“Concurrencia en Python”

conociendo al GIL y sus amigos..

PYCONAR2016

· BAHÍA BLANCA - ARGENTINA ·

8va Conferencia Argentina de Python

25, 26 y 27 de Noviembre - Bahía Blanca

Martin Alderete
@alderetemartin

Intro: Concurrencia y Paralelismo

Concurrencia

Composición de tareas independientes que se ejecutan.

TENER que hacer muchas cosas a la vez. Relacionado con la estructura.

Paralelismo

Ejecución simultánea de varias tareas relacionadas o no.

HACER muchas cosas a la vez. Relacionado con la ejecución

Rob Pike - 'Concurrency Is Not Parallelism' (Charla sobre golang)

CPython

Es la implementación oficial y más ampliamente utilizada del lenguaje. Está escrita en C. No confundir con Cython

Es la que usamos todo el día.

Esta charla se centra en CPython

Python: Threads

Procesos **livianos** o **tareas**

Son realmente thread del **SISTEMA**

- POSIX threads (pthreads)
- Windows threads

Completamente administrados por el Sistema Operativo
Comparten recursos (espacio de memoria) Sincronización manual
Representan la ejecución multithreading del intérprete (escrito en C)
Compiten por ejecutar código del intérprete!

Módulos: thread (bajo nivel, **no usar!**) threading (alto nivel)

Threading

Podemos crear thread y comunicarlos entre ellos (comparten memoria) debemos sincronizar (Lock, Condition, Semaphore).

```
import threading

t = threading.Thread(target=my_function, args=args_tuple, kwargs=data_dict)
t.start()


class Downloader(threading.Thread):
    def run(self):
        # codigo que corre en el thread
```

Queue.Queue

El módulo Queue provee una implementación de FIFO especialmente diseñada para multithreading. Permite que varios threads intercambien mensajes por medio de una cola de forma segura. Todo el mecanismo de “locking” está implementado dentro de la estructura de datos.

Permite Colas limitadas o infinitas.

Internamente utiliza un "deque", una lista doblemente enlazada, está escrito en C

Python: Process

Procesos del **SISTEMA**!
Utilizan primitivas del Sistema Operativo para lanzar nuevos proceso (fork, spawn)
Son un **NUEVO** proceso con su propio intérprete

Completamente administrados por el Sistema Operativo
Permite ejecución en paralelo (máquinas con multicore)
Comunicación por IPC (pipe, shared-memory, sockets, etc)
API similar a threading!

Módulos: os (bajo nivel, **no usar!**) multiprocessing (alto nivel)

Multiprocessing

Podemos crear procesos y comunicarlos entre ellos (NO comparten memoria) NO debemos sincronizar solo usar un mecanismo de comunicación con IPC.

```
import multiprocessing


p = multiprocessing.Process(target=target, args=args_tuple, kwargs=data_dict)
p.start()


class Downloader(multiprocessing.Process):
    def run(self):
        # codigo que corre en el proceso
```

multiprocessing.Queue

multiprocessing.Queue provee una implementación de FIFO especialmente diseñada para multiprocessing. Permite que varios procesos intercambien mensajes (los objetos deben ser “pickables”) por medio de una cola de forma segura. Todo el mecanismo de “locking” está implementado dentro de la estructura de datos.

Permite Colas limitadas o infinitas.

Internamente utiliza un "pipe", una especia de archivo donde se lee/escribe, está escrito en Python. Los Pipes se crean con la syscall **pipe()**

Poseen la misma API que Queue.Queue

Python: GIL (Global Interpreter Lock)

Ejecución en paralelo está **prohibida**.
Hay un “lock global del interprete”.
Asegura que **sólo un thread** ejecute dentro del intérprete a la vez.
Simplifica muchísimos detalles de **bajo nivel** del intérprete (manejo de memoria, garbage collection, comunicación con extensiones hechas en C, etc).

"Understanding the GIL (Python 2)"
"Understanding the NEW GIL (Python 3)"

Python: Y entonces…?

Con un CPU hay cooperación pero se pierde tiempo antes de lanzar un thread

El GIL condiciona el uso de múltiples CPUs
NO podemos hacer ejecución paralela de threads.

Los threads COMPITEN por el uso del intérprete (todos quieren el GIL)

Los thread en Python son útiles para tareas “semi” I/O bound
Con mucha I/O el GIL realiza “thrashing” **(release/acquire)** penaliza la ejecución

Mucho Context Switch 😩

Python: Estamos para atras?

NO! La mayoría de las aplicaciones son **I/O bound** entonces el GIL nos “perdona”!

Entonces usamos **threading** y todo va bien **(NO, esperen!)**

Debemos lograr minimizar el Context Switch

- Modelo 1:1 todos los threads son administrados por el Kernel
- Muchos threads == Mucha sincronización == Muchos bugs ✝

Ejemplo: **”Servidor web”**

- Si lanzamos un thread por cada conexión estamos muertos
- Si lanzamos un process por cada conexión estamos muertos

Cooperative Multitasking / Green-threads

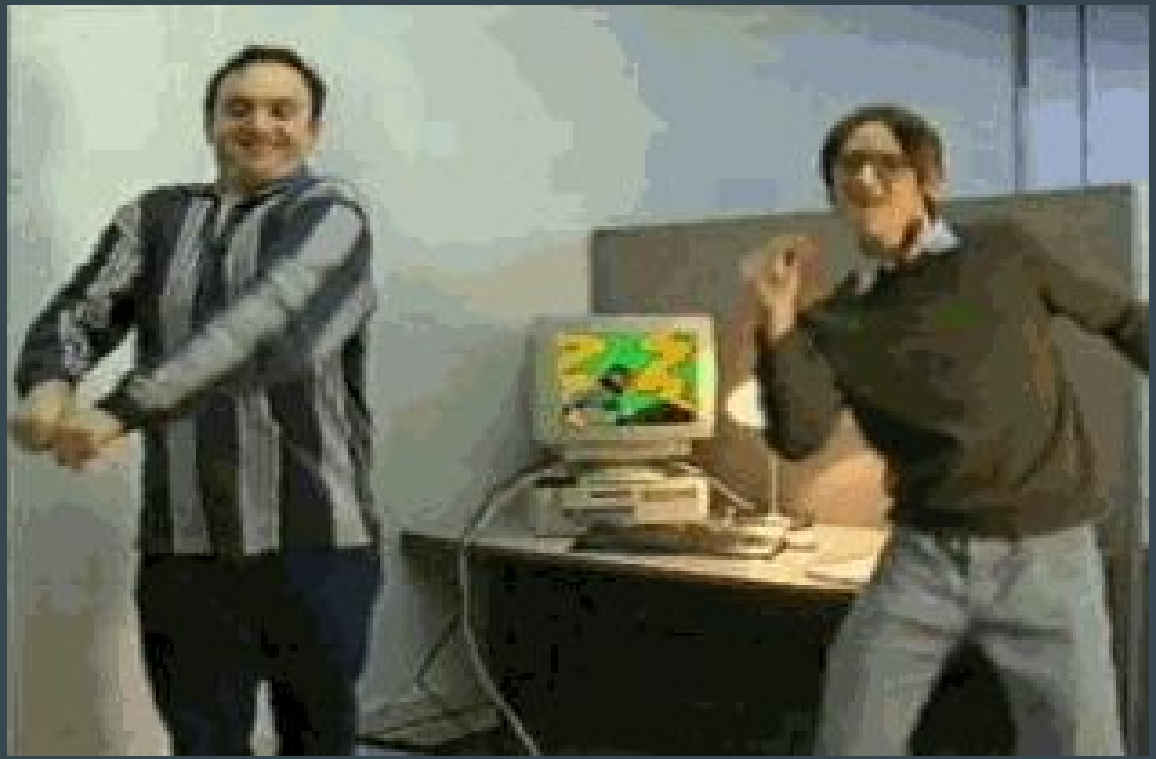
Estilo de “multitasking” donde el kernel nunca realiza un “Context Switch” sino que una tarea voluntariamente pausa su ejecución para darle la oportunidad a otra tarea.

Se utilizan **corrutinas** múltiples entry-point (PAUSE/RESUME)

Se dice que es **COOPERATIVO** porque las tareas deben cooperar cediendo el control.

Green-threads
Tareas planificadas por una librería o intérprete en lugar del SO.
No dependen de las capacidades del SO
Ejecutan en user-space land
Eliminan el Context Switch

Estamos salvados…



Cooperative Multitasking

Los Kernels no implementan este tipo de MULTI-TASKING

No es algo de propósito general

Tiene implicancias en la performance de ciertas tareas

Aplica muy bien al modelo de **I/O bound**

Quiero eso en Python!

Asyncio

Asyncio es una librería que provee la infraestructura para hacer codigo concurrente “single-thread “ usando **coroutine**, multiplexing I/O.

Event-loop
Abstracciones similares a Twisted
Implementación de TCP, UDP, Subprocess, pipe, etc
Futures, Coroutines
Primitivas de sincronización (Lock, Event, Condition)
Workers Pool (tareas en background)

Asyncio: Ejemplo

```python
import asyncio
import requests


async def fetch(url):
    loop = asyncio.get_event_loop()
    r = await loop.run_in_executor(None, requests.get, url)
    return r


def main():
    loop = asyncio.get_event_loop()
    urls = ('http://www.google.com/', 'http://python.org.ar/', 'http://python.org/')
    tasks = []
    for u in urls:
        t = loop.create_task(fetch(u))
        tasks.append(t)

    results = loop.run_until_complete(asyncio.gather(*tasks))
    for response in results:
        print('{} --> {}'.format(response.url, response.status_code))


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

Asyncio

Provee código **cooperativo** con una API **EXPLÍCITA** (Zen de Python) basada en **coroutines**, es decir, nosotros somos conscientes que estamos compartiendo recursos con otra tarea y cooperamos en los momentos que necesitamos “esperar/bloquear”, sin hacer Context-Switch

El event-loop está presente (diferencia con Gevent)
Utiliza el “mejor” mecanismo de Polling en el SO (poll/epoll/kqueue/select, etc)

Muy Eficiente para tareas **I/O BOUND**

Tenemos un ÚNICO thread ejecutando codigo (y los executors!)

“Conclusiones”

Necesitamos más CORES podemos lanzar más procesos con “**multiprocessing**”
Necesitamos cierto nivel (unos cuantos threads) de **I/O** podemos usar “**threading**”
Necesitamos gran nivel de **I/O** una buena opción para usar **Asyncio**
Las Queue (Queue.Queue, multiprocessing.Queue, asyncio.Queue) son **geniales**!

No hay “cosas” mágicas debemos probar y evaluar! Entender el stack con más detalles puede ayudar a decidir.

Cuando se usa Asyncio se debe estar seguro que no hay tareas bloqueantes!!!!!

Eficiencia != Menor tiempo de ejecución

Tratar de **aprovechar los recursos** lo mejor posible.

¿Preguntas?

Muchas Gracias!

```
while not manos.dormidas:
    aplaudir()
```

PYCONAR2016

· BAHÍA BLANCA - ARGENTINA ·

8va Conferencia Argentina de Python

25, 26 y 27 de Noviembre - Bahía Blanca

Martin Alderete
@alderetemartin

Links útiles

Python Github

Python C-API

Python Standard Library

Asyncio

David Beazley Home