Multithreading and multiprocessing in Python

Computing Methods for Experimental Physics and Data Analysis

Lecture 2A: Hands-on

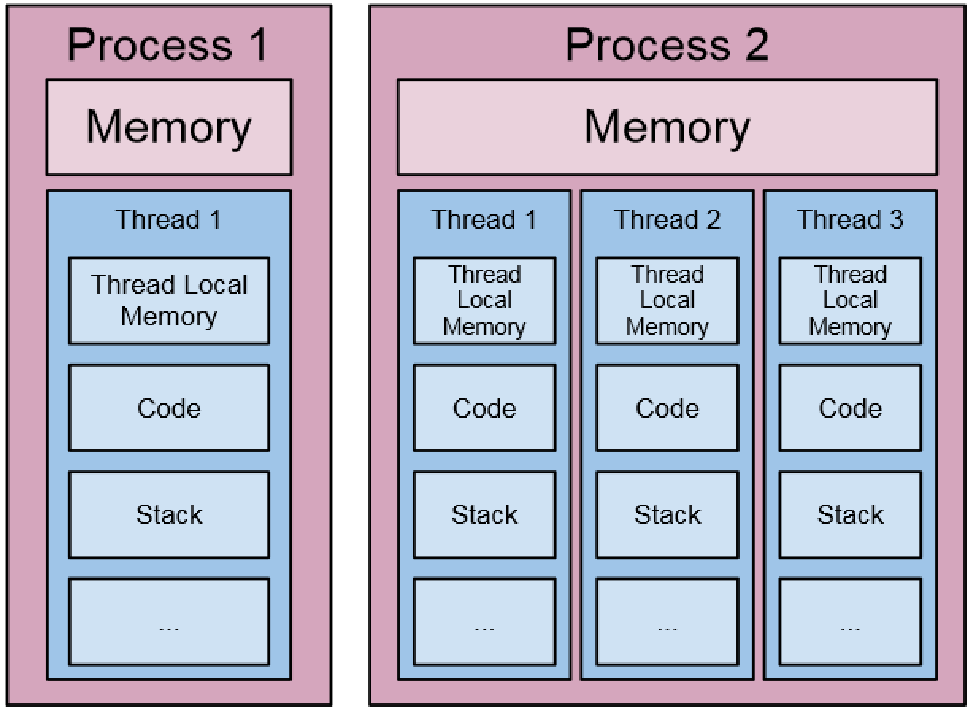
gianluca.lamanna@unipi.it

Recap

- Standard processors are designed for “sequential” programming
- Several “tricks” are applied at instruction level to better exploit the Von Neuman structure
  - → Vector processors, superscalars, pipeline, …
- Starting from about 2005 the performances serial processors start to show saturation
  - → Moore’s law, Denard’s scaling
- to overcome these limitations it is necessary to rethink the way of programming
  - → Concurrency & Parallelism
- The idea: divide the problem in sub-problems to be addressed simultaneously
  - → Different architectures for parallelism: Flynn’s taxonomy

Threads and processes

- Threads and processes are the way to use concurrency in python
- Python implements a very simple global thread-safe mechanism: Global Interpreter Lock (GIL).
  - → Avoid race condition on reference counting for memory management
  - → 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.
  - ➙In addition, the lock is released and reacquired around potentially blocking I/O operations.
- 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

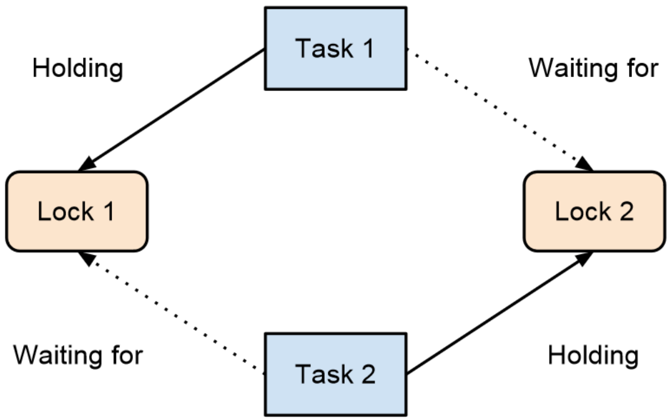
- Subject to GIL (although there are workarounds)
- Not killable
- Potential of race condition
- Same memory space

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.

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.



The multiprocessing module

- HelloWorld
  - ➙Create a process to run the function f()

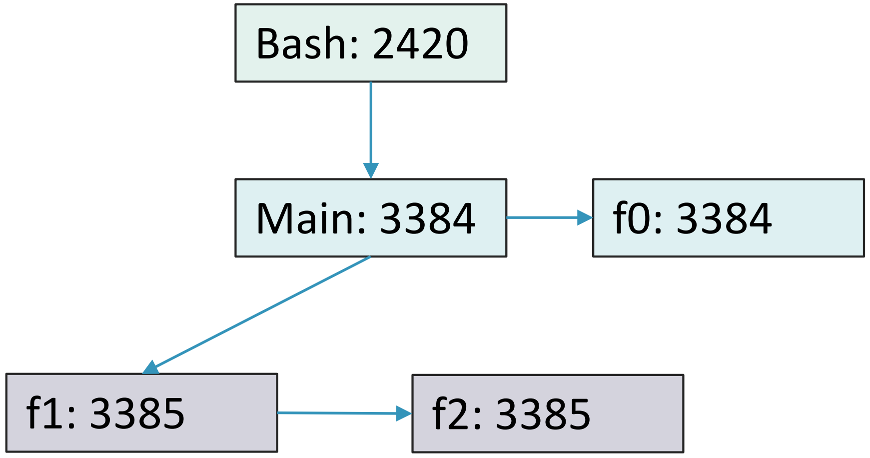
```
from multiprocessing import Process

def f(name):
    print('Hello '+name)

#MAIN
if __name__==“__main__”:
    p = Process(target=f, args=('World‘,))
    p.start()
    p.join()
```

HelloWorld.py

FourProcesses.py

The multiprocessing module

- FatherAndSons
  - ➔ Generate a tree of processes




```python
from multiprocessing import Process
import os
def f0(name):
    print()
    print( "-----> function “ +name)
    print ( "I am still the main process with ID "
          +str(os.getpid())+ " my father is ID:“ +str(os.getppid()))

def f1(name):
    print()
    print( "-----> function “ +name)
    print ( "I am the first sub-process with ID "
          +str(os.getpid())+ " my father is ID:“ +str(os.getppid()))
    f2('two')

def f2(name):
    print()
    print( "-----> function “ +name)
    print ( "I am still the first sub-process with ID "
          +str(os.getpid())+ " my father is ID:“ +str(os.getppid()))
    print("This is the end!")

#MAIN
if __name__==“__main__”:
    print ( "I am the main process with ID: “ +str(os.getpid()))
    f0('zero')
    p = Process(target=f1, args=('one‘ ,))
    p.start()
    p.join()
```

FatherAndSons.py

The multiprocessing module

- Use the Queue to get the result from multiple processes
  - → See later for a more important feature of the queues

```
import multiprocessing as mp

# define a example function
def Hello(pos,name):
    msg="Hello” +name
    output.put((pos, msg))

if __name__==“__main__”:
# Define an output queue
    output = mp.Queue()

# Setup a list of processes that we want to run
    processes = [mp.Process(target=Hello, args=(x, "Gianluca“)) for x in range(4)]

# Run processes
    for p in processes:
        p.start()

# Exit the completed processes
    for p in processes:
        p.join()

# Get process results from the output queue
    results = [output.get() for p in processes]

    print(results)
```

FourProcesses.py

The multiprocessing module

- How to distribute work to workers (aka cpu cores)
- Use the Pool class
  - ➞ Try Pool.map
  - ➞ Try Pool.map_async
- See also Pool.apply e Pool.apply_sync

```
def cube(x):
    print (str(os.getpid())+" "+str(os.getppid()))
    return x**3

#MAIN
if __name__==“__main__”:
    pool = mp.Pool(processes=4)
    results = pool.map(cube,range(1,7))
    print(results)
```

```
…
#MAIN
if __name__==“__main__”:
    pool = mp.Pool(processes=4)
    results = pool.map_async(cube,range(1,7))
    print(results.get())
…
```

PoolExample.py

The multiprocessing module

- Another example with pool.map and pool.map_async
  - → Notice the time measurement

```
import multiprocessing as mp
import time
import os

def doingstuffs(x):
    print ("Process: "+str(x)+" "+str(os.getpid()))
    time.sleep(1)

if __name__==“__main__”:
    start=time.time()
    pool = mp.Pool(processes=4)
    results = pool.map(doingstuffs,range(1,10))
    end=time.time()
    print("elapsed time: "+str(end-start))
```

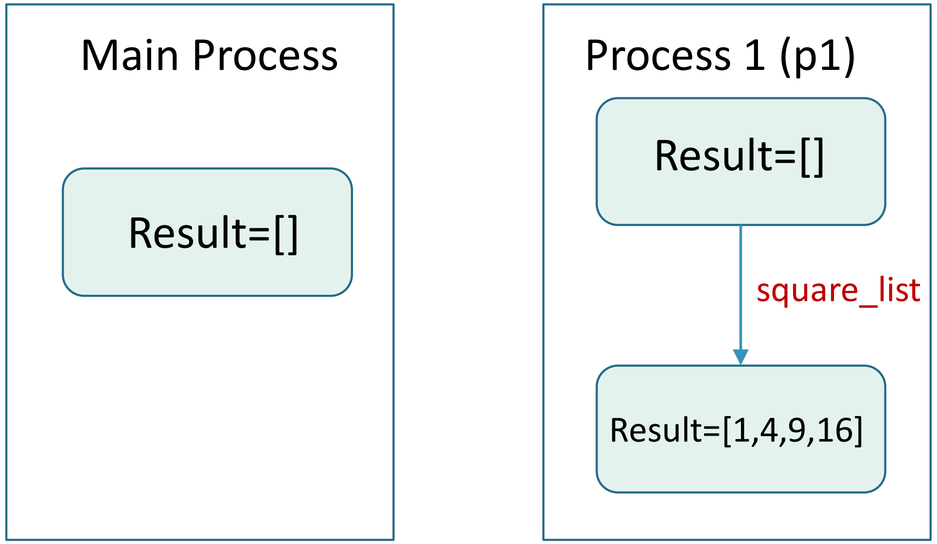
PoolExample2.py

```
…
results = pool.map_async(doingstuffs,range(1,10))
…
print(results.get())
```

Communication between processes

```python
import multiprocessing

# empty list with global scope
result = []

def square_list(mylist):
        global result
        for num in mylist:
                result.append(num * num)
        print("Result(in process p1):"+str(result))

#MAIN
if __name__==“__main__”:
    # input list
    mylist = [1,2,3,4]
    # creating new process
    p1 = multiprocessing.Process(target=square_list,args=(mylist,))
    # starting process
    p1.start()
    # wait until process is finished
    p1.join()

    # print global result list
    print("Result(in main program): "+str(result))
```

communication1.py

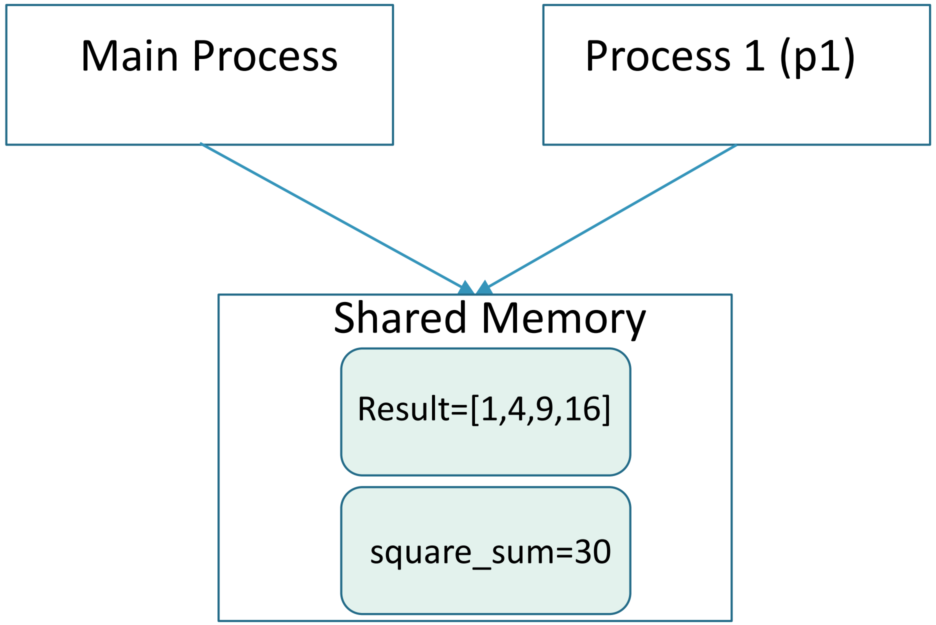


- Different memory spaces allocated for each process
  - → Try to print result in both processes

Comm. between processes: shared memory

```
import multiprocessing

def square_list(mylist, result, square_sum):
    for idx, num in enumerate(mylist):
        result[idx] = num * num
    # square_sum value
    square_sum.value = sum(result)
    # print result Array
    print("Result(in process p1): " +str(result[:]))
    # print square_sum Value
    print("Sum of squares(in process p1): " +str(square_sum.value))

 if __name__==“__main__”:
     # input list
     mylist = [1,2,3,4]
     # creating Array of int data type with space for 4 integers
     result = multiprocessing.Array('i', 4)
     # creating Value of int data type
     square_sum = multiprocessing.Value('i')
     # creating new process
     p1 = multiprocessing.Process(target=square_list, args=(mylist, result, square_sum))

     # starting process
     p1.start()
     # wait until process is finished
     p1.join()
     # print result array
     print("Result(in main program): " +str(result[:]))
     # print square_sum Value
     print("Sum of squares(in main program): " +str(square_sum.value))
```

