پایتون GIL به زبان ساده

در پایتون ۱۰ آبان, ۱۳۹۹

همه‌ی ما در مورد GIL و محدودیت‌هایی که برای ما ایجاد می‌کنه چیزهایی شنیدیم و مطالبی خوندیم. اما آیا واقعا می‌دونیم GIL چی هست؟ قبل از اینکه بخوایم در مورد GIL صحبت کنیم، بیایید یه تابع ساده بنویسیم. فرض کنید قطعه کدی به صورت زیر نوشتیم که عددی رو از ورودی دریافت می‌کنه و یکی یکی از اون کم می‌کنه.

```
def count_down(n):
    while n > ۰:
        n -= 1
```

خب حالا این تابع رو با یه عدد بزرگ فراخونی می‌کنیم تا ببینیم چقدر طول می‌کشه تا اجرا بشه.

```
from time import time

before = time()
count_down(100000000)
after = time()

print(after-before)
```

جستجو ...

کتاب ترفندهای پایتون

مطالب اخیر

چطور کد دیگران رو بررسی کنیم؟ راهنمایی برای code review

۲۸ آذر, ۱۳۹۹

پایتون GIL به زبان ساده

۱۰ آبان, ۱۳۹۹

توابع کاربردی پایتون در Functional Programming

۴ مهر, ۱۳۹۹

در ترمینال لینوکس مخفی بمونید

۱۳ شهریور, ۱۳۹۹

اجرای این تابع روی سیستم من حدود ۵.۴۴ ثانیه طول کشید. حالا این تابع رو ۲ مرتبه داخل برنامه‌ای که نوشتم فراخونی می‌کنم و دوباره زمان اجرای اون رو حساب می‌کنم.

```
from time import time

before = time()
count_down(100000000)
count_down(100000000)
after = time()

print(after-before)
```

همونطور که انتظار داشتیم زمان اجرای این کد حدود ۲ برابر حالت قبل یعنی ۱۱.۲۱ ثانیه شده.

اگه بخوایم زمان اجرای این برنامه کمتر بشه یا سریعتر این برنامه اجرا بشه باید چیکار کنیم؟ خب همونطور که در تئوری بهمون یاد دادن وقتی از thread استفاده کنیم، می‌تونیم چند تا کار رو داخل برنامه همزمان و به صورت موازی اجرا کنیم. فراخونی دو تابع به صورت موازی یعنی انگار داریم اون تابع رو یک بار فراخونی می‌کنیم. پس این کار باید نسبت به حالت قبل سریعتر انجام بشه. پس این بار تابع بالا رو در دو thread مختلف اجرا می‌کنیم:

```
from threading import Thread
from time import time

before = time()

thread1 = Thread(target=count_down, args=(100000000,))
thread2 = Thread(target=count_down, args=(100000000,))

thread1.start()
thread2.start()

thread1.join()
thread2.join()

after = time()
print(after-before)
```

زمان اجرا؟ ۱۶.۲۹ ثانیه! درسته. برنامه‌ی ما نه تنها بهینه تر نشد، بلکه زمان اجراش بیشتر هم شد! اجرای کد بالا روی سیستم من حدود ۱۶.۲۹ ثانیه طول کشید. حتما با خودتون می‌گید چرا چنین اتفاقی افتاد؟ خب بهتره برگردیم به همون حالت قبل و از thread استفاده نکنیم. جالبه که اگه از پایتون ۲ استفاده کنید حتی نتیجه‌ی بدتری هم می‌گیرید!

بایگانی تاریخ خورشیدی

آذر ۱۳۹۹

آبان ۱۳۹۹

مهر ۱۳۹۹

شهریور ۱۳۹۹

مرداد ۱۳۹۹

تیر ۱۳۹۹

اما چرا چنین اتفاقی افتاد؟

علت تمام این مشکلات چیزی نیست جز GIL یا Global Interpreter Lock. در واقع GIL در پایتون به شما اجازه می‌ده تا در یک زمان فقط و فقط یک thread اجرا کنید. در مثال بالا، ما عملا تونستیم فقط یک thread رو در یک زمان اجرا کنیم. به همین دلیل هیچ افزایش سرعتی در روند اجرای برنامه‌مون ندیدیم. اما خب چرا اجرای برنامه کند تر شد؟ چون در طول اجرای برنامه، پایتون سعی می‌کنه تا thread رو تغییر بده (چون ما از اون خواسته بودیم تا از ۲ thread استفاده کنه)، اما GIL مانع از انجام این کار می‌شه. همین تلاش‌های بدون نتیجه در روند اجرای برنامه باعث هدر رفتن زمان می‌شه و به همین دلیل هست که سرعت اجرا برنامه حتی کند تر هم می‌شه.

اما تئوری چی؟ همه‌ی ما می‌دونیم که استفاده از threadها باعث افزایش اجرای سرعت برنامه‌هامون میشن. چرا اصلا چیزی مثل GIL وجود داره؟

بهتره بدونید که GIL چیز بدی نیست. در واقع خیلی هم خوبه. مدیریت حافظه در پایتون در هنگام کار با threadها به هیچ وجه امن نیست. زمانی که شما چندین thread رو اجرا می‌کنید، برنامه‌ی شما ممکنه نتایج عجیب و اشتباهی به شما برگردونه. برای مثال اگه GIL وجود نداشت و دو thread می‌خواستن مقدار یک متغیر رو در برنامه همزمان افزایش بدن، اون متغیر بجای اینکه ۲بار به مقدارش اضافه بشه، فقط یکبار اضافه می‌شد (توضیحات بیشتر). GIL در چنین مواقعی به کمک ما میاد و نمیذاره تا چنین اتفاقاتی بیفته.

خب بهتر نیست کلاس Thread از پایتون حذف بشه؟

چرا اصلا کلاس Thread رو حذف نکردن تا برنامه نویس‌ها به اشتباه ازش استفاده نکنن؟ در هر صورت ما که نمی‌تونیم ازش استفاده کنیم!

در حقیقت مواقعی پیش میاد که ما می‌تونیم از thread داخل برنامه هامون استفاده کنیم. مثال‌هایی که بالا در موردشون صحبت کردیم، تمامشون وابسته به CPU بودن. یعنی برای انجام محاسبات فقط به CPU نیاز داشتن! زمان انتظار اون کدها برای اجرا وابسته به CPU بود.

اما مواقعی هست که کد شما وابسته به عملیاتی مثل I/O هست. یعنی عملیاتی برای خوندن و نوشتن داخل برنامتون دارید. در چنین شرایطی این عملیات در خارج از GIL انجام می‌شه. اینجاست که می‌تونید از کلاس Thread با خیال راحت استفاده کنید.

در مثال زیر تابعی رو می‌بینید که یک عمل I/O رو قراره برای ما انجام بده. این تابع، درخواستی به یک url ارسال می‌کنه و محتویات دریافتی رو به صورت متن برمی‌گردونه.

```
import requests

def get_content(url):
    response = requests.get(url)
    return response.text
```

فراخونی این تابع با آرگومان ورودی https://google.com حدود ۰.۸ ثانیه زمان می‌بره. اگر دو بار پشت هم این تابع رو فواخونی کنیم طبیعتا حدود ۱.۶ ثانیه زمان صرف می‌شه. حالا همین تابع رو دو بار در دو thread مختلف اجرا می‌کنیم:

```
before = time()

thread1 = Thread(target=get_content, args=
('https://google.com',))
thread2 = Thread(target=get_content, args=
('https://google.com',))

thread1.start()
thread2.start()

thread1.join()
thread2.join()

after = time()

print(after - before)
```

زمان اجرا؟ ۰.۸ ثانیه! زمان دو بار اجرای این تابع با کمک thread دقیقا برابر با حالتیه که انگار اون رو یک بار فراخونی کردیم. پس بالاخره موفق شدیم!

یه راه حل بهتر

در کد بالا هر thread حافظه‌ای اضافی اشغال می‌کنه و تغییر thread هم باعث از دست رفتن مقداری زمان می‌شه. زمانی که دو thread داریم

این زمان اصلا زیاد نیست. اما زمانی که هزاران thread قرار هست با هم اجرا بشن، شاید چند گیگابایت RAM و درصدی از CPU شما برای سوییچ کردن بین threadها هدر بره.

برای حل این مسئله، می‌تونیم از کتابخونه‌ای به اسم asyncio استفاده کنیم. این کتابخونه از نسخه‌ی پایتون ۳.۴ به بعد قابل استفاده است و باید حواستون باشه که asyncio با سایر کتابخونه‌های موجود در برنامتون سازگار باشه.

این کتابخونه تمام تسک‌های ما رو حول چرخه‌ی eventهای خودش قرار میده و تابعی که نوشتیم رو به صورت async (نا همگام) در یک thread اجرا می‌کنه. برخلاف کتابخونه‌ی Thread، سوییچ بین تسک‌ها در asyncio باید توسط خود برنامه نویس انجام بشه (با استفاده از await). حالا کد قبل رو این بار با کمک asyncio پیاده سازی می‌کنیم:

```
import asyncio
import aiohttp

loop = asyncio.get_event_loop()

async def get_content(pid, url):
    session = aiohttp.ClientSession(loop=loop)
    async with session.get(url) as response:
        content = await response.read()
        print(pid, content)
    await session.close()

loop.create_task(get_content(1, 'http://google.com/'))
loop.create_task(get_content(2, 'http://google.com/'))
loop.create_task(get_content(3, 'http://google.com/'))
loop.create_task(get_content(4, 'http://google.com/'))
loop.create_task(get_content(5, 'http://google.com/'))

loop.run_forever()
```

