11-Multiprocessing

September 16, 2022

1 Threads (Optional)

- The main idea is to more than one thing at a time (open to discussion!)
- Interests to programmers writing code for running on big iron, but also of interest for users of multicore PCs, e.g.:
  - A network server that communicates with several hundred clients all connected at once
  - A big number crunching job that spreads its work across multiple CPUs

[1]:
```
import time
import threading

class CountDownThread(threading.Thread):    # inherit from Thread

    def __init__(self, n, who_am_i=None):
        threading.Thread.__init__(self)
        self._n = n
        self._who_am_i = who_am_i

    def run(self):                          # redefine run()
        while self._n > 0:
            print("{}: {}".format(self._who_am_i if self._who_am_i else ".",␣
 ↪self._n))
            self._n -= 1
            time.sleep(.5)

class CountUpThread(threading.Thread):    # inherit from Thread

    def __init__(self, n, who_am_i=None):
        threading.Thread.__init__(self)
        self._n = n
        self._who_am_i = who_am_i

    def run(self):                          # redefine run()
        k = 0
        while k < self._n :
            print("{}: {}".format(self._who_am_i if self._who_am_i else ".", k))
            k += 1
            time.sleep(.5)
```

```
t1 = CountDownThread(5, 'A')       # executes until run() stops
t2 = CountDownThread(8, 'B')       # executes until run() stops
t3 = CountUpThread(5, 'C')

t1.start()
t2.start()
t3.start()

print('over and out! well... maybe not!')
```

```
A: 5
B: 8
C: 0
over and out! well… maybe not!
```

An alternative way of calling the threads

[2]:
```
import time
import threading

def countdown(n, name):
    while n > 0:
        print('{}:{}'.format(name, n))
        n -= 1
        time.sleep(.5)


# Creates a Thread object, but its run() method just calls the given function
threading.Thread(target=countdown, args=(5, 'A')).start()
threading.Thread(target=countdown, args=(5, 'B')).start()

print('over and out!')
```

```
A:5
B:5
over and out!
```

[3]:
```
import time
import threading

def countdown(n):
    while n > 0:
        print(n)
        n -= 1
        time.sleep(.5)

t1 = threading.Thread(target=countdown, args=(10,))
```

```
t2 = threading.Thread(target=countdown, args=(5,))

t1.start()
t2.start()

t1.join()       # Use t.join() to wait for a thread to exit
t2.join()

print('over and out!')
```

```
105

A: 4B: 7

C: 1
A:4
B:4
9
4
A: 3B: 6

C: 2
B:3
A:3
3
8
B: 5A: 2

C: 3
A:2
B:2
2
7
C: 4
B: 4
A: 1
A:1
B:1
6
1
B: 3
5
B: 2
4
B: 1
3
2
1
```

```
over and out!
```

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic

[4]:
```
import time
import threading

def my_print(s):
    time.sleep(.5)
    print(s + ' ', end='')

print('Why did the multithreaded chicken cross the road?')

for s in 'To get to the other side.'.split():
    threading.Thread(target=my_print, args=(s,)).start()
```

```
Why did the multithreaded chicken cross the road?
```

Accessing Shared Data

[5]:
```
import threading

M, k = 1000000, 0

def up():
    global k
    for i in range(M):
        k += 1

def down():
    global k
    for i in range(M):
        k -= 1

t1 = threading.Thread(target=up)
t2 = threading.Thread(target=down)

t1.start()
t2.start()

t1.join()
t2.join()

print(k)   # Oh! this (almost) never is equal to zero!?
```

```
-100848
```

1.1 Thread Synchronization Primitives: Mutex Locks

- Acquired locks must always be released
- However, it gets evil with exceptions and other non-linear forms of control-flow
- There are synchronization primitives, look for: Lock, RLock, Semaphore, Bounded-Semaphore, Event, and Condition

[6]:
```
import threading

lock = threading.Lock()

M = 1000000;
k = 0

def up():
    global k, lock
    for i in range(M):
        lock.acquire()
        k += 1
        lock.release()

def down():
    global k, lock
    for i in range(M):
        lock.acquire()
        k -= 1
        lock.release()

t1 = threading.Thread(target=up)
t2 = threading.Thread(target=down)

t1.start()
t2.start()

t1.join()
t2.join()
print(k)
```

```
side. other the get to To 0
```

Using the `with` command

[7]:
```
import threading

lock = threading.Lock()

M = 1000000;
k = 0
```

```
def up():
    global k, lock
    for i in range(M):
        with lock:
            k += 1

def down():
    global k, lock
    for i in range(M):
        with lock:
            k -= 1

t1 = threading.Thread(target=up)
t2 = threading.Thread(target=down)

t1.start()
t2.start()

t1.join()
t2.join()
print(k)
```

```
0
```

1.2 Queues

[8]:
```
import urllib.request, time, threading
from queue import Queue
from urls import url_list           # url_list - tuple of urls
```

[9]:
```
N = 20                              # number of urls to fetch
```

[ ]:
```
""" count the chars in a list of web pages"""  # SEQUENCIAL VERSION

chars_total = 0                     # chars counter

def get_page_size(url):             # sum the chars of each page
    global chars_total
    try:
        with urllib.request.urlopen(url) as response:
            chars_total += len(response.read())
    except:
        print('error reading {}'.format(url))

s = time.time()
for url in url_list[:N]:
    get_page_size(url)
```

```
print(chars_total)
print('took {} seconds'.format(time.time() - s))
```

```
error reading http://www.evillasforsale.com
error reading http://www.richardsonscharts.com
error reading http://www.electrichumanproject.com/
error reading http://www.besound.com/pushead/home.html
error reading http://www.lepoint.fr/
error reading http://www.samuraiblue.com/
error reading http://www.casbahmusic.com/
```

[ ]:
```
''' count the chars in a list of web pages '''  # THREADED VERSION

def queued_get_page_size():
    global chars_total
    while not q.empty():
        url = q.get()               # get a 'job' from the Queue
        try:
            with urllib.request.urlopen(url) as response:
                lock.acquire()      # just in case!
                chars_total += len(response.read())
                lock.release()
        except:
            print('error reading {}'.format(url))
        q.task_done()               # Signal that work is done


chars_total = 0                     # chars counter

s = time.time();

q = Queue();                        # define a queue
for url in url_list[:N]:            # 'put' jobs in the Queue
    q.put(url)

lock = threading.Lock()

workers = []
for _ in range(10):                 # create 10 (!) workers
    w = threading.Thread(target=queued_get_page_size)
    workers.append(w)
    w.start()

q.join()                            # Wait for all work to be done

for w in workers:
```

```
    w.join()

print(chars_total)
print('took {} seconds'.format(time.time() - s))
```

[ ]: