SPEED UP PYTHON WITH CONCURRENCY

What you will learn:

1. Different types of concurrency
2. Concurrency with Python libraries:
   1. `threading`
   2. `asyncio`
   3. `multiprocessing`
3. When to use concurrency

VERSIONS

Note:

- Code samples were tested using: Python 3.8.5
- `asyncio` was introduced in Python 3.4
- `async` and `await` usage was added in Python 3.5 and made keywords in Python 3.7
- `asyncio.run()` was added in Python 3.7

CONCURRENCY

- Concurrency, or parallelism, is doing multiple computation tasks at a time
- Computing workloads are often I/O bound
    - Waiting on disk or network
    - Take advantage of this and work on something else
- Modern computers have multiple processors

PYTHON

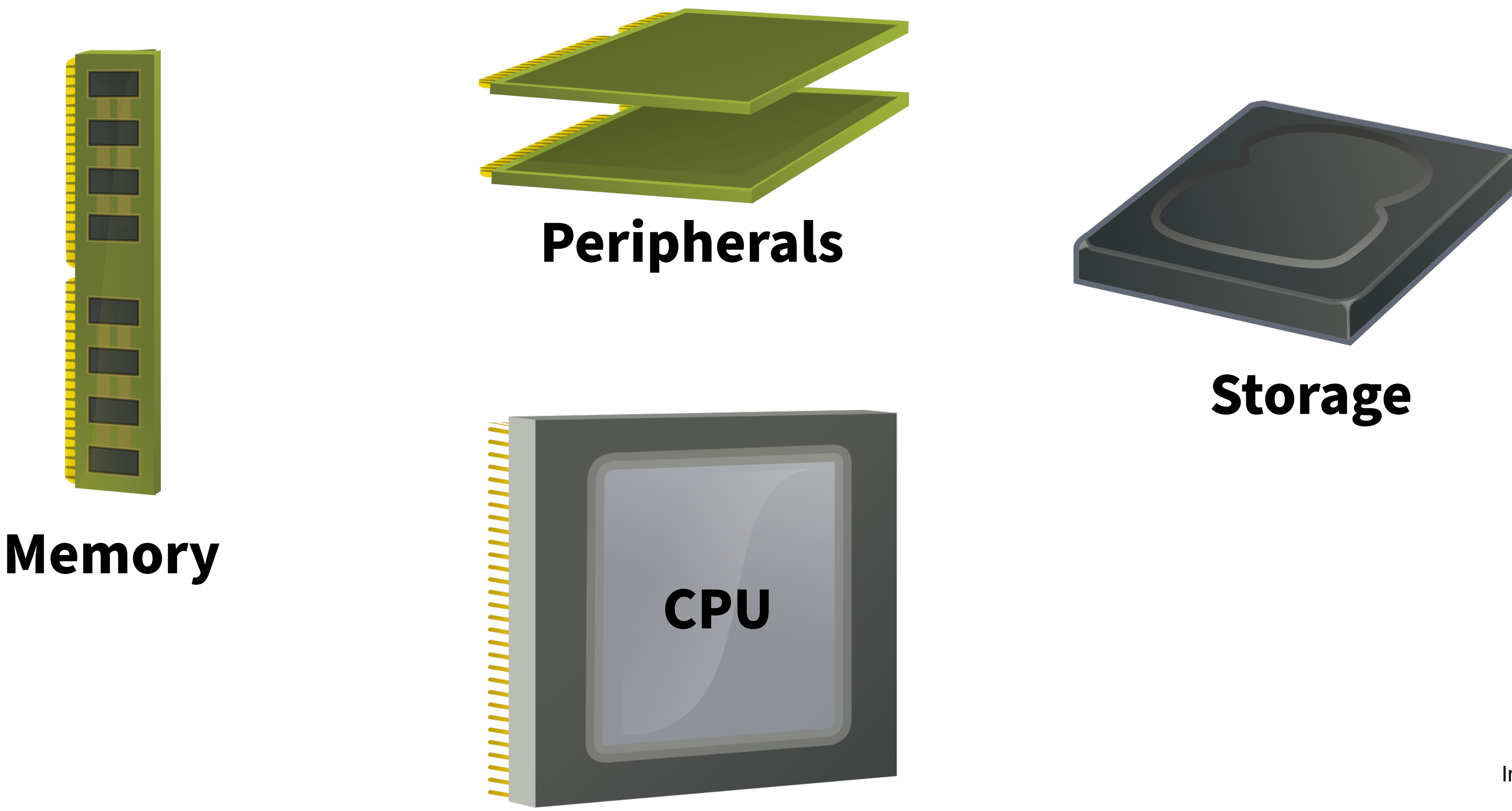
- Python provides three standard libraries for concurrency:
  - `threading`
  - `asyncio`
  - `multiprocessing`
- Global Interpreter Lock (GIL)

NEXT UP...

