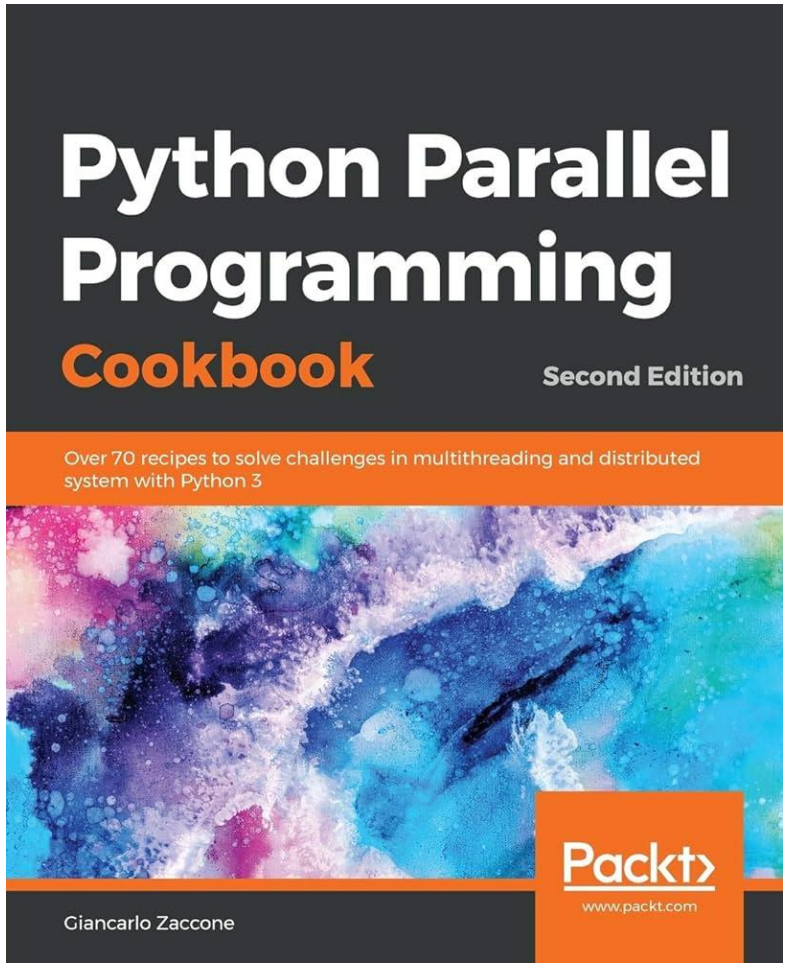
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE 2024-2025

CONCURENTA IN PYTHON

Ioana Leustean



[Python Parallel Programming Cookbook]
https://docs.python.org/3/library/

Bazata pe prezentarea similara facuta de Conf. Traian Serbanuta

Programare concurenta in Python

"The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O."

https://docs.python.org/3/search.html?q=Global+interpreter+lock

Programare concurenta in Python

"Python provides many libraries and frameworks that facilitate high-performance computations. However, doing parallel programming with Python can be quite insidious due to the **Global Interpreter Lock** (**GIL**).

In fact, the most widespread and widely used Python interpreter, **CPython**, is developed in the C programming language. The CPython interpreter needs GIL for thread-safe operations. The use of GIL implies that you will encounter a global lock when you attempt to access any Python objects contained within threads. And only one thread at a time can acquire the lock for a Python object or C API.

Fortunately, things are not so serious, because, outside the realm of GIL, we can freely use parallelism. This category includes all the topics that we will discuss in the next chapters, including multiprocessing, distributed computing, and GPU computing.

