فعالیت پنجم

سیستم‌های توزیع‌شده

سید امیرمسعود میرکاظمی

۴۰۳۶۱۶۰۲۰۰۳

بخش اول: سه مدل اجرای موازی

مدل اول: اجرای دستی با `threading.Thread`

تعریف:

در این مدل، برای هر وظیفه یک شیء از کلاس `threading.Thread` ایجاد می‌شود و با اجرای تابع thread، `start()` مربوطه کار خود را آغاز می‌کند. در پایان، با استفاده از `join()` منتظر می‌مانیم تا تمام threadها پایان یابند.

مکانیزم:

- هر thread مستقل از threadهای دیگر است.
- برنامه‌نویس باید به‌صورت دستی thread بسازد، اجرا کند و منتظر پایانش بماند.
- مدیریت تعداد زیاد threadها می‌تواند پیچیده، کند، و پرهزینه باشد.

در CPython، وجود GIL مانع می‌شود که بیش از یک thread در هر لحظه اجرای واقعی روی CPU داشته باشد (برای taskهای CPU-bound). بنابراین با وجود چندین thread، پردازش‌ها به‌صورت واقعی موازی نیستند، بلکه یکی‌یکی اجرا می‌شوند.

مدل دوم: استفاده از `ThreadPoolExecutor`

تعریف:

`ThreadPoolExecutor` یکی از امکانات کتابخانه‌ی `concurrent.futures` است که مدیریت اجرای همزمان وظایف را ساده و بهینه می‌کند. برخلاف مدل دستی، در اینجا از یک pool (استخر) از threadهای آماده استفاده می‌شود.

مکانیزم:

- یک pool با تعداد محدودی thread ثابت ایجاد می‌شود.
- وظایف (تابع‌هایی که باید اجرا شوند) به pool ارسال می‌شوند و در صف قرار می‌گیرند.
- فقط همان تعداد thread فعال می‌ماند و وظایف به‌نوبت اجرا می‌شوند.
- استفاده از `submit()` یا `map()` برای تخصیص وظایف به threadها انجام می‌شود.

محدودیت GIL:

همچنان تحت تأثیر GIL است، بنابراین برای taskهای CPU-bound عملکرد بهتری از مدل دستی ندارد، اما برای taskهای I/O-bound یا تعداد بالای وظایف سبک بسیار بهتر عمل می‌کند.

مزایا:

- مدیریت ساده و خودکار threadها
- کنترل بهینه بار با استفاده از صف
- مناسب برای وظایف سبک ولی زیاد

مدل سوم: استفاده از `multiprocessing.Pool`

تعریف:

ماژول `multiprocessing` در پایتون امکان اجرای موازی واقعی را از طریق processهای مستقل فراهم می‌کند. برخلاف threadها، هر process فضای حافظه، تفسیرگر، و زمان‌بندی مستقل دارد.

مکانیزم:

- یک pool از processها ایجاد می‌شود (معمولاً برابر با تعداد هسته‌های CPU).
- هر وظیفه به یکی از processها تخصیص داده می‌شود.
- چون processها مستقل هستند، GIL بی‌تأثیر است و همزمانی واقعی برقرار است.

بین processها، داده‌ها باید سریال‌سازی (serialization) شوند (مثلاً با pickle).
این باعث می‌شود overhead ارتباطی بیشتر از threading باشد، اما در وظایف سنگین قابل چشم‌پوشی است.

مناسب برای:
taskهای CPU-bound
بارهای سنگین که نیاز به استفاده کامل از چند هسته دارند

ویژگی	`threading.Thread`	`ThreadPoolExecutor`	`multiprocessing.Pool`
نحوه اجرا	دستی و جداگانه	خودکار با صف داخلی	موازی واقعی با process
تأثیر GIL	دارد	دارد	ندارد
مناسب برای	I/O-bound	I/O-bound / وظایف سبک	CPU-bound / سنگین
نیاز به مدیریت thread/process	بله	خیر	خیر

پیچیدگی کد	متوسط	کم	متوسط
استفاده از چند هسته‌ی CPU	نه	نه	بله

بخش دوم: کدهای پیاده‌سازی سه مدل اجرای موازی

در این بخش، ساختار کلی و عملکرد کد مربوط به هر سه مدل (مدل دستی با `threading.Thread`، مدل آماده با `ThreadPoolExecutor`، و مدل فرآیندی با `multiprocessing.Pool`) به‌صورت دقیق توضیح داده می‌شود. برای یکنواختی، ابتدا یک وظیفه سبک (بررسی اول بودن عدد) انتخاب شده و سپس همین الگو برای وظیفه سنگین‌تر (محاسبه بازگشتی فیبوناچی) استفاده شده است.

1. کد بررسی اول بودن عدد (`is_prime`)

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

این تابع عدد n را دریافت کرده و با یک حلقه ساده بررسی می‌کند که آیا n عدد اول است یا خیر. در صورتی که هیچ مقسوم‌علیه غیر از ۱ و خودش نداشته باشد، مقدار True بازمی‌گرداند.

2. پیاده‌سازی مدل اول: اجرای دستی با `threading.Thread`

```python
def run_manual_threads(numbers):
    threads = []
    start = time.time()
    for n in numbers:
        t = threading.Thread(target=is_prime, args=(n,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    end = time.time()
    return end - start
```

در این بخش از کد:

- ابتدا برای هر عدد در `numbers` یک thread جدید با وظیفه اجرای تابع `is_prime` ساخته می‌شود.
- همه threadها با `start()` آغاز به کار می‌کنند.
- پس از شروع همه، با استفاده از `join()` منتظر می‌مانیم تا همگی پایان یابند.
- مدت زمان اجرای کل عملیات محاسبه و بازگردانده می‌شود

3. پیاده‌سازی مدل دوم: استفاده از `ThreadPoolExecutor`

```python
from concurrent.futures import ThreadPoolExecutor

def run_executor_threads(numbers, max_workers=10):
    start = time.time()
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        list(executor.map(is_prime, numbers))
    end = time.time()
    return end - start
```

در این بخش:

- ابتدا یک استخر از threadها با حداکثر `max_workers` ساخته می‌شود.
- وظایف با استفاده از `executor.map` به threadها اختصاص داده می‌شود.
- این کار باعث اجرای پی‌درپی وظایف در تعداد محدودی thread فعال می‌شود.
- پس از اتمام همه وظایف، زمان اجرا محاسبه و بازگردانده می‌شود.

مزیت اصلی این مدل، مدیریت خودکار تعداد threadها و استفاده بهینه از منابع است.

4. پیاده‌سازی مدل سوم: استفاده از `multiprocessing.Pool`

```python
from multiprocessing import Pool, cpu_count

def run_multiprocessing(numbers, num_processes=None):
    if num_processes is None:
        num_processes = cpu_count()
    start = time.time()
    with Pool(processes=num_processes) as pool:
        pool.map(is_prime, numbers)
    end = time.time()
    return end - start
```

در این مدل:

- یک استخر از processها ساخته می‌شود (تعداد آن معمولاً برابر با تعداد هسته‌های CPU است).
- وظایف با `pool.map` بین processهای مختلف تقسیم می‌شود.
- چون هر process مستقل از دیگری است، GIL بی‌تأثیر بوده و هم‌زمانی واقعی حاصل می‌شود.
- زمان کل اجرا اندازه‌گیری و بازگردانده می‌شود.

تابع سنگین

```python
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

این تابع عدد n را دریافت کرده و مقدار فیبوناچی آن را به‌صورت بازگشتی محاسبه می‌کند. به دلیل ساختار بازگشتی و نبود بهینه‌سازی، پیچیدگی زمانی آن به‌شکل نمایی (O(2^n)) افزایش می‌یابد، و در نتیجه برای مقادیری مثل `fib(25)` بار پردازشی بالایی ایجاد می‌شود. این ویژگی باعث می‌شود که این تابع انتخاب خوبی برای شبیه‌سازی وظیفه‌های سنگین CPU-bound باشد.

۱. مدل اول: پیاده‌سازی دستی با `threading.Thread`

```python
def run_manual_threads_fib(numbers):
    threads = []
    start = time.time()
    for n in numbers:
        t = threading.Thread(target=fib, args=(n,))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    end = time.time()
    return end - start
```

توضیح:

- این مدل مشابه نسخه prime check است، با این تفاوت که تابع `fib` به‌عنوان وظیفه استفاده شده.
- به ازای هر مقدار در لیست `numbers`، یک thread جدید ایجاد شده و تابع `fib(n)` روی آن اجرا می‌شود.
- به دلیل استفاده از threading و وجود GIL، این مدل نمی‌تواند از چند هسته به‌طور واقعی استفاده کند و اجرای وظایف سنگین به‌صورت سریالی صورت می‌گیرد.

۲. مدل دوم: استفاده از `ThreadPoolExecutor`

```python
def run_executor_threads_fib(numbers, max_workers=10):
    start = time.time()
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        list(executor.map(fib, numbers))
    end = time.time()
    return end - start
```

توضیح:

- با ایجاد یک pool از threadها، وظایف `fib(n)` در یک صف قرار می‌گیرند و به صورت همزمان در تعداد محدودی thread اجرا می‌شوند.
- با وجود این ساختار بهینه‌تر نسبت به مدل دستی، همچنان تابع `fib` تحت GIL اجرا می‌شود، و بهره‌گیری از پردازنده در وظایف سنگین محدود باقی می‌ماند.
- این روش بیشتر برای وظایف سبک مناسب است و در اینجا سرعت چشمگیری ندارد.

۳. مدل سوم: استفاده از `multiprocessing.Pool`

```
def run_multiprocessing_fib(numbers, num_processes=None):
    if num_processes is None:
        num_processes = cpu_count()
    start = time.time()
    with Pool(processes=num_processes) as pool:
        pool.map(fib, numbers)
    end = time.time()
    return end - start
```

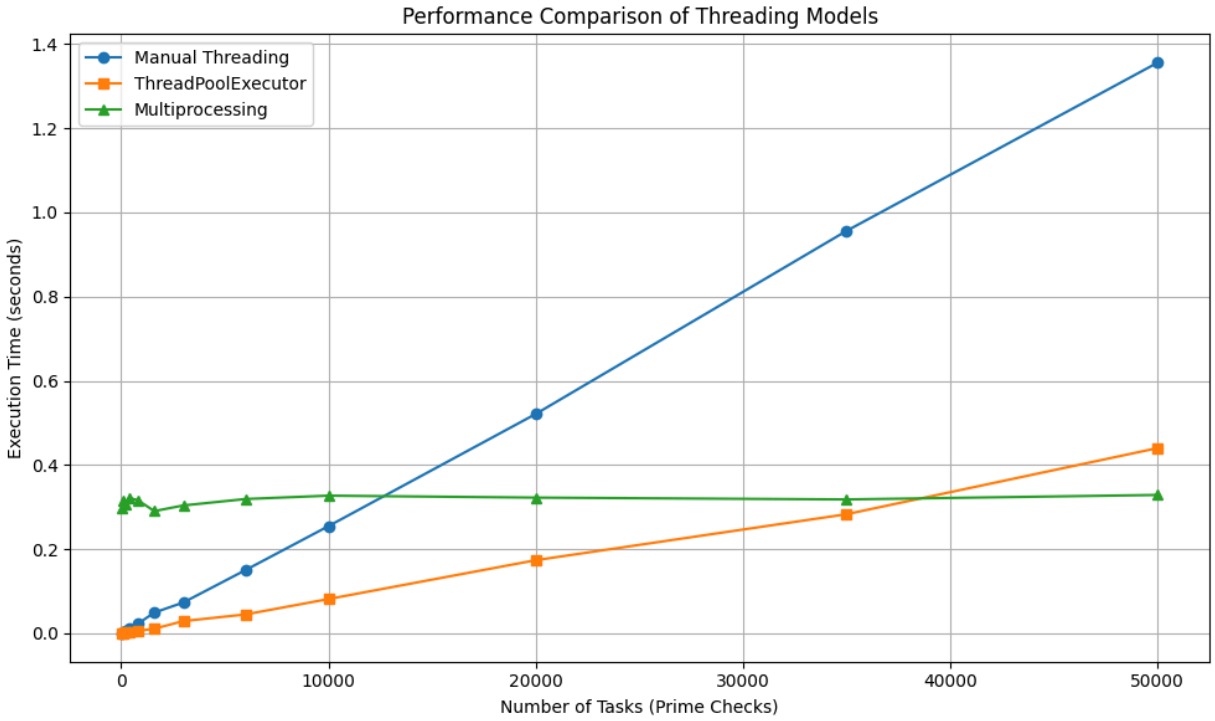
در این مدل، لیست `numbers` (که حاوی مقادیر تکراری مانند `[25, 25, ..., 25]` است) به pool از processهای مستقل فرستاده می‌شود.
هر process در فضای حافظه جداگانه کار می‌کند و GIL شامل حال آن‌ها نمی‌شود.
در این حالت، فرآیندها به‌صورت واقعی بر روی هسته‌های مختلف CPU اجرا می‌شوند و به همین دلیل، این مدل کاراترین عملکرد را برای وظایف سنگین ارائه می‌دهد.

بخش سوم: تحلیل نتایج و نمودارهای عملکرد

برای ارزیابی سه مدل اجرای همزمان، نتایج آزمایش‌ها در قالب نمودارهایی ارائه شده‌اند. دو نوع وظیفه با ماهیت کاملاً متفاوت انتخاب شد: یکی سبک و ساده (بررسی اول بودن اعداد) و دیگری سنگین و کاملاً CPU-bound (محاسبه عدد فیبوناچی به روش بازگشتی).

در هر آزمایش، تعداد وظایف (تعداد اعدادی که باید پردازش شوند) افزایش یافت و زمان کلی اجرای مجموعه وظایف برای هر مدل اندازه‌گیری شد.

۱. نمودار اول: وظایف سبک (بررسی اول بودن اعداد)



وظیفه‌ای با محاسبه کوتاه، بدون پیچیدگی زمانی بالا، مناسب برای اجرای همزمان در مقیاس بزرگ.

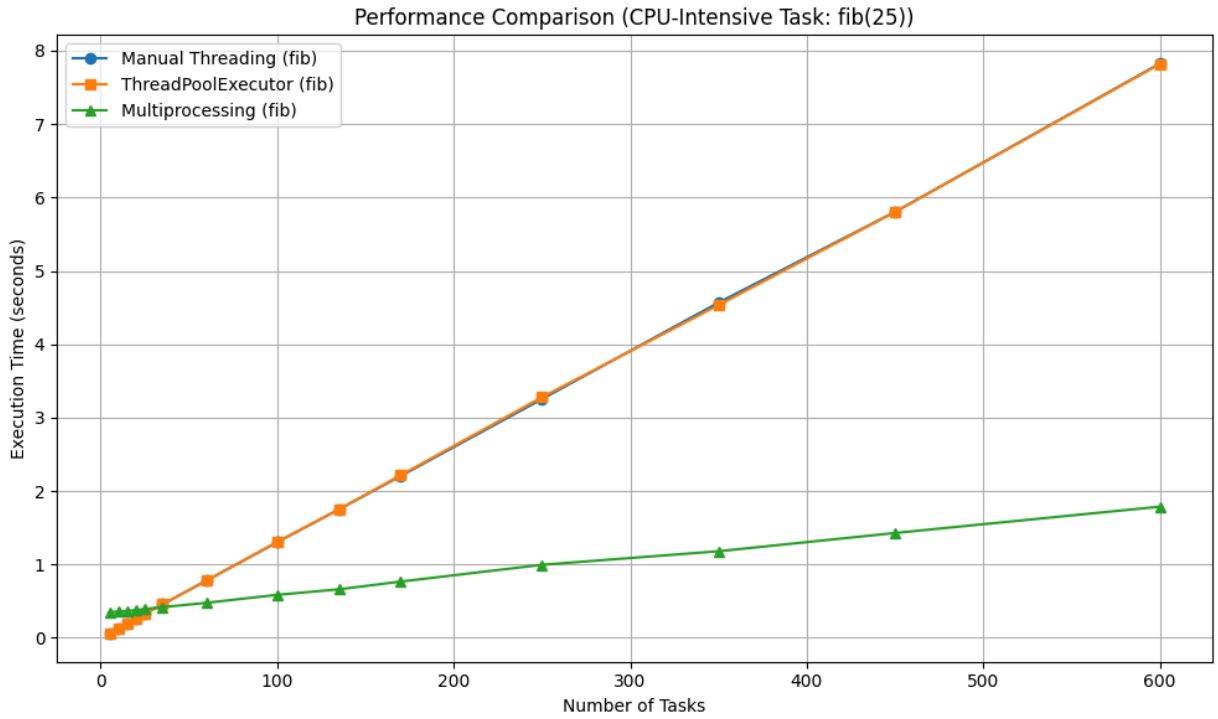
`Manual Threading`: با افزایش تعداد وظایف، زمان اجرا به‌شدت افزایش یافت. دلیل آن هزینه بالای ایجاد و زمان‌بندی threadهای زیاد است که در مدل دستی کنترل نمی‌شوند.

`ThreadPoolExecutor`: با مدیریت خودکار threadها و استفاده از صف، افزایش تعداد وظایف با رشد زمان بسیار ملایم‌تری همراه بود. این مدل عملکرد بهتری در این وظایف سبک دارد.

`Multiprocessing`: اگرچه موازی‌سازی واقعی انجام شد، ولی در وظایف بسیار سبک، سربار ایجاد processها و تبادل داده باعث شد که در برخی مقاطع، زمان اجرا تفاوت چندانی با مدل‌های دیگر نداشته باشد.

در وظایف سبک، `ThreadPoolExecutor` معمولاً بهترین توازن بین سرعت اجرا و سادگی پیاده‌سازی را ارائه می‌دهد.

۲. نمودار دوم: وظایف سنگین (محاسبه فیبوناچی بازگشتی)



محاسبه نمایی، کاملاً CPU-bound و نیازمند استفاده مؤثر از چند هسته واقعی برای دستیابی به بهینه‌ترین اجرا.

`Manual Threading` و `ThreadPoolExecutor`: هر دو تحت تأثیر مستقیم GIL قرار دارند. در نتیجه، با افزایش تعداد وظایف، اجرای آن‌ها به‌صورت سریالی انجام شده و زمان اجرا تقریباً خطی افزایش یافته است.

`Multiprocessing`: برخلاف مدل‌های قبلی، این روش با بهره‌گیری از processهای مستقل، قادر بود وظایف را به‌صورت واقعی بر روی هسته‌های مختلف CPU توزیع کند. بنابراین، حتی با افزایش حجم وظایف، رشد زمان اجرا ملایم‌تر و مدیریت‌شده بود.
در وظایف سنگین، تنها مدل `multiprocessing` قادر به عبور از محدودیت GIL و بهره‌برداری از قدرت پردازشی کامل سیستم است.

۳. مقایسه:

مدل اجرا	وظیفه سبک (prime)	وظیفه سنگین (fib)
Manual Threading	ناکارآمد در مقیاس بالا	کند، تحت تأثیر GIL
ThreadPoolExecutor	بسیار خوب	کند، به دلیل GIL
Multiprocessing	به‌طور نسبی خوب	بهترین عملکرد در وظیفه سنگین

بخش چهارم: نتیجه‌گیری

هدف این آزمایش، مقایسه عملکرد سه مدل اجرای موازی در پایتون برای وظایف سبک و سنگین بود. مدل‌های مقایسه شده شامل `threading.Thread` (مدل دستی)، `ThreadPoolExecutor` (مدل آماده)، و `multiprocessing.Pool` (مدل مبتنی بر فرآیند مستقل) بودند.

۱. اثر GIL و تفاوت میان threading و multiprocessing

مهم‌ترین نکته‌ای که در این آزمایش به‌وضوح مشاهده شد، اثر (GIL) بر عملکرد مدل‌های مبتنی بر threading در پایتون بود. GIL اجازه نمی‌دهد بیش از یک thread در یک زمان روی هسته‌های CPU به‌صورت واقعی اجرا شود؛ بنابراین حتی در صورت اجرای همزمان چندین thread، باز هم عملیات‌ها به‌صورت سریالی اجرا می‌شوند.
در مقابل، `multiprocessing` با استفاده از فرآیندهای جداگانه، به‌طور کامل از GIL عبور می‌کند و می‌تواند وظایف را به شکل واقعی و هم‌زمان روی چند هسته CPU اجرا کند. این ویژگی در وظایف سنگین، تأثیر بسیار قابل توجهی بر کارایی دارد.

۲. انتخاب مدل بر اساس نوع وظیفه

برای وظایف سبک، سریع، و تعداد زیاد، مثل بررسی اول بودن اعداد، مدل `ThreadPoolExecutor` بسیار مناسب است. این مدل با مدیریت داخلی threadها و استفاده از صف وظایف، تعادل خوبی بین

سادگی و کارایی ارائه می‌دهد.
برای وظایف سنگین و CPU-bound، مثل محاسبه بازگشتی فیبوناچی، تنها راهکار مؤثر، استفاده از `multiprocessing` است. این مدل اجازه می‌دهد تمام هسته‌های CPU به‌صورت موازی و واقعی وظایف را انجام دهند.
مدل `threading.Thread` به دلیل نبود کنترل داخلی روی تعداد threadها و هزینه بالای ایجاد thread، فقط برای موارد ساده و آموزشی مناسب است و در سناریوهای واقعی بهینه نیست.

ویژگی مسئله	مدل پیشنهادی
وظیفه I/O-bound یا سریع	`ThreadPoolExecutor`
وظیفه CPU-bound سنگین	`multiprocessing.Pool`
وظیفه سبک با کنترل دستی	`threading.Thread` (برای یادگیری)

این آزمایش نشان داد که درک نظری مفاهیم threading و multiprocessing به‌تنهایی کافی نیست، بلکه اندازه‌گیری عملی زمان اجرا و مشاهده رفتار سیستم در بارهای مختلف نقش مهمی در انتخاب درست مدل اجرایی دارد.