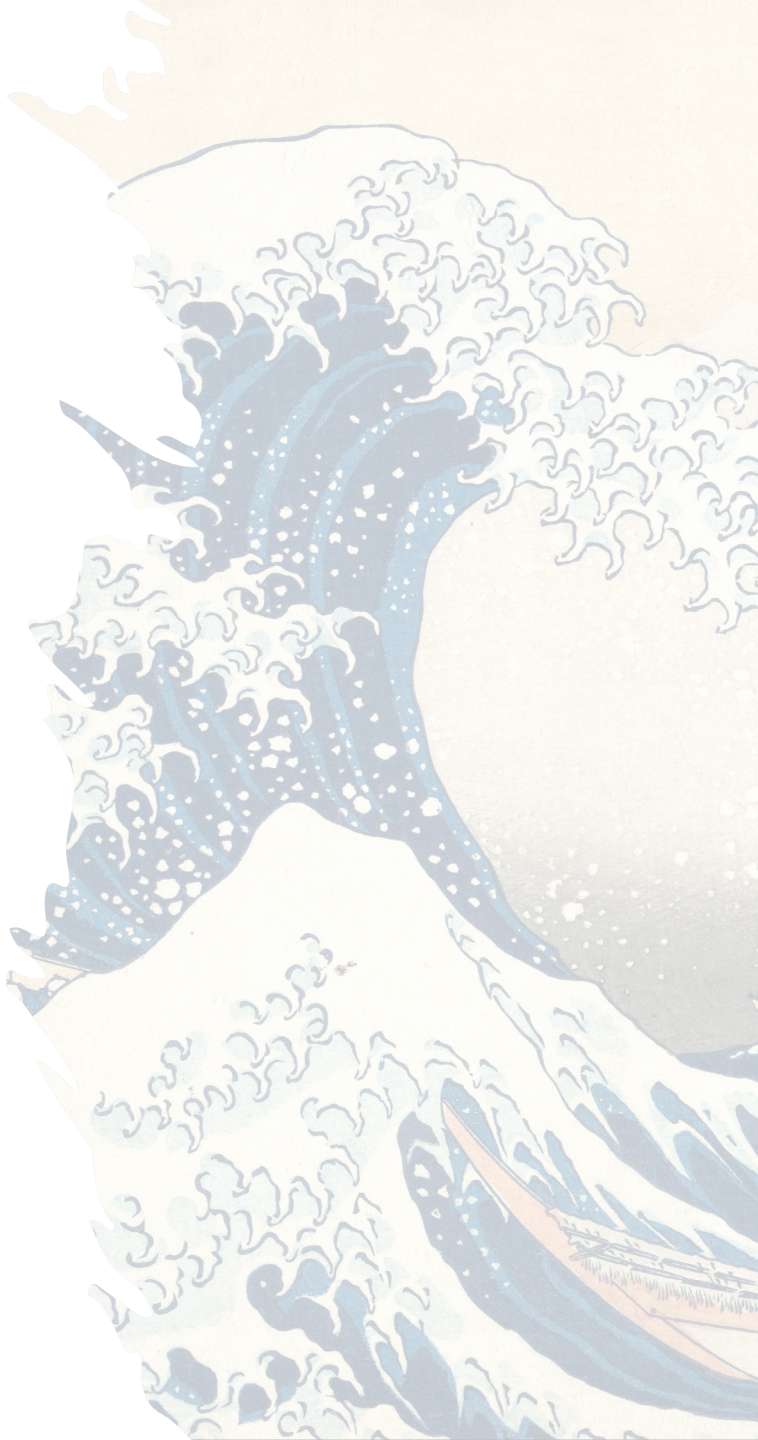
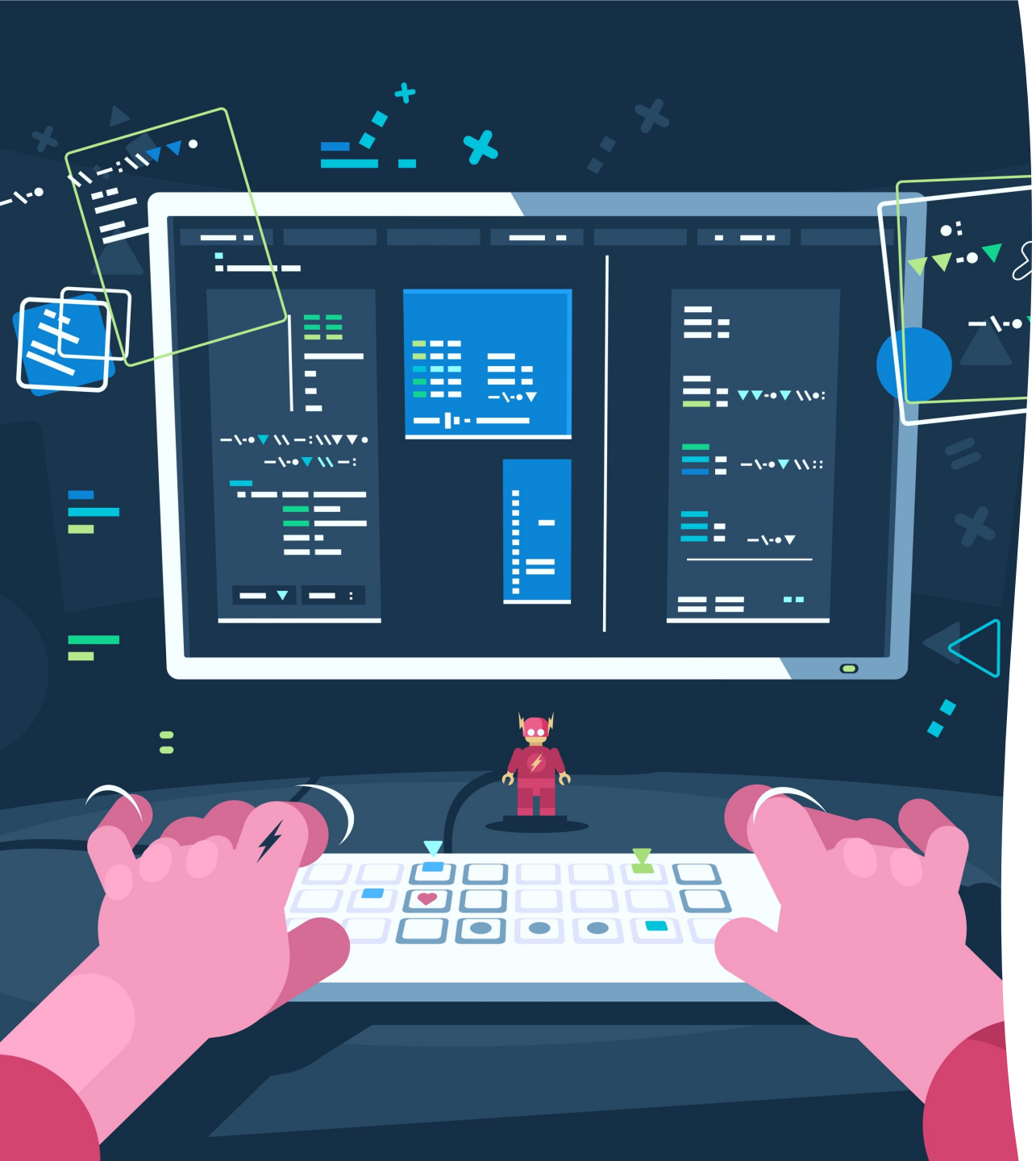
Stelios Sotiriadis

