Programming Principles in Python (CSCI 503/490)

Concurrency

Dr. David Koop

unittest

- Subclass from `unittest.TestCase`, write `test_*` functions
- Use `assert*` instance functions
-
```
import unittest

class TestOperators(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(3, 4), 7)

    def test_add_op(self):
        self.assertEqual(operator.add(3,4), 7)
unittest.main(argv=[''], exit=False)
```

Mock Testing

- Sometimes we don't want to actually execute all of the code that may be triggered by a particular test
- Examples: code that posts to Twitter, code that deletes files
- We can mock this behavior by substituting the actual methods with mockers
- Can even simulate side effects like having the function being mocked raise an exception signifying the network is done

Python Modules for Working with the Filesystem

- In general, cross-platform! (Linux, Mac, Windows)
- `os`: translations of operating system commands
- `shutil`: better support for file and directory management
- `fnmatch`, `glob`: match filenames, paths
- `os.path`: path manipulations
- `pathlib`: object-oriented approach to path manipulations, also includes some support for matching paths

Listing Files in a Directory

- Difference between file and directory
- `isfile`/`is_file` and `isdir`/`is_dir` methods
  - `os.path.isfile/isdir`
  - `DirEntry.is_file/is_dir`
  - `Path.is_file/is_dir`
- Test while iterating through
  -
```
from pathlib import Path
basepath = Path('my_directory/')
files_in_basepath = basepath.iterdir()
for item in files_in_basepath:
    if item.is_file():
        print(item.name)
```

[V. Ndlovu]

File Attributes

- Getting information about a file is "stat"-ing it (from the system call name)
- Names are similarly a bit esoteric, use documentation
- `os.stat` or use `.stat` methods on `DirEntry`/`Path`
- Modification time:
  - ```
    from pathlib import Path
    current_dir = Path('my_directory')
    for path in current_dir.iterdir():
        info = path.stat()
        print(info.st_mtime)
    ```
- Also can check existence: `path.exists()`

[V. Ndlovu]
    ```

Filename Pattern Matching

- `string.endswith/startswith`: no wildcards
- `fnmatch`: adds * and ? wildcards to use when matching (**not** just like regex!)
- `glob.glob`: treats filenames starting with . as special
  - can do recursive matchings (e.g. in subdirectories) using `**`
- `pathlib.Path.glob`: object-oriented version of glob
- ```
  from pathlib import Path
  p = Path('.')
  for name in p.glob('*.p*'):
      print(name)
  ```
- Also, can break apart paths:
  - `split/basename/dirname/join ~ parent/name/joinpath`

[V. Ndlovu]
  ```

Assignment 6

- Object-Oriented Programming
- Classes to create a library
  - Inheritance
  - Representations
  - Property
  - Exceptions
- Due this Friday, best to complete **before** the second test

Test 2

- This Wednesday, November 5, in class from 9:30-10:45am
- Similar Format to Test 1
- Emphasizes topics covered since Test 1, but still need to know core concepts from the first third of the course

Deleting Files and Directories

- Files: `os.remove` or `os.unlink`, or `pathlib.Path.unlink`
- ```
  from pathlib import Path
  Path('home/data.txt').unlink()
  ```
- Directories: `rmdir` or `shutil.rmtree`
  - `rmdir` only works if the directory is **empty**
  - **Careful:** this deletes the entire directory (and everything inside it)
    - `shutil.rmtree('my_documents/bad_dir')`

[V. Ndlovu]
  ```

Copying Files & Directories

- `shutil.copy`: copy file to specified directory
  - `shutil.copy('path/to/file.txt', 'path/to/dest_dir')`
- `shutil.copy2` preserves metadata, same syntax
- Copy entire tree: `shutil.copytree('data_1', 'data1_backup')`

[V. Ndlovu]

Moving and Renaming Files/Directories

- Moving files or directories:
  - `shutil.move('dir_1/', 'backup/')`
- Renaming files or directories:
  - `os.rename`
  - `pathlib.Path.rename`
  - ```
    data_file = Path('data_01.txt')
    data_file.rename('data.txt')
    ```

