Python - Concurrent and Parallel Programming

threads, locks, processes and events

Introduction

“The free lunch is over” - Herb Sutter, 2005 http://www.gotw.ca/publications/concurrency-ddj.htm

- Moore’s law no longer applies
- Single Core performance stagnates due to physical transistor limits
- Multi Core CPUs in commodity hardware, even Glasses
- It isn’t easy
- But it isn’t going anywhere

Disclaimer

- Python2.7
- Single-machine only
- Not a seasoned python programmer, feel free to correct

Concurrency

A chimpanzee writing Shakespeare ***and*** drinking coffee

Parallelism

N chimpanzees writing Shakespeare ***and*** drinking coffee, where N > 1.

Processes

- Kernel processes
- Have at least one running thread
- Are scheduled by the OS, not by Python
- `multiprocessing` allows for IPC
- more on that later

Processes

```
import multiprocessing as mp
class myProcess(mp.Process):
…
def my_process(foo,bar):
…
p = mp.Process(target=my_process, args=(someFoo, someBar))
p.start() # start the process
p.join() # block the current process until p completes
```

Threads

- Live within a process
- Share some resources within process
- Are scheduled by the OS

Threads

```
import threading
class myThread(threading.Thread):
…
def my_thread(foo,bar):
…
t = threading.Thread(target=my_thread, args=(someFoo,
someBar))

t.start() # start the thread
t.join() # block the current thread until t completes
```

Green threads (gevent, greenlet lib)

- User space threads
- Run within process thread
- Scheduled by the python interpreter
- Live on the memory heap, in process space

Green threads (gevent, greenlet lib)

```
import gevent
class myGreenlet(Greenlet):
…
def my_greenlet(foo,bar):
…
g = Greenlet(my_greenlet, someFoo, someBar)
g.start()
g.join()
```

Other cool stuff

- asyncore library
  - socket wrapper for asynchronous eventloop
- eventlet library
  - epoll based (think NodeJS eventloop, GUI)
- asyncio (Py3k, "tulip")
- Twisted (like asyncore, but more of a framework)

Difficulties with multithreading

"threads it tricky Why use is to"

Difficulties with multithreading

```
stdout_lock = threading.Lock()

[thread 1]
with stdout_lock:
    print "Why it is tricky"

[thread 2]
with stdout_lock:
    print "to use threads"
```

Difficulties with multithreading

- Threads are nondeterministic
- Scheduling is done by the OS, not by the Python interpreter
- It is unpredictable when a thread runs, hence code needs to be thread safe
- Threads that use I/O block (wait) for (example filesystem) resources to become available
- We must use locks to synchronise multithreaded access to shared state
- Synchronising state across threads using locks is difficult and error-prone

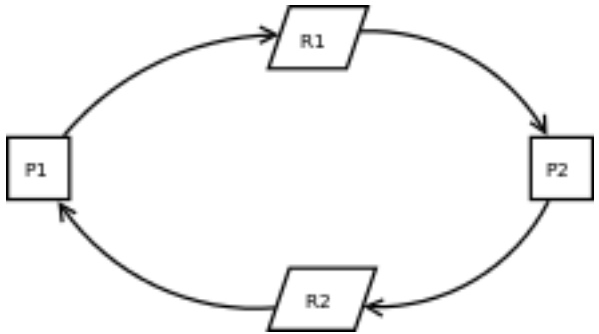
Thread locks & synchronisation

- threading.Lock : default lock
  - acquire() closes
  - release() opens
  - subsequent acquire calls on a closed lock block the acquiring thread - even the owning thread
- threading.Rlock : reëntrant lock with ownership counter
  - subsequent acquire calls *from the owning thread* do not block
    - useful when invoking thread-safe code on the same thread
- threading.Event : simple flag
  - set / clear toggles event to be flagged
  - wait() puts thread in wait state until the event is set
- threading.Condition : conditional lock with notification
  - acquire() blocks
  - notify() signals waiting thread that condition has changed state
  - wait() blocks thread until notify() is called on the lock by owning thread
- threading.Semaphore : lock with a counter
  - acquire() blocks when the amount of acquisitions == semaphore.max
  - release() decreases this