4. Parallel processing part 2

Agenda

- Introduction to threads with Python
  - Processes vs Threads
  - When to use?
- Critical section and synchronisation
- Introduction to asynchronous programming
- Lab 4 :
  - using the **`threading`** library
  - using a mutex and a semaphore



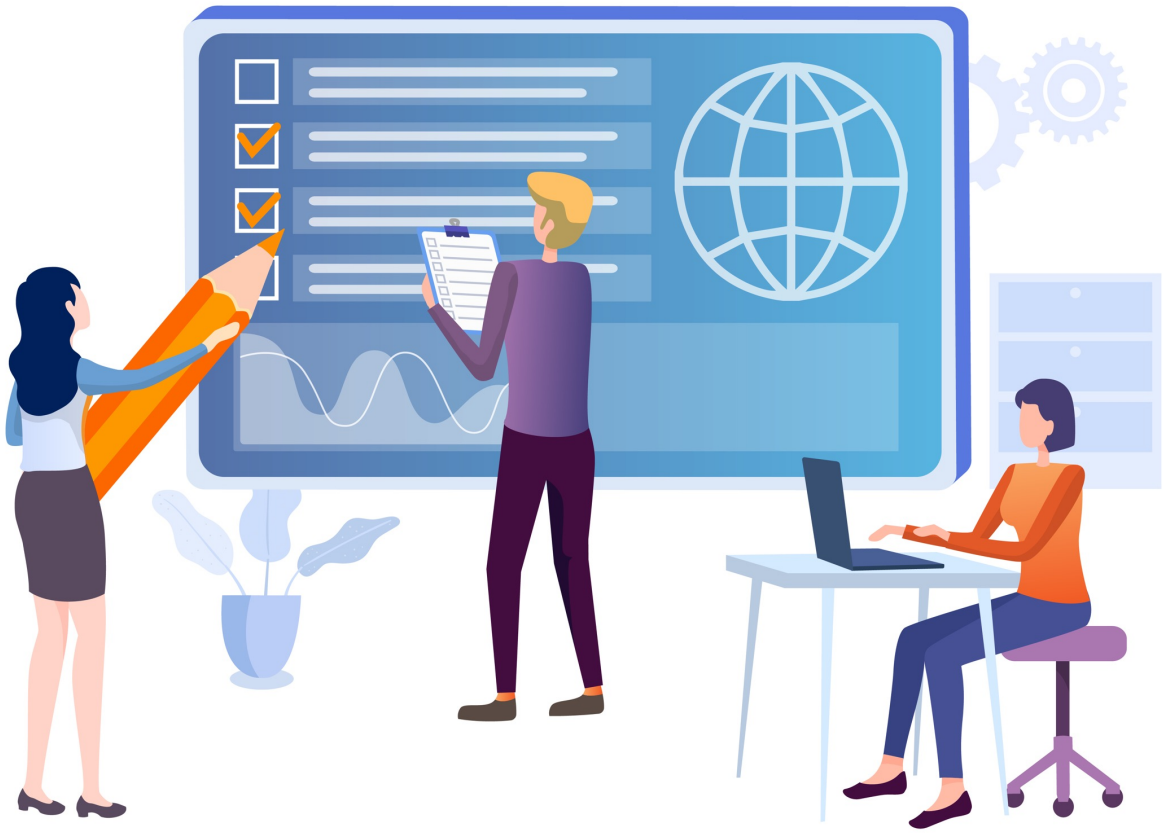


Assignment 1

- **CineSense** project!
  - 40% of your total score
  - Four weeks to submit
  - Submission guidelines will be shared soon!
- Good luck!

Quiz of the day

Get ready!



