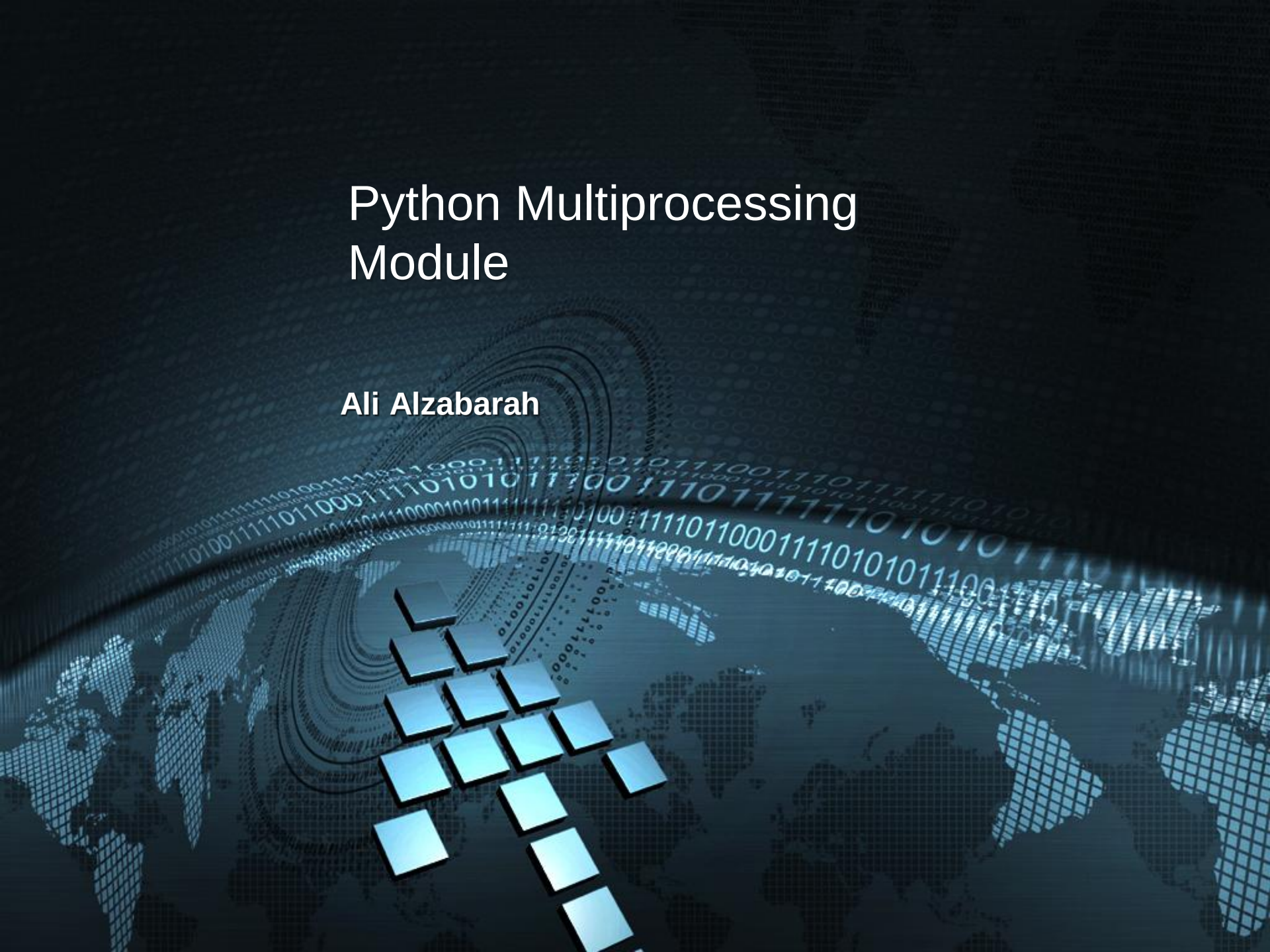
Python Multiprocessing Module

Ali Alzabarah



1. Introduction
2. Python and concurrency
3. Multiprocessing VS Threading
4. Multiprocessing module

5 Pool of worker

6 Distributed concurrency

7 Credit when credit is due

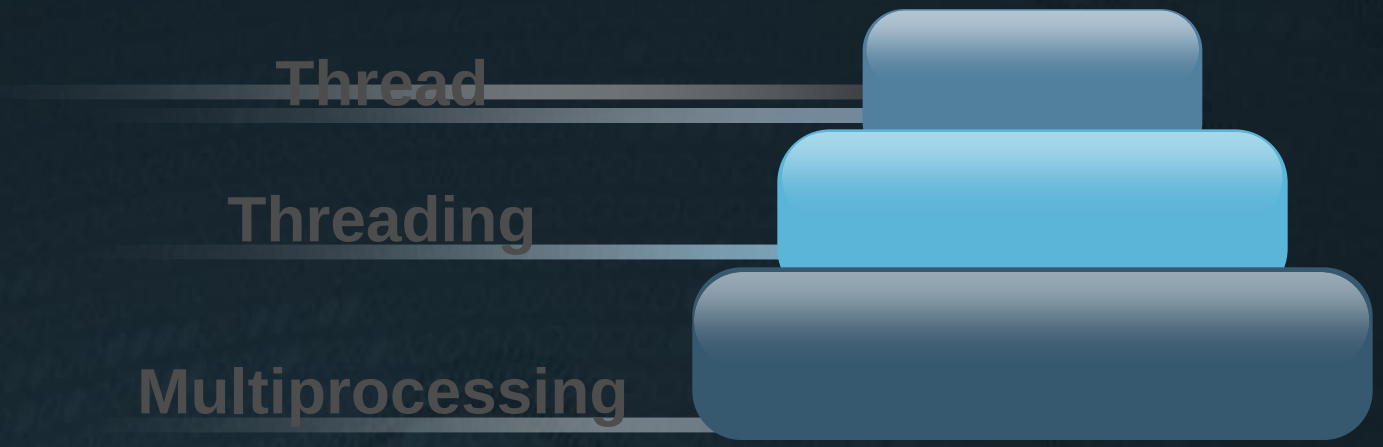
8 References

Introduction

- Thread : is a thread of execution in a program. Aka, lightweight process.
- Process : is an instance of a computer program that is being executed.
- Thread share the memory and state of the parent, process share nothing.
- Process use inter-process communication to communicate, thread do not.

Python and Concurrency

- Python has three concurrency modules :



Python and Concurrency

- **Thread :**
  - Provides low-level primitives for working with multiple threads.
  - Python first implementation of thread, it is old.
  - Not included in Python 3000 .
- **Threading :**
  - Construct higher-level threading interface on top of thread module.

Python and Concurrency

- **Multiprocessing :**
  - Supports spawning process.
  - Offer local and remote concurrency
  - New in python 2.6.
  - Solves the issue in the threading module.

Python and Concurrency

- Why new module?
  - **Python Global Interpreter Lock, GIL, limitation prevents a true parallelism in multi processors machines.**
    - What is GIL ?
      - Lock which must be acquired for a thread to enter the interpreter's space.
      - Lock assures that only one thread executes in the cPython VM at a time.

Python and Concurrency

- How GIL works ?
  - It controls the transfer of control between threads. Python interpreter determine how long a thread's turn runs, NOT the hardware timer.
  - Python uses the OS threads as a base but python itself control the transfer of control between threads.
- For the above reason, true parallelism won't occur with Threading module.
- So, They came up with Multiprocessing to solve this issue.

Python and Concurrency

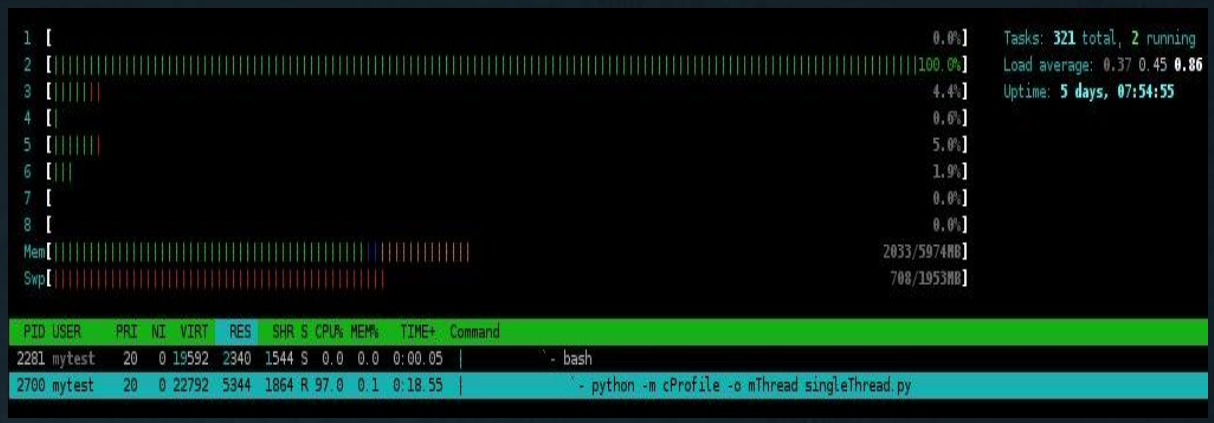
- “Nevertheless, you‘re right the GIL is not as bad as you would initially think: you just have to undo the brainwashing you got from Windows and Java proponents who seem to consider threads as the only way to approach concurrent activities “, Guido van Rossum.

Multiprocessing VS Threading

- Let’s see the problem in action. I analyzed the code that was written by Jesse Noller in depth. I used cProfile and pstats modules to gain an idea of how the code was handled by Python.
- I’m testing the program in Quad-Core machine, 8 CPU’s.

Multiprocessing VS Threading

- Single Thread :



  - **The program took 52.810 CPU seconds.**

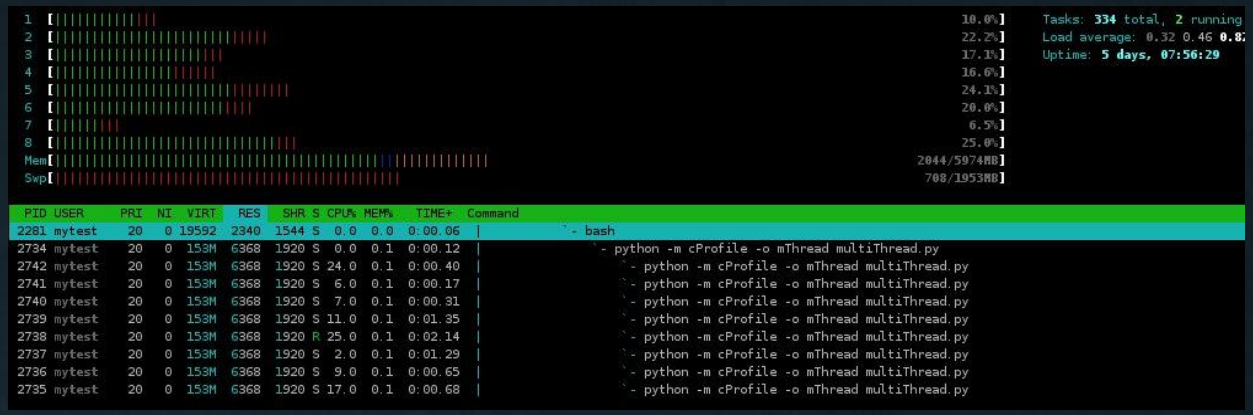
Multiprocessing VS Threading

- Most of the time, it was executing isPrime, sum_primes functions.

```
cumtime  percall filename:lineno(function)
 52.810   52.810 <string>:1(<module>)
 52.810   52.810 {execfile}
 52.809   52.809 singleThread.py:1(<module>)
 52.809    0.264 singleThread.py:17(sum_primes)
 50.002    0.000 singleThread.py:3(isPrime)
  1.545    0.000 {math.sqrt}
  1.309    0.000 {math.ceil}
  0.010    0.000 {sum}
  0.000    0.000 {method 'disable' of '_lsprof.
```

