SUPSI

Introduction to concurrency and parallelism (Part 2)

Massimo Coluzzi
Content realised in collaboration with:
Tiziano Leidi, Fabio Landoni

Parallel and Concurrent Programming
Bachelor in Data Science

RECAP

In previous episodes...

- We made a summary of the Von Neumann architecture.
- Difference between sequential and concurrent executions.
- Difference between concurrent and parallel executions.
- Limits of parallel performance: the Amdahl's Law

In this lecture

- Further stressing about computer architectures
- Processes vs Threads
- Parallel programming in Python
- CPU bound vs I/O bound
- Synchronous vs Asynchronous

Computer Architecture

Flynn’s taxonomy

Parallel computing requires parallel hardware with multiple processors to execute different parts of a program simultaneously.

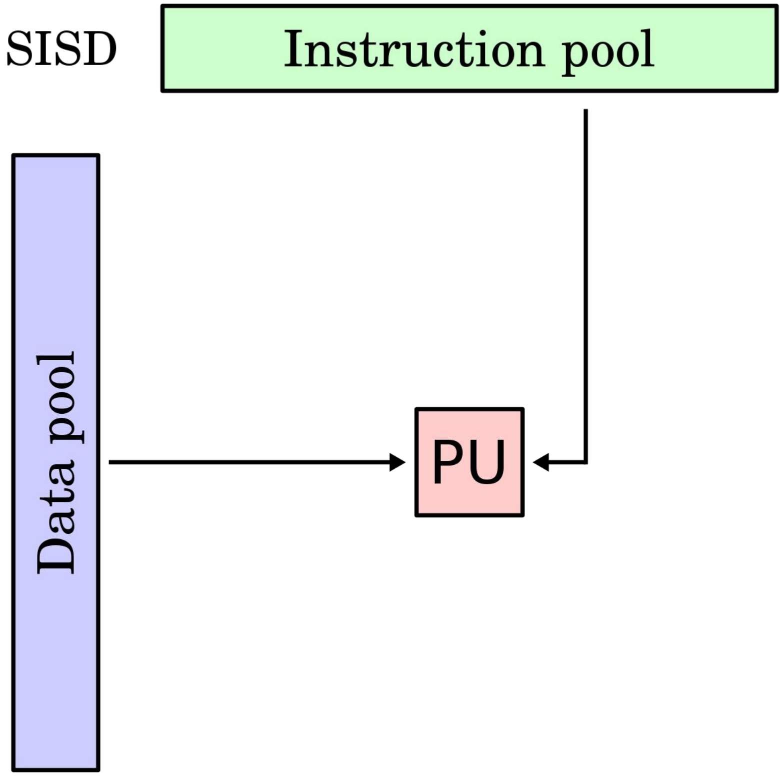
There are different types of parallel computer architectures.

Stanford University’s professor Michael J. Flynn defined a hardware taxonomy based on parallelization capabilities.

INFO

Single instruction, single data (SISD)

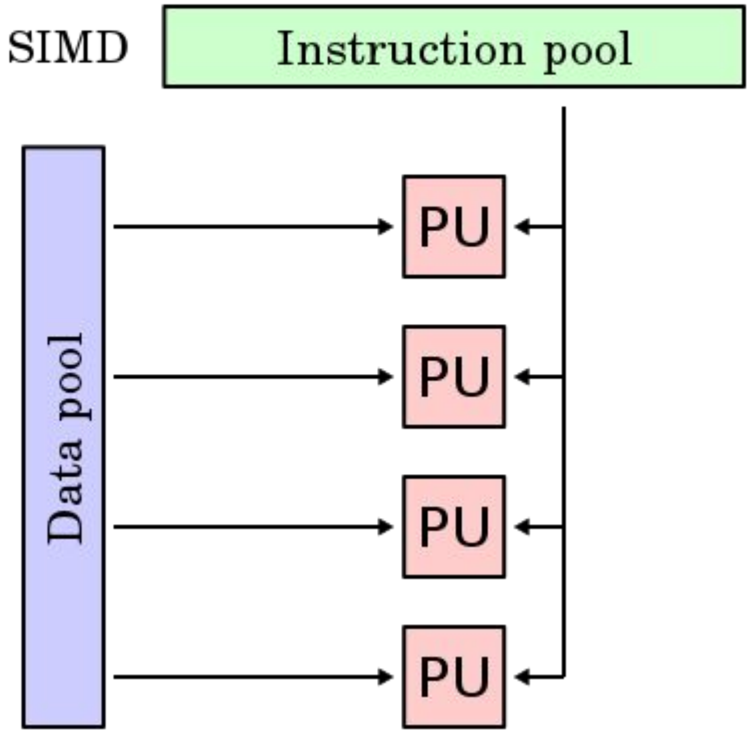
A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches a single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct a single processing element (PE) to operate on a single data stream (DS) i.e., one operation at a time.



