Chapter 13 - Concurrency

April 4, 2021

0.1 Overview

- concurrency is the art of making a computer do multiple things at once
- concurrency concepts are fairly straight fordward but the bugs that can occur are notoriously difficult to track down

0.2 Threads

- most often concurrency is created so that work can continue happening while the program is waiting for `I/O` to happen
- we can rely on python and the operating system to take care of the trickey switching part, while we create objects that appear to be running independently, by simultaneously
- these objects are called `threads`

[2]:
```python
from threading import Thread

class InputReader(Thread):
    def run(self):
        self.line_of_text = input()

print("Enter some text and press enter: ")
thread = InputReader()
thread.start()

count = result = 1
while thread.is_alive():
    result = count * count
    count += 1

print(f'calculated squares up to {count} * {count} = {result}')
print(f"while you typed {thread.line_of_text}")
```

```
Enter some text and press enter:
s
calculated squares up to 1991670 * 1991670 = 3966745405561
while you typed s
```

- example above runs two threads
- every program has a single thread, here we introduced another thread called `InputReader`
- to construct a thread, we must extend the `Thread` class and implement the `run` method

- any code executed by the `run` method happens in a seprate thread
- the new thread does not start untill we call `.start()` method on the object
- if we want to take out the concurrent call to see how it compares, we can call `thread.run()` in the place that we orginally called `thread.start()`

- interstingly, data we construct in one thread is accessible from other running threads
- remember, just because a method is on a `Thread` instance does not mean it is magically executed inside that thread
- there is a `thread.join()` method that says to *wait for the thread to complete before doing anything*

[ ]:
```
# code below ensures threads wont close untill all of them finish
for thread in threads:
    thread.join()
```

0.3 The Many Problems With Threads

0.3.1 Shared Memory

- the threads have accesss to all the programs memory and thus all its variables
- this can too easily cause inconsistencies in the program state
- the solution to this problem in threaded programs is to `synchronize` access to any code that reads or writes a shared variable
- synchronization works but you have to remember to try and use it and it intorduces bugs that are hard to track down

0.3.2 Global Interpreter Lock

- inorder to efficiently manage memeory, garbage collection and calls to machine code, Python has a utility called the `global interpreter lock (GIL)`
- this means threads are useless in python for the one thing they excel in other languages: `parallel processing`
- GLI prevent any two threads from doing work at the exact same time
- work means using the CPU, they can howerver use API calls or read data from the disk

0.3.3 Thread Overhead

- there is also a cost of maintaing each thread
- each thread takes up a certain amount of memory to record the state of that thread
- switching between threads also uses some CPU time
- this can be solved by structering our workload so that the threads can be reused to perfrom multiple jobs
- python provides a `ThreadPool` feature to handle this
- `ThreadPool` behaves simmilar to `ProcessPool`

0.4 Multiprocessing

- Multiprocessing libary was designed to mimic the `thread` API but it has evolved to provide more robust features

- multiprocessing is not useful when the processes spend a majority of their time waiting on `I/O` but it is the way to go for parallel computation
- multiprocessing module spins up new operating system processes to do the work
- this means there is an entire python interpreter running each process

[4]:
```
from multiprocessing import Process, cpu_count
import time
import os

class MuchCPU(Process):
    def run(self):
        print(os.getpid())
        for i in range(200000000):
            pass

if __name__ == "__main__":
    procs = [MuchCPU() for f in range(cpu_count())]
    t = time.time()
    for p in procs:
        p.start()
    for p in procs:
        p.join()
    print(f"work took {(time.time()) - t} seconds")
```

```
work took 0.2248551845550537 seconds
```

- each Process has a `pid` number which we can get using the `os.getpid()`
- also notice the `__name__` guardrail we put up so we dont acidentally import `MuchCPU`

- the difference between `Thead` and `Process` is that process is three times faster then thread

0.5 Multiprocessing Pools

for the following reasons, you should not have more processes than there are processors on the computer

- only `cpu_count()` process can run simultaneously
- each process consumes resources with a full copy of the `Python` interpreter
- communication between process is expensive
- crating process takes a non-zero amount of time

- you want to create at most `cpu_count()` processes when the program starts
- doing this on your own can be increadably difficult, but developers have done this for you in the form of `multiprocessing pools`

- `Pools` abstract away the overhead of figuring out what code is executing in the main process and which code is running in the subprocess
- the pool abstration restricts number of places in which code in different processes interacts, making it much easier to keep track of
- unlike threads, multiprocessing cannot directly access variables set up by other threads
- pools hide the process of passing data between processes