Serial vs multi-processing vs multi-threading

- Serial processing:

```
1 problem = 1 python program = 1 process = 1 task
```

- Multi-processing:

```
1 problem = 1 python program = n processes = n tasks
```

- Multi-threading:

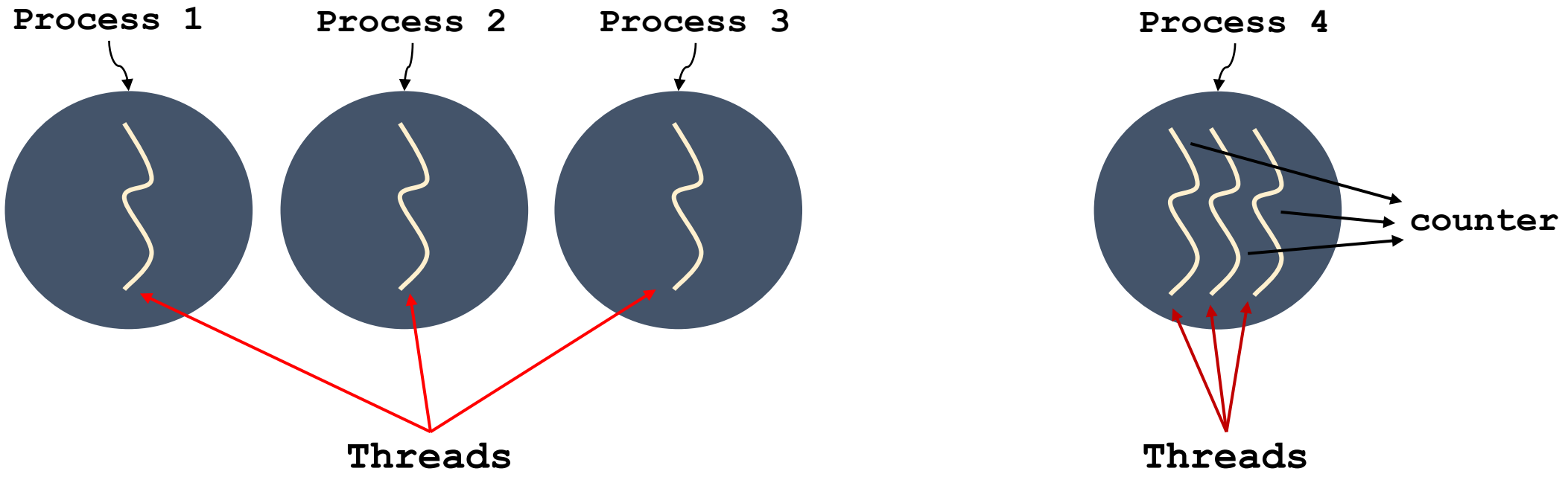
```
1 problem = 1 python program = 1 process = n threads = n tasks
```

Process types

- CPU-bound:
  - A process requiring significant computation – spends most of its time utilizing the CPU.
  - We want to reduce computation time or distribute the load across multiple CPU cores.
- Input/Output (I/O)-bound:
  - A process requiring significant computation – spends most of its time utilizing the disk.
  - The bottleneck is the speed at which the data is received or sent.

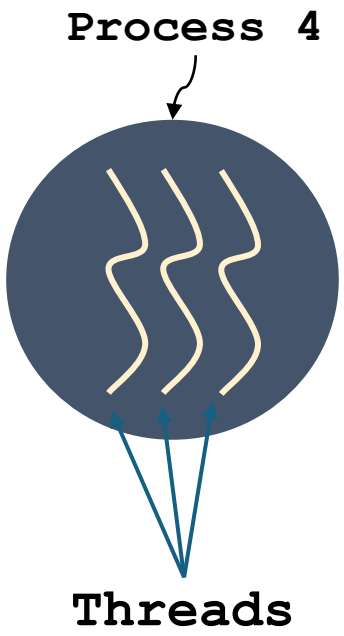
Processes and threads...

- Threads live inside processes, and they share memory.



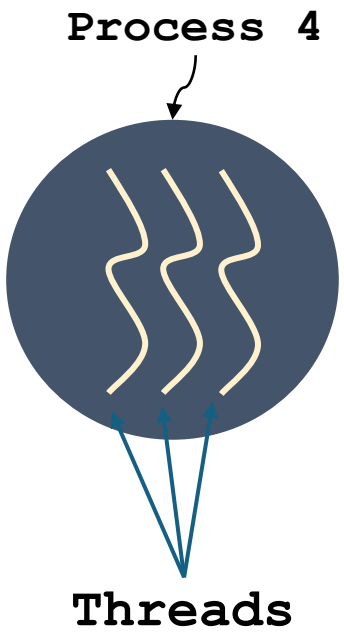
Why to use threads?

- Allow a single application to do many things at once
- Example
  - Thread to collect weather data.
  - Thread to analyze weather trends.
  - Thread to update forecasts.
- Ideal for I/O operations!



Threads

- A process can have multiple threads, which **share the same memory space and the variables declared in the program**.



Thread example

```python
import time
import threading

def foo():
    time.sleep(1)

# Parallel thread runner
def parallel_runner1():
    start = time.perf_counter()

    t1 = threading.Thread(target=foo)
    t2 = threading.Thread(target=foo)

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    end = time.perf_counter()
    print(f'Parallel: {end - start} second(s)')

if __name__ == '__main__':
    parallel_runner1()
```

script — Edited

Thread example (futures)

```python
import time
import concurrent.futures

def boo(sec):
    print(f"Running for {sec} second(s)")
    time.sleep(sec)

def parallel_runner(secs):
    start = time.perf_counter()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        executor.map(boo, secs)
    end = time.perf_counter()
    print(f'Parallel (thread pool map): {end - start} second(s)')

if __name__ == '__main__':
    secs = [1, 2, 3]
    parallel_runner(secs)
```

Summary

Processes

- An instance of a program that is being executed.
- Processes operate independently.
- **They do not share memory space with other processes directly.**
- Ideal for CPU bound tasks.