[V. Ndlovu]
    ```

Archives

- `zipfile`: module to deal with zip files
- `tarfile`: module to deal with tar files, can compress (`tar.gz`)
- Easier: `shutil.make_archive`
  - Specify base name, format, and root directory to archive
  - `shutil.make_archive('data/backup', 'tar', 'data/')`
- To extract, use `shutil.unpack_archive`

[V. Ndlovu]

Inspecting System Information

- `psutil`: cross platform library for information on running processes and system utilization
  - Monitor system
  - Profile system resources
  - Manage running processes

General System Information

- `cpu_count`: number of cpus
- `cpu_percent`:
- `virtual_memory`
- `swap_memory`
- `disk_partitions`
- `net_connections`
- `users`

Process Information

- Information on a specific process
- Identify process by its process id
  - `psutil.pids()`
  - `p = psutil.Process(7055)`
- Then, we can query process for information
  - `p.name()`
  - `p.exe()`
  - `p.cwd()`
  - `p.parent()`
  - `p.memory_info()`

Concurrency

What is concurrency?

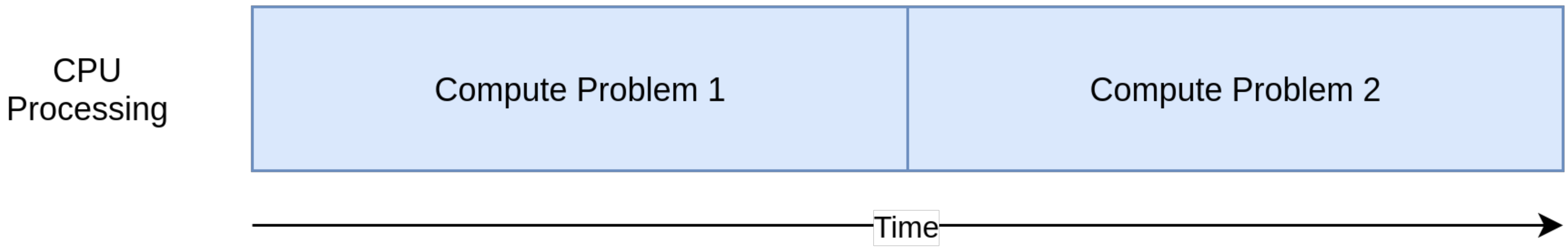
Why do we care about concurrency (multitasking and multiprocessing)?

Why concurrency?

- Speed:
  - Moore's Law and multiple cores
  - CPU-bound programs can use more cores
- Input/Output
  - Programs often sit waiting for data to load from disk/network

CPU-Bound

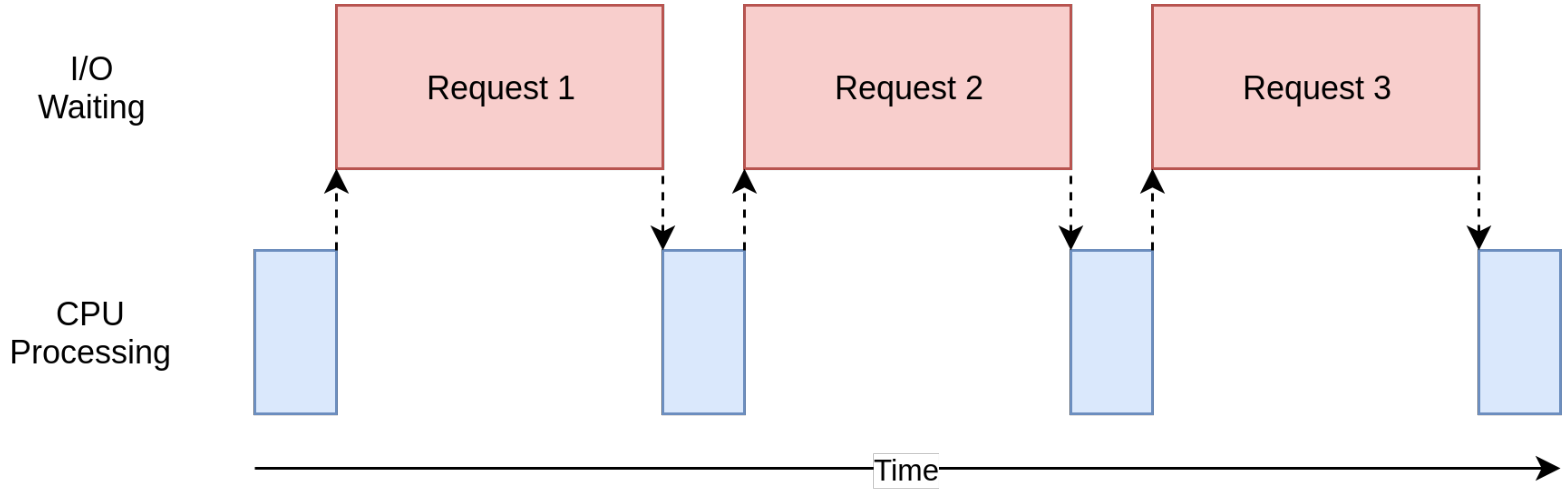
- Have to run each problem in sequence
- Wait for Problem 1 to finish before Problem 2 can start
- …even if they are totally separate problems!
- What if we could use another core for Problem 2?



[J. Anderson]

I/O-Bound

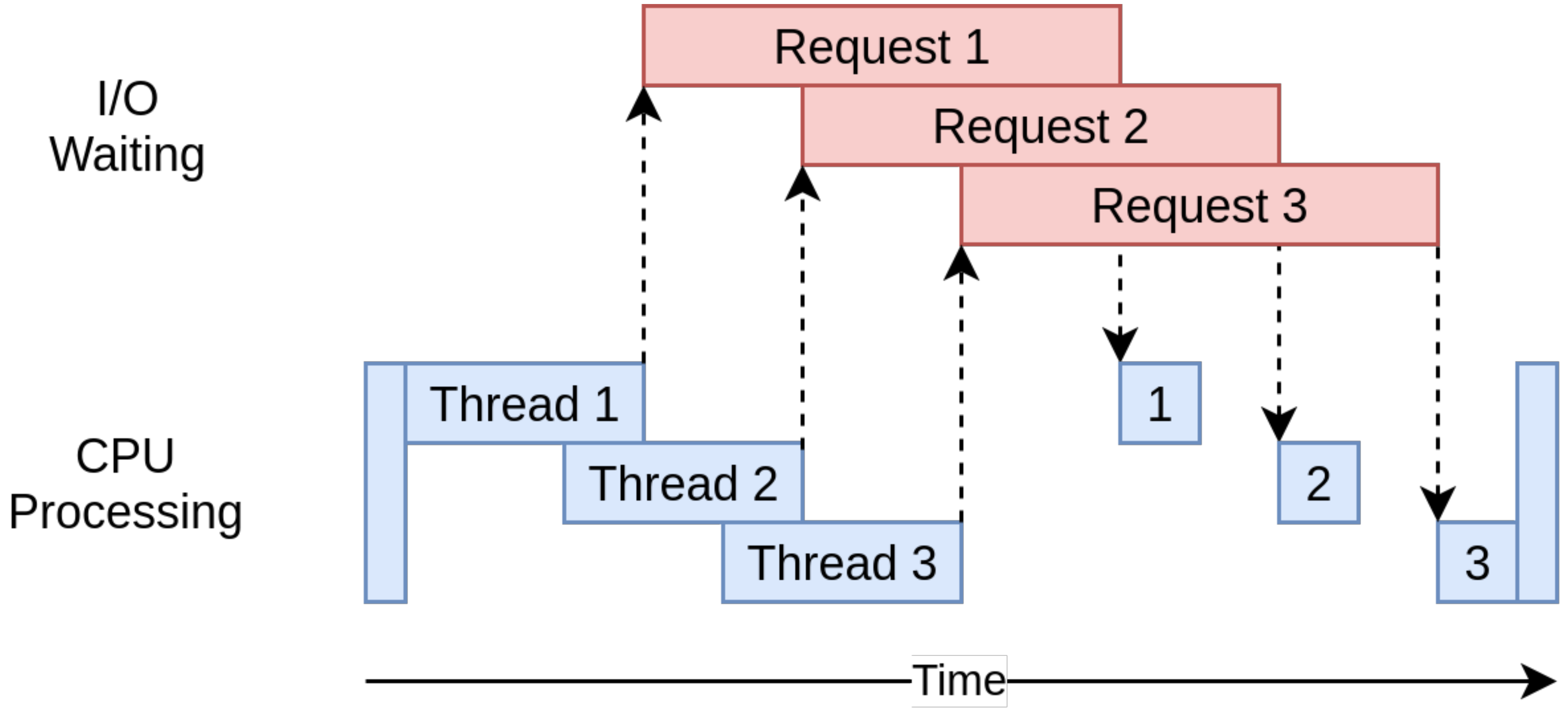
- Waiting for the file system or network to get data
- Nothing else happens while we wait for I/O to finish
- What if we could do something else while waiting for I/O?



[J. Anderson]

Threading

- Threading address the I/O waits by letting separate pieces of a program run at the same time
- Threads run in the same process
- Threads share the same memory (and global variables)
- Operating system schedules threads; it can manage when each thread runs, e.g. round-robin scheduling
- When blocking for I/O, other threads can run



[J. Anderson]

Threading Problem: Race Conditions

- Two threads, `T1` and `T2` that increment a variable `a = 42`
- We don't know when these threads will be **interrupted** by the OS
- `T1` reads the value of `a` (`42`)
  `T1` adds one and writes `a` (`43`) # `T1` finished
  `T2` reads the value of `a` (`43`)
  `T2` adds one and writes `a` (`44`) # `T2` finished
- `T1` reads the value of `a` (`42`) # `T1` INTERRUPT
  `T2` reads the value of `a` (`42`) # `T2` INTERRUPT
  `T1` adds one and writes `a` (`43`) # `T1` finished
  `T2` adds one and writes `a` (`43`) # `T2` finished
- Two different answers!

Threading Solution: Locking

- Ensure no two threads can access the same variable at the same time
- `T1` acquires a lock on `a`
  `T1` reads the value of `a` (`42`) # `T1` INTERRUPT
  `T2` waits for a lock on `a` # `T2` BLOCKED, sleeps
  `T1` adds one and writes `a` (`43`)
  `T1` releases lock on `a` # `T1` finished
  `T2` acquires a lock on `a`
  `T2` reads the value of `a` (`43`)
  `T2` adds one and writes `a` (`44`)
  `T2` releases lock on `a` # `T2` finished

Python and Threading

- ```
  import threading
  def printer(num):
      print(num)
  for i in range(5):
      t = threading.Thread(target=printer, args=(i,))
      t.start()
  ```
- Try this: you will likely see out-of-order outputs or weird formatting
- Why?
  ```

Python Locks

- ```
  my_lock = threading.Lock()
  def printer(num):
      with my_lock:
          print(num)

  for i in range(5):
      t = threading.Thread(target=printer, args=(i,))
      t.start()
  ```
- With statement provides context manager to acquire and release the lock
  ```