Multiprocessing VS Threading

- Multi Threads :



  - **The program took 59.337 CPU seconds. This is more than what it took the single version of the same program !**

Multiprocessing VS Threading

- Most of the time was used by a built-in method acquire !

```
cumtime  percall filename:lineno(function)
 59.313    0.125 {built-in method acquire}
 54.303    6.788 threading.py:622(join)
  0.001    0.000 threading.py:179(__init__)
  0.001    0.001 threading.py:1(<module>)
 59.337   59.337 multiThread.py:1(<module>)
 59.337   59.337 {execfile}
  0.001    0.000 Queue.py:107(put)
  0.000    0.000 heapq.py:31(<module>)
  0.000    0.000 {thread.start_new_thread}
  0.000    0.000 threading.py:270(notify)
```

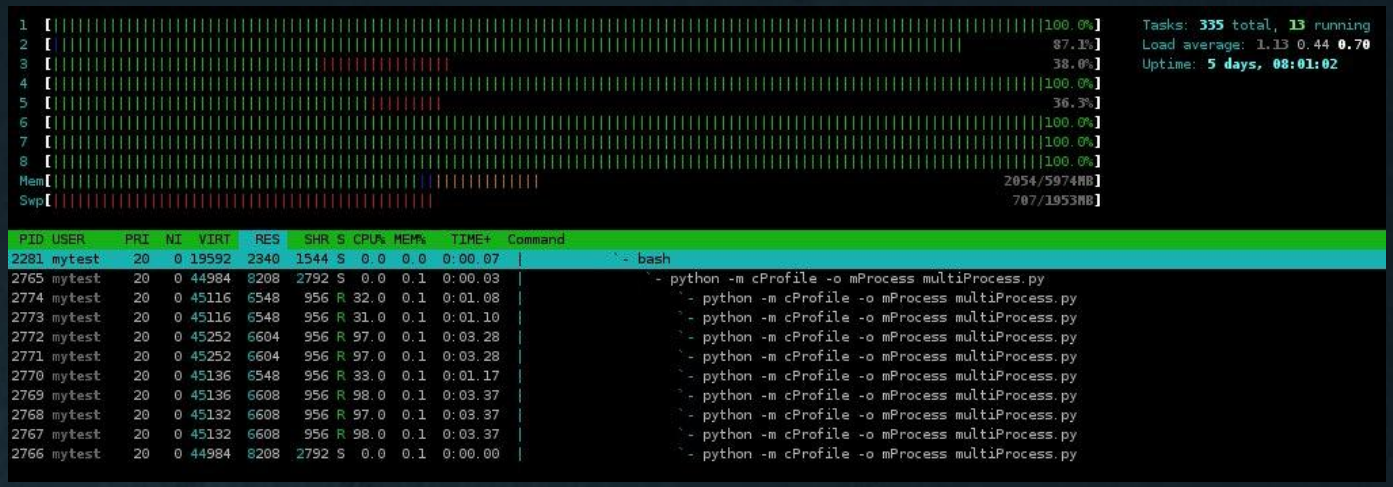
Multiprocessing VS Threading

- Wait ! Built-in method ? Threading acquire method was not in the code.

```
def isPrime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i  = 2
    while i <= max :
        if n % i == 0:
            return False
        i += 1
    return True
def sum_primes(n):
    return sum([x for x in xrange(2,n) if isPrime(x)])


def do_work(q):
    while True:
        try:
            x = q.get(block=False)
            sum_primes(x)
        except Empty:
            break
if __name__ == "__main__":

    work_queue = Queue()
    for i in xrange(0,100000,500):
        work_queue.put(i)

    threads = [ Thread(target=do_work, args=(work_queue,)) for i in range(8) ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

Multiprocessing VS Threading

- Built-in acquire ! This must be the GIL.
- Multi processes :



  - **Took only 11.968 seconds. ~ 5 times faster !**

Multiprocessing VS Threading

- Most of the time was spent in waiting for other processes to finish.

```
cumtime  percall filename:lineno(function)
 11.786    0.327 {posix.waitpid}
  0.166    0.021 {posix.fork}
  0.003    0.003 socket.py:44(<module>)
  0.002    0.001 sre_parse.py:385(_parse)
  0.004    0.004 __init__.py:43(<module>)
  0.001    0.001 socket.py:174(_socketobject)
  0.001    0.001 random.py:40(<module>)
  0.198    0.025 process.py:90(start)
  0.003    0.003 util.py:9(<module>)
  0.002    0.000 queues.py:73(put)
```

Multiprocessing VS Threading

- So, How does multiprocessing module solve the problem ?
  - **It uses subprocesses instead of thread.**
  - **Therefore, it allow the programmer to fully leverage multiple processors on a given machine.**

Multiprocessing VS Threading

- Differences between threading / multiprocessing syntax ? Almost the same.
  - **Threading :**
    - Thread(target=do_work,args=(work_queue,))
  - **Multiprocessing:**
    - Process(target=do_work,args=(work_queue,))
- I'm not going to cover all the functionality that multiprocessing module provides but I will discuss what is new. Any functionality that threading module provides is also in the multiprocessing module.

Multiprocessing Module

- Remember :
  - **Processes share nothing.**
  - **Processes communicate over inter-process communication channel .**
- This was not an issue with Threading module.
- Python developers had to find a way for processes to communicate and share date. Otherwise, The module will not be as efficient as it is.

Exchange Object between Processes

- Multiprocessing module has two communication channels :

Exchange Object between Processes

- Queues :
  - **Returns a process shared queue.**
  - **Any pickle-able object can pass through it.**
  - **Thread and process safe.**
- Pipes :
  - **Returns a pair of connection objects connect by a pipe.**
  - **Every object has send/recv methods that are used in the communication between processes.**

Exchange Object between Processes

- let’s see an example :
  - **Queues simple example :**
    - The program creates two queues :
      - Tasks : queue that has range of int.
      - Results : queue that is empty. It is used to store results.
    - Then creates n workers, each worker get a data , number, from shared queue, multiply it by 2 and store it in the result queue.

Exchange Object between Processes

```
from multiprocessing import Queue, Process, current_process

def worker(tasks,results):
        # get a tasks
        t = tasks.get()
        # do operation
        result = t * 2
        # put the result in results queue
        results.put([current_process().name,t,"*",20,"=",result])


if __name__ == '__main__':
    n = 100
    # create my tasks and results Queues.
    myTasks = Queue()
    myResults = Queue()
    # create n process :
    Workers = [ Process(target=worker, args=(myTasks,myResults)) for i in range(n) ]
    # start processes
    for each in Workers:
        each.start()