So, Python is not really multithreaded."

https://www.researchgate.net/publication/336413386_Python_Parallel_Programming_Cookbook_-_Second_Edition

```
import time
import threading

def fib(n):
    return n if n < 2 else fib(n - 2) + fib(n - 1)

def print_fib(n):
     print(fib(n))

if __name__ == "__main__":
          start = time.time()
          print_fib(30)
          print_fib(35)
          end=time.time()
          print("time ", end-start)
```

```
832040
9227465
 time  2.7712814807891846
```

```
if __name__ == "__main__":
          start = time.time()
          t1=threading.Thread(target=print_fib, args=(30,))
          t2=threading.Thread(target=print_fib, args=(35,))
          t1.start()
          t2.start()
          t1.join()
          t2.join()
          end=time.time()
          print(" time ", end-start)
```

```
832040
9227465
time  2.822628974914551
```

```
import time
import multiprocessing

def fib(n):
    return n if n < 2 else fib(n - 2) + fib(n - 1)

def print_fib(n):
     print(fib(n))

if __name__ == "__main__":
            start = time.time()
            p1=multiprocessing.Process(target=print_fib, args=(30,))
            p2=multiprocessing.Process(target=print_fib, args=(35,))
            p1.start()
            p2.start()
            p1.join()
            p2.join()
            end=time.time()
            print(" time ", end-start)
```

```
832040
9227465
 time  2. 2.6254284381866455
```

```
mport time
mport threading

ef fib(n):
  return n if n < 2 else fib(n - 2) + fib(n - 1)

ef print_fib(n):
    print(fib(n))

 __name__ == "__main__":
            start = time.time()
            t1=threading.Thread(target=print_fib, args=(30,))
            t2=threading.Thread(target=print_fib, args=(35,))
            t1.start()
            t2.start()
            t1.join()
            t2.join()
            end=time.time()
            print(" time ", end-start)
```

```
832040
9227465
time  2.822628974914551
```

Programare concurenta in Python

- Multithreading in Python
  threading — Thread-based parallelism — Python 3.13.0 documentation
- Multiprocessing in Python
  multiprocessing — Process-based parallelism — Python 3.13.0 documentation
- Programare asincrona folosind **asyncio**

  asyncio — Asynchronous I/O — Python 3.13.0 documentation

Thread-urile ruleaza pe un singur processor si sunt afectate de GIL

Procesele ruleaza in paralel, pe procesoare diferite.

Threads (lightweight processes)	**Processes**
Share memory.	Do not share memory.
Start/change are computationally less expensive.	Start/change are computationally expensive.
Require fewer resources (light processes).	Require more computational resources.
Need synchronization mechanisms to handle data correctly.	No memory synchronization is required.

Python Parallel Programming Cookbook

Multithreading in Python

```
import threading

t = threading.Thread(target=lambda : print("Hello "))
print("Greeting: ")
t.start()
print(" World!")
t.join()
```

Argumentele ale unui obiect Thread:

- target: functia executata de thread
- args: argumentele functiei target to be passed to target
- name: numele thread-ului
- daemon: daca este sau nu thread deamon
  ca si in Java, thread-urile deamon sunt terminate in momentul terminarii thread-ului parinte

Starile unui thread:

- Ready
- Running
- Blocked

Metode ale unui obiect Thread

- start() - porneste thread-ul
- run() - apeleaza functia target
- join() - asteapta pana cand thread-ul isi termina executia

Definirea unei subclase a clasei Thread

```
import threading

class myThread (threading.Thread):

  def __init__(self, what):
      super().__init__()
      self.what = what

 def run(self):
      print (self.what)

thread1 = myThread("Hello ")
thread2 = myThread("World!")
print("Greeting: ")
thread1.start()
thread2.start()
```

Interferenta thread-urilor

```
import threading
import time
import random


counter = 0
def increment_counter():
    global counter
    time.sleep(random.random())
    counter += 1
```

```
threads = []
for _ in range(10):
    thread = threading.Thread(target=increment_counter)
    threads.append(thread)

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

print("Final counter value:", counter)
```

Sincronizare folosind **Lock()**

```
import threading
import time
import random

counter = 0
lock = threading.Lock()

def increment_counter():
    global counter
    with lock:

time.sleep(random.random())
        counter += 1
```

```
threads = []
for _ in range(10):
    thread = threading.Thread(target=increment_counter)
    threads.append(thread)

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

print("Final counter value:", counter)
```

Obiecte ale clasei Thread

- **Lock**
- **RLock**
- **Condition**
- **Sempaphore**
- **Event**
- **Barrier**

Toate obiectele care au require() si release() pot fi folosite cu o instructiune **with**.