ThreadPoolExecutor

- Can be difficult to keep track of all threads
- Want to reuse threads instead of creating a new one each time
- Wait until all threads are done executing before next tasks
- `ThreadPoolExecutor` simplifies this
-
```
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=5) as executor:
    executor.map(printer, range(10))
```
- `max_workers` specifies the number of threads (can compute multiple times on one thread)
- `map` figures out how to assign the inputs to the threads

Python Threading Speed

- If I/O bound, threads work great because time spent waiting can now be used by other threads
- Threads **do not** run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
- Use threads when code is **I/O bound**, otherwise no real speed-up plus some overhead for using threads

Using multiple cores at once

- Python is linear/serial; only one thread executes at a time
- Python has **garbage collection**, releasing memory when not used
  - Requires keeping track of all objects by **reference counting**
  - ```
    a = {'IL','IN','OH'}
    b = {'states': a}
    ```
  - `{'IL','IN',OH'}` has a reference count of 2 (`a` and `b` both reference it)
- Problem: keeping track of references across different threads/processes
    ```

Python and the GIL

- Remember Python integrates other libraries, including those written in C
- Python was designed to have a thread-safe interface for C libraries (which were not necessarily themselves thread-safe)
- Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
- Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
- This effectively makes Python single-threaded (faster execution)
- Python requires threads to give up GIL after certain amount of time
- Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it

--disable-gil (No GIL Python)

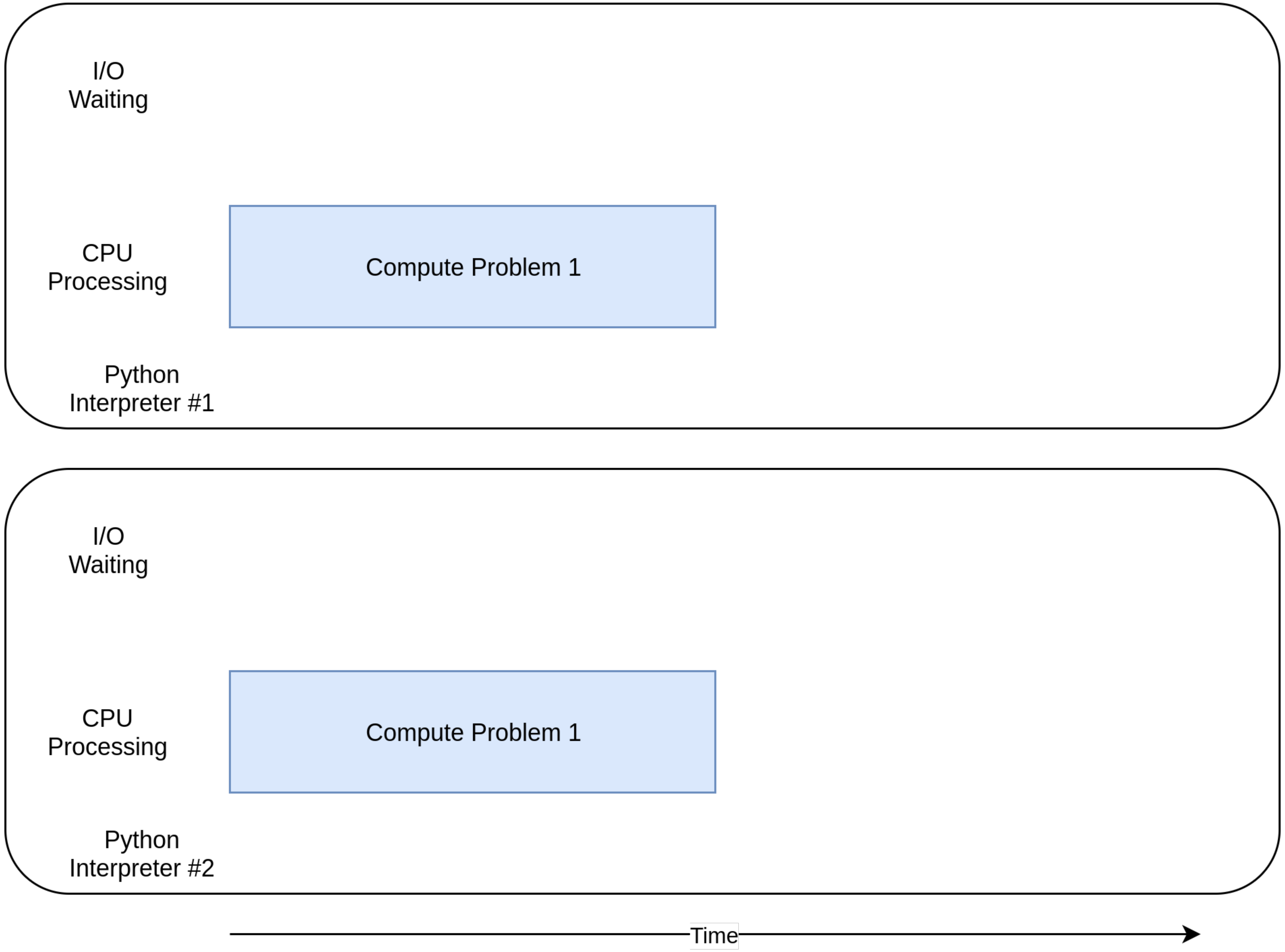
- GIL Problems:
  - Difficult to use multi-core CPUs effectively for scientific applications
  - GPU-Heavy workloads (AI) require effective multi-core CPU execution
  - Workarounds are complex, make libraries more difficult to use and maintain
- PEP 703: Making the Global Interpreter Lock Optional in Python
  - Use biased reference counting (most objects used by a single thread)
  - Change memory allocator to one that is thread-safe (pymalloc relies on GIL)
  - Use per-object locking for container thread safety
  - Updates to the garbage collector (non-generational) that also allow "stop-the-world" on threads

[PEP 703]

Multiprocessing

- Multiple processes do not need to share the same memory, interact less
- Python makes the difference between processes and threads minimal in most cases
- Big win: can take advantage of multiple cores!
- ```
  import multiprocessing
  with multiprocessing.Pool() as pool:
          pool.map(printer, range(5))
  ```
- **Warning**: known issues with running this in the notebook, use in scripts or look for alternate possibilities/library
- Set `__spec__ = None` to use the `%run` command in the notebook with a multiprocessing script
  ```