Threads

- The smallest unit of processing.
- Threads in the same process share the same memory.
- **Less resource-intensive than creating a process.**
- Ideal for I/O bound tasks.

Group quiz!

- Why do threads require extra care?
  - What potential problems do threads have that processes do not?

Threads have a shared memory space; they can access shared variables!

Pitfalls of parallel processing

- **Race Condition**
  - **A race condition occurs when multiple threads try to change the same variable simultaneously.**
  - The thread scheduler can arbitrarily swap between threads, so we cannot know the order in which the threads will try to change the data.

Pitfalls of parallel processing

- **Starvation**
  - **Starvation occurs when a thread is denied access to a particular resource for extended periods, slowing the overall program.**
  - This can be an unintended side effect of a poorly designed thread-scheduling algorithm.

Pitfalls of parallel processing

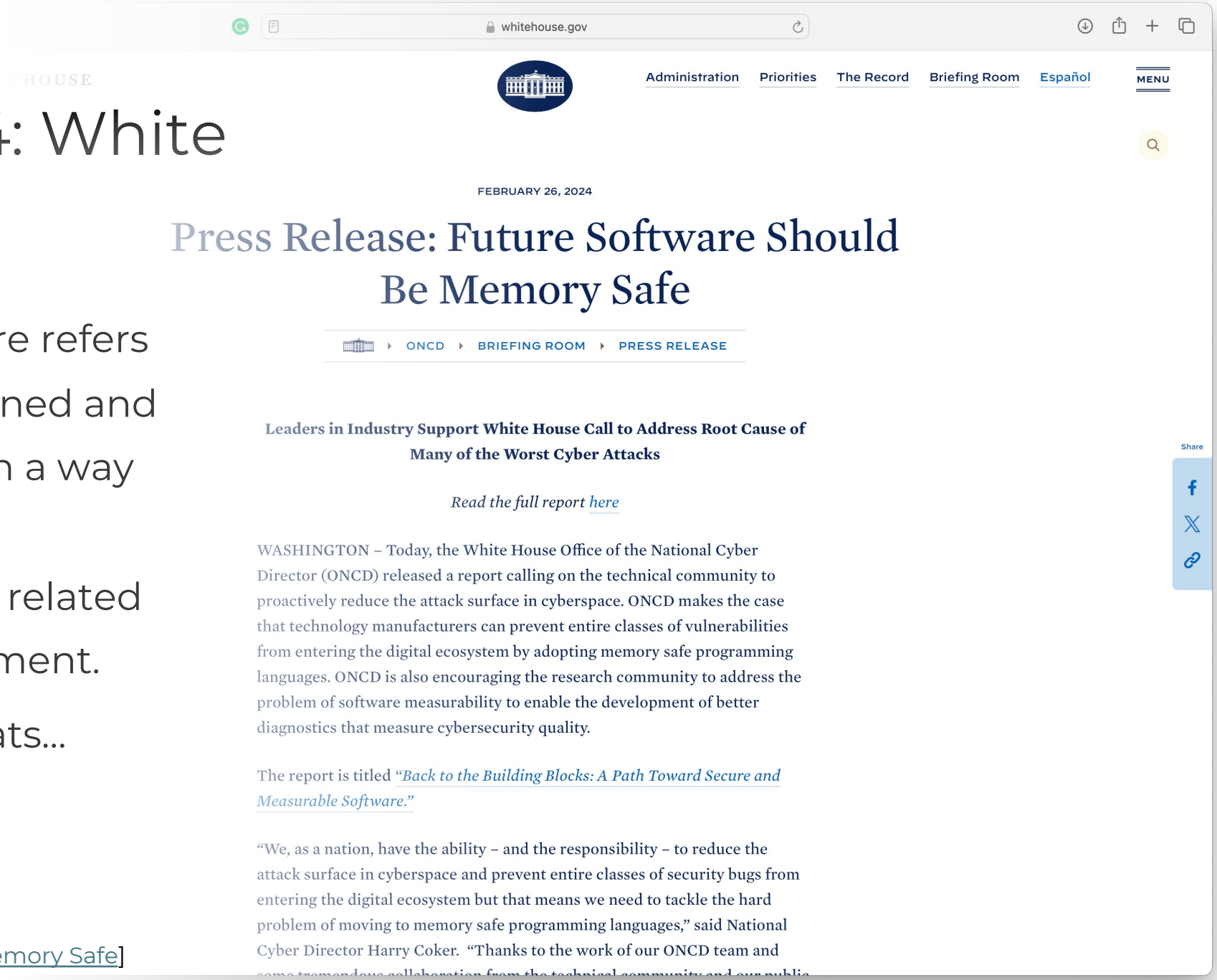
- **Deadlock**
  - **A state in which a thread is waiting for another thread to release a lock, but that other thread needs a resource to finish that the first thread is holding onto.**
  - This way, both threads come to a standstill, and the program halts.

Still, threads are the preferred way!

- Creating and managing threads involves significantly less overhead than processes.
  - You can achieve higher application throughput (serve more users).
- Threads can be dynamically created and destroyed based on the application's needs.
  - Threads share resources from their parent process, so the overall memory footprint of the application can be lower.

February 2024: White house report

- Memory-safe software refers to applications designed and implemented in such a way that they prevent programming errors related to memory management.
  - Threads are threats…
    - Not really!



[Press Release: Future Software Should Be Memory Safe]

Let’s talk about Python!

- CPython is the reference implementation of the Python programming language, written in the C programming language.
  - The global interpreter lock (GIL) protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously.
- This lock is necessary mainly because CPython's memory management is not thread-safe.

GIL (cont.)

- The GIL gets its job done, but at a cost.
- It effectively serializes the instructions at the interpreter level.
  - **For any thread to perform any function, it must acquire a global lock.**
- Only a single thread can acquire that lock at a time, which means the interpreter ultimately runs the instructions serially.

Still we can have the illusion..

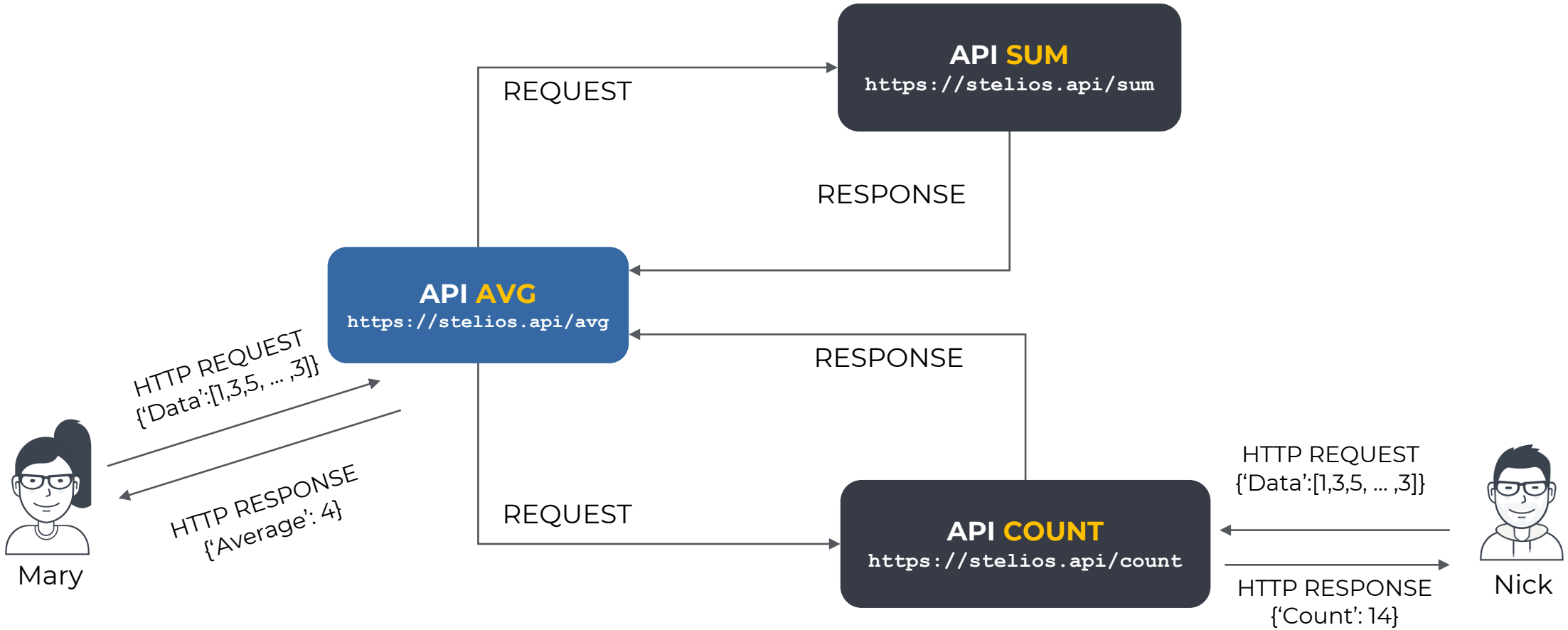
- This bottleneck, however, becomes irrelevant if your program has a more severe bottleneck elsewhere, for example , in network, IO, or user interaction.
  - In those cases, threading is an entirely effective method of parallelization.
- **Threading can make CPU-bound programs slower; in this case, it is better to use multi-processing.**

Use cases!

Video processing

- Develop an application to load videos in your script.
  - Videos will be processed for their content.
- Is this a CPU or I/O-bound task? **CPU**
- What is your strategy to develop this script? Load and process a video at a time.
- Can parallel programming be used? **Yes!**
- Multiprocessing of Threads? **Multiprocessing**

What is an Application Programming Interface (API)?



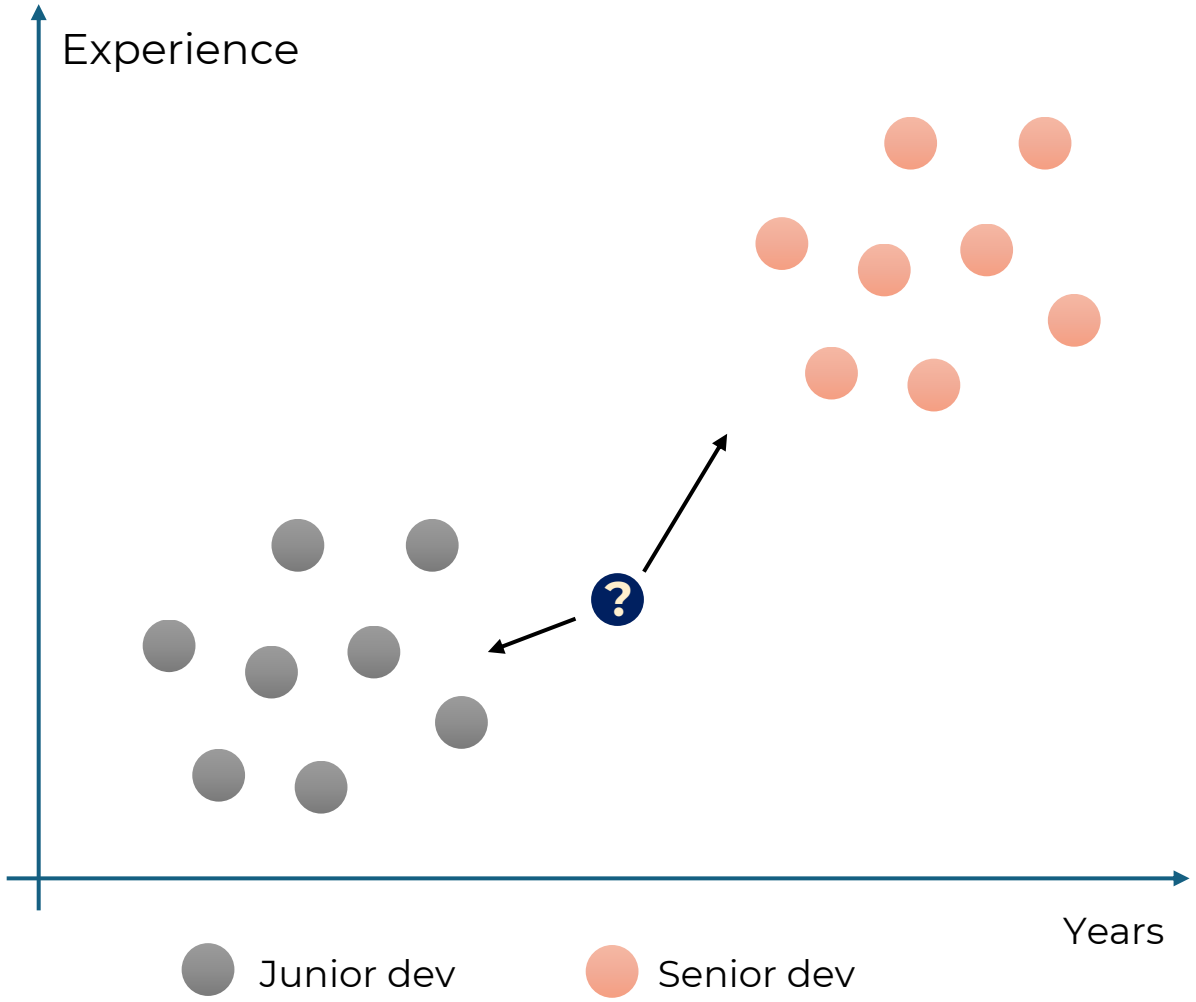
API fetcher

- Connect to a financial data API and collect real-time data.

  `https://api.frankfurter.app/latest?amount=10&from=GBP&to=EUR`

  - The data will be stored in a file for later use.
- Is this a CPU or I/O-bound task? **I/O**
- What is your strategy to develop this script? **Get one record at a time**
- Can parallel programming be used? **Yes!**
- Multiprocessing of Threads? **Threads is preferred**

K-Nearest Neighbors (KNN) ML model



1. **Choose k:** Select a `k`, which is the number of neighbouring data points to consider.
2. **Identify neighbours**: For a new data point, you look at the k closest data points (neighbours).
3. **Classify:** A new data point is classified based on the majority class of these `k` neighbours.

- For example, if `k=3`, you look at the `3` nearest neighbors.

- **How to find the best `k`?**

KNN model tester

- Find the best k for a KNN algorithm.
- Is this a CPU or I/O-bound task? **CPU**
- What is your strategy to develop this script? **Test for a k at a time**
- Can parallel programming be used? **Yes!**
- Multiprocessing of Threads? **Multiprocessing is preferred**

API tester for data analysis

- Test all your API endpoints with combinations of random test cases and inputs.
  - There might be thousands of tests to run!
- Each failed test must be logged into a file so you can review it afterwards.
- Is this a CPU or I/O-bound task? **I/O**
- What is your strategy to develop this script? **Test for a k at a time**
- Can parallel programming be used? **Yes!**
- Multiprocessing of Threads? **Threads is preferred (logger)**

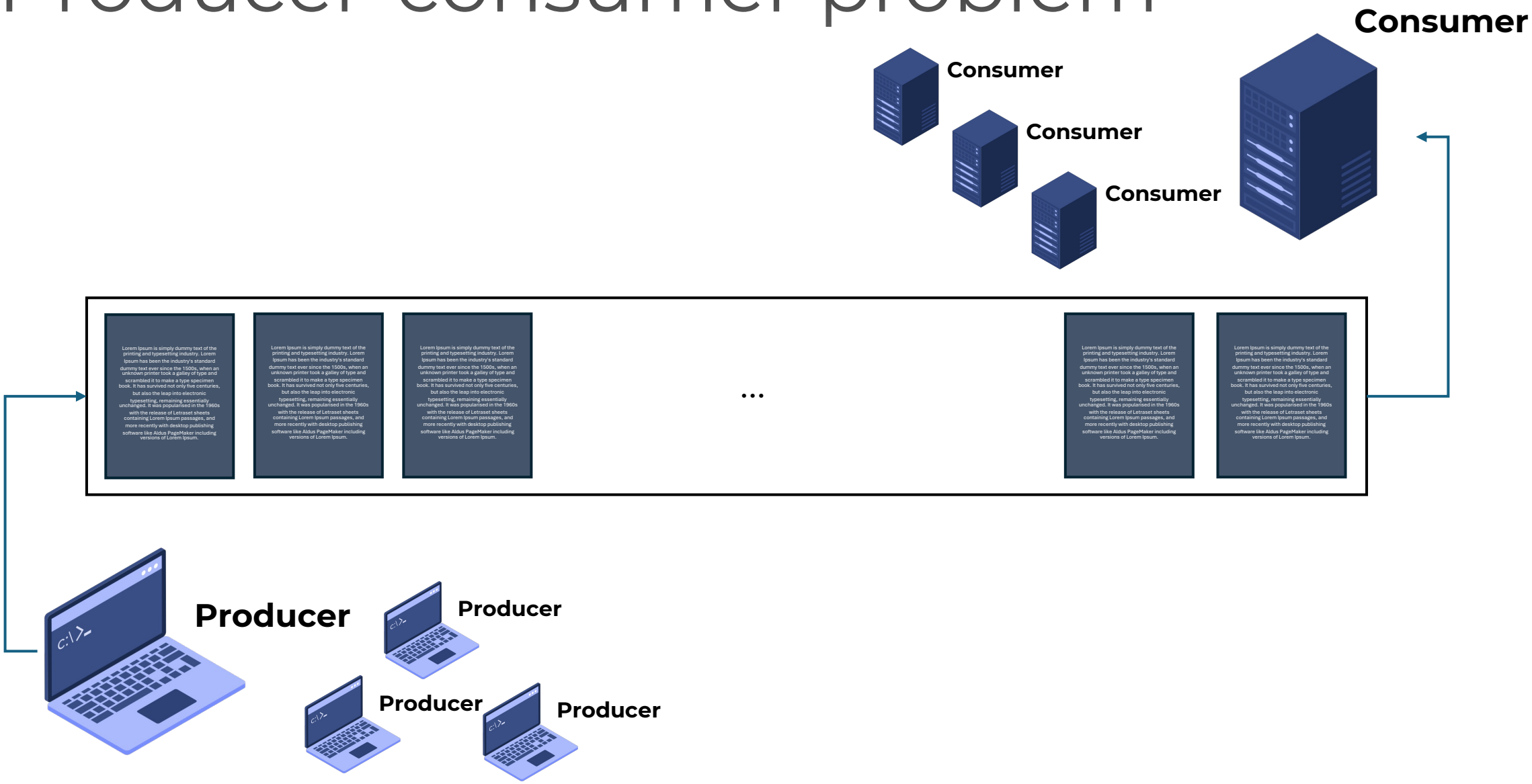
Examples of problems with threads!

- **Web Server Logging**
  - A web server handles multiple client requests simultaneously and logs the details of each request.
    - **A lock ensures that log entries are not corrupted due to concurrent access.**
- **Bank accounts**
  - A bank account where multiple transactions (deposits and withdrawals) happen simultaneously.
    - **A lock ensures that the account balance remains accurate.**

Real time data processing

- Data collection from news websites for sentiment analysis.
  - Producer thread (Data adder): A thread that continuously collects data from news websites and adds it to a list for further processing.
  - Consumer Thread (Data processor): A thread that processes the collected website data from the list, removing each processed data point after processing.
- You might have hundreds of producer and consumer threads!

Producer-consumer problem



Producer consumer problem

- The producer-consumer problem is a classic example of a multi-threading synchronization problem, with two types of threads: **producers** and **consumers**.
- The problem involves ensuring that **producers do not add data to a full buffer** and **consumers do not take data from an empty buffer**.
  - No two threads access the same data at the same time!

What is synchronization?

- Multiple threads can safely access **shared resources** without causing data inconsistency or race conditions.
  - Critical section!
- Common synchronization mechanisms include **locks**, **semaphores**, and **mutexes**.

What is a critical section?

- A part of a program that accesses shared resources
  - Data structures, files, or external devices (printers) must not be executed simultaneously by multiple threads or processes.
- The critical section involves operations that can lead to race conditions!
  - This causes data inconsistency, corruption or software bugs.

What are examples of the critical section?

- Banking system → Account Balance updates
- Inventory management → Stock updates
- Ticket booking system → Seat allocation
- File system operations → File access

Example

What is wrong with this script?

```
from threading import Thread
from time import sleep

data = [1,2,3]
def handle_data():
    while len(data)>0:
        sleep(0.5)
        print(data.pop(0))

t1 = Thread(target=handle_data)
t2 = Thread(target=handle_data)

t1.start()
t2.start()

t1.join()
t2.join()
```

Critical section

```
1
2
Exception in thread Thread-41 (handle_data):
Traceback (most recent call last):
  File "/usr/lib/python3.10/threading.py", line 1016, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.10/threading.py", line 953, in run
    self._target(*self._args, **self._kwargs)
  File "<ipython-input-21-30bea9524e7e>", line 8, in handle_data
IndexError: pop from empty list
3
```

Use a lock!

```
from threading import Thread,Lock
from time import sleep
data = [1,2,3]
data_lock = Lock()
def handle_data():
    data_lock.acquire()
    while len(data)>0:
      sleep(0.5)
      print(data.pop(0))
    data_lock.release()

t1 = Thread(target=handle_data)
t2 = Thread(target=handle_data)
t1.start()
t2.start()
t1.join()
t2.join()
```

What is synchronization?

