.
- Threads can still be run in separate processes, sometimes.
- NumPy, TensorFlow, PyTorch – written to release the GIL when possible (or not use Python interpreter)
- "Premature optimization is the root of all evil." – Donald Knuth

Global Interpreter Lock – continued

How to deal with GIL?

- multithreading → asyncio
- multithreading → multiprocessing (+ asyncio)
- CPU-intensive functions → Cython (no GIL!)
- Python → a faster language. . . ?

Global Interpreter Lock – continued

How to deal with GIL?

- multithreading $\rightarrow$ asyncio
- multithreading $\rightarrow$ multiprocessing (+ asyncio)
- CPU-intensive functions $\rightarrow$ Cython (no GIL!)
- Python $\rightarrow$ a faster language...?

Global Interpreter Lock – continued

How to deal with GIL?

- multithreading $\rightarrow$ asyncio
- multithreading $\rightarrow$ multiprocessing (+ asyncio)
- CPU-intensive functions $\rightarrow$ Cython (no GIL!)
- Python $\rightarrow$ a faster language. . . ?

Global Interpreter Lock – continued

How to deal with GIL?

- multithreading $\rightarrow$ asyncio
- multithreading $\rightarrow$ multiprocessing (+ asyncio)
- CPU-intensive functions $\rightarrow$ Cython (no GIL!)
- Python $\rightarrow$ a faster language. . . ?

Outline

Thank you!

That's it. Questions?

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`

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
- *"Thread State and the Global Interpreter Lock"*, https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock
- Christoph Heer, *"Is it me, or the GIL?"*, presented at EuroPython 2019 in Basel, Switzerland, July, 2019. Slides available at https://ep2019.europython.eu/media/conference/slides/Lj9n5pc-is-it-me-or-the-gil.pdf