    # create tasks
    for each in range(n):
        myTasks.put(each)

    # get the results :
    while n :
        result = myResults.get()
        print "Res : " , result
        n -= 1
```

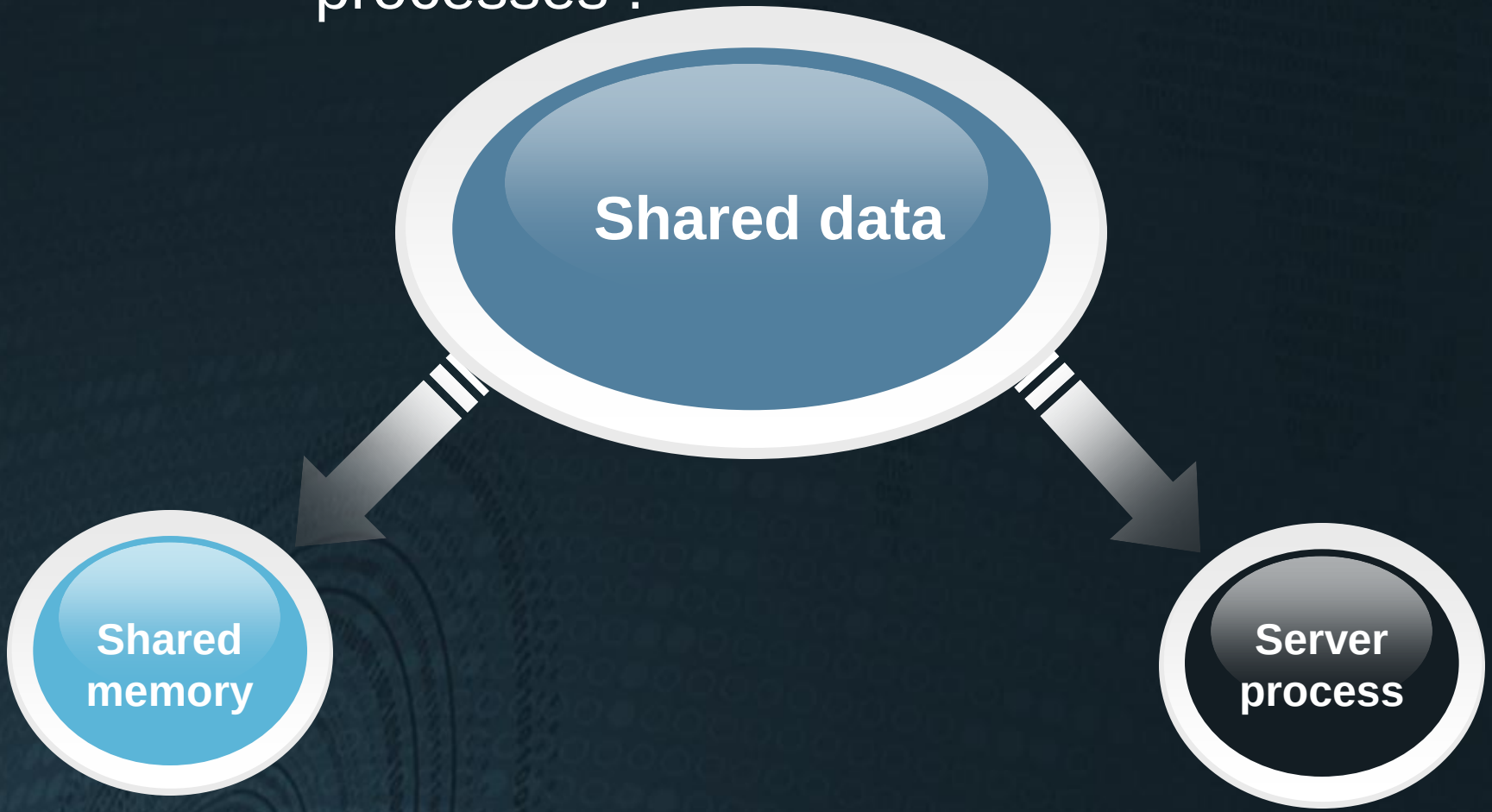
Exchange Object between Processes

- Observation :
  - **Result is not in order even if our tasks' queue was in order. This is because the program run in parallel.**
  - **Queue.get() return the data to the worker and delete it.**
- Part of the output :

