Multithreading and Multiprocessing

Threading

- typically, concurrency is created so that we can do some task while I/O is happening (e.g., a server can start processing a new request while waiting for data from a previous request to arrive)
- we can create objects that appear to be running independently, but simultaneously
- the job of threading is to enable an application to be responsive
- CPython, the default implementation of Python, has a Global Interpreter Lock (GIL), which prevents your application from doing two things at once, but rather, the CPU time is being rationed across your threads

Simple threading example

In [ ]:

```
from threading import Thread

class InputReader(Thread):
    """Thread example, extends Thread class"""

    def run(self):
        """
        Whatever is in the run method (or called from
        it) is executed in a separate thread
        """
        self.line_of_text = input('Enter some text: ')

input('Are you ready? When you hit return the thread will start.')
thread = InputReader() # create thread object
thread.start() # cf. thread.run() for no concurrency

count, result = 1, 1

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

print('calculated squares up to {0:,} * {0:,} = {1:,}'
      .format(count, result))
print('while you typed "{}"'.format(thread.line_of_text))
```

In [ ]:

```python
from threading import Thread
import json
from urllib.request import urlopen
import time

cities = ['Boulder', 'Atlanta', 'San Francisco',
          'Reno', 'Honolulu', 'Zurich', 'Dubai',
          'Dublin', 'Hyderabad', 'Rome']

class TempGetter(Thread):
    def __init__(self, city):
        """Initialize our thread

In the previous example, our class which extended
Thread did not need an __init__ method, because
there was no per-thread information to store. Which
means that the __init__ method from the superclass
(Thread) was called automatically. Here, because we
need to store per-thread information (the city), we
have to explicitly call the__init__ method of Thread.
        """
        super().__init__()
        self.city = city

    def run(self):
        url_template = (
            'http://api.openweathermap.org/data/2.5/'
            'weather?q={}&units=imperial'
                        '&&APPID=10d4440bbaa8581bb8da9bd1fbea5617')
        response = urlopen(url_template.format(self.city))
        data = json.loads(response.read().decode())
        self.temperature = data['main']['temp']

threads = [TempGetter(c) for c in cities] # creates 10 threads
start = time.time()

# start all 10 threads
for thread in threads:
    thread.start() # not run()

# wait for all 10 threads to complete
for thread in threads:
    thread.join()

for thread in threads:
    print(f"it is {thread.temperature:.0f}°F in {thread.city}")
print(f"Got {len(threads)} temps in {time.time() - start} seconds")
```

In [ ]:

```
%%bash
python3 getweather.py
```

Threading (cont'd)

- the main problem with threads is also their primary advantage–shared memory
    - all threads have access to all the memory
    - what if two threads access the same data?
- synchronization is the solution, but it's tricky
    - bugs due to incorrect synchronization can be very difficult to find due to ordering issues
- one solution is to force communication between threads to occur using a data structure that has built in locking, such as queue.Queue
- disadvantages could be outweighed by the fact that shared memory is FAST, except for the GIL

Lab: threads

- create a program which uses threads to simulate a database server
- your "database server" should simply be a thread which sleeps for a random interval (check out **`time.sleep()`** and **`random.randint()`** if you're not familiar with them)
- your main thread should get input from the user and respond to it (perhaps reversing the input given by the user) while the database thread is busy

Multiprocessing

- the Python multiprocessing library is designed for cases where CPU-bound jobs needs to happen in parallel and multiple cores are available
- advantages
    - separate memory space for each process
    - code is usually straightforward compared to threads
    - avoids GIL limitation
    - eliminates synchronization (assuming no shared memory)

A Simple Multiprocessing Example

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

class MuchCPU(Process):
    def run(self):
        print(os.getpid()) # get process ID
        for i in range(80_000_000):
            result = i * i

if __name__ == '__main__':
    print('Running...')
    procs = [MuchCPU() for f in range(2)]
    t = time.time()

    for p in procs:
        p.run()

    #for p in procs:
        #p.join()

    print('work took {} seconds'.format(time.time() - t))
```

Multiprocessing (cont'd)

- no reason for more processes than there are processors
  - only `cpu_count()` procs can run simultaneously
  - each proc consumes resources with a full copy of Python interpreter
  - interproc communication is expensive
  - creating procs takes a nonzero amount of time
- so we create at most `cpu_count()` processes when the program starts and have them execute tasks as needed
- easy to implement a basic series of communicating processes to do this, but it can be tricky to debug, test, and get correct–we don't have to do all this work because the Python developers have already done it for us–multiprocessing pools

Multiprocessing Pools

- pools abstract away the overhead of figuring out what code is running in main process and what code is running in subprocess
- abstraction restricts the number of places that code in different processes interact with each other, making it easier to keep track of
- pools also hide the passing of data between processes
    - using a pool looks much like a function call–you pass data into a function, it's executed in another process or processes, and when the work is complete, a value is returned
    - under the hood, a lot of work is being done to support this–objects in one process are being pickled (serialized) and passed into a pipe, then another process retrieves data from the pipe and unpickles it. Work is done in the subprocess and a result is produced. The result is pickled and passed into a pipe. Eventually, the original process unpickles it and returns it.

Multiprocessing Pool Example

In [ ]:

```
import random
import math
import os
from multiprocessing.pool import Pool

def prime_factor(value, level=0):
    factors = []
    if level:
        print('    ' * level, 'prime_factor(', value, ', ', level, ') ', os.getpid(), sep='')
        pass
    for divisor in range(2, value - 1):
        quotient, remainder = divmod(value, divisor)
        if not remainder:
            factors.extend(prime_factor(divisor, level + 1))
            factors.extend(prime_factor(quotient, level + 1))
            break
    else:
        factors = [value]
    return factors

if __name__ == '__main__': # distiguishes between running and importing
    pool = Pool()

    to_factor = [
        random.randint(40_000_000, 80_000_000)
                for _ in range(64)
    ]
    print(to_factor)
    results = pool.map(prime_factor, to_factor)
    for value, factors in zip(to_factor, results):
        print("The factors of {} are {}".format(value, factors))
    #print(results)
```

Lab: Multiprocessing Pool

- write a program to compute 1!...48! using a multiprocessing pool
- won't be much of a parallelism example, but it's easy to code
- use previous example as a template

Multithreading/Multiprocessing for Python 3

- Python 3.2 introduced the **`concurrent.futures`** module for multithreading via the ThreadPoolExecutor, or multiprocessing, using ProcessPoolExecutor
- it's been backported to Python 2.6+ and can be installed using **`pip install futures`**

In [ ]:

```
import concurrent.futures
import urllib.request
import time

URLS = ['https://www.japan.go.jp/',
        'http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://www.python.org',
        'http://www.wikipedia.org',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://www.apple.com',
        'http://blahblahblah.org']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

start = time.time()
# We use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # start the load operations and mark each future with its URL
    future_to_url = {
        executor.submit(load_url, url, 60): url for url in URLS }

    # asynchronously wait for threads to complete...
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print(f'{url} generated an exception: {exc}')
        else:
            print(f'{url} is {len(data)} bytes')

print(f'Completed in {time.time() - start:.2f} seconds')
```

In [ ]:

```
def serial():
    start = time.time()
    for url in URLS:
        try:
            data = load_url(url, 60)
        except Exception as exc:
            print(f'{url} generated an exception: {exc}')
        else:
            print(f'{url} is {len(data)} bytes')

    print(f'Completed in {time.time() - start:.2f} seconds')
```

In [ ]:

```
serial()
```

In [ ]:

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False
    sqrt_n = int(math.floor(math.sqrt(n)))

    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print(f'{number} is prime: {prime}')
```

In [ ]:

```
%timeit -r 5 main()
```

In [ ]:

```
def serial():
    for prime in PRIMES:
        print(f'{prime} is prime: {is_prime(prime)}')
```

In [ ]:

```
%timeit -r 5 serial()
```