Multithreading and multiprocessing in Python

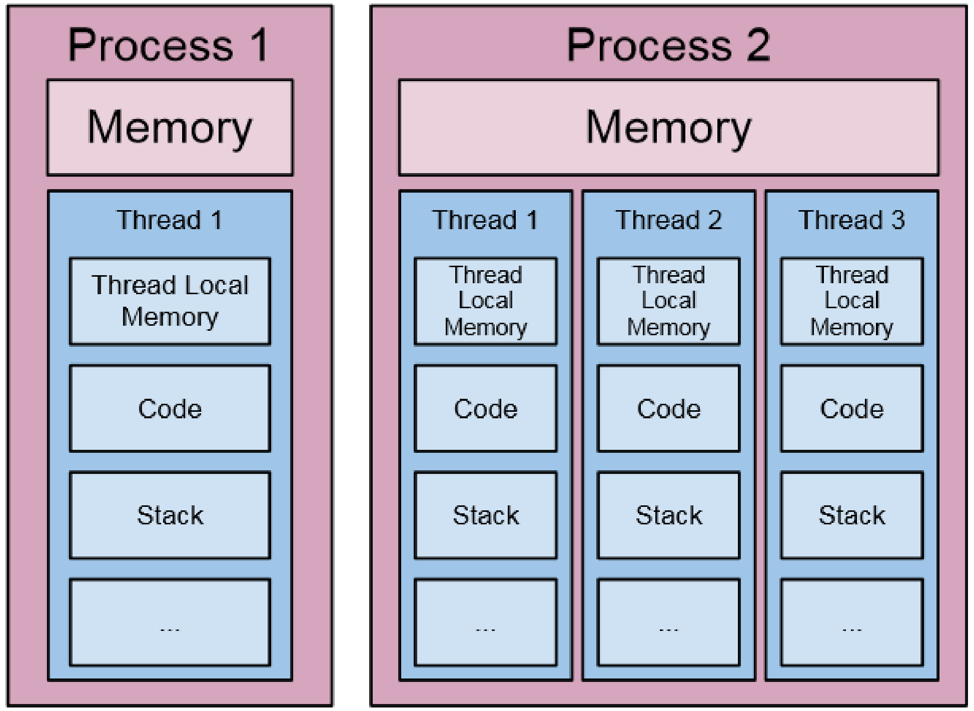
Computing Methods for Experimental Physics and Data Analysis

Lecture 1B

gianluca.lamanna@unipi.it

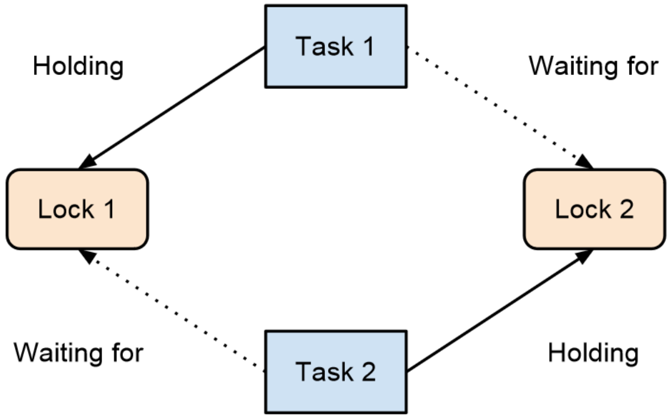
Threads and processes in Python

- Threads and processes are the way to use concurrency in python
  - ➙ a thread is a single sequence of execution within a program that can run in parallel with other threads. Threads are used to achieve concurrent execution, allowing a program to perform multiple operations simultaneously.
  - ➙a process is an independent instance of a program that is executed by the operating system. Each process runs in its own memory space and has its own resources, which means it does not share memory directly with other processes. Processes are used to achieve parallel execution, where different processes can run simultaneously on different CPU cores, making full use of multi-core processors.



Things to be affraid of! (not only in python…)

- Starvation
  - a task is costantly denied necessary resource
  - The task can never finish (starves)
- Deadlock
  - Usually a deadlock occurs when two or more tasks wait cyclically for each other.



Thread-safe

- **thread-safe** refers to an operation, function, or code that can be safely executed in a multithreaded environment without causing unexpected behavior or errors, even when accessed by multiple threads simultaneously
  - → Race conditions
  - → Deadlocks
  - → Data corruption

Mutex and Semaphores

- Mechanism to avoid threads conflicts. OS features to reserve the use of resources through *synchronization primitives*:
  - Mutex
  - Semaphore
  - Atomic operation (es. variable assignment)
  - Thread-safe data structures (es. Queue in Python)
- Mutex: it's a locking mechanism
  - Only one task acquire the mutex, only the owner can release the lock
- Semaphore: it's a signaling mechanism
  - ISR (Interrupt signals routing) can be issued by a manager to active specific threads
  - Semaphores are generalization of Mutex

Example of lock in multi-thread

- Not Thread-safe:

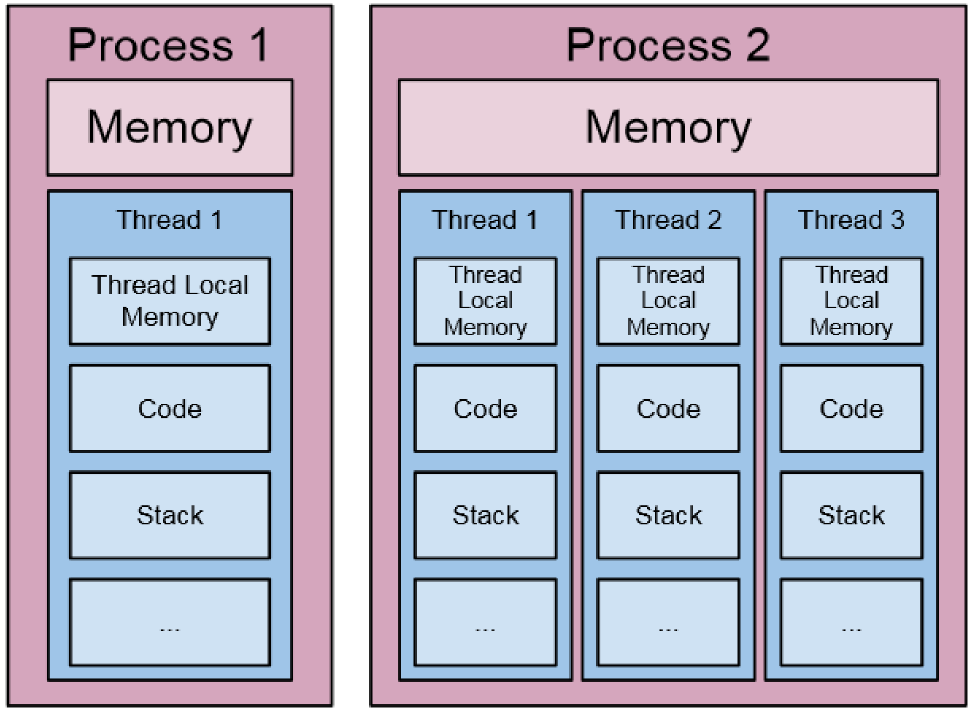
```
counter = 0
def increment_counter():
    global counter
    counter += 1
```

- Thread-safe with mutex:

```
lock = threading.Lock()
counter = 0
def increment_counter():
    global counter
    with lock:
        counter += 1
```

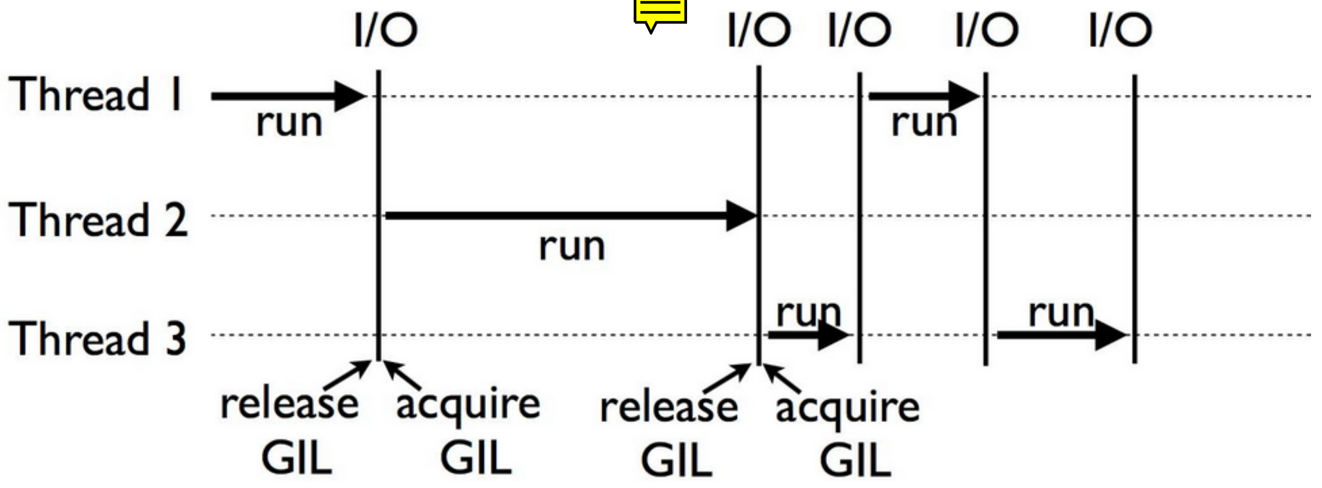
Threads and processes in Python

- Threads and processes are the way to use concurrency in python
- Python implements a very simple thread-safe mechanism: Global Interpreter Lock (GIL).
  - → In order to prevent conflicts only one statement in one thread is executed at a time (single-threading)

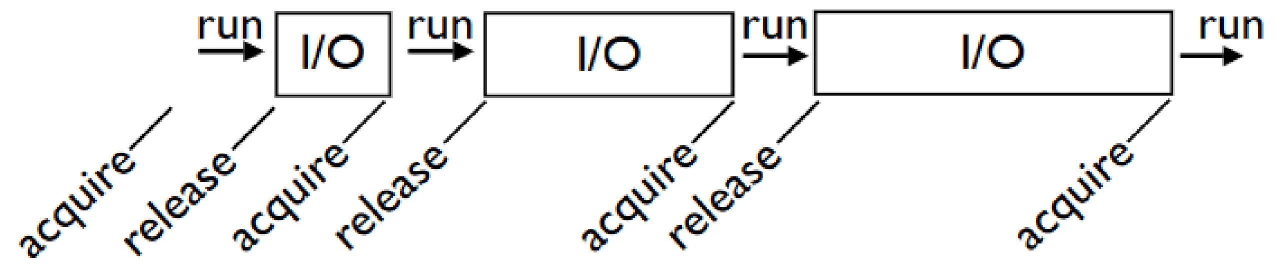


The Global Interpreter Lock (GIL)

- The Global Interpreter Lock refers to the fact that the Python interpreter is not thread safe.
- There is a global lock that the current thread holds to safely access Python objects.
- Because only one thread can acquire Python Objects/C API, the interpreter regularly releases and reacquires the lock every 100 bytecode of instructions. The frequency at which the interpreter checks for thread switching is controlled by the sys.setcheckinterval()function.



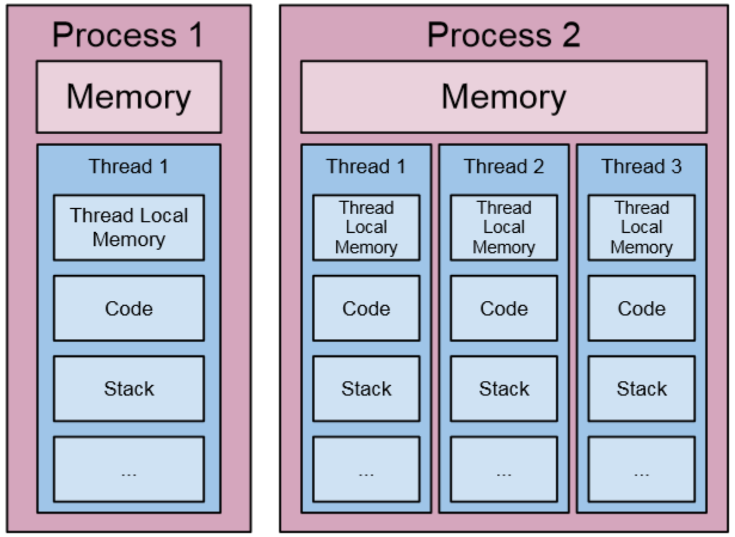
- The lock is release by a thread in case of a I/O operation
  - →During I/O other threads can work concurrently
- It is important to note that, because of the GIL, the CPU-bound applications won't be helped by threads.
  - →In Python, it is recommended to either use processes, or create a mixture of processes and threads.



Process: pros and cons

- A process is an instance of a program
- Managed by operating system
  - ➙ Memory space allocated by the kernel
- Two processes can execute code simultaneously in the same python program
- Separated memory space
- Takes advantage of multiple cores and CPUs
- Child processes are killable
- Avoid GIL limitations

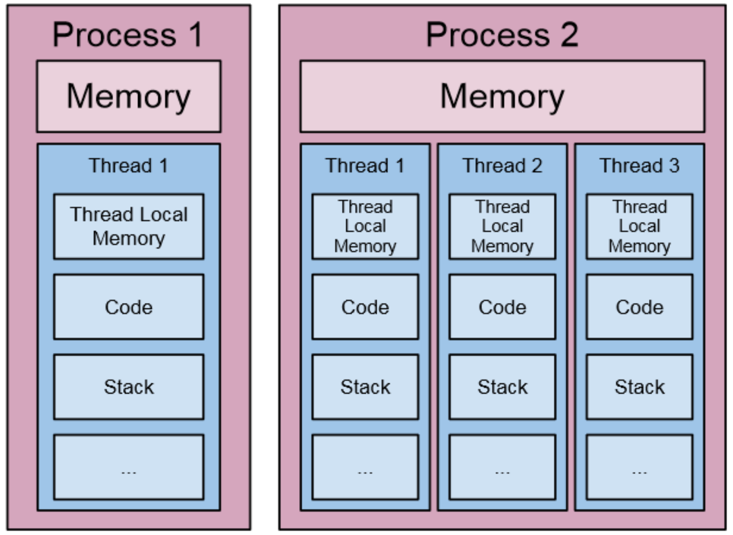
- Relatively high overhead
  - ➙ Open and close processes takes more time
- Sharing information between processes is very slow
- Model not adaptable to parallelism



Threads: pros and cons

- Processes produce threads (sub-processes) to handle sub tasks
  - → Threads live inside the process and share the same memory space
- Can use shared memory
  - → Threads communication
- Lightweight
- Very small overhead
- Great option for I/O bound application

- Subject to GIL (although there are workarounds)
- Not killable
- Potential of race condition
- Same memory space



When to use threads vs processes?

- **Processes** speed up Python operations that are CPU intensive because they benefit from multiple cores and avoid the GIL.
- **Threads** are best for IO tasks or tasks involving external systems because threads can combine their work more efficiently. Processes need to pickle their results to combine them which takes time.
  - → Threads provide no benefit in python for CPU intensive tasks because of the GIL.

Threads, processes and concurrency modules

- The management of threads and processes is carried out in python with special modules
  - ➙ Be sure that you have “multiprocessing” and “threading” modules on your python installation
- The module “concurrent” (python >3.2) is an abstraction of multiprocessing and multithreading to simplify some function
  - ➙ Check
- Bring your laptop for hands-on