```
Res :  ['Process-16', 14, '*', 2, '=', 28]
Res :  ['Process-12', 15, '*', 2, '=', 30]
Res :  ['Process-17', 16, '*', 2, '=', 32]
Res :  ['Process-18', 17, '*', 2, '=', 34]
Res :  ['Process-20', 18, '*', 2, '=', 36]
Res :  ['Process-21', 19, '*', 2, '=', 38]
Res :  ['Process-22', 20, '*', 2, '=', 40]
Res :  ['Process-19', 22, '*', 2, '=', 44]
Res :  ['Process-24', 23, '*', 2, '=', 46]
Res :  ['Process-25', 24, '*', 2, '=', 48]
Res :  ['Process-26', 25, '*', 2, '=', 50]
Res :  ['Process-27', 26, '*', 2, '=', 52]
Res :  ['Process-28', 27, '*', 2, '=', 54]
Res :  ['Process-58', 56, '*', 2, '=', 112]
Res :  ['Process-59', 57, '*', 2, '=', 114]
Res :  ['Process-60', 59, '*', 2, '=', 118]
Res :  ['Process-63', 61, '*', 2, '=', 122]
Res :  ['Process-65', 63, '*', 2, '=', 126]
Res :  ['Process-61', 64, '*', 2, '=', 128]
```

Sharing state between processes

- Multiprocessing module has two ways to share state between processes :



Sharing state between processes

- Shared memory :
  - **Python provide two ways for the data to be stored in a shared memory map:**
    - Value :
      - The return value is a synchronized wrapper for the object.
    - Array :
      - The return value is a synchronized wrapper for the array.

Sharing state between processes

- Server process:
  - **A Manager object control a server process that holds python objects and allow other process to manipulate them.**
    - What is Manager ?
      - Controls server process which manages shared object.
      - It make sure the shared object get updated in all processes when anyone modifies it.

Sharing state between processes

- Let's see an example of sharing state between processes :
  - **The program create a Manager list, share it between n number of workers, every worker update an index.**
  - **After all workers finish, the new list is printed to stdout.**

Sharing state between processes

- Server process simple example :

```
from multiprocessing import Manager, Process, current_process

# function that takes a list, every worker access on index. The index
# is actually the worker number. The worker will update the value of the index
# to be the old value to the power2.

def worker(aList):
        aList[ int(current_process().name.split("-")[1]) -1 ] = (int(current_process().name.split("-")[1]) -1) * \
                                (int(current_process().name.split("-")[1]) -1)

if __name__ == '__main__':
    n = 100
    # create a Manager
    manager = Manager()
    # create a list that is managed by the manager
    l = manager.list()
    # fill the list with number from 0 to n
    l.extend(range(n))
    # create processes
    p = [ Process(target=worker, args=(l,)) for i in range(n -1)]

    # start the processes
    for each in p :
            each.start()
    # join the processes
    for each in p :
            each.join()
    # print the final result
    print "Final result : ", l
```

Sharing state between processes

- Observation :
  - **We did not have to worry about synchronizing the access to the list. The manager took care of that.**
  - **all processes see the same list and act on one shared list.**
- Result when n = 10000 :

```
Final result :  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121,
144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529,
576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156,
1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764,
1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704,
2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844,
3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184,
5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724,
6889, 7056, 7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464,
8649, 8836, 9025, 9216, 9409, 9604, 9801]
```

Multiprocessing module

- Summary of the last 10 slides :
  - **Communication channels :**
    - Queues
    - Pipes
  - **Shared data :**
    - Shared memory :
      - value
      - array
    - Server :
      - Manager
- Let’s discover other cool features in our module :

Pool of worker

- Multiprocessors module has a Pool class that :
  - **Distribute the work between worker.**
  - **Collect the return value as a list.**
- You do not have to worry about managing queue , processes, shared date/stats yourself.
- It makes it easy to implement quick/simple concurrent program.
- Lets see an example :

Pool of worker

- Program reads list of words from a file and returns a list of list where every list contains a word and its length.