communication2.py

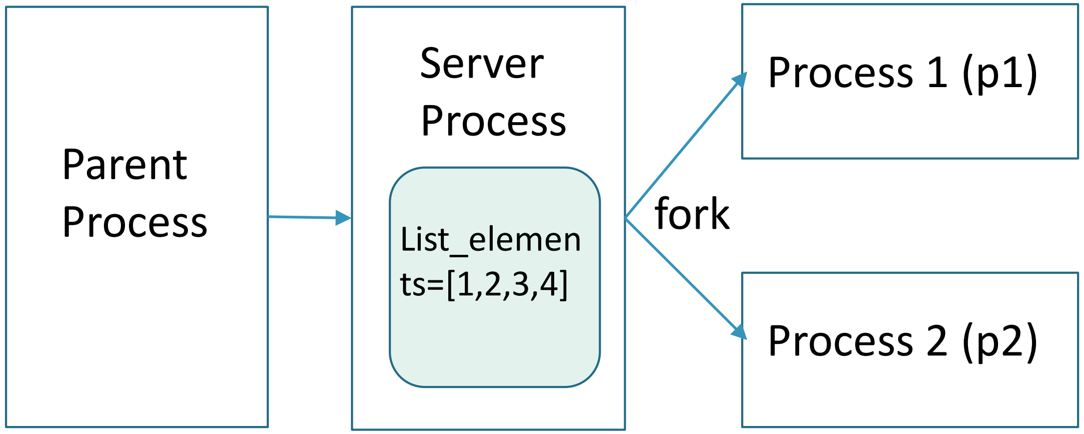


- Shared memory : multiprocessing module provides Array and Value objects to share data between processes.
  - ➔ Array: array allocated from shared memory.
  - ➔ Value: object allocated from shared memory.

Comm. between processes: server process

```python
import multiprocessing

def add_element(record,records):
    records.append(record)
    print("New element added to records list")

def sum_elements(records):
    summ=sum(records)
    print("New sum is: "+str(summ))

#MAIN
with multiprocessing.Manager() as manager:
    list_elements=[1,2,3,4]
    records=manager.list(list_elements)
    new_element=5

    print("Old sum is: "+str(sum(list_elements)))
    #creating new processes
    p1 = multiprocessing.Process(target=add_element, args=(new_element,records))
    p2 = multiprocessing.Process(target=sum_elements, args=(records,))

    #running process p1 to insert new element
    p1.start()
    p1.join

    #running process p2 to sum list elements
    p2.start()
    p2.join()
```

communication3.py

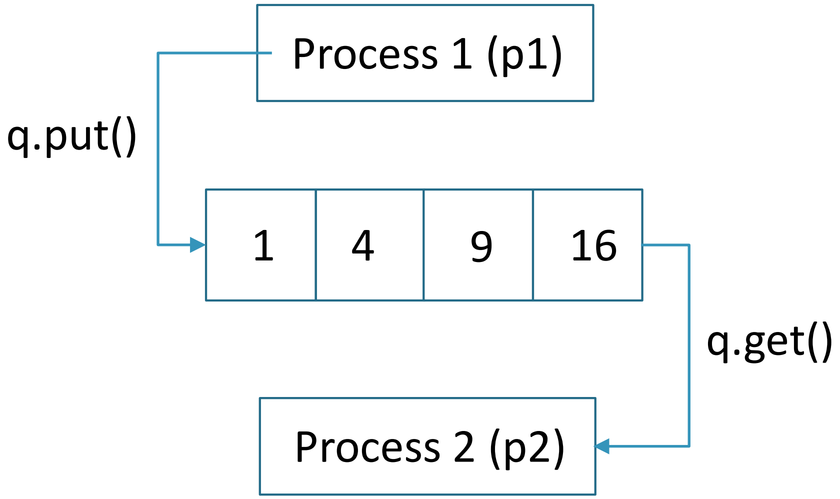


- Server process : Whenever a python program starts, a server process is also started. From there on, whenever a new process is needed, the parent process connects to the server and requests it to fork a new process.
  - ➔ A server process can hold Python objects and allows other processes to manipulate them.
  - ➔ multiprocessing module provides a Manager class which controls a server process. Hence, managers provide a way to create data which can be shared between different processes.
  - ➔ Server process allows to share any type of object (dict, lists,…). It is also possible to connect a server process to the network

Comm. between processes: queue

```python
import multiprocessing

def square_list(mylist, q):
    # append squares of mylist to queue
    for num in mylist:
        q.put(num * num)

def print_queue(q):
    print("Queue elements:")
    while not q.empty():
        print(q.get())
    print("Queue is now empty!")

#MAIN
if __name__==“__main__”:
    # input list
    mylist = [1,2,3,4]
    # creating multiprocessing Queue
    q = multiprocessing.Queue()

    # creating new processes
    p1 = multiprocessing.Process(target=square_list, args=(mylist, q))
    p2 = multiprocessing.Process(target=print_queue, args=(q,))

    # running process p1 to square list
    p1.start()
    p1.join()
    # running process p2 to get queue elements
    p2.start()
    p2.join()
```