Multiprocessing address CPU-bound processes



[J. Anderson]

Multiprocessing using concurrent.futures

- ```
  import concurrent.futures
  import multiprocessing as mp
  import time

  def dummy(num):
      time.sleep(5)
      return num ** 2

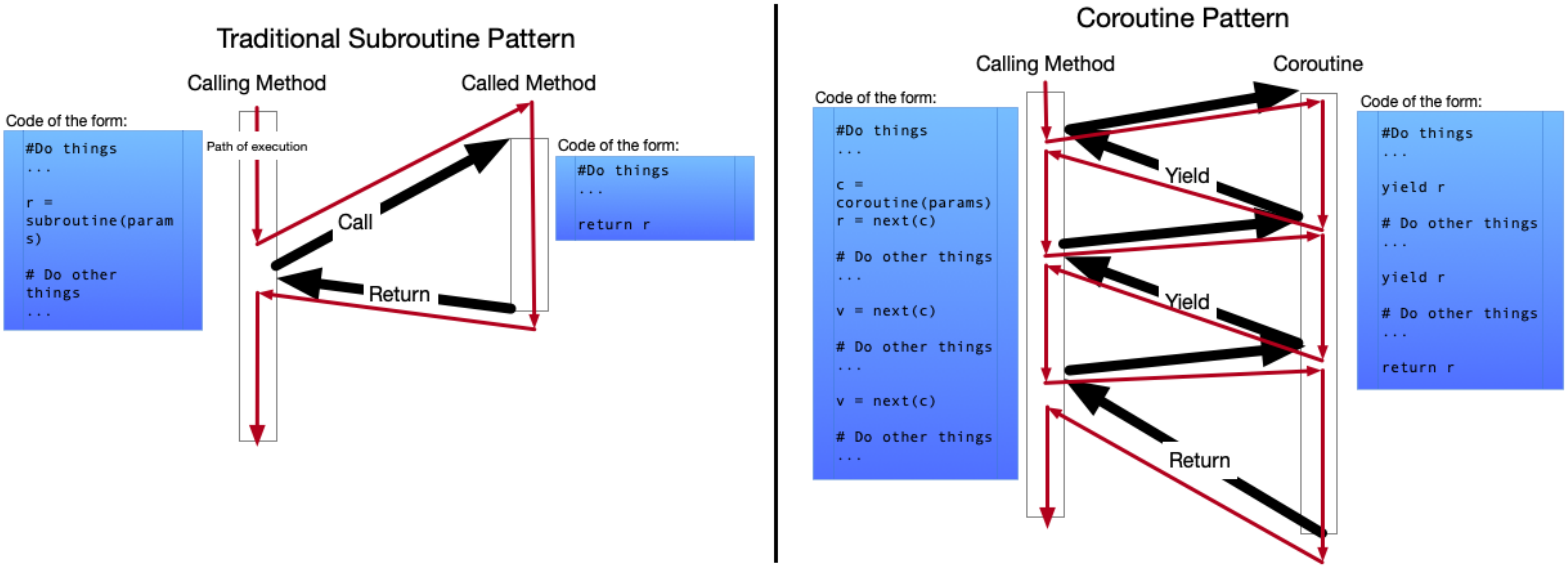
  with concurrent.futures.ProcessPoolExecutor(max_workers=5,
              mp_context=mp.get_context('fork')) as executor:
      results = executor.map(dummy, range(10))
  ```
- `mp.get_context('fork')` changes from `'spawn'` used by default in MacOS, works in notebook
  ```

When to use threading or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - Multithreading is your best bet because of its low overhead
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)

[J. Anderson]

Subroutines vs. Coroutines



[J. Weaver]

Generators basically do this!

- ```
  def random_numbers(start=1, end=1000):
      while True:
          yield random.randint(start, end)
  for x in random_numbers():
      print(x)
  ```
- The `yield` statements pause execution of the function and go back to the main function
- They are almost coroutines except you can't pass anything in
- Hard to have multiple things going on
  ```