INFO

Single instruction, multiple data (SIMD)

A single instruction is simultaneously applied to multiple different data streams. Instructions can be executed sequentially, such as by pipelining, or in parallel by multiple functional units.



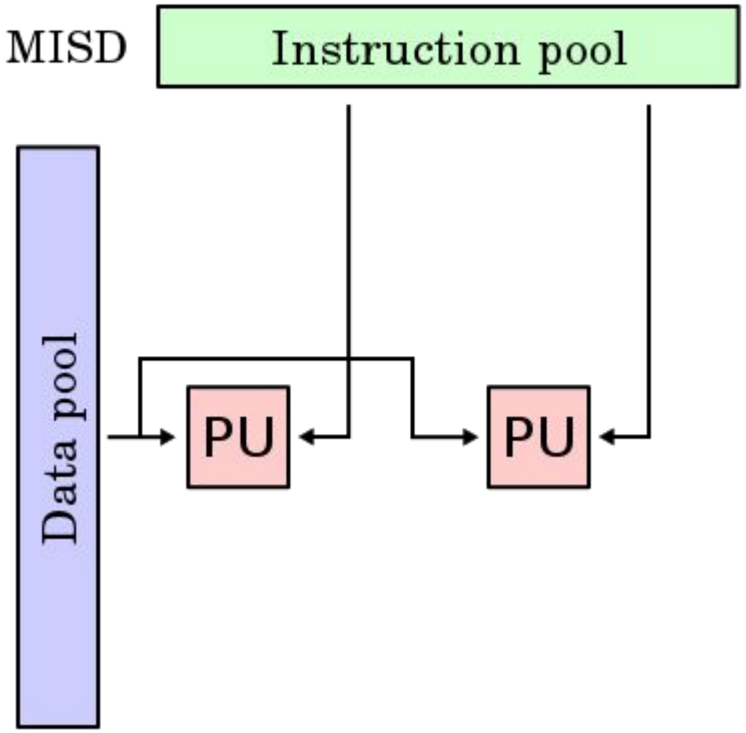
INFO

Multiple instruction, single data (MISD)

Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance.

Heterogeneous systems operate on the same data stream and must agree on the result.

Examples include the Space Shuttle flight control computer.

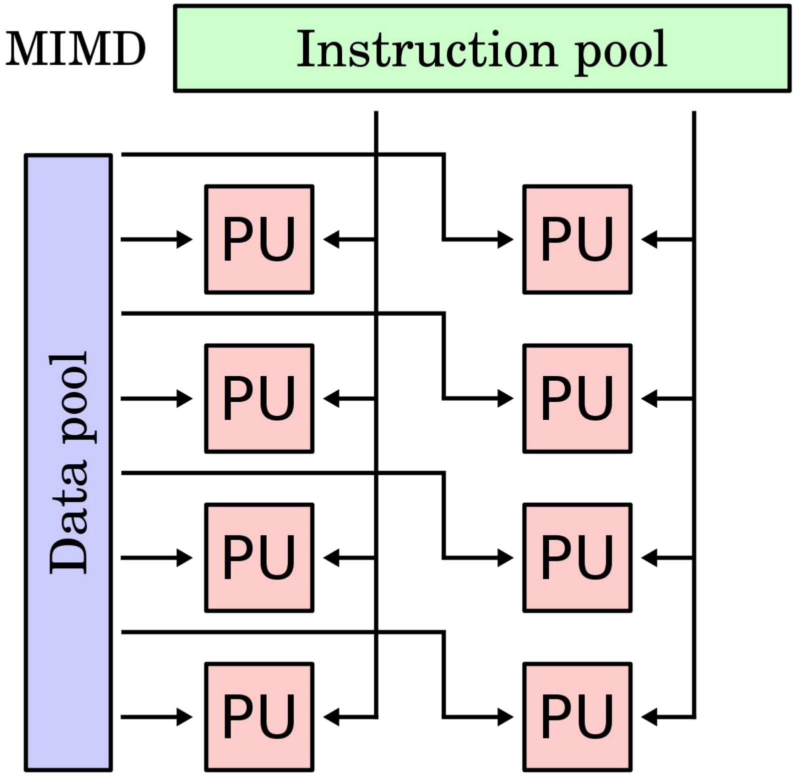


!!!

Multiple instruction, multiple data (MIMD)

Multiple autonomous processors simultaneously executing different instructions on different data.

MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space.



Memory architectures

All modern computer architectures belong to the MIMD category.

They also rely on a shared memory approach:

- All processors access the same memory
- The operating system allows every processor to access the memory equally fast (Uniform Memory Access).

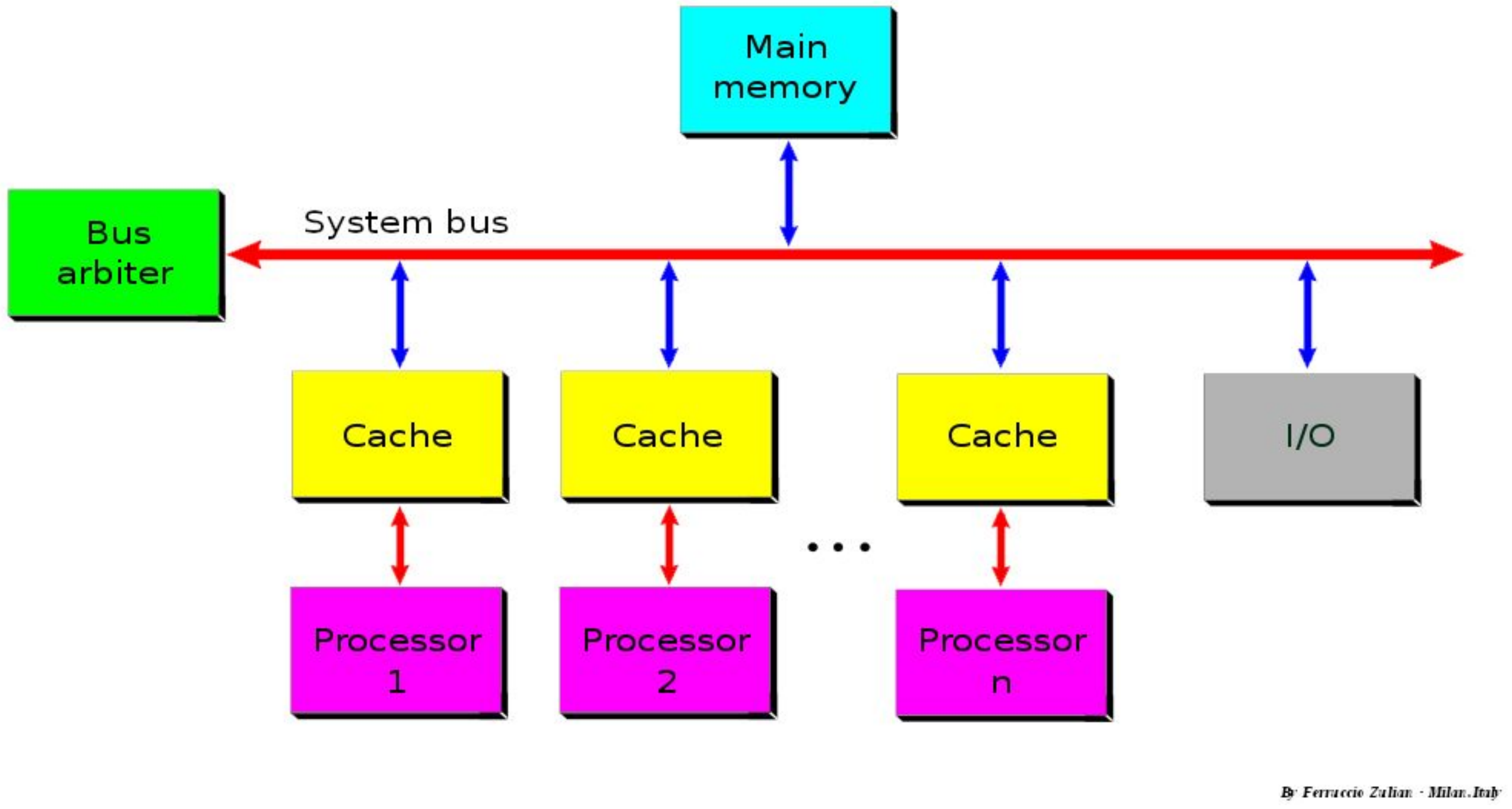
Symmetric multiprocessing (SMP)

In particular, modern architectures use a type of UMA called Symmetric multiprocessing (SMP).

Two or more identical processors connected to a single shared memory, controlled by a single operating system.

All processors are treated equally (none of them has a special purposes).

Memory architectures



Access time of different resources

The following table shows the time required to access different resources inside a computer.

All values have been normalized to 1 second to make you better understand the difference in magnitude between the different operations.