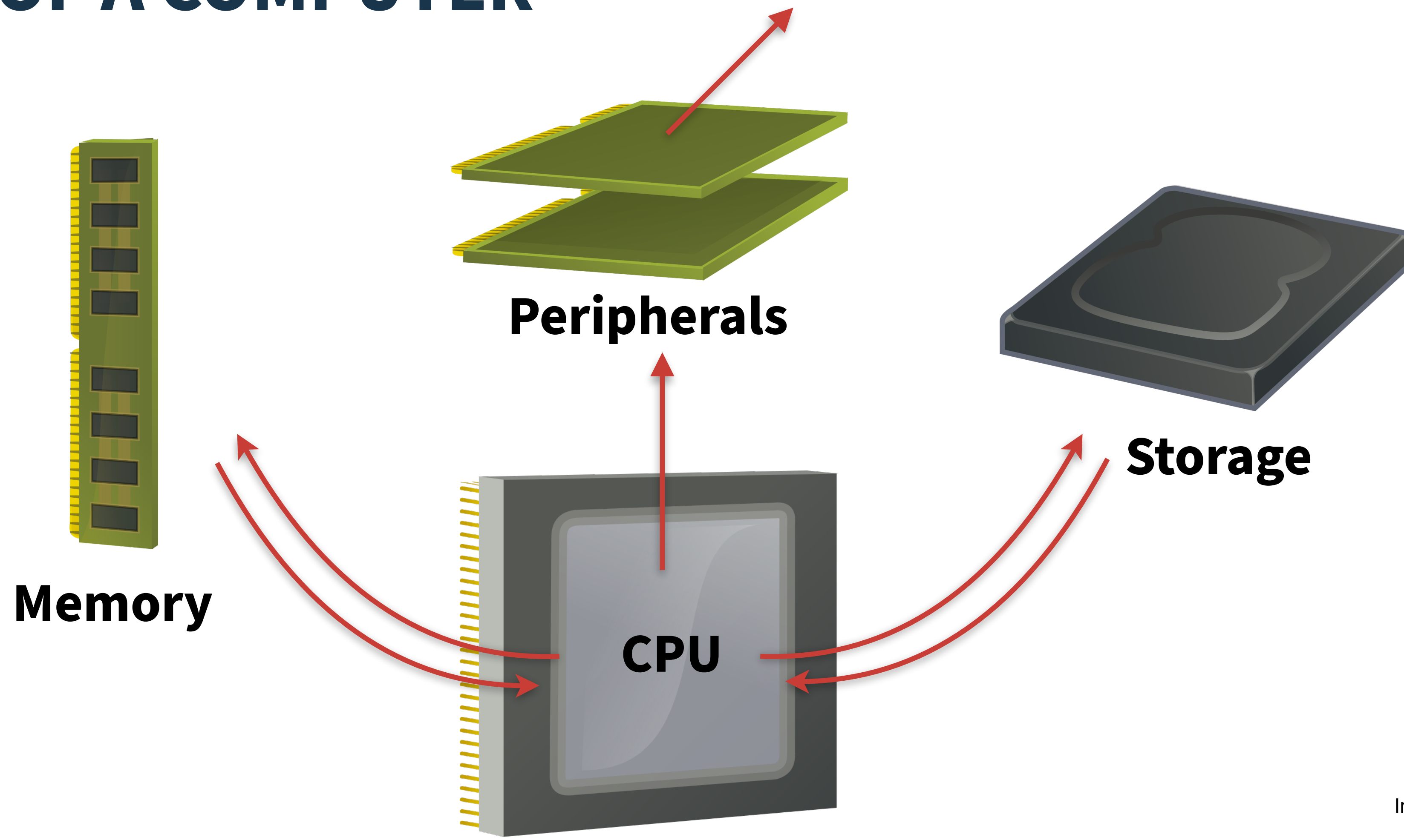
Latency in processing

TABLE OF CONTENTS

PARTS OF A COMPUTER



PARTS OF A COMPUTER



LATENCY

1 ns = 1/1,000,000,000 s

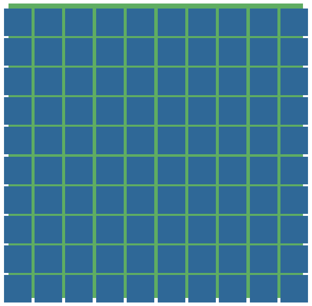
Intel i7 => 100 instructions

LATENCY

100 ns

Main memory reference

LATENCY



1 µs = 1000ns
Read 500kB from memory

x10 = 

LATENCY

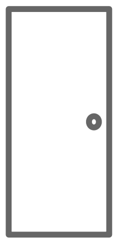
1 ms = 1000 μs

2 ms =~ Disk seek

150 ms =~ Ping time USA to Europe

x10 = 

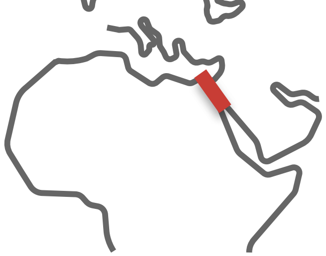
LATENCY



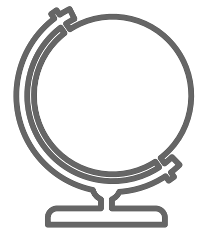
0.01 ns
1 cpu instruction
1 metre (=~ 1 yard)
Height of door knob