Instructiunea **with** declara contextul in care o resursa este protejata:
la intrarea in bloc se apeleaza acquire iar la iesirea din bloc se apeleaza release

```
with some_lock:
    # do something...
```

este echivalenta cu

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

https://docs.python.org/3/library/threading.html#using-locks-conditions-and-semaphores-in-the-with-statement

Lock, Rlock, Condition, Semaphore

```
def threading_with(syncObject):
    with syncObject:
        logging.debug('%s acquired via with'  %

def threading_not_with(syncObject):
    syncObject.acquire()
    try:
        logging.debug('%s acquired directly' %s
    finally:
        syncObject.release()
```

```
if __name__ == '__main__':

    logging.basicConfig(level=logging.DEBUG,
                        format='(%(threadName)-10s) %(message)s',)

    lock = threading.Lock()
    rlock = threading.RLock()
    condition = threading.Condition()
    mutex = threading.Semaphore(1)
    threading_synchronization_list = [lock ,rlock , condition , mutex]

    for statement in threading_synchronization_list :
        t1 = threading.Thread(target=threading_with, args=(statement,))
        t2 = threading.Thread(target=threading_not_with, args=(statement,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

clasa Condition

Metode:

- condition.acquire()
- condition.release()
- condition.wait(t)

class threading.Condition(lock=None)

This class implements condition variable objects.
A condition variable allows one or more threads to wait until they are notified by another thread.

If the lock argument is given and not None, it must be a Lock or RLock object,
and it is used as the underlying lock. Otherwise, a new RLock object is created
and used as the underlying lock.

https://docs.python.org/3/library/threading.html#threading.Condition

Producer-Consumer cu Condition

```
import logging
import threading
import time
items = []
condition = threading.Condition()
class Consumer(threading.Thread):
class Producer(threading.Thread):
def main():
    producer = Producer(name='Producer')
    consumer = Consumer(name='Consumer')

    producer.start()
    consumer.start()
    producer.join()
    consumer.join()


if __name__ == "__main__":
    main()
```

```
class Consumer(threading.Thread):
    def __init__(self, name):
        super().__init__()

    def consume(self):
        with condition:
            if len(items) == 0:
                logging.info('no items to consume')
                condition.wait()
            items.pop()
            logging.info('consumed 1 item')
            condition.notify()

    def run(self):
        for i in range(20):
            time.sleep(2)
            self.consume()
```

Producer-Consumer cu Condition

```
import logging
import threading
import time
items = []
condition = threading.Condition()
class Consumer(threading.Thread):
class Producer(threading.Thread):
def main():
    producer = Producer(name='Producer')
    consumer = Consumer(name='Consumer')

    producer.start()
    consumer.start()
    producer.join()
    consumer.join()

if __name__ == "__main__":
    main()
```

```
class Producer(threading.Thread):
    def __init__(self, name):
        super().__init__()

    def produce(self):
        with condition:
            if len(items) == 10:
                logging.info('items produced {}. Stopped'.format(len(items)))
                condition.wait()
            items.append(1)
            logging.info('total items {}'.format(len(items)))
            condition.notify()

    def run(self):
        for i in range(20):
            time.sleep(0.5)
            self.produce()
```

Python Parallel Programming Cookbook

Procese in Python (multiprocessing)

"multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. "

https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing

The Python multiprocessing module, which is a part of the standard library of the language, implements the shared memory programming paradigm, that is, the programming of a system that consists of one or more processors that have access to a shared memory.

Spawning a process is the creation of a child process from a parent process. The latter continues its execution asynchronously or waits until the child process ends.

Python Parallel Programming Cookbook