- using a pool looks like a function call; you pass data into a function, it is executed in another process or processes, and when the work is done, a value is returned
- behidn the scenes, there is alot of work being done such as an object getting pickled and the pased, etc
- pickling takes alot of time, and thus you should only pass a minumum amount of data betwen pols

[ ]:
```python
import random
from multiprocessing.pool import Pool

def prime_factor(value):
    factor = []
    for divisor in range(2, value -1):
        quotient, remainder = divmod(value, divisor)
        if not remainder:
            factors.extend(prime_factor(divisor))
            factor.extend(prime_factor(quotient))
            break
        else:
            factors = [value]
        return factor

if __name__ == "__main__":
    pool = Pool()

    to_factor = [random.randint(10000, 5000000) for i in range(20)]
    results = pool.map(prime_factor, to_factor)
    for value, factors in zip(to_factor, results):
        print(f'the factors of {value} are {factors}')
```

- we first construct a multiprocessing pool isntance
- by default this pool creates a seprate process for each of the CPU cores in the machine
- the `map` method accepts a function and an iterable
- the pool pickles each of the values in the iterable and passes it into an avaliable process, which executes the function on it
- when the process is finished doing its work, it pickles the resulting list of factors and passes it back to the pool
- then the pool has more work avaliable, it takes on the next job
- there are funcky `await/sync` stuff you can do with the pool
- you can also `close` a pool or `terminate` it

0.6 Queues

- if we need more control over communication between processes we can use a queue
- `Queue` data structures are useful for sending messages from one process into another processor
- any pickled object can be sent into a `Queue`
- this idea of a queue can acually become a distributed system

0.7 Problem with Multiprocessing

- primary drawback is that sharing data between processes is costly
- excessive pickling quickly dominates processing time
- multiprocessing works best when relatively small objects are passed between processes and a tremendous amount of work need to be done on each one
- the other major problem, is that it can be hard to tell which process a variable or method is being accessed

0.8 Futures

- there are also asynchronous ways of implementing concurrency
- `Futures` wrap either multiprocessing or threading depending on which concurrency we need
- whether you need `I/O` (threading) or `CPU` (multiprocessing)
- `Futures` are useful for *call and answer* type interactions
- meaning, processing can happen in another thread and at some point in the future, you can ask it for the results
- futures are just a wraper around multiprocessing pools and thread pools

[6]:
```python
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from os.path import sep as pathsep
from collections import deque

def find_files(path, query_string):
    subdirs = []
    for p in path.iterdir():
        full_path = str(p.absolute())
        if p.is_dir() and not p.is_symlink():
            subdirs.append(p)
        if query_string in full_path:
            print(full_path)
    return subdirs

query = '.py'
futures = deque()
basedir = Path(pathsep).absolute()

with ThreadPoolExecutor(max_workers=10) as executor:
    futures.append(
        executor.submit(find_files, basedir, query))
    while futures:
        future = futures.popleft()
        if future.exception():
            continue
        elif future.done():
            subdirs = future.result()
            for subdir in subdirs:
```

```
                futures.append(executor.submit(
                    find_files, subdir, query))
        else:
            futures.append(future)
```

```
C:\$Recycle.Bin\S-1-5-21-2588907532-1068130059-599424490-1001\$IO7ZBC4.py
C:\Users\Vicktree\Desktop\.test.py.un~
C:\Users\Vicktree\Desktop\.test.py~.un~
C:\Users\Vicktree\Desktop\test.py
C:\Users\Vicktree\Desktop\test.py~
C:\Users\Vicktree\Downloads\password_ongoing.py
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-6-d584e51053b2> in <module>
     24         future = futures.popleft()
---> 25         if future.exception():
     26             continue

c:
 ↪\users\vicktree\appdata\local\programs\python\python39\lib\concurrent\futures\_base.
 ↪py in exception(self, timeout)
    467
--> 468             self._condition.wait(timeout)
    469

c:\users\vicktree\appdata\local\programs\python\python39\lib\threading.py in␣
 ↪wait(self, timeout)
    311             if timeout is None:
--> 312                 waiter.acquire()
    313                 gotit = True

KeyboardInterrupt:

During handling of the above exception, another exception occurred:

KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-6-d584e51053b2> in <module>
     31                     find_files, subdir, query))
     32         else:
---> 33             futures.append(future)

c:
 ↪\users\vicktree\appdata\local\programs\python\python39\lib\concurrent\futures\_base.
 ↪py in __exit__(self, exc_type, exc_val, exc_tb)
    626
    627     def __exit__(self, exc_type, exc_val, exc_tb):
--> 628         self.shutdown(wait=True)
```

```
    629         return False
    630 

c:
↪\users\vicktree\appdata\local\programs\python\python39\lib\concurrent\futures\thread.
↪py in shutdown(self, wait, cancel_futures)
    227         if wait:
    228             for t in self._threads:
--> 229                 t.join()
    230     shutdown.__doc__ = _base.Executor.shutdown.__doc__

c:\users\vicktree\appdata\local\programs\python\python39\lib\threading.py in␣
↪join(self, timeout)
   1031 
   1032         if timeout is None:
-> 1033             self._wait_for_tstate_lock()
   1034         else:
   1035             # the behavior of a negative timeout isn't documented, but

c:\users\vicktree\appdata\local\programs\python\python39\lib\threading.py in␣
↪_wait_for_tstate_lock(self, block, timeout)
   1047         if lock is None:  # already determined that the C code is done
   1048             assert self._is_stopped
-> 1049         elif lock.acquire(block, timeout):
   1050             lock.release()
   1051             self._stop()

KeyboardInterrupt:
```

- the core of the program above is the event loop
- we can construct a `ThreadPoolExector` as a context manager so that is is automatically cleaned-up and closes its threads when it is done
- it requires a `max_workers` argument to indicate the number of threads running at the time
- `ProcessPoolExecutor` normally is constrained to the number of CPU's on the machine, but with threads, it can be much higher, depending how many are waiting on `I/O` at the time
- once the executor had been constructed, we submit a job to it using the root directory
- the `submit()` method immediately returns a `Future` object, which promises to give us a result
- the future is placed inside the queue
- the loop then repeatedly removes the first future from the queeu and inspects it
- if it is still running, it gets added back to the end of the queue
- if no errors occur, we can call `result()` to get the return value

0.9 AsyncIO

- combines the concepts of futures and event loops with the `coroutines`
- this was specifically designed for network `I/O`
- this library provides its own event loop
- the cost of this event loop is that when we run code in the `async` task on the event loop, the

code must return immediately
- blocking neither on I/O nor on long-running calculation
- AsyncIO solves this by creating a set of `coroutines` that use `async` and `await` syntax to return control to the event loop immediately when the code will block
- the keywork replaces the `yeild`, `feild from` and `send` syntax we used with `raw coroutines`

0.10 AsyncIO in Action

[8]:
```python
import asyncio
import random

async def random_sleep(counter):
    delay = random.random() * 5
    print(f"{counter} sleeps for {delay}")
    await asyncio.sleep(delay)
    print(f"{counter} awakens")

async def five_sleepers():
    print("creating five tasks")
    tasks = [asyncio.create_task(random_sleep(i)) for i in range(5)]
    print("Sleeping after starting five tasks")
    await asyncio.sleep(2)
    print("Waking and waiting for five tasks")
    await asyncio.gather(*tasks)

asyncio.get_event_loop().run_until_complete(five_sleepers())
print("Done five tasks")
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-8-efcc171606e9> in <module>
     16     await asyncio.gather(*tasks)
     17
---> 18 asyncio.get_event_loop().run_until_complete(five_sleepers())
     19 print("Done five tasks")

c:\users\vicktree\appdata\local\programs\python\python39\lib\asyncio\base_event
↪py in run_until_complete(self, future)
    616         """
    617         self._check_closed()
--> 618         self._check_running()
    619
    620         new_task = not futures.isfuture(future)

c:\users\vicktree\appdata\local\programs\python\python39\lib\asyncio\base_event
↪py in _check_running(self)
    576     def _check_running(self):
```

```
    577         if self.is_running():
--> 578             raise RuntimeError('This event loop is already running')
    579         if events._get_running_loop() is not None:
    580             raise RuntimeError(

RuntimeError: This event loop is already running
```

- A task in this context, is an obejct that `asyncio` knows how to schedule on the event loop

This includes: - coroutines defined with the `async` and `await` syntax - coroutines decorated with the `@asyncio.coroutine` and using the `yeild from` syntax (deprecated in favor of the `async` and `await` - `asyncio.Future` objects. These are almost identical to the `concurrent.futures` but for the use with `asyncio` - any awaitable obkect, that is, one with an `__await__` function

examining the `five_sleepers` future:

- the coroutine first constructs five instances of the `random_sleep` coroutine
- these are wrapped in a `asyncio.create_task` call, which adds the future to the loops taks queue so they can execute and start immediately, when control is returned to the loop
- the control is returned whenever we call `await`
- in this case, we call `await asyncio.sleep` to pause the execution of the coroutine for two seconds
- during the break, the event loop executes the taks that it has queue up, namely the five `random_sleep` tasks
- when the sleep call in the `five_sleepers` task wakes up, it calls `asyncio.gather`. this function accepts tasks as `varargs` and awaits each of them before returning
- each of the `random_sleep` coroutines prints a starting message, then sends control back tot he event loop for a specific amount of time using its own `await` call
- when the sleep has completed, the event loop passes contorl back tot he relevent `random_sleep` tasks, which prints its awakening message before returning
- when the event queue is empty, the `run_until_complete` call is able to terminate and the program ends

- `async` keyworkd acts as documentation notifiying the python interpreter that the coroutine contains the `await` calls

0.11 Reading an AsyncIO Future

- an AsyncIO coroutine executes each line in order untill it encounter an `await` statement at which point, it returns control to the event loop
- the event loop then executrs any other tasks that are ready to run, including the one that the orginal coroutien was waiting on
- whenever that child task completes, the event loop sends the result back into the coroutine so that it can pick up execution untill it encounters another await statement or returns

0.12 AsyncIO for Networks

- AsyncIO was specifically designed for use with network sockets
- AsyncIO networking resolves around the intimately linked concepts of transporting and protocols

- a protocal is a class that has specific methods that are called when relevant events happen
- Since DNS runs on top of `UDP` **(User Datagram Protocol)**, we build our protocol class as a subclass of `DatagramProtocol`
- the transport essentially represents a communication stream
- behind the scenes the transport has set up a task on the event loop that is listening for incomming `UDP connections`
- all we have to do, then is start the event loop running with the call to `loop.run_forver()` so that the task can process these packets
- when the packets arrive, they are processed on the protocol and everything just works fine

- there is alot of boilerplate in setting up a protocol class and the underlying transport
- AsyncIO provides an abstraction on top of these two key concepts, called **streams**
- we will see an example of streams in the TCP server

0.13 Using Executors to Wrap Blocking Code

- AsyncIO provides its own version of hte futures library to allow us to run code in a separate thread or process when there is not an appropriate non-blocking call to be made
- this allows us to combine threads and processes with the asynchronous model
- this is useful when an application has bursts of `I/O` bound and CPU-bound activity

0.14 Streams

- below is description of code I chose not to include due to its verbosity

- `create_server` hooks inot AsyncIOs strams instead of using the underlying transport/protocol code
- it allows us to pass in a normal coroutine, whihc receives reader and writer parameters
- these both represent streams of bytes that can be read from and written, like files or sockets
- secondly, because this is a TCP server, instead of **UDP**, there is some socket cleanup requied
- this cleanup is a blocking call, so we have to run `wait_closed` coroutine on the event loop

- Streams reading is a potentially blocking call so we have to call it with `await`
- writing doesent blokc; it just puts the data in queue, which AsyncIO sends out in the background

0.15 AsyncIO clients

- because it can handle many thousands of simultaneous connections, `AsyncIO` is a very common for implementing servers
- clients can be much simpler than servers, as they dont have to be set up to wait for incomming connections
- like most networking libraries, you just open a conenction, submit your requests and process any responses
- the main difference is that you need to use `await` any time you make a potientally blocking call

[9]:
```
import asyncio
import random
import json
```

```python
async def remote_sort():
    reader, writer = await asyncio.open_connection("127.0.0.1", 2015)
    print("Generating random list...")
    numbers = [random.randrange(10000) for r in range(10000)]
    data = json.dumps(numbers).encode()
    print("List Generated, Sending data")
    writer.write(len(data).to_bytes(8, "big"))
    writer.write(data)

    print("Waiting for data...")
    data = await reader.readexactly(len(data))
    print("Received data")
    sorted_values = json.loads(data.decode())
    print(sorted_values)
    print("\n")
    writer.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(remote_sort())
loop.close()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-9-ee1349fe2c32> in <module>
     20
     21 loop = asyncio.get_event_loop()
---> 22 loop.run_until_complete(remote_sort())
     23 loop.close()
     24

c:\users\vicktree\appdata\local\programs\python\python39\lib\asyncio\base_events.
↪py in run_until_complete(self, future)
    616         """
    617         self._check_closed()
--> 618         self._check_running()
    619
    620         new_task = not futures.isfuture(future)

c:\users\vicktree\appdata\local\programs\python\python39\lib\asyncio\base_events.
↪py in _check_running(self)
    576     def _check_running(self):
    577         if self.is_running():
--> 578             raise RuntimeError('This event loop is already running')
    579         if events._get_running_loop() is not None:
    580             raise RuntimeError(
```

```
RuntimeError: This event loop is already running
```