Bài 52 - Parallel Computing on Python

30 Nov 2020 - phamdinhkhanh

Menu

1. Tại sao cần xử lý song song

Trong quá trình xây dựng các ứng dụng deep learning trên python, mình nhận ra rằng để tạo ra một ứng dụng thì không khó. Nhưng để tạo ra một ứng dụng đáp ứng được tốc độ xử lý, độ chính xác và mức độ sử dụng resource thì cần phải tối ưu rất nhiều thứ. Bạn sẽ phải quan tâm đến các khía cạnh như:

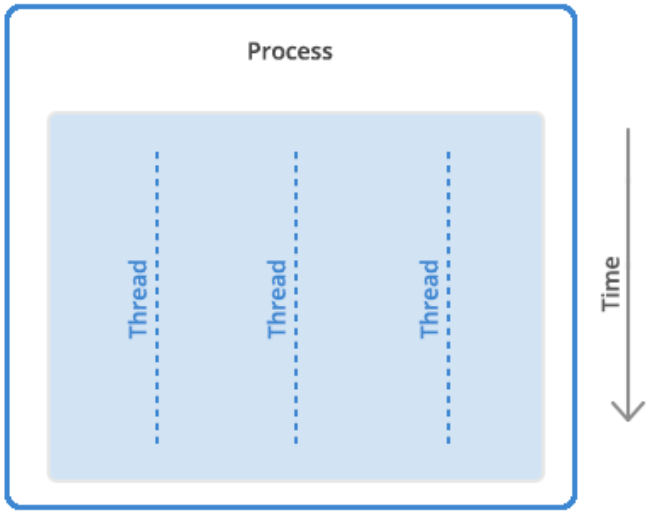
- Giảm nhẹ kích thước mô hình thông qua: Quantization và compress mô hình.
- Optimize lại code.
- Chuyển từ single-thread sang multi-thread.
- Allocate lại tài nguyên như CPU, Memory.

Đặc biệt là các ứng dụng trên python thì tối ưu tốc độ xử lý là một challenge bởi python bị ràng buộc bởi cơ chế GIL (Global Interpreter Lock). Tức là nó chỉ cho phép một thread hoạt động truy suất và chỉnh sửa bộ nhớ tại một thời điểm. Do đó python không tận dụng được các tính toán đa luồng. Tuy nhiên ở python 3.2 trở đi thì python đã bắt đầu hỗ trợ đa luồng. Và thông qua bài viết này mình sẽ hướng dẫn các bạn có thể accelerate các ứng dụng của mình thông qua đa luồng.

Nhưng trước tiên chúng ta cần hiểu về thread/process là gì? Vì blog dành cho đa dạng bạn đọc ở trình độ khác nhau nên bạn nào đã biết thì có thể bỏ qua phần kiến thức rất sơ đẳng này.

1.1. Thread và Process

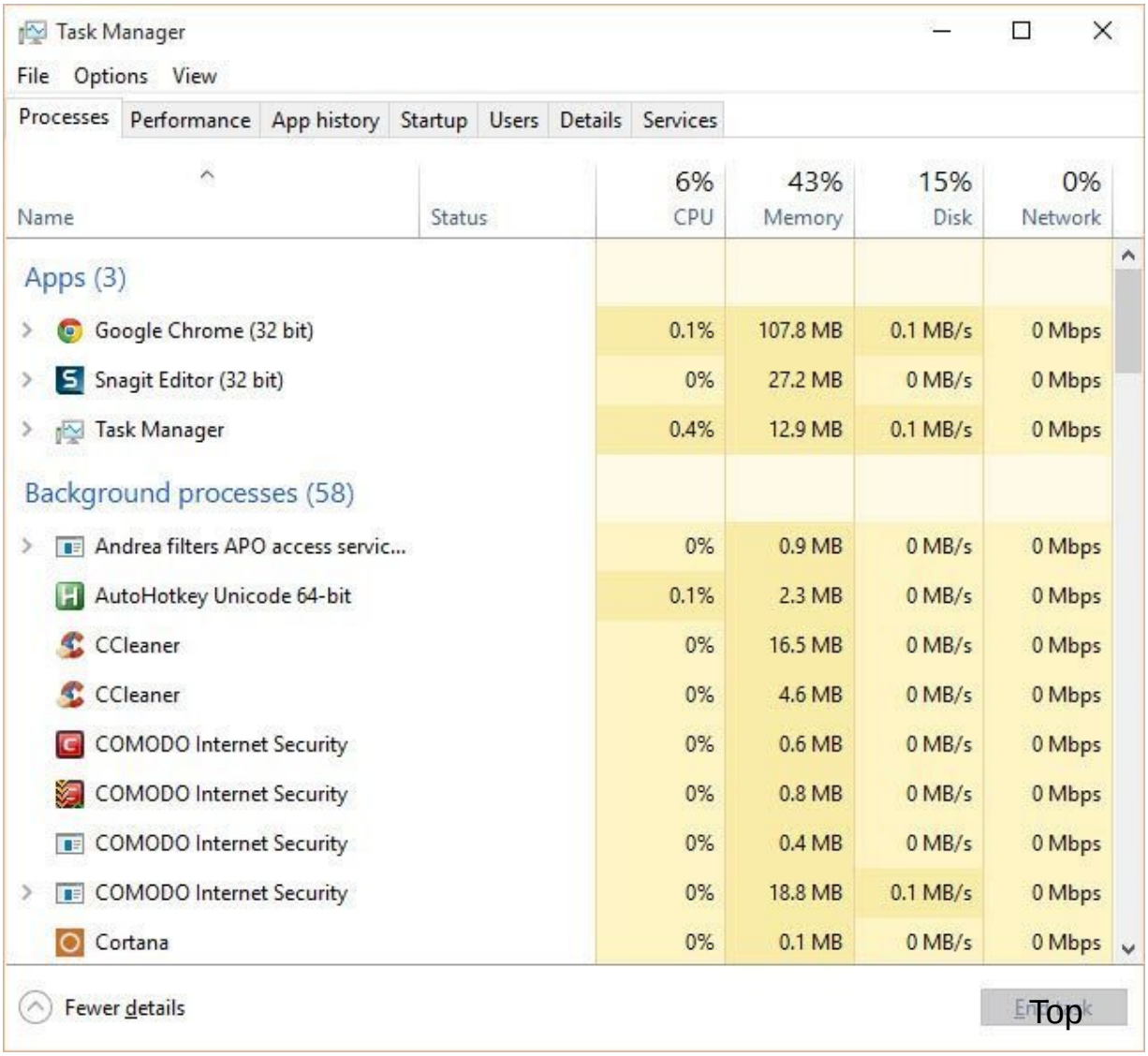
Top



Thread và process là hai khái niệm cơ bản trong lập trình và cũng có nhiều định nghĩa từ các nguồn khác nhau cho chúng.

Process là gì?

Chúng ta hiểu một cách đơn giản thì process là tiến trình để chạy một phần mềm. Khi bạn start một program thì tức là bạn đang khởi tạo một process. Hệ điều hành khi đó sẽ cung cấp các tài nguyên về memory, cpu, disk, bandwidth cho process để cho chạy ứng dụng của bạn.



Hình 1: Khi bạn vào task management của window bạn có thể theo dõi các process đang chạy với mã PID của process. Mỗi một process sẽ phụ trách một instance của OS system và được cung cấp các thành phần như memory, cpu, disk, bandwith,….

Lịch xử lý của các processes sẽ được OS sắp xếp dựa trên một số thuật toán lập lịch như round robin, first come first serve,… Mình sẽ không đi sâu vào phần này, các bạn có thể tham khảo thêm Operating System Scheduling algorithms (https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm).

Threads là gì?

Chắc hẳn bạn đã từng nghe đến thông số số cores của CPU. Các CPU càng hiện đại, số lượng cores sẽ càng nhiều. Các core sẽ hỗ trợ cho việc tính toán multi-task tốt hơn. Các threads sẽ được vận hành và tính toán trên các core của CPU.

Một process khi được khởi tạo sẽ sinh ra các threads để run application. Bạn sẽ thắc mắc vậy thì chỉ cần một thread cũng được ? Tại sao lại cần nhiều threads? Nhiều threads sẽ giúp cho việc tính toán multi-task tốt hơn. Tức là bạn có thể làm nhiều nhiệm vụ một lúc. Nếu coi mỗi thread là một công nhân, thì việc sản xuất sẽ nhanh hơn nếu có nhiều công nhân phối hợp cùng làm việc. Bạn có thể hình dung dễ hơn qua ví dụ:

Khi bạn làm việc với microsoft word, bạn gõ bàn phím thì có những công việc sau cần thực hiện:

- Đọc thông tin input từ keyboards.
- Hiển thị lên màn hình các thông tin đã nhập trong quá trình gõ.
- High light những chỗ bị sai chính tả.
- Suggest các từ để có thể gõ nhanh hơn.

Mỗi công việc được phụ trách bởi một thread và chúng phối hợp với nhau để giúp ứng dụng của bạn mượt hơn. Nếu chỉ có một thread làm tất cả mọi công việc thì nó sẽ bị quá tải và bạn có thể gặp phải giới hạn về tốc độ xử lý của CPU, thuật ngữ hay được gọi là CPU bound.

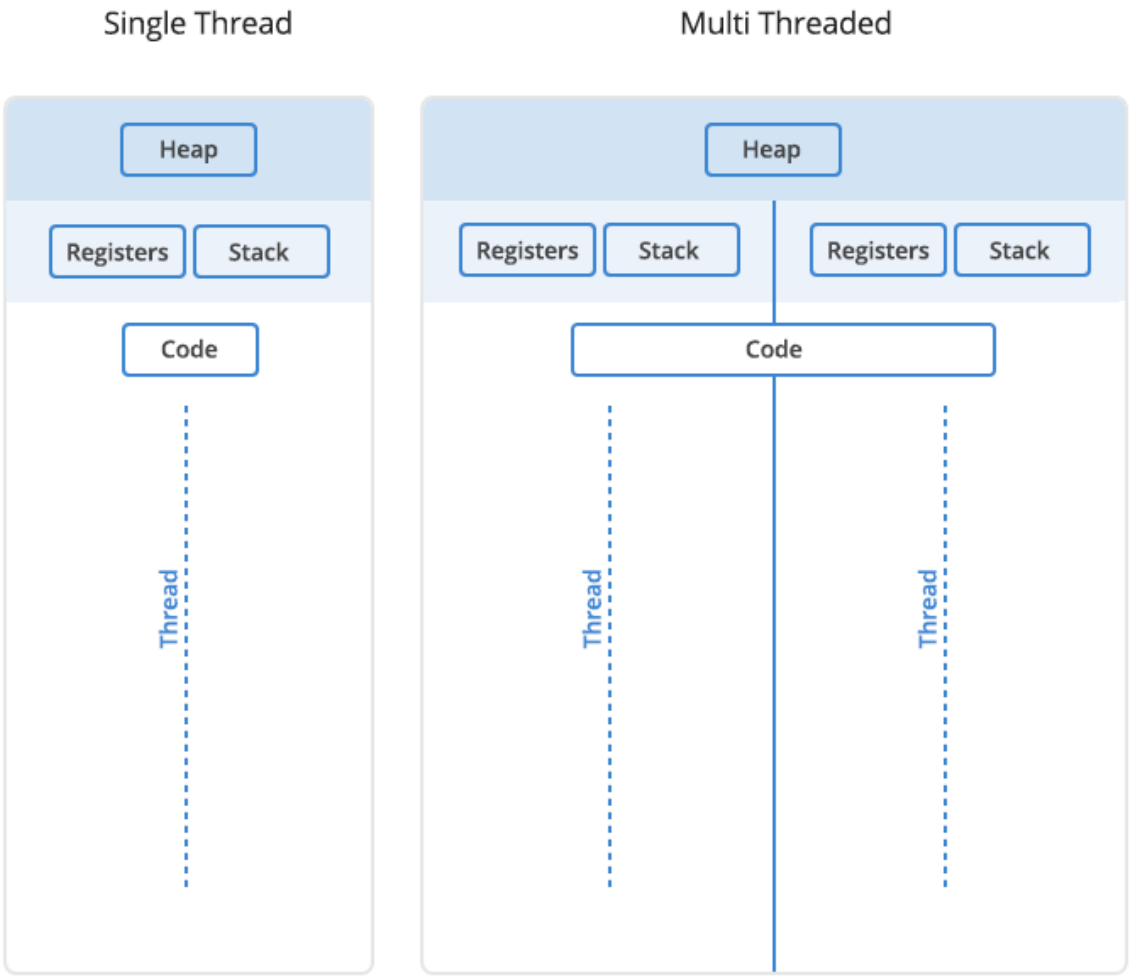
Một process có thể là single-thread hoặc multiple-threads tùy thuộc vào số lượng là một hoặc nhiều. Khi có nhiều threads thì đòi hỏi phải có sự phối hợp tính toán song song (parrallel computing) giữa các threads với nhau. Từ đó sinh ra các khái niệm về đồng bộ (`synchronous`) và bất đồng bộ (`asynchronous`). Chúng ta sẽ làm rõ hai khái niệm này ở các phần tiếp theo.

Bạn có thể thắc mắc multiple-threads thì có khác gì khác biệt so với việc sử dụng multiple-processes? Chúng ta vẫn có thể tính toán song song được trên cả hai? Vậy tại sao lại cần phải tách một process thành nhiều threads làm gì ? Thực tế là trong python thì process và thread cùng kế thừa chung một interface là một base thread. Chúng sẽ có những đặc tính chung, nhưng thread là một phiên bản nhẹ hơn so với process. Do đó việc khởi tạo thread sẽ nhanh hơn. Một điểm khác biệt nữa đó là thread được thiết kế để có thể hoạt động tương tác lẫn nhau. Các threads trong cùng một process sẽ chia sẻ được dữ liệu qua lại nên có lợi thế về I/O. Dữ liệu của process thì được thiết kế private nên một process không thể chia sẻ dữ liệu với các process khác. Đây là lý do chúng ta cần nhiều threads hoạt động trong một process.

Tiếp theo môi trường hoạt động của multiple-threads sẽ như thế nào ?

Khi các threads chạy song song trên cùng một process, chúng sẽ khởi tạo dữ liệu như thế nào? Dữ liệu sẽ được lưu vào đâu? Chúng chia sẻ chung một code như thế nào? Chúng ta cùng làm rõ qua sơ đồ bên dưới.

Top



Hình 2: Cấu trúc của single thread và multiple threads.

Đầu tiên ứng dụng của bạn sẽ khi khởi chạy sẽ load code lên. Phần main của chương trình sẽ được compiler khởi chạy đầu tiên. Lần lượt các method sẽ được load vào môi trường stack theo trình tự chạy. Compiler chạy lần lượt các hàm trong stack. Các hàm được compiler biên dịch thành mã máy (`byte code`) và được thực thi để sinh ra dữ liệu. Dữ liệu sau đó được lưu trữ tại hai bộ nhớ là Heap và Stack (cái này cũng tùy thuộc vào virtual machine của từng ngôn ngữ). Stack lưu trữ method và các local variable còn heap lưu trữ object, array từ chương trình của bạn (phần lưu trữ này cũng có thể thay đổi tùy vào cách sắp xếp bộ nhớ của các ngôn ngữ). Nếu load trên stack thì không bị phân mảng dữ liệu và có thời gian load/access nhanh hơn. Còn heap sẽ allocate vùng nhớ ngẫu nhiên, các ô nhớ không liên tục nên do đó bị phân mảng.

Okie, mình nghĩ lý thuyết như vậy là đủ rồi. Tiếp theo chúng ta sẽ cùng thực hành khởi tạo các thread và process trong python.

2. Khởi tạo thread trong python

2.1. Khởi tạo từ hàm

Trên python3 để khởi tạo một thread thì chúng ta sử dụng module `_thread` , trên python2 là `thread` . Để start một method trên thread thì chúng ta chỉ cần truyền vào `_thread.start_new_thread()` tên method và các giá trị đối số của nó. Ví dụ bên dưới chúng ta sử dụng hàm `_counter()` để đếm lùi các số từ trên xuống dưới.

Top

```
import _thread
import time

def _counter(counter, thread_name):
  while (counter):
    time.sleep(0.01)
    print("{}: {}".format(thread_name, counter))
    counter -= 1

counter = 5

# Khởi tạo 2 threads 1 và 2
try:
  _thread.start_new_thread(_counter, (counter, "khanh thread")) # pas
  print("\n")
  _thread.start_new_thread(_counter, (counter, "ai thread"))
except:
  print("Error: unable to start thread")

# Running counter
while (counter):
  counter -= 1
  pass
```

```
ai thread: 5
khanh thread: 5
ai thread: 4
khanh thread: 4
ai thread: 3
khanh thread: 3
ai thread: 2
khanh thread: 2
ai thread: 1
khanh thread: 1
```

Ta thấy hai thread đã xen kẽ nhau cùng thực hiện tác vụ đếm ngược. Tuy nhiên về bản chất thì chúng vẫn là đơn luồng vì cơ chế GIL của python ép buộc một thời điểm chỉ có một thread được tương tác với dữ liệu. Có khá nhiều developer tỏ ra thất vọng về hạn chế này nhưng một số khác thì bảo vệ quan điểm này bởi nó giúp một dữ liệu không bị sử dụng và thay đổi cùng lúc bởi nhiều threads. Hiện tượng này dẫn tới `concurrency`, một trong những bug thường gặp ở các ngôn ngữ đa luồng như java hay C++.

Trong ví dụ trên thì các method trên hai threads `khanh` và `ai` khởi chạy độc lập nhau mà không ưu tiên một thread hoàn thành thì mới chạy thread tiếp theo. Cách chạy như vậy được gọi là bất đồng bộ `asynchronous`, một khái niệm cơ bản của parallel application. Trái ngược lại thì là đồng bộ `synchronous`, các method sẽ chạy theo tuần tự, sau khi method trước đó đã hoàn thành.

2.2. Khởi tạo kế thừa

Một cách khác để khởi tạo một thread đó là kế thừa lại Threading module. Kiểu kế thừa này khá phổ biến trong lập trình, chắc các bạn còn nhớ khi khởi tạo model trên pytorch chúng ta cũng kế thừa lại nn.Module (https://phamdinhkhanh.github.io/2019/08/10/PytorchTurtorial1.html#31-

top

ki%E1%BA%BFn-tr%C3%BAc-m%E1%BA%A1ng-cnn) chứ ? Khi đó chúng ta chỉ cần override lại các method cần điều chỉnh từ class cha.

```
import threading
import time

class FirstThread(threading.Thread):
  def __init__(self, thread_id, thread_name, counter):
    threading.Thread.__init__(self)
    self.thread_id = thread_id
    self.thread_name = thread_name
    self.counter = counter

  def run(self):
    print("Start thread {}!".format(self.thread_name))
    while (self.counter):
      time.sleep(0.01)
      print("{} : {}".format(self.thread_name, self.counter))
      self.counter -= 1
    print("End thread {}".format(self.thread_name))


thread1 = FirstThread(1, "khanh thread", 5)
thread2 = FirstThread(2, "ai thread", 5)

thread1.start()
thread2.start()
```

```
Start thread khanh thread!
Start thread ai thread!
khanh thread : 5
ai thread : 5
khanh thread : 4
ai thread : 4
khanh thread : 3
ai thread : 3
khanh thread : 2
ai thread : 2
khanh thread : 1
End thread khanh thread
ai thread : 1
End thread ai thread
```

2.3. Cơ chế Thread Lock

Như đã giới thiệu chương trước, trong ví dụ ở trên các threads là bất đồng bộ (`asynchronous`). Hai threads chạy độc lập với nhau mà không theo thứ tự. Chúng ta có thể đồng bộ (synchronous) các thread. Tức là cho phép một thread chạy xong thì thread khác mới được phép chạy bằng cách sử dụng Thread Lock trong python.

Top

```
import threading

class FirstThread(threading.Thread):
  def __init__(self, thread_id, thread_name, counter):
    threading.Thread.__init__(self)
    self.thread_id = thread_id
    self.thread_name = thread_name
    self.counter = counter

  def run(self):
    threadLock.acquire()
    print("Start thread {}!".format(self.thread_name))
    while (self.counter):
      time.sleep(0.01)
      print("{} : {}".format(self.thread_name, self.counter))
      self.counter -= 1
    print("End thread {}".format(self.thread_name))
    threadLock.release()

threadLock = threading.Lock()
thread1 = FirstThread(1, "khanh thread", 5)
thread2 = FirstThread(2, "linh thread", 5)

thread1.start()
thread2.start()

threads = [thread1, thread2]

for t in threads:
  t.join()
```

```
Start thread khanh thread!
khanh thread : 5
khanh thread : 4
khanh thread : 3
khanh thread : 2
khanh thread : 1
End thread khanh thread
Start thread linh thread!
linh thread : 5
linh thread : 4
linh thread : 3
linh thread : 2
linh thread : 1
End thread linh thread
```

Trong hàm `run()` của thread thì chỉ cần thêm hàm `thread.acquire()` và `thread.release()` vào đầu và cuối hàm thì luồng sẽ được locking cho đến khi thread chạy xong thì thread khác mới được xử lý tiếp. Như chúng ta thấy, sau khi thread1 xử lý xong thì mới đến lượt thread2 xử lý.

3. Khởi tạo process trong python

Top

Để khởi tạo một `process` trong python chúng ta sử dụng class `Process` của thư viện `multiprocessing` . Chúng ta cũng truyền vào hàm và đối số như đã thực hiện với thread.

```
from multiprocessing import Process
import time

def _counter(counter, process_name):
  while (counter):
    time.sleep(0.01)
    print("{}: {}".format(process_name, counter))
    counter -= 1

counter = 5

exec1 = Process(target=_counter, args=(counter, "khanh thread")) # pa
exec2 = Process(target=_counter, args=(counter, "ai thread"))

exec1.start()
exec2.start()

for exec in execs:
  exec.join()
```

```
khanh thread: 5
ai thread: 5
khanh thread: 4
ai thread: 4
khanh thread: 3
ai thread: 3
khanh thread: 2
ai thread: 2
khanh thread: 1
ai thread: 1
```

Khi làm việc với multi-process, chúng ta luôn cần một lệnh `join()` để đảm bảo main process hoàn thành sau cùng sau khi các child process khác kết thúc.

Ta nhận thấy rằng các process được thực hiện một cách độc lập và bất đồng bộ. Để đồng bộ các process với nhau thì chúng ta đơn giản là `lock` chúng lại.

Top

```
from multiprocessing import Process, Lock
import time

def _counter_lock(counter, process_name, lock):
  lock.acquire()
  while (counter):
    time.sleep(0.01)
    print("{}: {}".format(process_name, counter))
    counter -= 1
  lock.release()

counter = 5

lock = Lock()
exec1 = Process(target=_counter_lock, args=(counter, "khanh thread", 
exec2 = Process(target=_counter_lock, args=(counter, "ai thread", loc
execs = [exec1, exec2]

for exec in execs:
  exec.start()
```

```
khanh thread: 5
khanh thread: 4
khanh thread: 3
khanh thread: 2
khanh thread: 1
ai thread: 5
ai thread: 4
ai thread: 3
ai thread: 2
ai thread: 1
```

Bạn thấy đó, chúng cũng khá na ná thread phải không nào ?

3.1. Sử dụng chung dữ liệu

Khi làm việc với các ứng dụng concurrent thì chúng ta nên hạn chế nhất có thể việc chia sẻ dữ liệu giữa các process để tránh xảy ra các lỗi phát sinh do concurency. Tuy nhiên python vẫn cung cấp một cơ chế giúp chia sẻ dữ liệu giữa các process, đó chính là các shared memory object trong multiprocessing như Value, Array. Thật vậy, giả sử ở ví dụ bên dưới chúng ta sử dụng 2 processes để thay đổi dấu các phần tử của một list các số nguyên.

Top

```
from multiprocessing import Process, Lock
import time

def _counter_arr(arrs, process_name):
  lock.acquire()
  for i, el in enumerate(arrs):
    time.sleep(0.01)
    arrs[i] = -arrs[i]
    print("{}: {}".format(process_name, arrs[i]))
  lock.release()

arrs = [1, 2, 3, 4]
lock = Lock()
exec1 = Process(target=_counter_arr, args=(arrs, "khanh process")) # 
exec2 = Process(target=_counter_arr, args=(arrs, "ai process"))
execs = [exec1, exec2]

exec1.start()
exec2.start()

for exec in execs:
  exec.join()
```

```
khanh process: -1
khanh process: -2
khanh process: -3
khanh process: -4
ai process: -1
ai process: -2
ai process: -3
ai process: -4
```

Ta nhận thấy dữ liệu là không được chia sẻ giữa 2 processes vì process thứ hai đổi lại dấu của process thứ nhất sẽ khiến các phần tử của 2 processes này đảo dấu. Tiếp theo nếu chúng ta sử dụng Array trong multiprocessing thì sao ?

Top

```
from multiprocessing import Process, Value, Array, Lock
import time

def _counter_arr(arrs, process_name):
  lock.acquire()
  for i, el in enumerate(arrs):
    time.sleep(0.01)
    arrs[i] = -arrs[i]
    print("{}: {}".format(process_name, arrs[i]))
  lock.release()

arrs = Array('i', range(1, 5, 1))
lock = Lock()
exec1 = Process(target=_counter_arr, args=(arrs, "khanh process")) # 
exec2 = Process(target=_counter_arr, args=(arrs, "ai process"))
execs = [exec1, exec2]

exec1.start()
exec2.start()

for exec in execs:
  exec.join()
```

```
khanh process: -1
khanh process: -2
khanh process: -3
khanh process: -4
ai process: 1
ai process: 2
ai process: 3
ai process: 4
```

Các bạn đã thấy gì chưa ? Dữ liệu đã được chia sẻ qua lại giữa hai processes. Vậy thì chúng ta sẽ thường sử dụng shared memory khi nào ? Giả định bạn đang có một pipeline biến đổi dữ liệu gồm nhiều step khác nhau, mỗi một process sẽ phụ trách một step trong pipeline. Khi đó dữ liệu cần được shared chung giữa các process.

3.2. Shared object giữa các Process

Queue là một định dạng stack an toàn khi làm việc với multi thread và process. Chúng ta có thể tạo ra một queue và cho phép các thread, process truy cập dữ liệu mà không bị hiện tượng concurrency vì dữ liệu được truy suất và sử dụng một lần bởi một thread hoặc process.

Bên dưới chúng ta sẽ lấy ví dụ về việc sử dụng 2 process để đọc các dữ liệu trong một queue. Hai process này tới phiên của mình sẽ lấy ra các phần từ nằm trong queue theo kiểu FIFO (First Come First Out).

Top

```
from multiprocessing import Process, Queue
import time

def _counter_queue(queue, process_name, max_count):
  # lock.acquire()
  while max_count:
    time.sleep(0.01)
    value = queue.get()
    print("{}: {}".format(process_name, value))
    max_count -= 1
  # lock.release()

q = Queue()
for i in range(10):
  q.put(i)
max_count = 5
# lock = Lock()
exec1 = Process(target=_counter_queue, args=(q, "khanh process", 5)) ;
exec2 = Process(target=_counter_queue, args=(q, "ai process", 5))
execs = [exec1, exec2]

exec1.start()
exec2.start()

for exec in execs:
  exec.join()
```

```
khanh process: 0
ai process: 1
khanh process: 2
ai process: 3
khanh process: 4
ai process: 5
khanh process: 6
ai process: 7
khanh process: 8
ai process: 9
```

Như vậy không có bất kỳ một data nào được sử dụng chung giữa 2 processes nên tránh được concurrency.

3.3. Pool trong multiprocess

Trong python chúng ta có thể sử dụng pool để tận dụng được các tính toán song song trên nhiều process một lúc. Cơ chế của pool đã loại bỏ hạn chế của GIL trong python, cho phép nhiều luồng hoạt động đồng thời và giúp đẩy nhanh quá trình tính toán.

Trong Pool chúng ta có thể khai báo nhiều workers cùng thực hiện chương trình. Các chương trình có thể thực hiện một cách bất đồng bộ thông qua hàm `apply_async()`. Tức là cho phép thực hiện song song nhiều method trên các workers. Đồng thời `apply_async()` cũng cho phép đưa vào các hàm callback để xử lý giữa liệu sau cùng.

Top

Ví dụ bên dưới chúng ta sẽ sử dụng 5 workers để tính toán bất đồng bộ bình phương của các số trong phạm vi 20. Kết quả sau khi tính sẽ được lưu vào một list.

```
import multiprocessing as mp
import time

def _square(x):
  return x*x

def log_result(result):
  # Hàm được gọi bất kỳ khi nào _square(i) trả ra kết quả.
  # result_list được thực hiện trên main process, khong phải pool wor
  result_list.append(result)

def apply_async_with_callback():
  pool = mp.Pool(processes=5)
  for i in range(20):
    pool.apply_async(_square, args = (i, ), callback = log_result)
  pool.close()
  pool.join()
  print(result_list)

if __name__ == '__main__':
  result_list = []
  apply_async_with_callback()
```

```
[0, 1, 9, 25, 16, 36, 49, 64, 81, 100, 144, 4, 169, 225, 256, 289, 324
```

Ta thấy thứ tự của list không theo tuần tự từ thấp tới cao do hàm được gọi bất đồng bộ.

Bên cạnh cách khởi tạo Pool cho process như trên, chúng ta còn có thể khởi tạo từ `concurrent.futures` như phần tiếp theo mình giới thiệu.

4. Process Pool vs Thread Pool

4.1. Process Pool

Trong python thì bắt đầu từ version 3.2 chúng ta có thể sử dụng module `concurrent.futures` để xử lý bất đồng bộ các tasks. Đây là một abstract layer được kế thừa trên cả hai modules là threading và multiprocessing để tạo ra một interface cho phép khởi tạo các task sử dụng pool của các processes và threads.

Để khởi tạo một Process Pool, chúng ta sử dụng `ProcessPoolExecutor` trong `concurrent.futures` module.

Top

```
from concurrent.futures import ProcessPoolExecutor
from time import sleep
import timeit

def _counter(counter, task_name):
  print("Start process {}!".format(task_name))
  while (counter):
    print("{} : {}".format(task_name, counter))
    counter -= 1
  print("End process {}!".format(task_name))
  return "Completed {}!".format(task_name)

def _submit_process():
  executor = ProcessPoolExecutor(max_workers=5)
  future = executor.submit(_counter, 10, "task1")
  print('State of future: ', future.done())
  print('futre result: ', future.result())
  print('State of future: ', future.done())

_submit_process()
```

```
Start process task1!
task1 : 10
task1 : 9
task1 : 8
task1 : 7
task1 : 6
task1 : 5
task1 : 4
task1 : 3
task1 : 2
task1 : 1
End process task1!
State of future:  False
futre result:  Completed task1!
```

Trong `ProcessPoolExecutor()` chúng ta cần truyền vào số lượng các worker để chạy process. Số lượng worker càng lớn thì càng nhiều threads được sinh ra để tính toán process.

Hàm submit()

Hàm `submit()` được sử dụng để load các task vào process pool. Tham số truyền vào là tên hàm và các đối số của hàm. Hàm `done()` để kiểm tra trạng thái của task. Lúc đầu ngay sau khi submit thì task chưa hoàn thành nên `done()` là `False` . Hàm `result()` thường được dùng để kiểm tra kết quả sau khi task cuối cùng trong process pool đã thực thi xong. Do đó trạng thái `done()` sau khi result được trả về là `True` .

Hàm map()

Nhắc đến hàm map trong python, nếu bạn đã có kinh nghiệm thì sẽ hiểu ngay nó sẽ map các đối số từ một list vào hàm.

Ví dụ: Để tính diện tích của các bounding box dựa trên tọa độ `(x1, x2, y1, y2)` thì chúng ta thực hiện hàm map trong process pool như sau:

Top

```
from concurrent.futures import ProcessPoolExecutor
from concurrent.futures import as_completed
x1s = [5, 10, 20, 35]
x2s = [15, 20, 30, 55]
y1s = [5, 10, 10, 15]
y2s = [15, 20, 20, 35]

def _bbox(x1, x2, y1, y2):
  w = x2-x1
  h = y2-y1
  area = w*h
  return area

with ProcessPoolExecutor(max_workers = 5) as executor:
  results = executor.map(_bbox, x1s, x2s, y1s, y2s)

for result in results:
  print(result)
```

```
100
100
100
400
```

4.2. Thread Pool

Thread pool cũng tương tự như Process Pool nhưng là tập hợp của các các threads thay vì processes. Các khởi tạo `ThreadPoolExecutor` trên `concurrent.futures` cũng hoàn toàn tương tự như `ProcessPoolExecutor` . Ta thực hiện như sau:

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def _counter(counter, task_name):
  print("Start process {}!".format(task_name))
  while (counter):
    print("{} : {}".format(task_name, counter))
    counter -= 1
  print("End process {}!".format(task_name))
  return "Completed {}!".format(task_name)

def _submit_thread():
  executor = ThreadPoolExecutor(max_workers=5)
  future = executor.submit(_counter, 10, "task1")
  print('State of future: ', future.done())
  print('futre result: ', future.result())
  print('State of future: ', future.done())

_submit_thread()
```

Top

```
Start process task1!
task1 : 10
task1 : 9
task1 : 8
task1 : 7
task1 : 6
task1 : 5
task1 : 4
task1 : 3
task1 : 2
task1 : 1
End process task1!
State of future:  True
futre result:  Completed task1!
```

So sánh thời gian xử lý của process và thread. Để đo lường, các bạn cần cài package `cProfile`.

```
import cProfile
cProfile.run('_submit_process()')
cProfile.run('_submit_thread()')
```

```
State of future:  True
         1268 function calls in 0.072 seconds

...

State of future:  True
         249 function calls in 0.001 seconds
...
```

Với cùng các tác vụ như nhau thì ta thấy thời gian thực thi của Thread Pool chỉ là `0.001` seconds, nhanh thời gian thực thi của Process Pool là `0.072` seconds. Lý do là vì thread là một phiên bản light weight hơn process rất nhiều. Bạn có thể nhận thấy điều này một cách trực quan thông qua số hàm được gọi ở cả hai phương pháp.

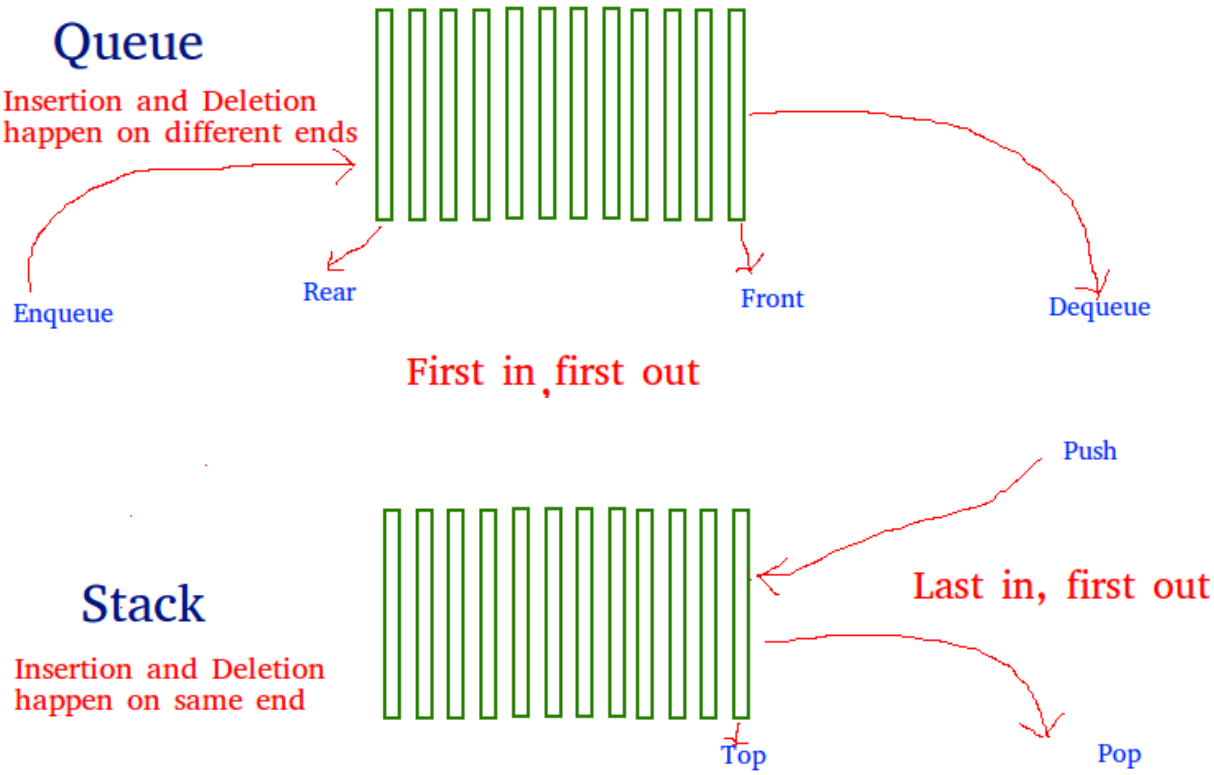
Vậy lựa chọn thế nào giữa process pool và thread pool?

Chúng ta đã biết rằng khi sử dụng threads thì sẽ có lợi về I/O vì các threads có thể chia sẻ data qua lại lẫn nhau. Còn giữa các processes thì data được sử dụng hoàn toàn độc lập nên không có lợi về I/O. Tuy nhiên khi sử dụng process thì chúng ta sẽ được allocate về CPU, Memomory,… nên lời khuyên là nếu task của bạn gặp phải giới hạn về I/O bound thì nên sử dụng thread pool và giới hạn về CPUs bound thì nên sử dụng process pool.

5. Queue

Khi chạy một process trên multiple-threads, queue thường được sử dụng để làm nơi lưu trữ dữ liệu chung giữa các threads với nhau.

Top



Hình 3: Cấu trúc FIFO và LIFO của một queue.

Queue là một cấu trúc dữ liệu tuyến tính (linear data structure). Nó có tính chất tương tự như list. Cho phép chúng ta thêm, sửa, xóa, truy xuất các phần tử bên trong. Trong python, Queue có ưu điểm lớn hơn list đó là tốc độ truy xuất nhanh hơn. Độ phức tạp thời gian (`time complexity`) của queue là `O(1)` trong khi của list là `O(n)` . Queue là một lựa chọn thay thế tốt hơn cho list trong trường hợp dữ liệu của bạn có số lượng phần tử lớn.

Khi làm việc với Queue ban có thể truy xuất các phần tử bên trong khối theo kiểu FIFO (first in first out) hoặc LIFO (last in first out) thông qua hàm `pop()` .

Queue thường được sử dụng trong các tác vụ liên quan tới threads synchronous. Các thread sẽ sử dụng chung một dữ liệu và thay đổi các phần tử bên trong nó một cách tuần tự.

Top

```
from concurrent.futures import ThreadPoolExecutor
import queue


def _sum_queue(name, work_queue):
  sum = 0
  while not work_queue.empty():
    print(f"Task {name} running")
    count = work_queue.get()
    sum += count
  print(f"Task {name} total: {sum}")
  return sum

def task(name, work_queue):
  if work_queue.empty():
    print(f"Task {name} nothing to do")
  else:
    print("Start ThreadPoolExecutor!")
    with ThreadPoolExecutor(max_workers = 5) as executor:
      print("Submit task!")
      future = executor.submit(_sum_queue, name, work_queue)
      sum = future.result()
    return sum

# Create the queue of work
work_queue = queue.Queue()

# Put some work in the queue
for work in [15, 10, 5, 2]:
    work_queue.put(work)

# Create some synchronous tasks
tasks = [("one", work_queue), ("two", work_queue)]

# Run the tasks
for n, q in tasks:
    print(task(n, q))
```

```
Start ThreadPoolExecutor!
Submit task!
Task one running
Task one running
Task one running
Task one running
Task one total: 32
32
Task two nothing to do
None
```

Trong ví dụ trên giải sử chúng ta có hai threads hoạt động một cách synchronous là `one` và `two` . Hai threads này sử dụng chung một nguồn dữ liệu là `work_queue` . Khi thread `one` chạy xong thì toàn bộ các phần tử của queue đã được trích xuất xong nên ở thread `two` chúng ta không có gì để chạy tiếp.

Top

6. Kết luận

Như vậy qua bài viết này mình đã giới thiệu với các bạn những khái niệm cơ bản nhất trong xử lý song song như thread, process, khác biệt giữa chúng và các khái niệm về đồng bộ, bất đồng bộ, concurrency. Thông qua ví dụ minh họa chi tiết và cách khởi tạo và áp dụng thread và process trong một chương trình bạn đọc sẽ có thể áp dụng vào các project của mình.

Đồng thời để loại bỏ hạn chế của GIL trong python, tận dụng được tính toán song song trên nhiều CPUs thì bạn đọc có thể áp dụng Process Pool, Thread Pool như một giải pháp hữu hiệu.

Để xây dựng bài viết này mình đã khảo cứu và tổng hợp từ rất nhiều nguồn dữ liệu bên dưới.

7. Tài liệu

https://docs.python.org/3.1/library/multiprocessing.html

https://docs.python.org/3/library/queue.html#queue.Queue

https://www.geeksforgeeks.org/stack-in-python/?ref=lbp

https://realpython.com/python-async-features/

https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_pool_of_processes.htm

https://medium.com/@bfortuner/python-multithreading-vs-multiprocessing-73072ce5600b

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

https://realpython.com/python-async-features/

Top