communication4.py

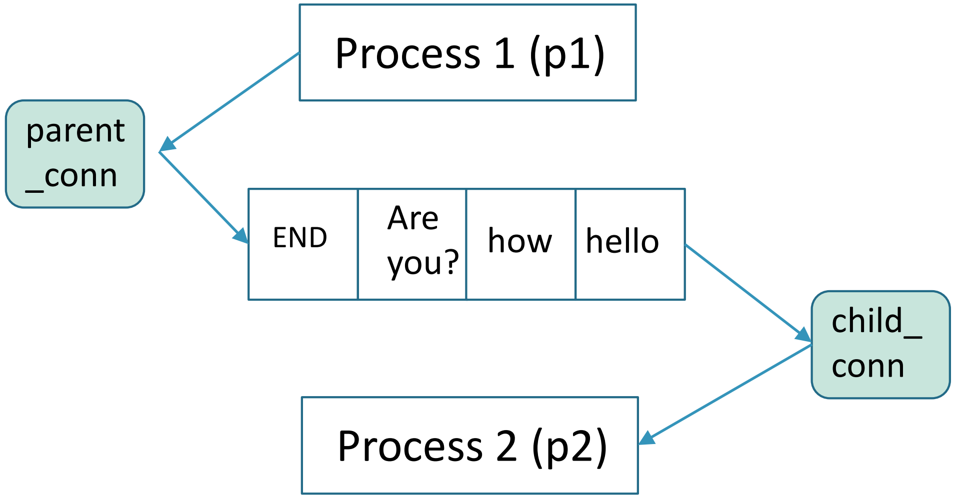


- Queue : A simple way to communicate between process with multiprocessing is to use a Queue to pass messages back and forth.
  - → Any Python object can pass through a Queue.

Comm. between process: pipe

```python
import multiprocessing
import time
def sender(conn, msgs):
    for msg in msgs:
        time.sleep(1)
        conn.send(msg)
        print("Sent the message: "+str(msg))
    conn.close()

def receiver(conn):
    time.sleep(2)
    while 1:
        msg = conn.recv()
        if msg == "END":
            break
        print("Received the message: "+str(msg))
#MAIN
if __name__==“__main__”:
    # messages to be sent
    msgs = ["hello,", "how", "are you?", "END"]
    # creating a pipe
    parent_conn, child_conn = multiprocessing.Pipe()
    # creating new processes
    p1 = multiprocessing.Process(target=sender, args=(parent_conn,msgs))
    p2 = multiprocessing.Process(target=receiver, args=(child_conn,))
    # running processes
    p1.start()
    p2.start()
    # wait until processes finish
    p1.join()
    p2.join()
```

communication5.py



- Pipes : A pipe can have only two endpoints.
  - ➙ Hence, it is preferred over queue when only two-way communication is required. Queue is slower (it's built on topo of pipe)
- multiprocessing module provides Pipe() function which returns a pair of connection objects connected by a pipe.
  - ➙ The two connection objects returned by Pipe() represent the two ends of the pipe.
  - ➙ Each connection object has send() and recv() methods (among others).

Synchronization between processes

- Process synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as critical section.
- A race condition occurs when two or more processes can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

```
import multiprocessing

def withdraw(balance):
        for x in range(10000):
                balance.value = balance.value - 1
def deposit(balance):
        for x in range(10000):
                balance.value = balance.value + 1

def perform_transactions():
        # initial balance (in shared memory)
        balance = multiprocessing.Value('i', 100)
        # creating new processes
        p1 = multiprocessing.Process(target=withdraw, args=(balance,))
        p2 = multiprocessing.Process(target=deposit, args=(balance,))
        # starting processes
        p1.start()
        p2.start()
        # wait until processes are finished
        p1.join()
        p2.join()
        # print final balance
        print("Final balance = {}".format(balance.value))

#MAIN
if __name__==“__main__”:
    for x in range(10):
        # perform same transaction process 10 times
        perform_transactions()
```

synchro1.py

Synchronization between processes

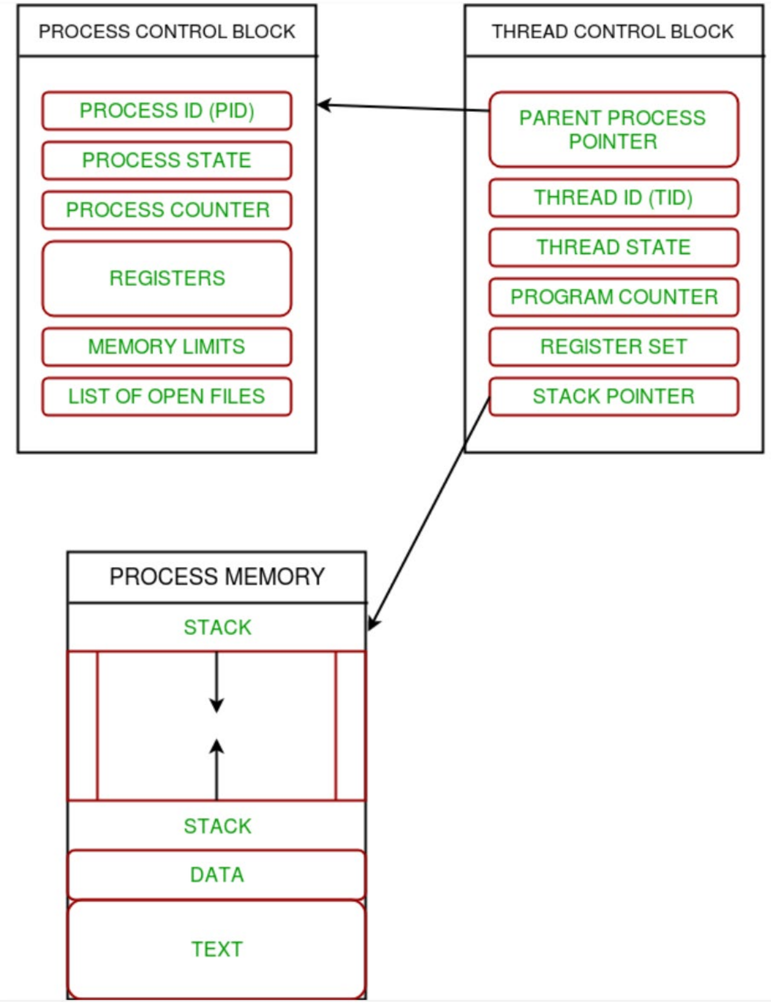
- multiprocessing module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System.
  - A semaphore is a synchronization object that controls access by multiple processes to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again.

```python
import multiprocessing

# function to withdraw from account
def withdraw(balance, lock):
    for x in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()
# function to deposit to account
def deposit(balance, lock):
    for x in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()
def perform_transactions():
        # initial balance (in shared memory)
        balance = multiprocessing.Value('i', 100)
        # creating a lock object
        lock = multiprocessing.Lock()
        # creating new processes
        p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
        p2 = multiprocessing.Process(target=deposit, args=(balance,lock))
        # starting processes
        p1.start()
        p2.start()
        # wait until processes are finished
        p1.join()
        p2.join()
        # print final balance
        print("Final balance = "+str(balance.value))
#MAIN
for x in range(10):
    # perform same transaction process 10 times
    perform_transactions()
```

