Python 6: Concurrency & Parallelism

IN608: Intermediate Application Development Concepts

Kaiako: Tom Clark & Grayson Orr

Last Session’s Content

- Exceptions
  - Try
  - Catch
  - Finally
- Automation testing
  - Unit testing
  - Integration testing
  - Selenium Python

Today’s Content

- Introduction
  - Performance pressures
  - Concurrency & parallelism
- Multithreading
- Multiprocessing
- Asyncio

Introduction

Performance Pressures

- Two things that can slow down a process:
  - Waiting for I/O - I/O bound
  - Heavy computational load - CPU bound

Concurrency & Parallelism

- We can improve performance in these situations by using concurrency & parallelism
- If our program is waiting on I/O, then we can use concurrency to something else while waiting
- Sometimes a complex computation can be divided into multiple parts that be run in parallel
- These techniques can yield great performance gains, but they are somewhat complex & can lead to difficult bugs

Example - Slow Code

- This code spends most of its time waiting

```
from time import sleep, time

def slow(x):
    sleep(10)
    return x

def do_tasks(num):
    for n in range(num):
        slow(n)

do_tasks(5)
```

Multithreading

Concurrency with Multithreading

- This code still spends most of its time waiting
- All of the invocations of `slow` overlap in time, so it is much faster

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

def slow(x):
    sleep(10)
    return x

def do_threaded_tasks(num):
    tasks = list(range(num))
    with ThreadPoolExecutor(max_workers=10) as ex:
        results = ex.map(slow, tasks)

do_threaded_tasks(5)
```

Threading Challenges

- This code spends most of its time waiting
- Threads do not necessarily run in parallel, especially in Python
  - This means that they may not speed up CPU bound code
- We do not know exactly when threads will execute, or in what order they will run
- Threads that read & write to shared memory in an uncontrolled way produce unpredictable results
- Code that does not have this problem is said to be thread-safe
- Resource: https://docs.python.org/3/library/threading.html

Programming Activity (30 Minutes)

Programming Activity

- Checkout to master - `git checkout master`
- Create a new branch called 06-practical - `git checkout -b 06-practical`
- Copy 06-practical.ipynb from the course materials repository into your practicals repository
- Open up the Anaconda Prompt (it should be install on all lab computers) & `cd` to your practicals repository
- Run the following command: `jupyter notebook`

Programming Activity

- Please open 03-practical.ipynb
- Please **ONLY** answer questions 1-3
- We will go through the solutions after 30 minutes

Multiprocessing

Parallelism with Multiprocessing

- Each invocation of `slow` run in its own process

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

def slow(x):
    sleep(10)
    return x

def do_multiprocessing_tasks(num):
    tasks = list(range(num))
    with ProcessPoolExecutor() as ex:
        ex.map(slow, tasks)

do_multiprocessing_tasks(5)
```

Multiprocessing Challenges

- Since each task is run in a separate process with its own memory, race conditions are less of an issue, but it also means that it is harder to share information
- Processes can run on other cores & thus run in parallel
- The number of processes that can run at one time is limited to the number of cores on the host
- Resource: https://docs.python.org/3/library/multiprocessing.html

Asyncio

Multitasking Model

- The threading model we looked at earlier uses preemptive multitasking
- This means that the operating system decides when to switch execution between threads
- Another model is cooperative multitasking. In this model a thread continues executing until it indicates that it can give up control to another thread
- In Python, this is implemented with Asyncio
- Resource https://docs.python.org/3/library/asyncio.html

Concurrency with Asyncio

```
import asyncio

async def slow():
    await asyncio.sleep(10)
    return 0

async def do_async_tasks(num):
    tasks = []
    for n in range(num):
        tasks.append(slow(n))
    await asyncio.gather(*tasks, return_exceptions=True)

asyncio.run(do_async_tasks(10))
```

Concurrency with Asyncio

- Functions defined with `async` are different from ordinary functions
- Rather than just producing their results, they return Coroutine objects
- These objects can be used in an event loop or in other async tasks

```
async def slow():
    await asyncio.sleep(10)
    return 0
```

Concurrency with Asyncio

- When an async task awaits something, it indicates to the event loop that this may take a while
- Some other task should use the time to run its code
- The target of an await must be an awaitable like a coroutine

```
async def do_async_tasks(num):
    tasks = []
    for n in range(num):
        tasks.append(slow(n))
    await asyncio.gather(*tasks, return_exceptions=True)
```

Concurrency with Asyncio

- Runs an event loop
- Async task can be added to the loop
- The system loops over its tasks, checking on their status & giving them time to run their code when they are ready

```
asyncio.run(do_async_tasks(10))
```