1 ns
100 cpu instruction
100 metre (=~ 100 yard)
1 football field

100 ns
1 memory reference
10km, 6 miles
1/4 of a Marathon



3 µs
Read 1MB from memory
300km, 186 miles
3x length Suez Canal



825 µs
Read 1MB from disk
82,500 km, ~ 51k miles
2x Earth Circumference

2 ms
Disk seek
200k km, 125k miles
½ Distance to Moon

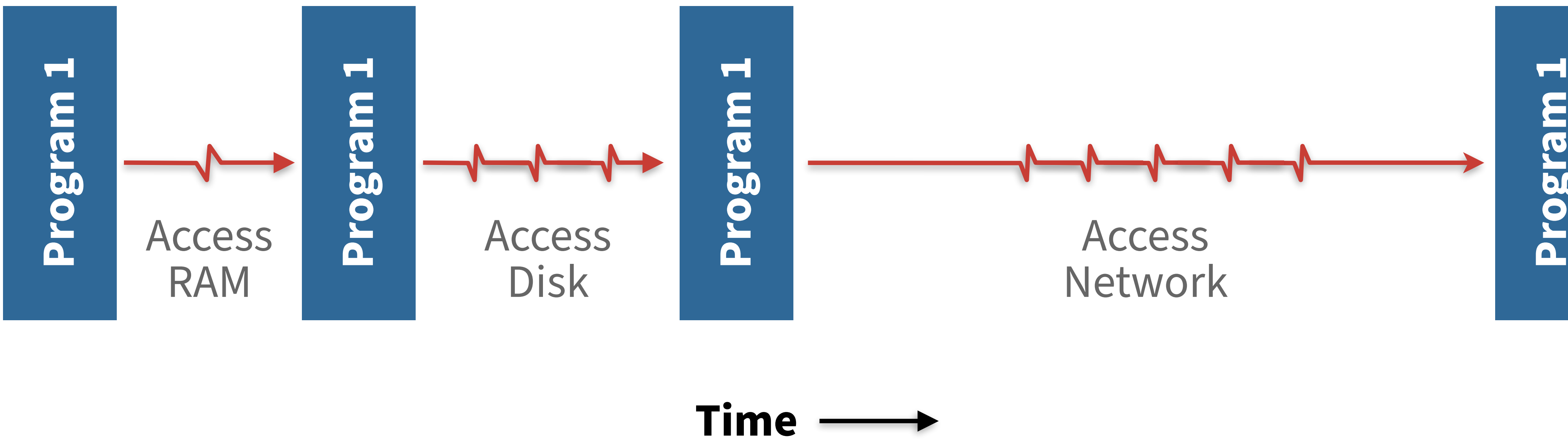
150 ms
Ping USA to Europe
15M km, ~10M miles
1/10 distance to sun

LATENCY

- If it took 1 second to perform 1 instruction

`Referencing Memory`	2 hours 47 minutes	10k instructions
`Disk Seek`	6 years, 4 months	200M instructions
`Seek + Read 1MB`	8 years, 11 months	285M instructions
`Ping Europe`	475 years, 8 months	15B instructions

HURRY UP AND WAIT

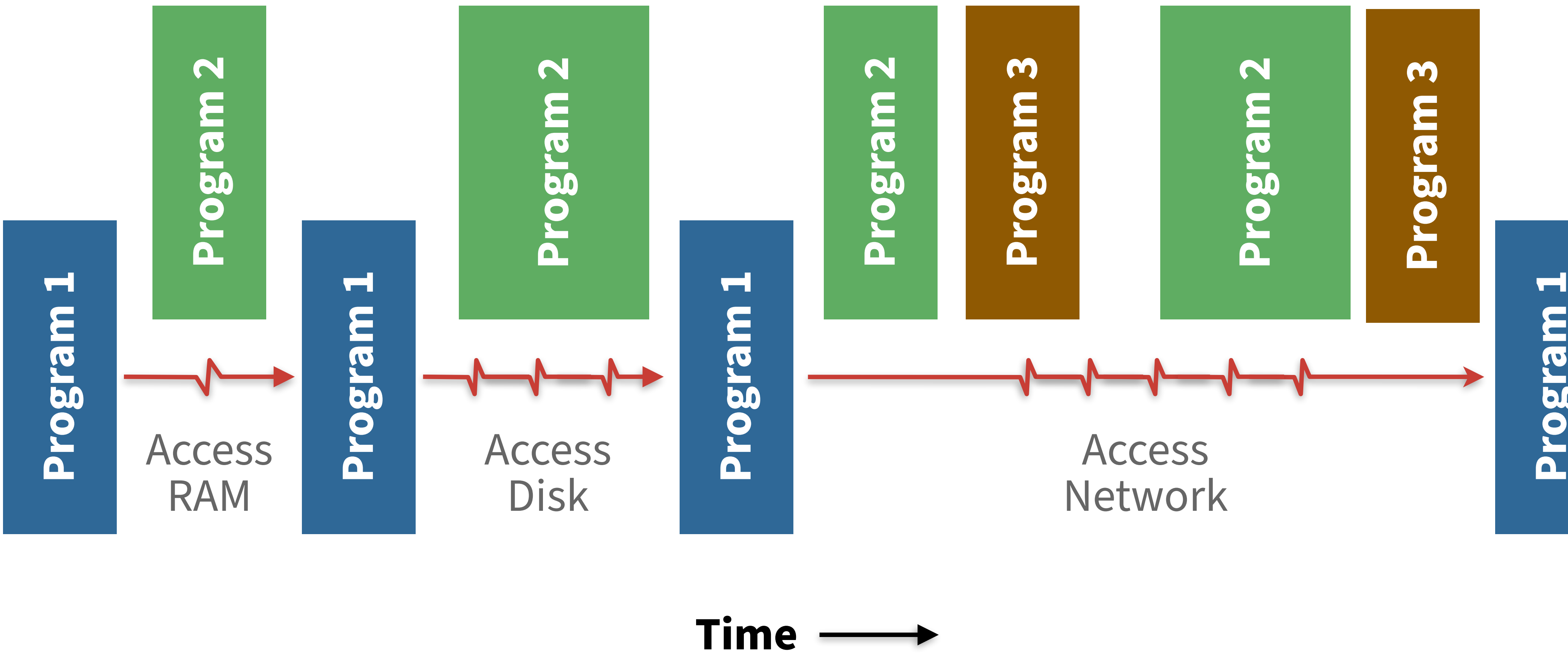


NEXT UP...

Types of concurrency

TABLE OF CONTENTS

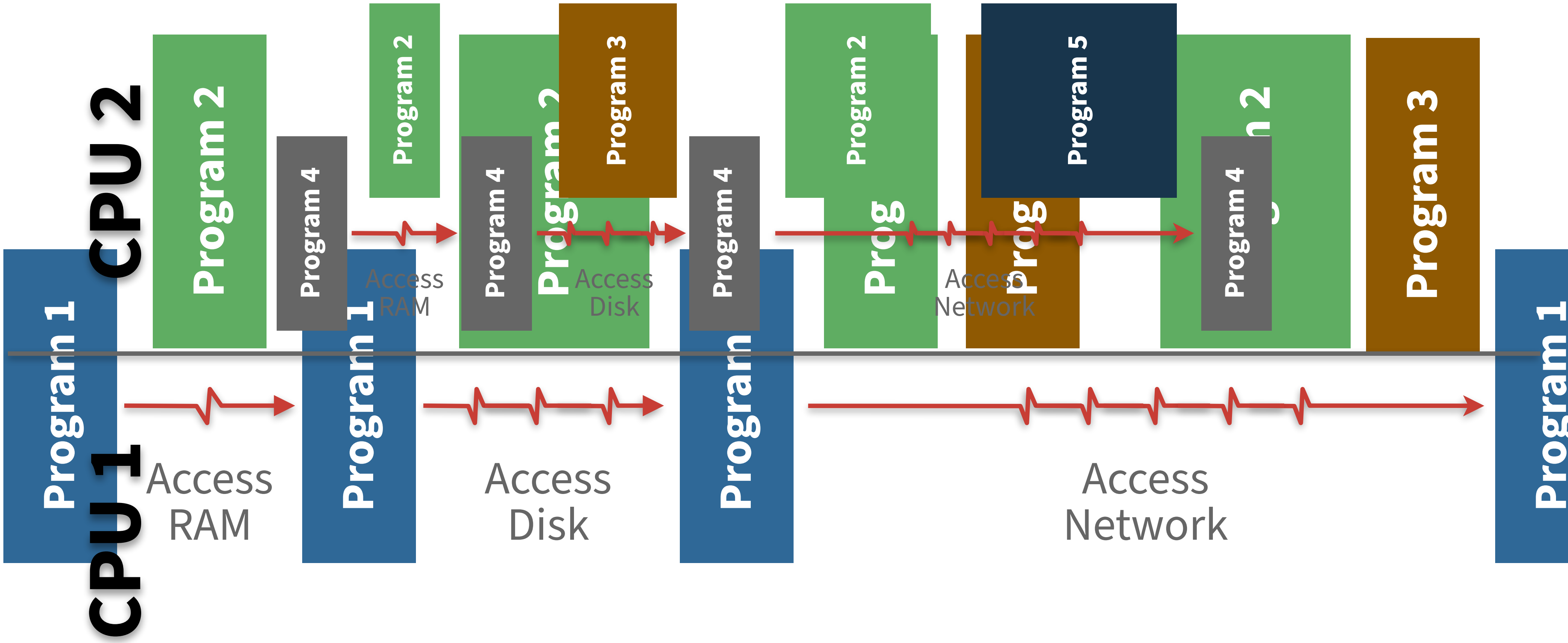
TIME SLICING



COOPERATIVE vs PRE-EMPTIVE MULTITASKING

- No multitasking:
  - DOS operating system
- Cooperative multitasking:
  - Program willing gives up CPU
  - Signals that it is going into a wait-state
  - Windows 3.1
- Pre-emptive multitasking:
  - Program can be interrupted by the Operating System
  - Mainframes, Unix based, Windows NT/95 and forward