synchro2.py

Threading

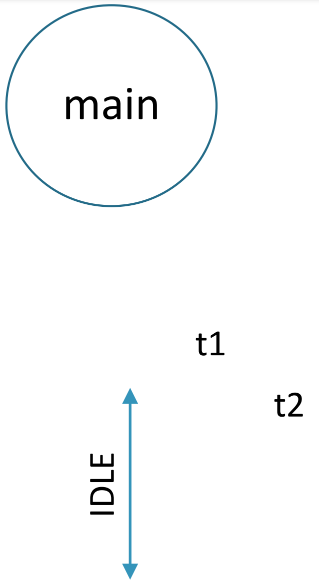
- A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
  - ➡ In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!
- Multiple threads can exist within one process where:
  - ➡ Each thread contains its own register set and local variables (stored in stack).
  - ➡ All thread of a process share global variables (stored in heap) and the program code.



Threading module

```
import threading
import os

def task1():
     print("Task 1 assigned to thread: "+threading.current_thread().name)
     print("ID of process running task 1: "+str(os.getpid()))
def task2():
     print("Task 2 assigned to thread: "+threading.current_thread().name)
     print("ID of process running task 2: "+str(os.getpid()))
#MAIN
if __name__==“__main__”:
    # print ID of current process
    print("ID of process running main program: "+str(os.getpid()))
    # print name of main thread
    print("Main thread name: "+threading.main_thread().name)

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')
    # starting threads
    t1.start()
    t2.start()
    # wait until all threads finish
    t1.join()
    t2.join()
```

thread1.py



- The threads aren’t different processes
- Due to GIL the parallelism is only «Logic»

Threads synchronization

```python
import threading
# global variable x
x = 0

def increment():
    global x
    x += 1

def thread_task():
    for _ in range(100000):
        increment()

def main_task():
    global x
    # setting global variable x as 0
    x = 0
    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)
    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()

#MAIN
for i in range(10):
    main_task()
    print("Iteration {0}: x = {1}".format(i,x))
```

thread2.py
thread2b.py

```python
def thread_task(lock):
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()
```

```python
# creating a lock
lock = threading.Lock()

# creating threads
t1 = threading.Thread(target=thread_task, args=(lock,))
t2 = threading.Thread(target=thread_task, args=(lock,))
```

- Python is not thread-safe
- The scope of the memory is shared by all threads (global x)
- Unpredictable behaviour without lock

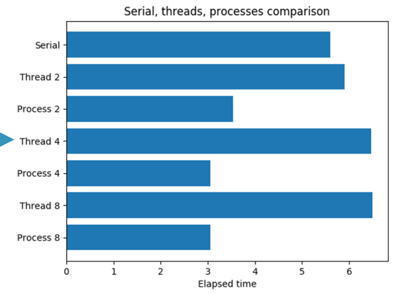
Comparison between Threads and Processes

- Write a code to factorize a list of numbers
  - → The 300 odd numbers from 1000000000001 and 1000000000597
- Try to benchmark the time needed to factorize this list by using:
  - → Serial code
  - → 2,4,8 Threads
  - → 2,4,8 Processes
- Produce a plot with the results

MAIN → Benchmark → Serial_factorizer / Threads_factorizer / process_factorizer → Plot_results →

Benchmark:
- Measure the execution time for each launch

Threads_factorizer:
- Define the correct number of threads/processes
- For each threads/processes assign the correct chunck of numbers to examine



