Session 4 : Thread and process pools

Razmig Kéchichian

Learning outcomes

The aim of this session is to familiarize you with Python process and thread pools and how to make use of them in scenarios of data and task parallelism.

I. Process pools and data parallelism

Recall that the principal aim behind creating process and thread pools is to avoid the overhead of creating individual processes and threads and limiting their numbers thus improving the use of system resources. Python provides 2 APIs for process and thread pools:

1. the `Pool` class defined in the module `multiprocessing` for synchronous and asynchronous execution,
2. the `concurrent.futures` module providing both thread and process pools for asynchronous execution with identical API.

We shall examine the `multiprocessing.Pool` class in this section and study its application to data parallelism.

The `Pool` class represents a pool of worker processes. It has methods which allow tasks to be submitted to the worker processes in a few different ways. Take a moment to have a look on its documentation. To illustrate the use of `Pool`, we return to the example of Fibonacci series generation. The `fibonacci()` function below returns a tuple consisting of the index it has received and the corresponding Fibonacci series element it has calculated. Notice that the `Pool` object is used in a context manager, that is, in a `with ... as ...` statement. Execute the example and peruse the documentation to make sure that you understand each of the ways in which the pool is used.

```
import time
import random
import multiprocessing

def fibonacci(n):
    print(multiprocessing.current_process().name)
    a, b = 0, 1
    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    return (n, a)

if __name__ == "__main__":
    indexes = [random.randint(0, 100) for i in range(10)]

    with multiprocessing.Pool(processes = 4) as pool:
        print("*** Synchronous call in one process")
        result = pool.apply(fibonacci, (10,))
        print(result)

        print("*** Asynchronous call in one process")
        result = pool.apply_async(fibonacci, (10,))
        print(result.get())

        print("*** Synchronous map")
        for x in pool.map(fibonacci, indexes):
            print(x)

        print("*** Asynchronous map")
        for x in pool.map_async(fibonacci, indexes).get():
            print(x)

        print("*** Deliberate timeout")
        result = pool.apply_async(time.sleep, (5,))
        print(result.get(timeout=1))
```

Exercise 1: Prime or not ?

We'd like to carry out a primality test on a large number of relatively large integers. A process pool would be ideal for such CPU-bound tasks. Write a Python program that generates a number of random integers in the range

$[10^3, 10^6]$ the primality of which is tested by worker processes in a `multiprocessing.Pool`. Submit tasks to the pool by at least one synchronous and one asynchronous method. Experiment with the number of worker processes and the number of generated integers. Try to measure execution times by calculating the elapsed time in the following manner:

```python
import time
start = time.time()
# some code
end = time.time()
seconds = end - start
```

Avoid input and output operations (i.e. `input()` and `print()`) in execution time measurements. For the primality test itself, you can use the following code:

```python
import math

def is_prime(n):
    if n < 2:
        return (n, False)
    if n == 2:
        return (n, True)
    if n % 2 == 0:
        return (n, False)

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return (n, False)
    return (n, True)
```

II. Thread pools and task parallelism

The Python module `concurrent.futures` provides the classes `ThreadPoolExecutor` and `ProcessPoolExecutor` which implement asynchronous thread and process pools. We shall examine the former class in this section. Everything that we mention about `ThreadPoolExecutor` also applies to `ProcessPoolExecutor`.

A `ThreadPoolExecutor` object can be instantiated with a maximum number of worker threads. Tasks can be submitted to a `ThreadPoolExecutor` object in two manners, either via its `map()` method which applies a function to iterables, or via its `submit()` method which schedules a function call and returns a `Future` object representing its execution. The following example illustrates both task submission methods through the above Fibonacci series generation example, this time implemented as a thread pool. Execute the example and peruse the documentation to make sure that you understand each of the ways in which the pool is used.

```python
import random
import concurrent.futures

def fibonacci(n):
    a, b = 0, 1
    i = 0
    while i < n:
        a, b = b, a+b
        i += 1
    return (n, a)

if __name__ == "__main__":
    indexes = [random.randint(0, 100) for i in range(10)]

    with concurrent.futures.ThreadPoolExecutor(max_workers = 3) as executor:
        print("Results returned via asynchronous map:")
        for result in executor.map(fibonacci, indexes):
            print(result)

        print("Results returned as Future objects as they complete:")
        futures = [executor.submit(fibonacci, index) for index in indexes]
        for future in concurrent.futures.as_completed(futures):
            print(future.result())
```

Exercise 2: What time is it, again ?

In Exercise 2 of Session 2, we designed client-server programs around System V-style message passing to request and serve current date and time. Both programs were single threaded. What this means for the server in particular, is that it cannot serve (read current date/time from system, create and send message) multiple clients at the same time. In this exercise, you will develop the previous server program so that it handles every client date/time request in a separate thread. You will work this exercise in two stages:

1. the server creates a separate thread every time it receives a client

date/time request, then

2. the server avoids thread creation by maintaining a thread pool to which requests are submitted.

The client program follows:

```
import os
import sys
import time
import sysv_ipc

key = 666

def user():
    answer = 3
    while answer not in [1, 2]:
        print("1. to get current date/time")
        print("2. to terminate time server")
        answer = int(input())
    return answer

try:
    mq = sysv_ipc.MessageQueue(key)
except ExistentialError:
    print("Cannot connect to message queue", key, ", terminating.")
    sys.exit(1)

t = user()

if t == 1:
    pid = os.getpid()
    m = str(pid).encode()
    mq.send(m)
    m, t = mq.receive(type = (pid + 3))
    dt = m.decode()
    print("Server response:", dt)

if t == 2:
    m = b""
    mq.send(m, type = 2)
```

You will notice that this version of the client program defines the type of the date/time response message as `pid` + 3, thus preventing multiple processes waiting for the server's response to receive each other's messages. For testing purposes, you can keep multiple clients alive by adding a small delay in the client code or in the worker thread of the server before receiving or sending the date/time response.

Note that the message passing facilities underlying the `sysv_ipc` module are thread-safe, thus allowing us to use them in multithreaded Python programs.