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
- Multithreading
- Multiprocessing
- Asyncio

Concurrency & Parallelism Introduction

Performance Pressures

Two things that can slow down a process:

- Waiting for I/O - (I/O bound)
- Heavy computational load - (Processor bound)

Concurrency & Parallelism

We can improve performance in these situations by using concurrency and parallelism

- If our program is waiting on I/O, then we can use concurrency to do something else while waiting.
- Sometimes a complex computation can be divided into multiple parts That may be run in parallel.

These techniques can yield great performance gains, but they are somewhat complex and can lead to difficult bugs.

Example slow code

```
from time import sleep, time

def slowfunc(i):
    sleep(10)
    return i

def do_tasks(num):
    for k in range(num):
        slowfunc(k)

count = 10
do_tasks(count)
```

This code spends most of its time just waiting.

Concurrency with Multithreading

Concurrency with multithreading

```
from time import sleep
import concurrent.futures

def slowfunc(i):
    sleep(10)
    return i

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

count = 10
do_threaded_tasks(count)
```

This code still spends most of its time just waiting, but all of the invocations of `slowfunc()` overlap in time, so it's much faster.

Threading challenges

- Threads don’t necessarily run in parallel, especially in Python. This means that they may not speed up processor-bound code.
- We don’t know exactly when threads will execute, or in what order they will run.
- Threads that read and 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 02-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 06-practical.ipynb
- Please **ONLY** answer questions 1-2
- We will go through the solutions after 30 minutes

Concurrency & Parallelism with Multiprocessing

Parallelism with multiprocessing

```
from time import sleep
import multiprocessing

def slowfunc(i):
    sleep(10)
    return i

def do_multiprocess_tasks(num):
    tasks = list(range(num))
    with multiprocessing.Pool() as pool:
        pool.map(slowfunc, tasks)

count = 10
do_multiprocess_tasks(count)
```

In this code each invocation of `slowfunc()` runs in its own process.

Multiprocessing issues

- 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’s harder to share information.
- Processes can run on other cores and 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

Concurrency with Asyncio

Multitasking models

- 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. This is implemented in Python with *Asyncio*
- Resource https://docs.python.org/3/library/asyncio.html

Concurrency with asyncio

```
import asyncio

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

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

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

What's going om

```
asyncio.run(do_async_tasks(count))
```

This runs an *event loop*. Async tasks can be added to the loop. The system loops over its tasks, checking on their status, and giving them time to run their code when they're ready to do so.

What’s going on

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

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.

What’s going on

```
await asyncio.sleep(10)
```

When an async task *awaits* something, it indicates to the event loop that this may take a while, and maybe some other task should use the time to run its code. But they can’t await just any code. The target of an await must be an *awaitable*, like a coroutine.