توجه کنید که در کد بالا از کتابخونه‌ی aiohttp بجای requests استفاده کردیم. aiohttp یک کتابخونه‌ی مشابه با requests هست با این تفاوت که می‌تونیم از اون به صورت async استفاده کنیم. در مثال بالا ابتدا یک تابع async تعریف کردیم (می‌تونیم بهش coroutine هم بگیم)، سپس اون رو ۵ بار با idهایی فراخونی کردیم تا نتیجه‌ی خروجی برای ما واضح تر بشه. خروجی اجرای کد بالا به صورت زیر هست:

```
۳ b'<!DOCTYPE html PUBLIC...
۴ b'<!DOCTYPE html PUBLIC...
۲ b'<!DOCTYPE html PUBLIC...
۱ b'<!DOCTYPE html PUBLIC...
۵ b'<!DOCTYPE html PUBLIC...
```

همونطور که می‌بینید ترتیب نمایش خروجی به همون ترتیب فراخوانی تابع در برنامه نیست. به عبارت دیگه برنامه‌ی ما وقتی به کلمه‌ی await برمی‌خوره، عمل سوییچ بین تسک‌ها رو انجام می‌ده. ما می‌تونیم همین نتایج رو با استفاده از کتابخونه‌ی Thread هم بگیریم. ولی استفاده از asyncio سربار کمتری برای برنامه‌مون ایجاد می‌کنه.

- توجه کنید ما اینجا در مورد جزییات کتابخونه‌ی asyncio و برنامه نویسی async در پایتون صحبت نمی‌کنیم. بلکه فقط در مورد کارایی این کتابخونه صحبت می‌کنیم. برای اطلاعات بیشتر در مورد asyncio می‌تونید از اینجا اطلاعات خوبی بدست بیارید.

خب حالا که در مورد تسک‌های I/O صحبت کردیم، بهتره به همون مشکل تسک‌های وابسته به CPU برگردیم.

راهکاری برای تسک‌های وابسته به CPU

کلاسی در پایتون با نام multiprocessing.Process هست که عملکرد و استفاده از اون تقریبا مشابه با کلاس Thread هست. با این تفاوت که این کلاس از sub-processها بجای threadها استفاده می‌کنه. یعنی بجای اینکه یک thread برای شما ایجاد کنه، یک پروسه‌ی جدا برای شما می‌سازه (os.fork). به همین دلیل عملیاتی که داره انجام می‌ده توسط GIL مسدود نمی‌شه. خب بیایید امتحان کنیم. فقط کافیه کدی که قبلا با Thread نوشتیم رو کمی تغییر بدیم و از Process بجای Thread در اون استفاده کنیم.

```
from multiprocessing import Process
from time import time

before = time()

process1 = Process(target=count_down, args=(100000000,))
process2 = Process(target=count_down, args=(100000000,))

process1.start()
process2.start()

process1.join()
process2.join()

after = time()
print(after-before)
```

اجرای این کد روی سیستم من حدود ۶ ثانیه طول کشید. زمان اجرا حدودا برابر با زمانیه که انگار این تابع رو یکبار فراخونی کردیم. اگه این تابع رو سه بار در سه Process مختلف هم اجرا کنید بازهم همین مقدار طول می‌کشه. خب این عالیه! اما توجه کنید که این پردازش‌ها هر کدوم در یک فضای حافظه‌ی جداگونه اجرا می‌شن. بنابراین پردازش‌ها نمی‌تونن داده‌ها یا آبجکت‌ها رو بین خودشون به اشتراک بذارن (در حالی در threadها می‌تونستن).

یادتون باشه ما اینجا فقط در مورد CPython صحبت کردیم. پیاده سازی‌های دیگه‌ای از زبان پایتون مثل Jython و IronPython وجود دارن که محدودیت‌های GIL رو ندارن. اما پیشنهاد می‌شه که اکثر مواقع (مگر در مواردی که دقیقا می‌دونید هدفتون چیه) از CPython استفاده کنید. همچنین پروژه‌هایی مثل Jython معمولا خیلی بروز نیستن و سرعت توسعه‌ی اون‌ها همیشه کندتر از CPython هست. پس اگه می‌خواید از تمام ویژگی‌های بروز و خوب پایتون بدون هیچ دردسری بهره ببرید بهتره از CPython استفاده کنید. برای اینکه در مورد انواع پیاده سازی‌های زبان پایتون اطلاعات بیشتری بدست بیارید، می‌تونید اینجا رو ببینید.

مطلب قبلی:
توابع کاربردی پایتون در Functional Programming

مطلب بعدی:
چطور کد دیگران رو بررسی کنیم؟ راهنمایی برای code review

مطالب مرتبط: