High Performance Python Lab

Overview of multithreading and multiprocessing tools

November 22, 2023

Difference between multithreading and multiprocessing

Multithreading:

+ less overhead

• common address space

- need to synchronize access to resources

+ easy to communicate with others

Multiprocessing:

- more overhead

• independent address spaces for each process

+ less need to synchronize

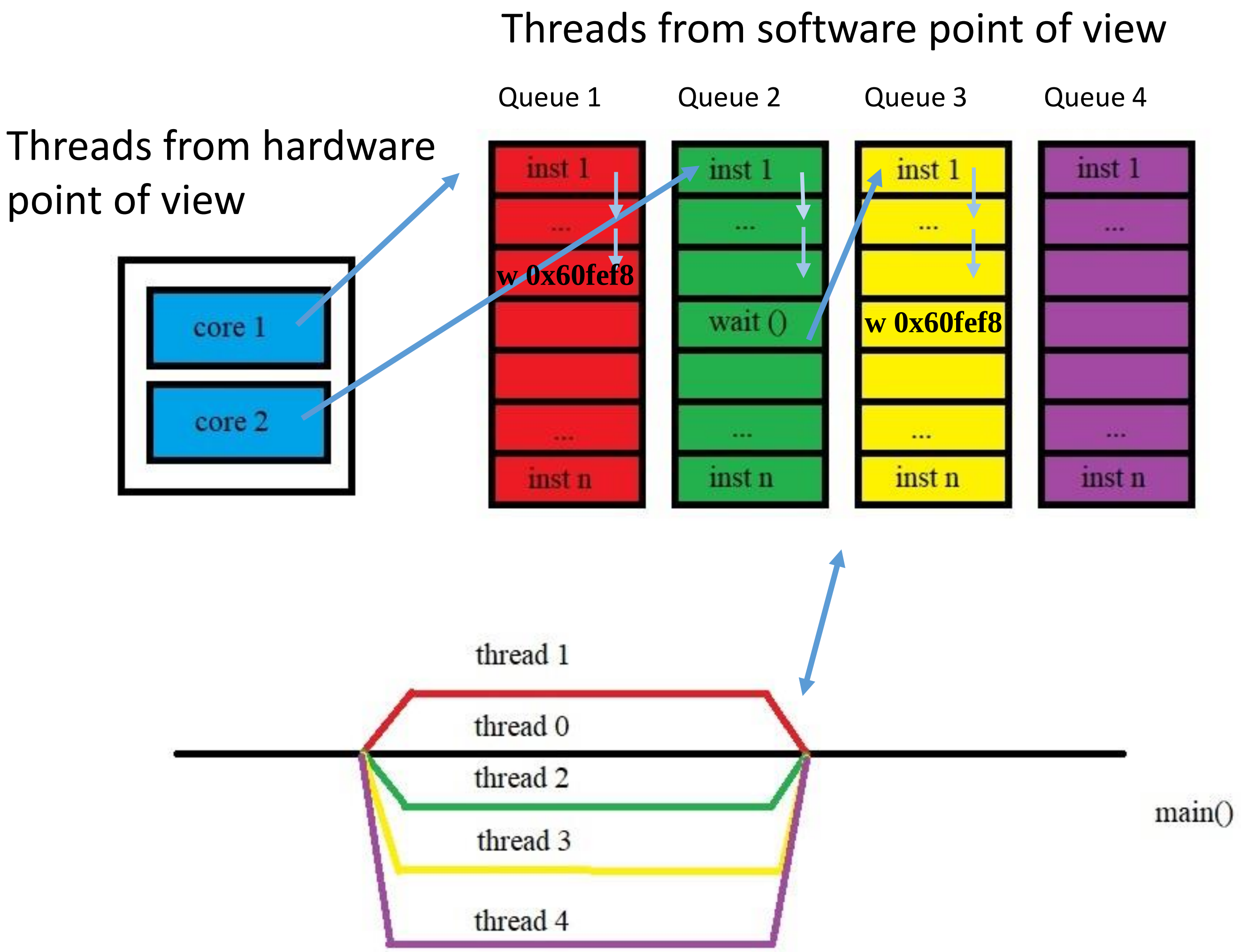
- need to find ways to communicate with others

Overview of libraries

- Multithreading
- Multiprocessing
- Concurrent
- Joblib

Also: Dask, ray, asyncio

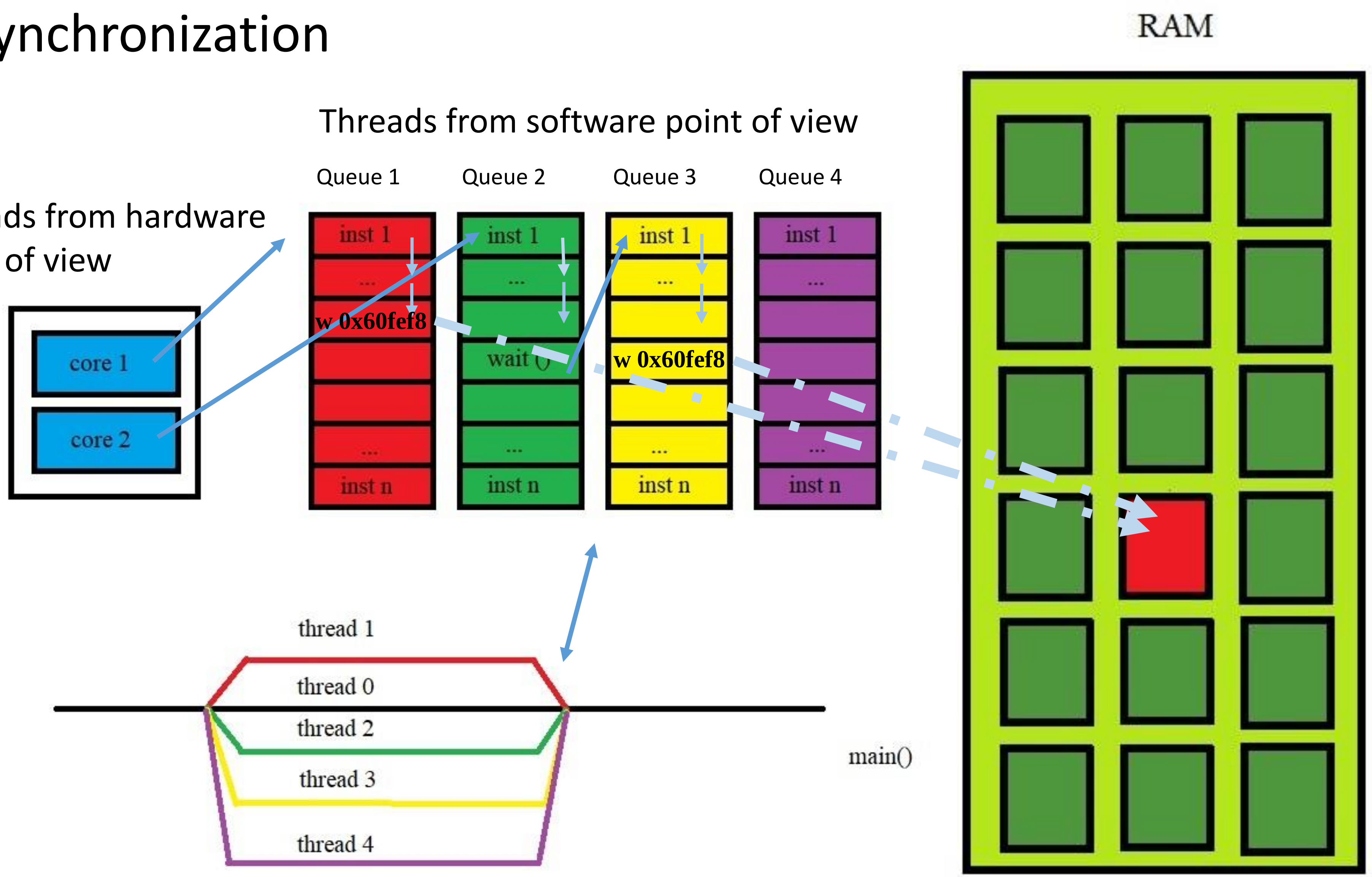
Synchronization



Synchronization

Threads from software point of view

Threads from hardware point of view



Data access needs to be synchronized

Synchronization techniques

Busy waiting – repeatedly check some condition. Once it's true you can modify some shared variable.

Mutex – a technique of allowing only one thread to access a variable at a given point in time using locks. Once a thread **acquires the lock** the shared variable cannot be accessed by other threads.

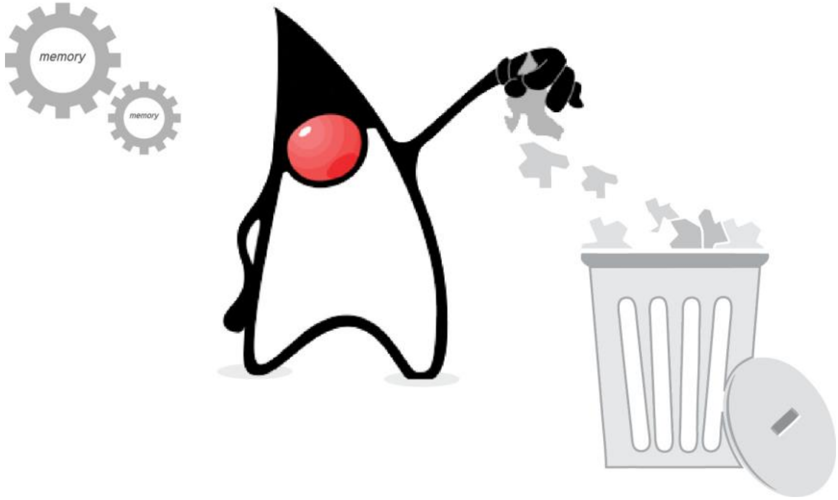
Semaphore – a generalization of mutex. When acquiring the lock semaphore decrements the counter. Once the counter gets to zero other threads have to wait while others unlock the semaphore.

Model examples

Pipeline – if process can be broken down into stages of a single pipeline independent parts can be done in parallel. Or they can be done in parallel on different data (**query processing, web server administration**).

Background Task – a thread running in the background doing "menial" jobs (**back up data, collect garbage**).

User interface – GUI use multithreading to ensure responsiveness of the interface to the user (**progress bars, loading animation**).



Semaphore use case description

Web application:

- it has to process incoming requests from users. Seeing as these requests are independent the processing can be done simultaneously i.e using threads.

- there are millions of requests and the server can only provide a hundred of threads. There is no way this number of threads can deal with such number of requests in a runtime fashion.

- that is why the requests need to be queued up. To do that the servers have queuing threads.

- the length of the queue is limited by the memory.

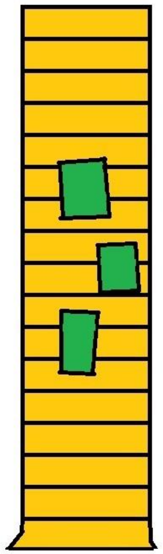
Implementation

The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.

What do we need?

Implementation

The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.

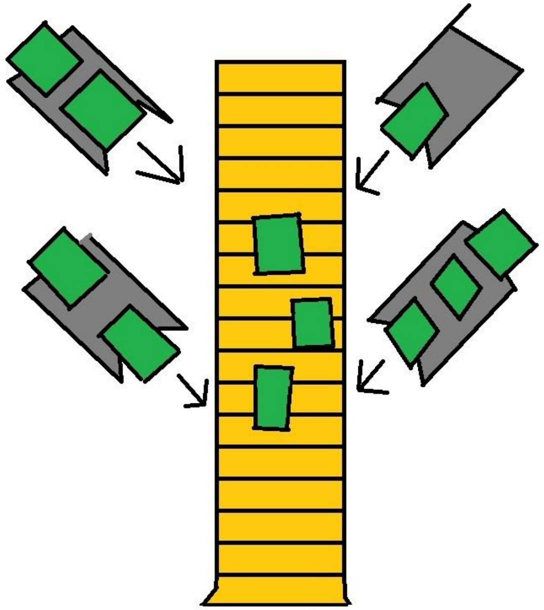


What do we need?

1) Queue – to serialize request processing

Implementation

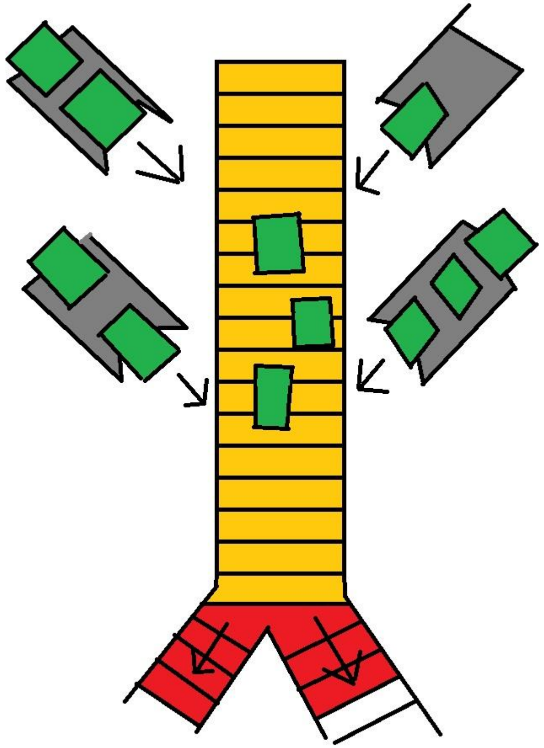
The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.



What do we need?
1) Queue – to serialize request processing
2) Producer/s – thread/s that try to fill the queue

Implementation

The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.