MULTIPLE PROCESSORS



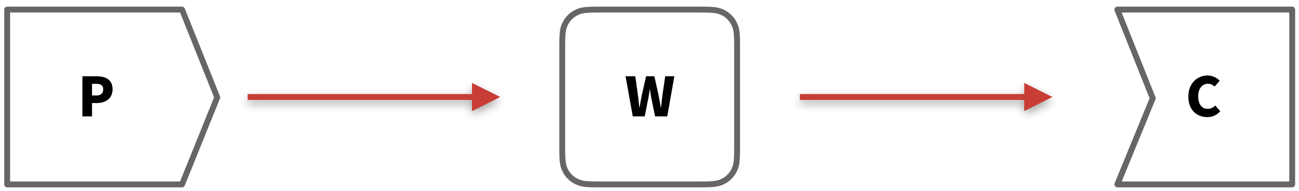
CONCURRENCY TYPES

- Not all algorithms can take full advantage of concurrency
- **Trivial concurrency**:
  - Comprised of activities that are independent of each other
  - No shared data
  - Example: a web server handling multiple clients at a time
- **Shared data concurrency**:
  - Software typically has three steps: input, compute, output
  - Splitting up the compute portion means that co-ordination is required at the input and output stages
  - May require co-ordination amongst compute nodes

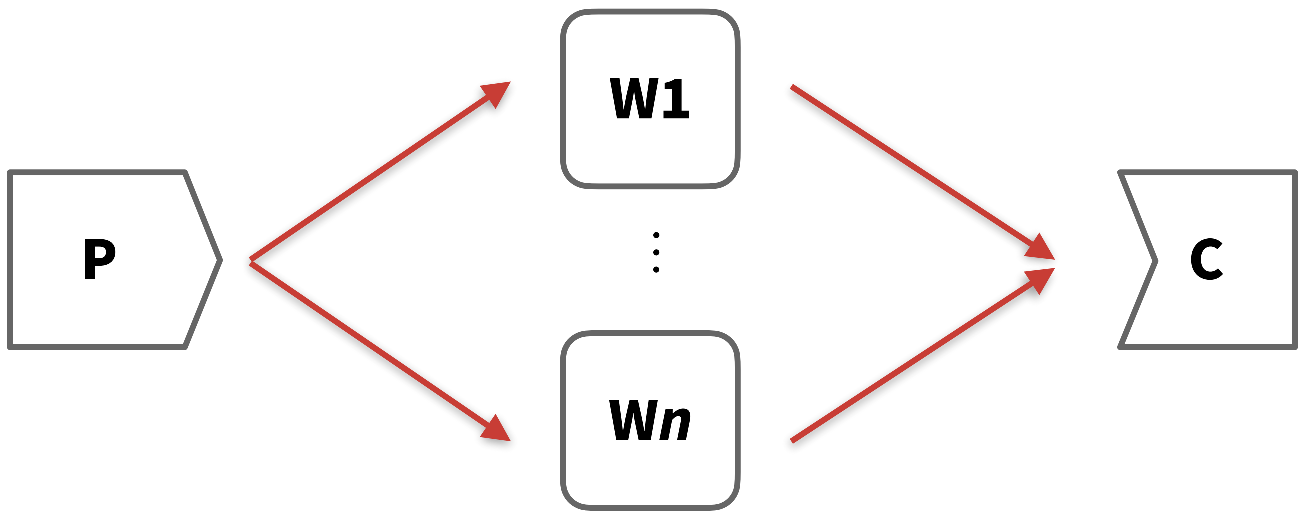
CONCURRENCY COMPONENTS

- Concurrent programs can often be categorized into three parts:
    1. **Producer:** component that produces data
    2. **Worker:** computation component that does work
    3. **Consumer:** component that consumes data
- These concepts can be mixed and matched