- Coordination of concurrent operations
  - Ensure that shared resources or data are accessed and modified consistently and controlled.
- We need a lock!
  - A **mutex** is a locking mechanism ensuring that only one thread can access a resource at a time.
  - A **semaphore** is a signalling mechanism that can be used to control access to a resource by multiple threads.

Mutex

- The mutex is a particular case of a counting semaphore with an initial count of **1**.
  - Only one thread at a time accesses the critical section.
  - Processes include the following steps:

  acquire lock – access critical section - release lock.

```python
import time, threading, concurrent.futures

mutex = threading.Lock()

def boo(sec):
    mutex.acquire()
    print(f"Running for {sec} second(s)")
    time.sleep(sec)
    mutex.release()

def parallel_runner(secs):
    start = time.perf_counter()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        executor.map(boo, secs)
    end = time.perf_counter()
    print(f'Parallel mutex: {end - start} second(s)')

if __name__ == '__main__':
    secs = [1, 2, 3]
    parallel_runner(secs)
```

Mutex example

How long does it take?

6 seconds!

Semaphore

- Semaphores restrict the number of concurrent accesses to a specific part of the code or data.
  - It aims to limit the parallelism in critical sections to ensure data integrity and avoid race conditions.
  - A set of threads at a time accesses the critical section.

  acquire lock – access critical section - release lock.

```python
import time, threading, concurrent.futures

semaphore = threading.Semaphore(2)

def boo(sec):
    semaphore.acquire()
    print(f"Running for {sec} second(s)")
    time.sleep(sec)
    semaphore.release()

def parallel_runner(secs):
    start = time.perf_counter()
    with concurrent.futures.ThreadPoolExecutor() as executor:
        executor.map(boo, secs)
    end = time.perf_counter()
    print(f'Parallel semaphore: {end - start} second(s)')

if __name__ == '__main__':
    secs = [1, 2, 3]
    parallel_runner(secs)
```

Semaphore example

How long does it take?

1+3 seconds = **4**

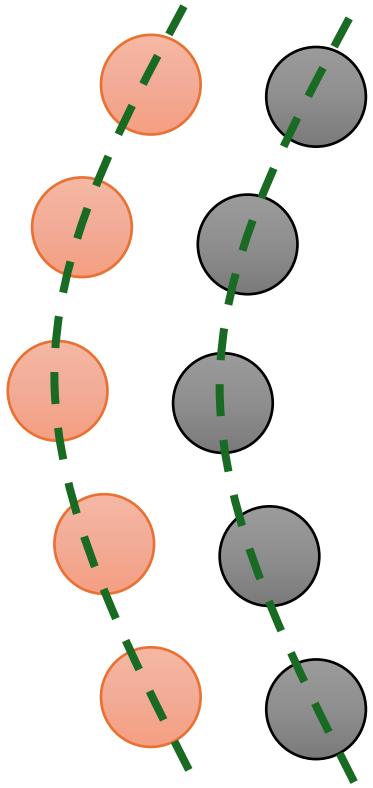
Parallelism vs Concurrency

Parallelism

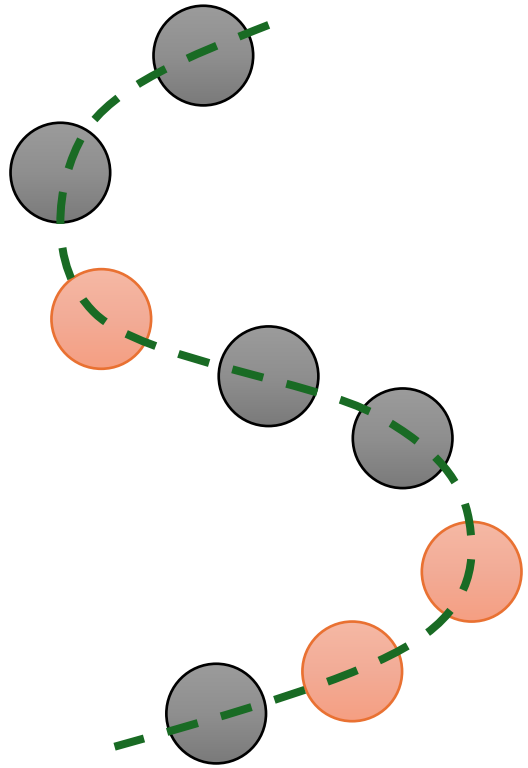
- Parallelism consists of performing multiple operations at the same time.
  - Processes: CPU tasks
  - Threads: I/O tasks that share memory

Concurrency

Parallel processing



Concurrent processing



How does it work?

- Chess master Garry Kasparov hosts a chess exhibition where he plays multiple amateur players.
- He conducts the exhibition synchronously and asynchronously.
  - 24 opponents
  - Garry makes each chess move in 5 seconds
  - Opponents each take 55 seconds to make a move
  - Games average 30 pair-moves (60 moves total)

Synchronous version

- Garry plays one game at a time, never two simultaneously until the game is complete.
- Each game takes **(55 + 5) * 30 = 1800** seconds, or **30 minutes**.
- The exhibition takes **24 * 30 = 720** minutes, or **12 hours**

Asynchronous version

- Garry moves from table to table, making one move at each table.
- He leaves the table and lets the opponent make their next move during the wait time.
- One move on all 24 games takes Garry:
  - **24 * 5 = 120** seconds, or **2 minutes**.
- The exhibition is now reduced to:
  - **120 * 30 = 3600** seconds, or just **1 hour**.

Thank you!

- The lab starts soon!

  (404-405)

O(no!)

Lab 4

Big Data Analytics

Lab activities

- Complete lab 4 activities and exercises.
- Use your preferred python IDE.