```
def do_something():
p=multiprocessing.Process(target=do_something,  args=(…,))
p.start()
p.join()
```

Paralelism folosind subprocese - Pool

```
#Using a Process Pool – Chapter 3: Process Based Parallelism

import multiprocessing

def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(0,100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)

    pool.close()
    pool.join()
    print ('Pool    :', pool_outputs)
```

Comunicarea folosind o structura **Pipe**

```
if __name__ == '__main__':
    conn1, conn2 = Pipe()

    sender_process = Process(target=sender, args=(conn2,))
    sender_process.start()



    receiver_process = Process(target=receiver, args=(conn1,))
    receiver_process.start()

sender_process.join()
    receiver_process.join()
```

```
def sender(connection):
    print('Sender: Running', flush=True)
    for i in range(10):
        value = random()
        sleep(value)
        connection.send(value)

    connection.send(None)
    print('Sender: Done')
```

Comunicarea folosind o structura **Pipe**

```
if __name__ == '__main__':
    conn1, conn2 = Pipe()

    sender_process = Process(target=sender, args=(conn2,))
    sender_process.start()



    receiver_process = Process(target=receiver, args=(conn1,))
    receiver_process.start()

sender_process.join()
    receiver_process.join()
```

```
def receiver(connection):
    print('Receiver: Running')
    while True:
        item = connection.recv()
        print(f'>receiver got {item}')
        if item is None:
            break

     print('Receiver: Done', flush=True)
```

Executie asincrona in Python

- **concurrent.futures** Python module
- libraria **asyncio**

"The most important feature of this type of programming is that the code is not performed on multiple threads, as in the classic concurrent programming, but on a single thread. Thus,it is not at all true that two tasks are executed at the same time, but, according to this approach, they are performed at almost the same time."

Python Parallel Programming Cookbook

concurrent.futures – executors and futures

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
         'http://www.bbc.co.uk/',]

def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

if __name__ == '__main__':
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
      future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
      for future in concurrent.futures.as_completed(future_to_url):
            ……
```

https://docs.python.org/3/library/concurrent.futures.html

concurrent.futures – executors and futures

```
if __name__ == '__main__':
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:

    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}

    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)
```

https://docs.python.org/3/library/concurrent.futures.html

obiecte Future, corutine, task-uri

"A Future is a special low-level awaitable object that represents an eventual result of an asynchronous operation.
....
When a Future object is awaited it means that the coroutine will wait until the Future is resolved in some other place.
....
Normally there is no need to create Future objects at the application level code."

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications.
...
Wrap the coro coroutine into a Task and schedule its execution. Return the Task object.

https://docs.python.org/3/library/asyncio-task.html

Corutine fara task-uri

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

```
started at 07:24:51
hello
world
finished at 07:24:54
```

Corutine cu task-uri

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():

        task1 = asyncio.create_task(say_after(1, 'hello'))
        task2 = asyncio.create_task(say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")
        await(task1)
        await(task2)
        print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Task -urile sunt folosite pentru a executa corutinele *concurrent*.

```
started at 07:33:06
hello
world
finished at 07:33:08
```

```
import concurrent.futures
import urllib.request
import asyncio

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://www.bbc.co.uk/']

async def load_url(url, timeout):
…
return len(data)

async def main():
  with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
            future_to_url = {asyncio.create_task(load_url(url, 60)): url for url in URLS}
  for future in future_to_url:
            print(await(future))

asyncio.run(main())
```

```
async def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        data = conn.read()
        print('%r page is %d bytes' % (url, len(data)))
        return len(data)
```

https://docs.python.org/3/library/asyncio-task.html

Pe saptamana viitoare!