```
from multiprocessing import Pool

def aFunction(x):
    return [ [len(z),z] for z in x]

if __name__ == '__main__':
    # create a pool that has 8 workers
    pool = Pool(processes=8)
    # read the file in
    f = open('stopWord.txt','r')
    l = f.readlines()
    f.close()
    # remove new line character
    d = [ w.strip() for w in l ]
    # get the result asynchronously
    result = pool.apply_async(aFunction, [d])
    # print out the result
    print result.get()
```

Pool of worker

- Observation :
  - **we did not do any work beside telling the program how many workers we want in the pool.**
- Part of the result :

```
[3, 'any'], [3, 'app'], [11, 'application'], [9, 'available'], [4, 'back'],
[7, 'because'], [6, 'before'], [5, 'being'], [5, 'below'], [4, 'best'], [6, 'better'],
[7, 'between'], [4, 'both'], [3, 'box'], [4, 'buch'], [3, 'but'], [6, 'called'], [3, 'can'],
[7, 'certain'], [6, 'change'], [5, 'check'], [5, 'click'], [5, 'could'], [6, 'create'],
[7, 'created'], [8, 'creating'], [7, 'current'], [9, 'currently'], [3, 'did'],
[9, 'different'], [4, 'does'], [5, 'doesn'], [3, 'don'], [4, 'down'], [4, 'each'],
[4, 'easy'], [4, 'edit'], [3, 'end'], [3, 'etc'], [4, 'even'], [5, 'first'], [5, 'fixme'],
[9, 'following'], [5, 'found'], [4, 'full'], [4, 'full'], [7, 'general'], [8, 'generate'],
[3, 'get'], [4, 'give'], [5, 'going'], [4, 'good'], [3, 'got'], [3, 'had'], [3, 'has'],
[4, 'have'], [4, 'help'], [4, 'here'], [9, 'highlight'], [4, 'home'], [7, 'instead'],
[3, 'its'], [4, 'just'], [3, 'key'], [4, 'know'], [4, 'like'], [4, 'line'], [4, 'link'],
[5, 'links'], [4, 'list'], [4, 'look'], [4, 'many'], [4, 'menu'], [5, 'might'], [4, 'more'],
```

- It can not be easier than this ☺

Distributed concurrency

- Recall that the Manager in multiprocessing module controls a server process that manages a share object.
- That server can be accessed remotely and the shared object can be distributed to many clients. Whenever a client update the shared object, every other client will see the change.

Distributed concurrency

- To create the server :
  - **Create a class that inherit BaseManager class.**
  - **Call the class method “register” to assign a name to what you want to share.**
  - **Define the address which your server will be listening on.**
  - **Call the function get_server and serve_forever to run the server.**
- To create the client :

Distributed concurrency

- Register the name of the object that the server is sharing.
- Connect to the server address.
- Call the name of the shared object.

- Lets see an Example :

```
# Server :

# create class that inherit the BaseManager class
class QueueManager(BaseManager):
    pass
# register our queue
QueueManager.register('get_queue', callable=lambda:queue)
# define the address of the server
m = QueueManager(address=('', 50000))
# get the server
s = m.get_server()
# run for ever
s.serve_forever()
###################################
# client :
from multiprocessing.managers import BaseManager
class QueueManager(BaseManager):
    pass
QueueManager.register('get_queue')
m = QueueManager(address=('127.0.0.1', 50000))
m.connect()
queue = m.get_queue()
queue.put("Hi There")
```

Conclusion

- Multiprocessing module is a powerful addition to python. it solved the GIL problem and introduced easy way to achieve true parallelism.
- Implementing a concurrent program is not easy, but with the way this module works, I think it makes the programmer job much easier.

Credit when credit is due

- The example in slide 10 was based on an example on a presentation by Jesse Noller on PyWorks,Atlanta 2008.
- My explanation of the GIL problem and the solution were in Jesse Noller presentation and also in a tutorial by Norman Matloff and Francis Hsu.

References

- Python 2.6 documentation, http://docs.python.org/library/multiprocessing.html
- PyMOTW by Doug Hellmann, http://www.doughellmann.com/PyMOTW/multiprocessing/
- “Tutorial on Threads Programming with Python” by Norman Matloff and Francis Hsu, University of California, Davis.

Thank You!