Access time of different resources

Operation	**Scaled time**
1 CPU cycle	1 second
L1 cache access	3 seconds
L2 cache access	10 seconds
L3 cache access	33 seconds
Main memory access (DRAM)	6 minutes
Solid-state disk access (SSD)	9 to 90 hours
Magnetic disk access (HDD)	1 to 12 months
San Francisco to New York TCP request	4 years
San Francisco to London TCP request	8 years

Access time of different resources

The previous table should give you an idea of how much asynchronous and concurrent programming can speed up performance.

For example, a CPU can perform more than 380 million operations while waiting for a TCP request to respond.

Processes and Threads

RECAP

Program vs Process vs Thread

Program:

- set of instructions and associated data
- resides on the disk
- is loaded by the operating system to perform a task

Process:

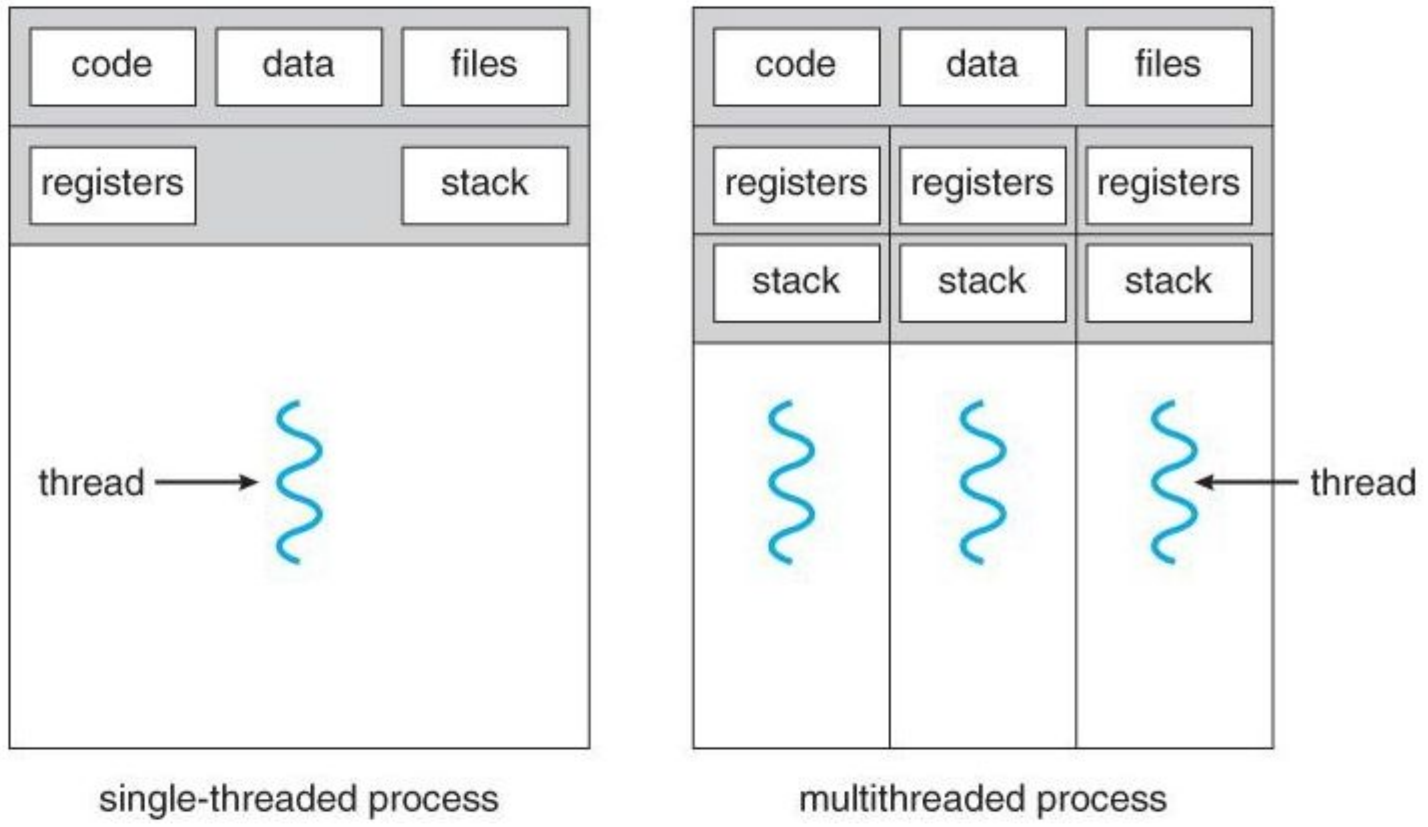
- execution environment of a program
- it consists of instructions (code), data, state information and other resources (CPU, memory, address-space, disk and network I/O acquired at runtime)
- independent instance of a running program

Program vs Process vs Thread

Thread:

- A thread is the smallest unit of execution in a process.
- Threads execute instructions sequentially.
- The scheduling of threads execution is handled by the operating system.
- Threads can share all the resources allocated to the process they belong to.

Process vs Thread



Process vs Thread - Cooking analogy

Cooking	**Computing**
Preparing a vegetable soup	The process
Two cooks working on potatoes and carrots	Two threads executing as part of the same process
The kitchen	Shared address space of the process
Cookbook with instructions used by both cooks	The code
Ingredients (potatoes, carrots, etc.)	Data and variables

!!!

Process

- Processes are isolated program executions.
- Processes do not share any resources.
- Process execution is scheduled by the operating system.
- The time a process can access the CPU is determined by the operating system based on the process priority.
- Interactive processes (e.g., mouse driver) have higher priority than background processes (e.g., network monitor).

!!!

Context switching

When the operating system suspends a process, its state is frozen.

All CPU registries and L1 cache of the suspended process are copied to memory.

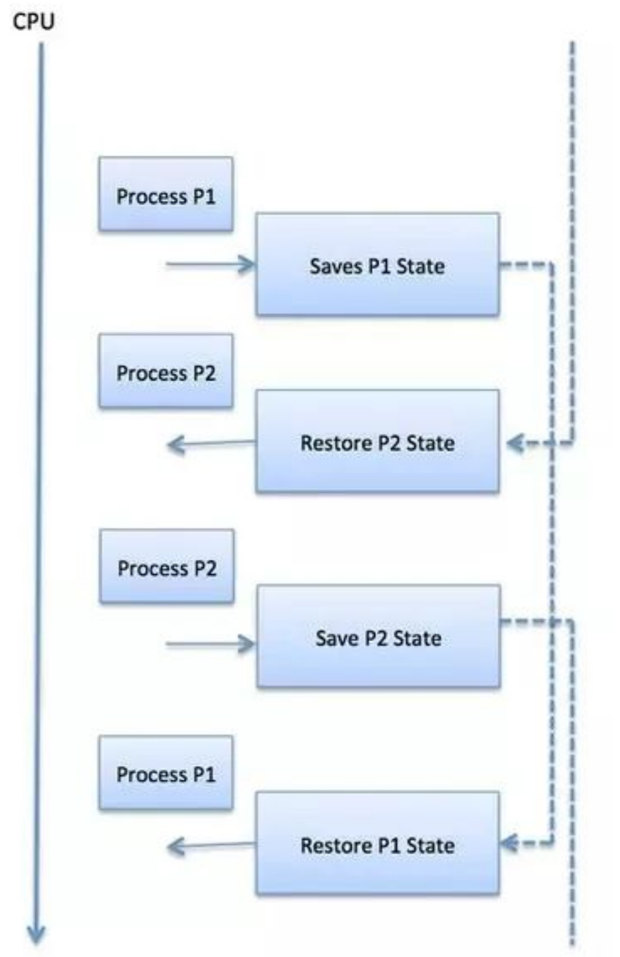
Registries and L1 cache of the new process are loaded to the CPU, and the new process is started.

This procedure is called context switching.

Context switching

Since memory (RAM) is about 1000 times slower than CPU, context switching is a pretty expensive operation.

Threads was conceived to reduce the overhead caused by context switching in parallel processing.



!!!

Thread

Threads are also known as lightweight processes.

- They are less resource-intensive.
- They require less overhead to create and terminate.
- Threads can share all the resources allocated to the process they belong to.
- Changing the running thread does not require complete context switching. Only the CPU registries are dumped to and loaded from the L1 cache.

Question

It is better to use processes or threads?

Answer

It depends on what we are doing and the environment we are running in.

Implementation of threads and processes differs between operating systems and programming languages.

Rule of thumb: if you can take advantage of multiple threads, stick with using threads rather than multiple processes.

Parallel programming in Python

SERIOUSLY ?
memegen.com

Thread safety

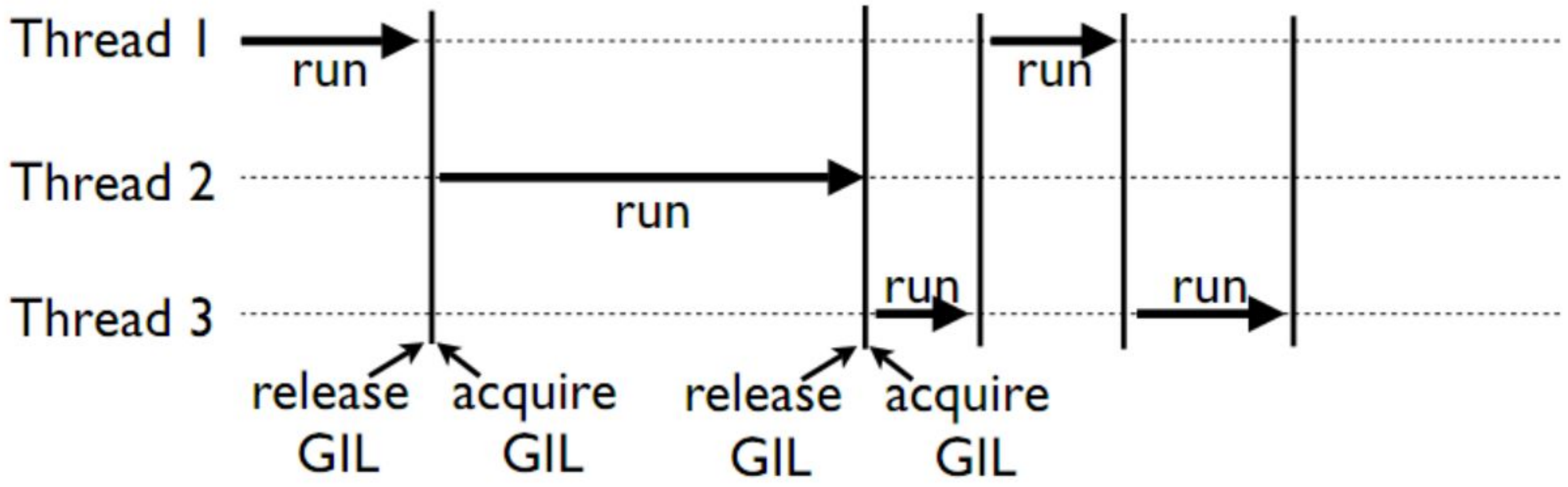
During this course, we will see most problems related to parallel processing, particularly accessing the same variables simultaneously, known as a race condition (or data racing).

Avoiding race condition issues is known as thread safety.

The most popular Python implementation (CPython) ensures thread safety by preventing multiple threads from executing in parallel.

Global Interpreter Lock (GIL)

Global Interpreter Lock (GIL) is a mechanism that prevents multiple threads from executing Python code in parallel.



CPython and the GIL

CPython:

- is the default and most widely used Python interpreter,
- is written in C and Python,
- uses the GIL to guarantee thread safety,
- actually does not support parallel threads executions.

There are heated debates on whether the GIL was the right choice or not. In the words of Larry Hastings, the design decision of the GIL is one of the things that made Python as popular as it is today.

noGIL implementations

There are several attempts to remove the GIL.

So far, all attempts have broken C extensions and degraded the performance of single and multithreaded I/O bound programs.

We will use a proof of concept noGIL implementation during the exercise labs. It is about 10 times slower than default CPython and should not be used in production environments.

But allows us to study the effects of parallel thread executions.

CPython and the GIL

As well argued by Larry Hastings during PyCon 2015, just because CPython has the GIL does not mean there is no value in writing multi-threaded Python programs.

Multi-threaded workloads can be grouped into categories:

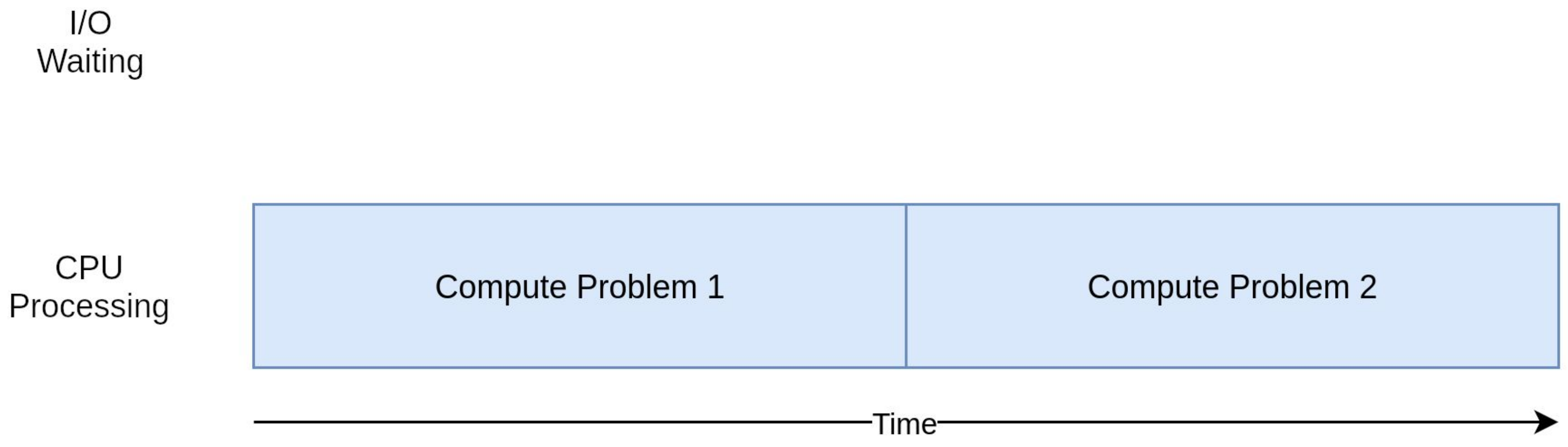
- Synchronous vs. Asynchronous
- CPU bound vs. I/O bound

Only synchronous CPU bound workloads are affected by the GIL.

CPU bound and I/O bound

CPU Bound

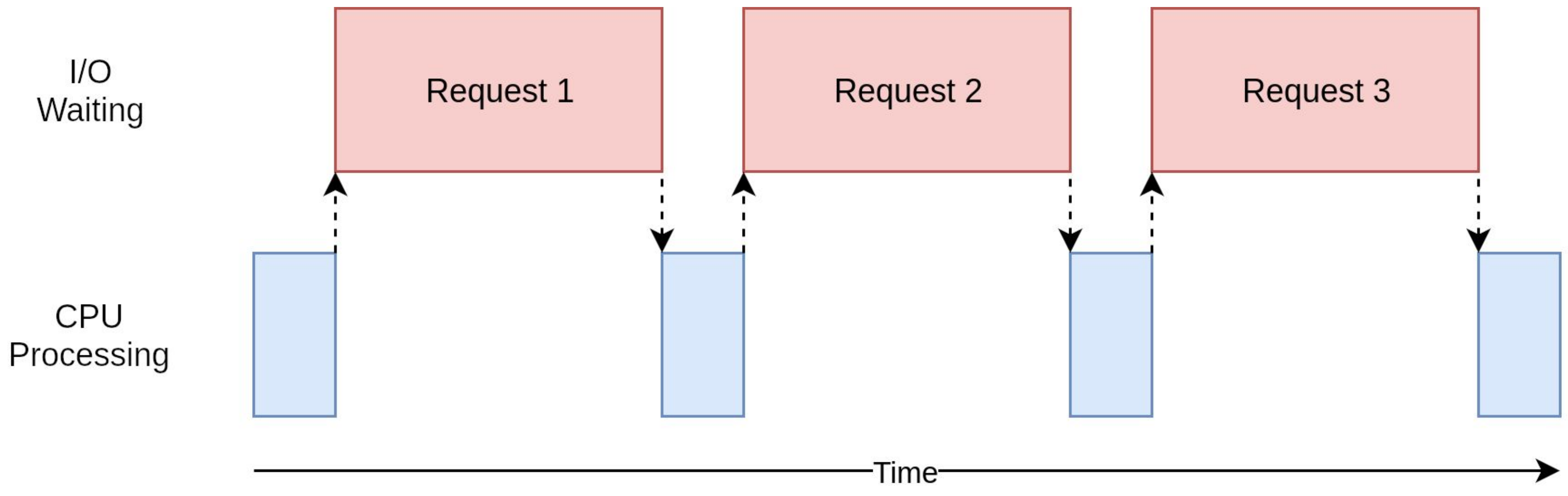
CPU-bound executions require significantly high utilization of the CPU (e.g., data crunching, image processing, matrix multiplication).



If a CPU bound program is provided a more powerful CPU, it can complete faster. The same could be true if the program can be structured to take advantage of multiple CPU units available.

I/O Bound

I/O bound programs spend most of the time waiting for I/O operations to complete (such as write/read from main memory, network interfaces, or user input) while the CPU sits idle.



GIL, I/O Bound and CPU Bound

- For I/O-intensive tasks, the GIL does not create a significant bottleneck, and you can gain a lot by using multiple concurrent threads.

- For CPU-intensive tasks, the GIL can negatively impact multi-threaded performance.
  Ways to get around:
  - Perform processor-heavy computation outside of the GIL's restrictions using parallel threads. This can be done using extension modules where computation is performed on data structures managed outside of CPython's runtime (e.g., C++).
  - Use Python's multiprocessing package.

Python multiprocessing package

Python multiprocessing package:

- Used to get around GIL limitations for CPU bounded applications.
- Implements the program with multiple processes instead of multiple threads.
- Each process will have its own instance of the Python interpreter with its own GIL, so the separate processes can execute in parallel.
- Downsides:
    - Communication between processes is more complicated than between threads
    - Creating multiple processes uses more system resources than creating multiple threads.

I/O Bound vs CPU Bound

Both types of programs can benefit from concurrent architectures.

I/O bound process	**CPU bound process**
The program spends most of its time talking to a slow device, like a network connection, a hard drive, or a printer.	The program spends most of its time doing CPU operations.
Speeding it up involves overlapping the times spent waiting for these devices.	Speeding it up involves finding ways to do more computations in the same amount of time.
Multiple threads	Multiple processes

Synchronous and Asynchronous

Synchronous

- When a function is invoked, we talk about synchronous execution when the program execution waits until the function call is completed.
- Synchronous execution blocks at each method call before proceeding to the next line of code.
- Synchronous execution is synonymous with serial execution.

Asynchronous (or async)

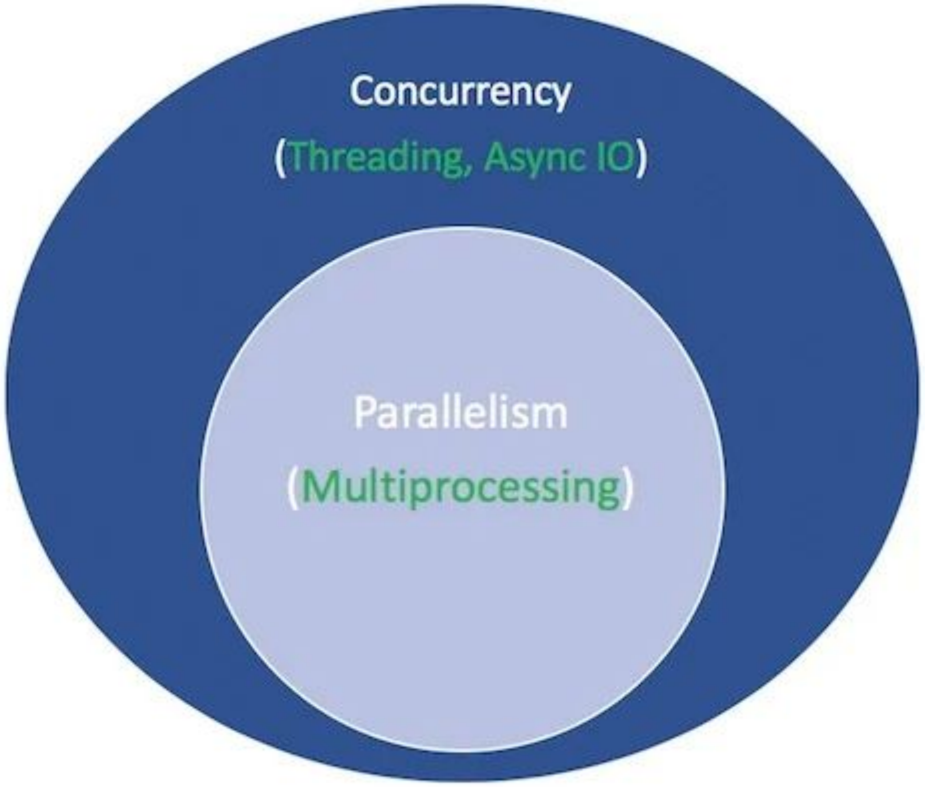
- Asynchronous executions do not wait for a task to complete before moving on to the next one.
- In non-threaded environments, asynchronous programming provides an alternative to threads in order to achieve concurrency (cooperative multitasking model).

- Two approaches:
  - Called subroutines return an entity called future or promise. The main program can query for the status of the computation and retrieve the result once completed.
  - Pass a callback function to the asynchronous function call, which is invoked with the results when the asynchronous function is done processing.

Async IO

- Asynchronous IO (async IO) is a language-agnostic paradigm (model) that has implementations across a host of programming languages.
- Async IO is a single-threaded, single-process design that uses cooperative multitasking.
- Async IO is a concurrent programming design that has received dedicated support in Python, evolving rapidly from Python 3.4.
- The Python library to write concurrent code using the async/await syntax is calle `asyncio`.

Multithreading, Multiprocessing & Async IO in Python



Summary

- Von Neumann architecture
- Process
- Thread
- GIL
- I/O Bound and CPU Bound
- Async IO

Further readings

- What Is the Python Global Interpreter Lock (GIL)?
- Python stands to lose its GIL, and gain a lot of speed