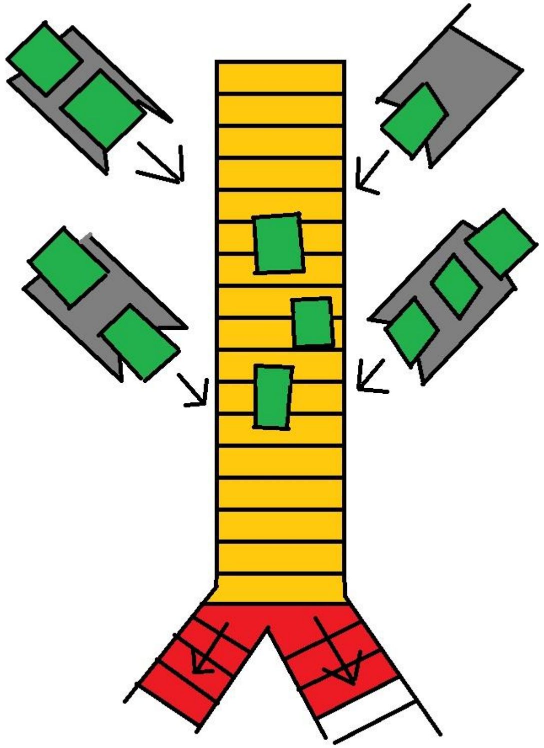


What do we need?

1) Queue – to serialize request processing
2) Producer/s – thread/s that try to fill the queue
3) Consumer/s - thread/s that try to empty the queue

Implementation

The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.

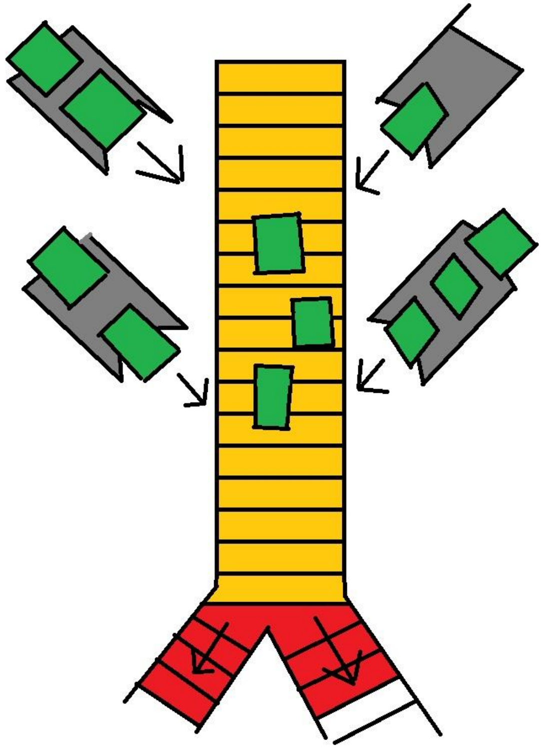


What do we need?
1) Queue – to serialize request processing
2) Producer/s – thread/s that try to fill the queue
3) Consumer/s - thread/s that try to empty the queue

??? Synchronization ???

Implementation

The model for the case is called **Consumer-Producer model**. And this is a typical use case for semaphroes.



What do we need?
1) Queue – to serialize request processing
2) Producer/s – thread/s that try to fill the queue
3) Consumer/s - thread/s that try to empty the queue

??? Synchronization ???

1) Semaphore to protect queue size going over capacity
2) Semaphore to protect queue size going negative

Global interpreter lock (GIL)

Python code is interpreted line by line. In a multithreaded program the interpreter is locked by a single thread. This is needed to avoid segmentation faults.

Hence multithreading can’t properly help with performance if the code needs constant interpretation – **CPU bound programs**.

Alternatively, if the program needs significant amount of time to carry out the already interpreted instruction (**I/O calls**, for instance) then GIL less of an issue

Practice session

1) Introductory examples
2) Multithreading/multiprocessing with Python on more realistic use cases
3) Little bit of practice

Practice session

1) Introductory examples
2) Multithreading/multiprocessing with Python on more realistic use cases
3) Little bit of practice

Conclusions:

1) If the program/function occasionally lets go of the CPU then GIL is less of a problem
2) Multiprocessing adds time overhead on creating and destroying processes
3) Multithreading and multiprocessing add memory usage load
4) Popular libraries (scikit, torch, numpy etc.) usually have multiprocessing/multithreading modules already implemented

Multiprocessing/multithreading inside libraries

Numpy:
- Blas and Lapack via Cython
- ...

Pytorch:
- OpenMP and TBB via Cython
- JIT compilation
- ...

Scikit:
- joblib
- ...