asyncio

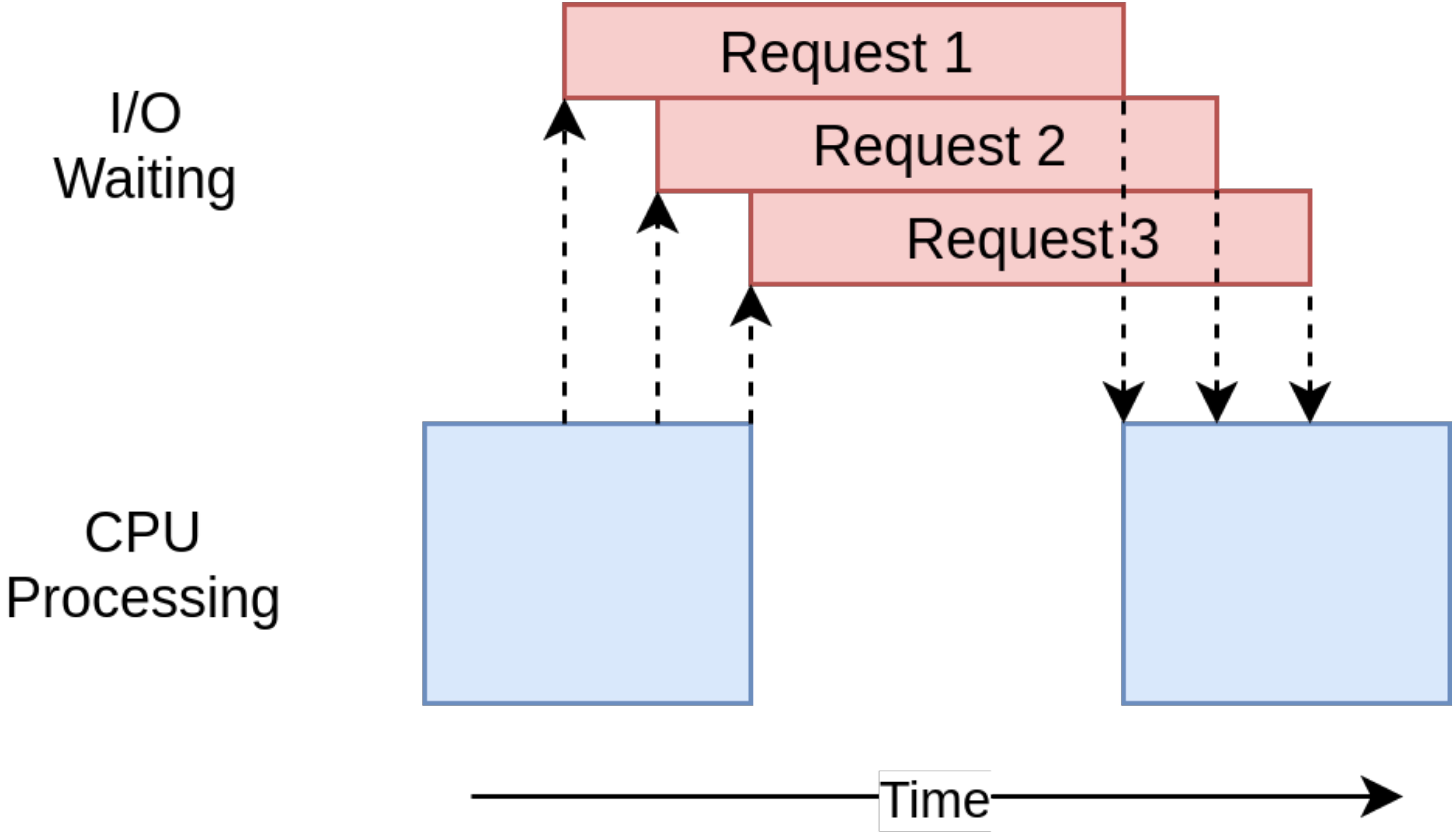
- Single event loop that controls when each task is run
- Tasks can be ready or waiting
- Tasks are **not interrupted** like they are with threading
  - Task controls when control goes back to the main event loop
  - Either waiting or complete
- Event loop keeps track of whether tasks are ready or waiting
  - Re-checks to see if new tasks are now ready
  - Picks the task that has been waiting the longest

[J. Anderson]

async

- `async` is a keyword that tells Python that the function uses await
- Also `async with` context manager
- ```
  async def download_site(session, url):
      async with session.get(url) as response:
          print("Read {0} from {1}".format(
                          response.content_length, url))
  ```
- `asyncio` uses a single thread
- Requires special libraries (`aiohttp`)
- Tends to have less overhead than multiprocessing
  ```

asyncio


Request 1
Request 2
Request 3
I/O
Waiting
CPU
Processing
Time

When to use threading, asyncio, or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - If there is library support, use asyncio
  - Otherwise, multithreading is your best bet (lower overhead)
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)

[J. Anderson]

Concurrency Comparison

Concurrency Type	**Switching Decision**	**Number of Processors**
Pre-emptive multitasking (`threading`)	The operating system decides when to switch tasks external to Python.	1
Cooperative multitasking (`asyncio`)	The tasks decide when to give up control.	1
Multiprocessing (`multiprocessing`)	The processes all run at the same time on different processors.	Many

[J. Anderson]