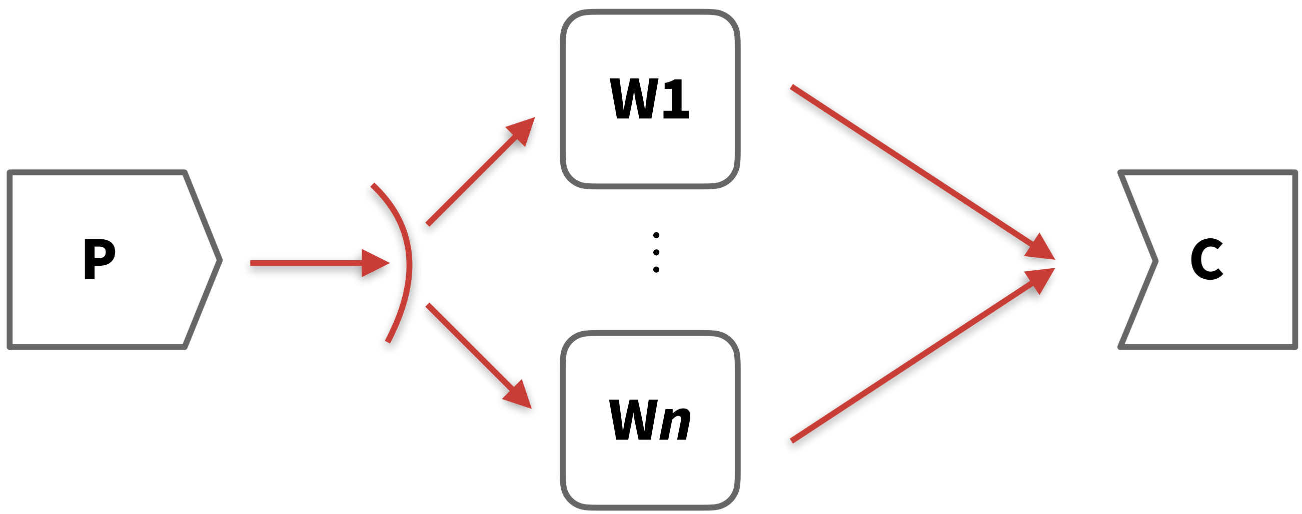
CONCURRENCY PATTERNS: PIPELINE



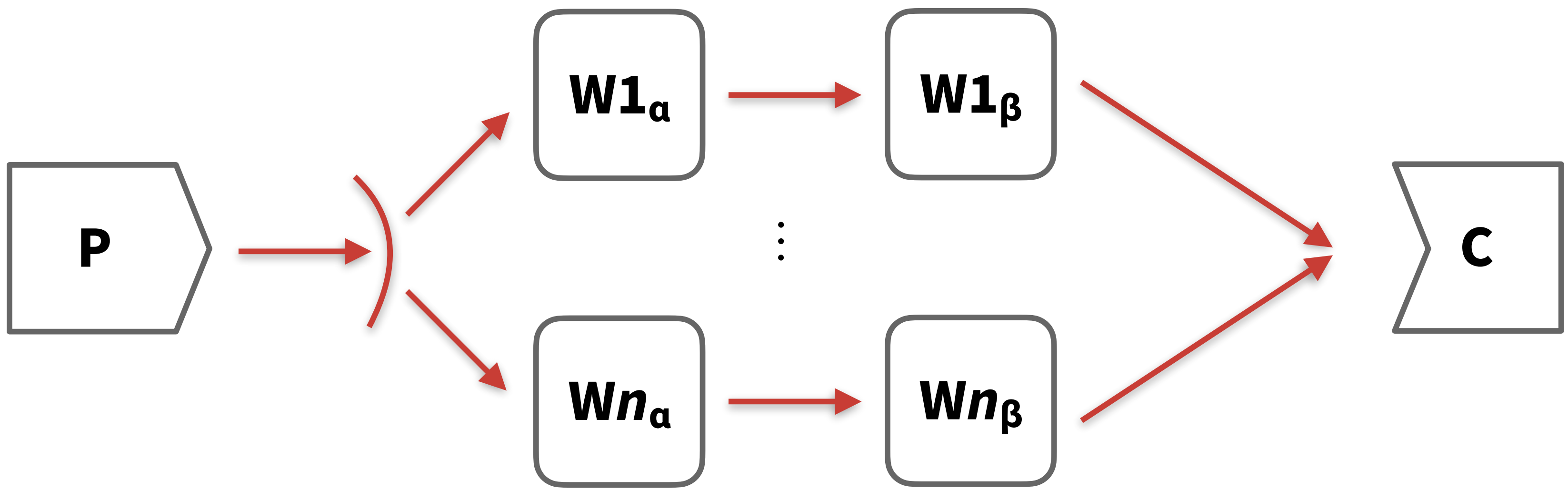
CONCURRENCY PATTERNS: N-WORKERS



CONCURRENCY PATTERNS: BROADCAST



CONCURRENCY PATTERNS: MIX-AND-MATCH



CONCURRENCY CHALLENGES

- Execution co-ordination: how to sync up different processes
- Memory allocation: which processes get what memory
- Scheduling: when are which processes active
- Throughput: managing above concepts to work done per unit time
- Distribution: threads, processes, machines
- Deadlocks: two or more components waiting on each other
- Resource Starvation: running out of memory, disk space, processes

CONCURRENCY IN PYTHON

- Operating system level
- Multi-processor
- Threads
- `asyncio`

PYTHON GIL

- Global Interpreter Lock
- Mutex (thread lock) ensuring only one thread controls the interpreter at a time
- Limits multi-threaded execution
- In place to prevent race conditions with memory and reference allocation
- Particularly important when Python interacts with C-extensions
- Lots of discussions on what to do about the GIL
    - Guido: only remove GIL if new code does not decrease the performance of a single-threaded program
    - CPython and PyPy thing! Jython and IronPython do not use a GIL

PEP 554

- “Multiple Interpreters in the Stdlib”
  `https://www.python.org/dev/peps/pep-0554/`
- CPython supports **subinterpreters**
- Subinterpreters are a feature at the C-extension level allowing for concurrency
- Interpreters are independent of each other
- PEP 554 proposes exposing these interpreters in the Python standard library
- Does not fix the GIL
- As changes around the GIL happen, they can be exposed to programmers earlier

NEXT UP...