Comparison between Threads and Processes

```python
import math
import multiprocessing
import random
import threading
import time
import matplotlib.pyplot as plt
import numpy


class Timer(object):
    def __init__(self, name=None):
        self.name = name
        self.timee=0

    def __enter__(self):
        self.tstart = time.time()

    def __exit__(self, type, value, traceback):
        if self.name:
            print('[%s]' % self.name, end=' ')
        self.timee=(time.time() - self.tstart)
        print('Elapsed: %s' % (time.time() - self.tstart))
        self.output()

    def output(self):
        return self.timee
```

```python
def factorize_naive(n):
    """ A naive factorization method. Take integer 'n', return list of
        factors.
    """
    if n < 2:
        return []
    factors = []
    p = 2

    while True:
        if n == 1:
            return factors
        r = n % p
        if r == 0:
            factors.append(p)
            n = n // p
        elif p * p >= n:
            factors.append(n)
            return factors
        elif p > 2:
            # Advance in steps of 2 over odd numbers
            p += 2
        else:
            # If p == 2, get to 3
            p += 1
    assert False, "unreachable"
```

Final_example.py

Comparison between Threads and Processes

```python
# Each "factorizer" function returns a dict mapping num -> factors
def serial_factorizer(nums):
    return {n: factorize_naive(n) for n in nums}

def threaded_factorizer(nums, nthreads):
    def worker(nums, outdict):
        """ The worker function, invoked in a thread. 'nums' is a
            list of numbers to factor. The results are placed in
            outdict.
        """
        for n in nums:
            outdict[n] = factorize_naive(n)

    # Each thread will get 'chunksize' nums and its own output dict
    chunksize = int(math.ceil(len(nums) / float(nthreads)))
    threads = []
    outs = [{} for i in range(nthreads)]

    for i in range(nthreads):
        # Create each thread, passing it its chunk of numbers to factor
        # and output dict.
        t = threading.Thread(
                target=worker,
                args=(nums[chunksize * i:chunksize * (i + 1)],
                      outs[i]))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    # Merge all partial output dicts into a single dict and return it
    return {k: v for out_d in outs for k, v in out_d.items()}
```

```python
def mp_worker(nums, out_q):
    """ The worker function, invoked in a process. 'nums' is a
        list of numbers to factor. The results are placed in
        a dictionary that's pushed to a queue.
    """
    outdict = {}
    for n in nums:
        outdict[n] = factorize_naive(n)
    out_q.put(outdict)

def mp_factorizer(nums, nprocs):
    # Each process will get 'chunksize' nums and a queue to put his out
    # dict into
    out_q = multiprocessing.Queue()
    chunksize = int(math.ceil(len(nums) / float(nprocs)))
    procs = []
    for i in range(nprocs):
        p = multiprocessing.Process(
                target=mp_worker,
                args=(nums[chunksize * i:chunksize * (i + 1)],
                      out_q))
        procs.append(p)
        p.start()
    resultdict = {}
    for i in range(nprocs):
        resultdict.update(out_q.get())
    # Wait for all worker processes to finish
    for p in procs:
        p.join()
    return resultdict
```

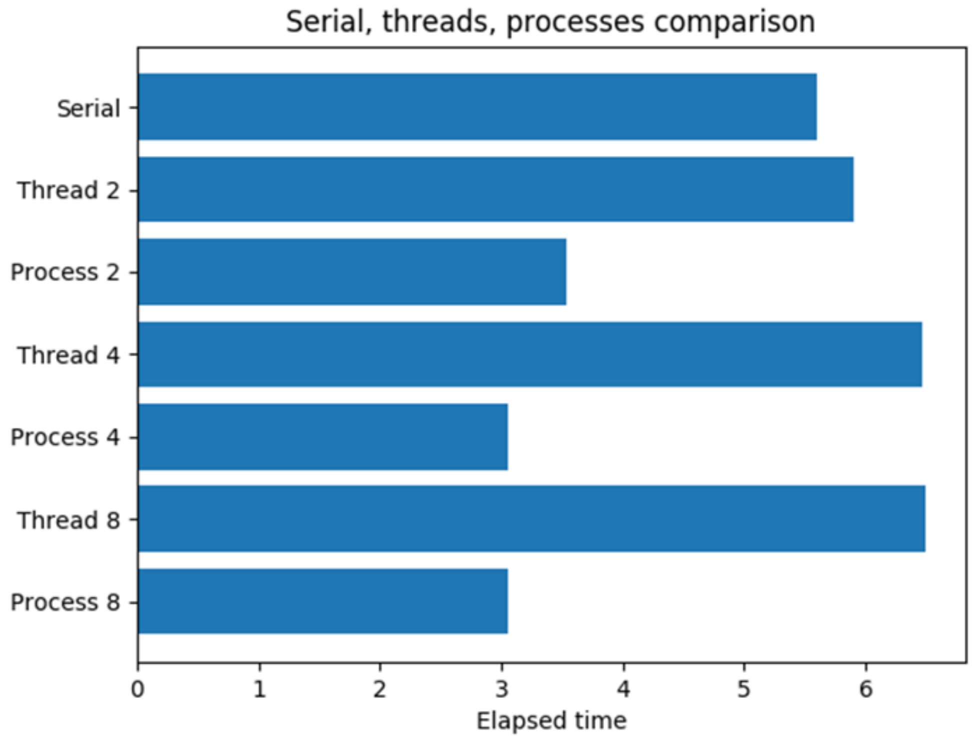
Final_example.py

Comparison between Threads and Processes

Final_example.py

```python
def plot_results(elapsed):
    plt.rcdefaults()
    fig, ax = plt.subplots()
    laby = ('Serial','Thread 2','Process 2','Thread 4','Process 4','Thread 8','Process 8')
    y_pos = numpy.arange(len(laby))
    ax.barh(y_pos, elapsed, align='center')
    ax.set_yticks(y_pos)
    ax.set_yticklabels(laby)
    ax.invert_yaxis()  # labels read top-to-bottom
    ax.set_xlabel('Elapsed time')
    ax.set_title('Serial, threads, processes comparison')
    plt.show()
    wait()
def benchmark(nums):
    print('Running benchmark...')
    elapsed_times=[]
    tserial=Timer('serial')
    with tserial as qq:
        s_d = serial_factorizer(nums)
    elapsed_times.append(tserial.output())
    for numparallel in [2, 4, 8]:
        tthread=Timer('threaded %s' % numparallel)
        with tthread as qq:
            t_d = threaded_factorizer(nums, numparallel)
        elapsed_times.append(tthread.output())
        tmpar=Timer('mp %s' % numparallel)
        with tmpar as qq:
            m_d = mp_factorizer(nums, numparallel)
     elapsed_times.append(tmpar.output())
     plot_results(elapsed_times)
```

```python
#MAIN
if __name__==“__main__”:
    N = 299
    nums = [999999999999]
    for i in range(N):
        nums.append(nums[-1] + 2)
    benchmark(nums)
```



Why should I use threads?

```
import requests
import threading as thr
from time import perf_counter

buffer_size=1024

#define a function to manage the download
def download(url):
    response = requests.get(url, stream=True)
    filename = url.split("/")[-1]
    with open(filename,"wb") as f:
        for data in response.iter_content(buffer_size):
            f.write(data)

#MAIN
if __name__ == "__main__":
    urls= [
        "http://cds.cern.ch/record/2690508/files/201909-262_01.jpg",
        "http://cds.cern.ch/record/2274473/files/05-07-2017_Calorimeters.jpg",
        "http://cds.cern.ch/record/2274473/files/08-07-2017_Spectrometer_magnet.jpg",
        "http://cds.cern.ch/record/2127067/files/_MG_3944.jpg",
        "http://cds.cern.ch/record/2274473/files/08-07-2017_Electronics.jpg",
    ]

    t = perf_counter()

#sequential download
    for url in urls:
        download(url)

    print("Time: "+str(perf_counter()-t))
```

threadIO_seq.py

- GIL is bypassed in two cases:
  - ➜ running programs in external C code (ex: numpy)
  - ➜ in case of I/O operation: Python release the lock waiting for I/O
- A tipical application is the use of the network
  - ➜ writing to a disk, display an image to the screen, print on a printer,…

Why should I use threads?

```
import threading as thr
import requests
import os
from time import perf_counter

buffer_size=1024

#define a function to manage the download
def download(url):
    response = requests.get(url, stream=True)
    filename = url.split("/")[-1]
    with open(filename,"wb") as f:
        for data in response.iter_content(buffer_size):
            f.write(data)

#MAIN
if __name__ == "__main__":
    urls= [
        "http://cds.cern.ch/record/2690508/files/201909-262_01.jpg",
        "http://cds.cern.ch/record/2274473/files/05-07-2017_Calorimeters.jpg",
        "http://cds.cern.ch/record/2274473/files/08-07-2017_Spectrometer_magnet.jpg",
        "http://cds.cern.ch/record/2127067/files/_MG_3944.jpg",
        "http://cds.cern.ch/record/2274473/files/08-07-2017_Electronics.jpg",
    ]

#define 5 threads
    threads = [thr.Thread(target=download, args=(urls[x],)) for x in range(4)]

    t = perf_counter()
```

threadIO_par.py

```
#start threads
    for thread in threads:
        thread.start()

#join threads
    for thread in threads:
        thread.join()


    print("Time: "+str(perf_counter()-t))
```

- Performaces depend on network speed
- Overheads for thread start and lock release

Process vs Thread

Process	Thread
Separate memory	Shared memory
More memory	Less memory
Killable children (but can become zombies)	No zombies
More overhead	Less Overhead
Slower creation and destruction	Faster creation and destruction
Easier to code and debug	Harder to code and debug
No GIL: yes for CPU-bound problems	GIL: No for CPU-bound problems (ok for I/O)

Assignment

- Calculate for each number in a list, the sum of all primes which are smaller than the given number. It should output the pairs [n, sum_primes(n)] sorted by n.
  - Example: The first number is n, the second number is the sum of all primes < n.
- Your task is to calculate range(100000, 2500000, 100000). Please use the multiprocessing module and compare the result (in term of execution time) with the serial version of your code.
- Try to do the same with the threading module.

```
>> [[10, 17],
>>  [20, 77],
>>  [30, 129],
>>  [40, 197],
>>  [50, 328],
>>  [60, 440],
>>  [70, 568],
>>  [80, 791],
>>  [90, 963]]
```