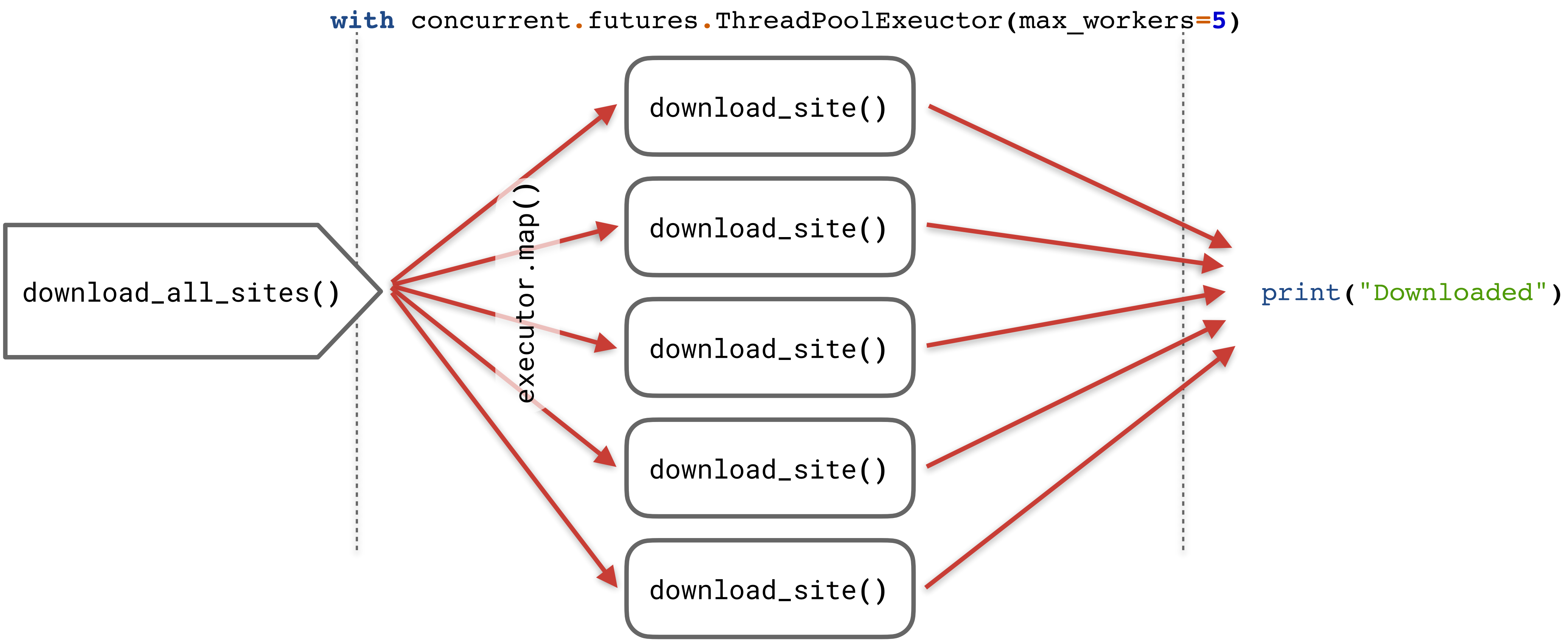
Python `threading` library

TABLE OF CONTENTS

I/O BOUND CONCURRENT PROGRAM

- Most programs spend a lot of time waiting for I/O
- Threads allow you to time slice your computation, doing processing work while waiting
- Threads work within the GIL
- Significant speed-up can result for disk or network heavy software

N-WORKERS PATTERN



THREAD SAFETY

- Memory is shared across threads
- Consider two threads using a single `requests.Session()` object
- Thread 1 starts downloading from Jython, gets interrupted
- Thread 2 tries to start downloading from RealPython, but the session object wasn't done
- Low-level primitives fix this using a mechanism called **locking**
- Higher level primitive: `threading.local()`
    - Looks like a global variable, but is actually created per thread
    - `get_session()` created a new `requests.Session()` object per thread

MORE THREADS FOR THE WIN?

- Thread pool size set to 5 even though downloading 160 URLs
- There is overhead creating threads
- There is overhead switching between threads
- Too many threads means code spends all its time managing threads

WHAT ABOUT?

- Basic thread primitives:

```
Thread.start()
Thread.join()
Queue
```

- The `concurrent.futures` library and **Executors** abstracts these away
- Introduced in Python 3.2

NEXT UP...

Race conditions

TABLE OF CONTENTS

CONCURRENCY IS HARD

- Shared memory and objects can be problematic
- Locks and `threading.local()` help, but you must remember to use them
- Most things aren’t thread safe!
    - `requests.Session()`
    - `print()`
- Better to assume not thread safe

NEXT UP...

Python `asyncio` library

TABLE OF CONTENTS

EVENT LOOPS

- `asyncio` was introduced in Python 3.4
- Independent of the operating system
- Event loop and coroutines
- Concurrency is achieved co-operatively with your tasks giving up their turns
- Keywords:
  - `async`
  - `await`

LIBRARIES

- `asyncio` still fairly new
- Libraries are just starting to take advantage of it
- Instead of `requests`, need `aiohttp`

```
$ python -m pip install aiohttp
...
Installing collected packages: multidict, async-timeout, yarl, aiohttp
Successfully installed aiohttp-3.6.2 async-timeout-3.0.1 multidict-4.7.6 yarl-1.5.1
```

ERROR HANDLING

```
await asyncio.gather(*tasks, return_exceptions=True)
```

- `return_exceptions` dictates what to do if something goes wrong
- Either:
    - Cause a normal exception (default)
    - Register the exception in the task object
- Tasks can be introspected
- `Task.result()` will return result of a coroutine or raise:
    - Exception caused by task
    - `CancelledError`
    - `InvalidStateError`

THREADS vs `aysncio`

- `asyncio` concurrency requires less overhead, tends to out-perform threads
- Coding with `asyncio` is slightly more complicated
- `asyncio` is still new
- Co-operative vs pre-emptive multitasking

NEXT UP...

Multiple processes

TABLE OF CONTENTS

MULTIPROCESSING

- Everything so far has been on a single CPU
- The `multiprocessing` library gives the ability to run on multiple CPUs
- Each CPU gets its own instance of the interpreter

MULTIPROCESSING

```
with multiprocessing.Pool(initializer=set_global_session) as pool:
    pool.map(download_site, sites)
```

- By default `Pool` gives you one process per CPU in your computer
- Each process has its own memory space, `initializer` is run per process within its local memory
- The `multiprocessing` library gives the ability to run on multiple CPUs
- Each CPU gets its own instance of the interpreter

MULTIPROCESSING vs THREADING

- Lots of overhead in creating a process
  - Operating System specific differences
  - Tends to require more memory
  - Tends to be slower to initialize
- Sharing information between processes must be done using explicit constructs:
  - `Queue` and `Pipe`
  - Shared memory: `Value` & `Array`
- Usually used to map processes to CPUs

NEXT UP...

CPU Bound computation

TABLE OF CONTENTS

I/O BOUND vs CPU BOUND WORKLOADS

- **I/O Bound**: waiting on input and output
- **CPU Bound**: waiting on computation
- Threading and `asyncio` only see speed-up in I/O bound cases
- CPU bound requires multiple CPUs

WHEN TO USE CONCURRENCY

Premature optimization is the root of all evil (or at least most of it) in programming.

-- Donald Knuth

- Concurrency introduces extra complications:
  - More code
  - Types of concurrency
  - Thread-safety and memory sharing
- Ask yourself: do you really need it?
- Decide on a model
- Favor `asyncio` over threads if you can

NEXT UP...

Summary

TABLE OF CONTENTS

SUMMARY

- I/O bound vs CPU bound computing
- Latency in I/O bound computing
- Concurrency patterns
  - Pipes
  - N-Workers
  - Broadcast

CONCURRENCY IN PYTHON

- Python includes in the standard library:
  - `threading`
  - `asyncio`
  - `multiprocessing`

CHOOSING CURRENCY

- Are you sure you need it?
- Is it I/O bound?
- Prefer `asyncio` over threading
- Careful with thread safety and shared memory

FURTHER READING

- Latency:
  - https://colin-scott.github.io/personal_website/research/interactive_latency.html
  - http://norvig.com/21-days.html
- Concurrency:
  - https://en.wikipedia.org/wiki/Concurrency_(computer_science)
- Python GIL:
  - https://realpython.com/python-gil/
  - https://wiki.python.org/moin/GlobalInterpreterLock
- Subinterpreters
  - https://www.python.org/dev/peps/pep-0554/
  - https://medium.com/@carreira.mktp/python-3-9-subinterpreters-and-c-extension-wars-f140f1460fd5

FURTHER READING

- Old school threading:
  - https://docs.python.org/3/library/threading.html
  - https://realpython.com/intro-to-python-threading/
- Threads and Futures:
  - https://docs.python.org/3/library/concurrent.futures.html
- `asyncio`:
  - https://realpython.com/async-io-python/
  - https://stackoverflow.com/questions/49005651/how-does-asyncio-actually-work/51116910#51116910
- Multi-processing:
  - https://docs.python.org/3/library/multiprocessing.html
  - http://zetcode.com/python/multiprocessing/

FURTHER READING

- Distributed computing:
  - Amazon Web Services, Google Cloud Platform, and Azure
  - `https://wiki.python.org/moin/DistributedProgramming`
  - `https://distributed.dask.org/en/latest/`
  - `https://celeryproject.org`

Thanks!

Dankie ju faleminderit faleminderit شكرا Grazias Շնորհակալություն Sağ ol eskerrik asko Дзякуй তোমাকে ধন্যবাদ hvala trugéré
благодаря Akeva Chezu ba gràcies Salamat zikomo 谢谢 hvala děkuji Tak dank u Dankon aitäh takk fyri salamat kiitos Merci
Grazas დიდი მადლობა Danke σας ευχαριστώ આભાર Mèsi poutèt ou Na gode Mahalo תודה Dhanyawaad köszönöm þakka þér
Daalụ terima kasih Go raibh maith agat grazie ありがとう matur nuwun ಧನ್ಯವಾದಗಳು សូមអរគុណអ្នក Kamsahamnida ຂໍຂອບໃຈທ່ານ
Lorem ipsum dolor paldies ačiū ви благодариме Mauruuru Dhanyawaadh Welálin баярлалаа
barka Ahéhee' Dhanyabaad miigwetch manana شکر از شما dziękuję obrigado ਤੁਹਾਡਾ ਧੰਨਵਾਦ mulțumesc спасибо tapadh leibh хвала
ďakujem hvala Waad ku mahadsan tahay Gracias Asante Tack Salamat rahmat நன்றி ధన్యవాదాలు ขอบคุณ tualumba teşekkür
ederim Спасибі آپ کا شکریہ rahmat cảm ơn bạn Diolch yn fawr אַ דאַנק Balika o ṣeun Ngiyabonga