Locking pitfalls

- deadlock
  - thread A → waits on thread B → waits on thread A
  - A waits on C, C waits on B, B waits on A
    - usually circular dependencies or not releasing lock (thread might be waiting on C for a long time, never releasing resource A)

```
a.resource_a_lock.acquire() #locks
b.resource_b_lock.acquire() #locks
b.resource_a_lock.acquire() #blocks
b.resource_c_lock.acquire() #blocks
a.resource_b_lock.acquire() #blocks
```



Locking pitfalls

- race condition
  - thread A retrieves value of bank account f00b4r (EUR 100)
  - thread B gets scheduled and does the same (EUR 100)
  - thread B increases the value by 10% (EUR 110)
  - thread A gets scheduled and increases by 15% (EUR 126,5)

Within shared state multithreading, use a form of lock or event to keep these Get-Update-Store operations (transactions) atomic (thread-safe)

Use queues to share state between threads

Queue - synchronised task container (it uses simple locks internally)

```
import Queue

q = Queue.Queue()
q.put() # append a task
q.get() # retrieve a task, wait() if no task available
q.task_done() # decrease task counter
q.join() # block until all tasks are complete
```

B^5 - Buy Beer Brought By Bicycle

How it works

I have:

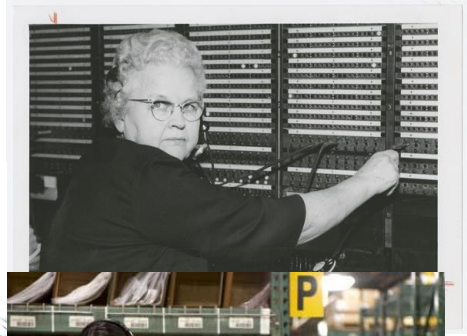
- a warehouse filled with tasty beer
- a cargo bike
- a cell phone where I can get orders

Pseudocode

```
while (1)
    continue unless order = pop_order_off_stack()
        collect_order(order)

    ready_bicycle()
        route = determine_route(order.address)
        bike_to_address(route)
        fullfill_transaction(order.price)
        return_home(route)
```

So it got a bit popular



- Hire three employees
  - three messengers (threads)
  - a dispatcher (scheduler)
  - Two order pickers (threads)
- Buy another cargo bike (pool)

1. The dispatcher places the orders in the order queue
2. The order pickers process the order queue and readily available cargo bikes with orders
3. The messengers wait for available ready orders and deliver them. The dispatcher guides them to their destination.
4. Upon delivery they notify the dispatcher that the order has been delivered. They park their bike back in the empty bike queue.

```
def simulate_business():
    n_messengers = 10, n_bikes = 15, n_order_pickers = 3, n_orders = 30

    q_orders = Queue.Queue()
    q_loaded_bikes = Queue.Queue(maxsize=n_bikes)
    q_empty_bikes = Queue.Queue(maxsize=n_bikes)

    [q_orders.put("order %d" % o) for o in xrange(n_orders)]
    [q_empty_bikes.put("bike %d" % b) for b in xrange(n_bikes)]

    for m in xrange(n_messengers):
        thread = threading.Thread(target=messenger_worker, args=(q_loaded_bikes, q_empty_bikes, q_orders, m))
        thread.daemon = True
        thread.start()

    for o in xrange(n_order_pickers):
        thread = threading.Thread(target=order_picker_worker, args=(q_orders, q_empty_bikes, q_loaded_bikes, o))
        thread.daemon = True
        thread.start()

    q_orders.join()
```

```
def order_picker_worker(order_queue, empty_queue, ready_queue, n):
    def process_order(order):
        while True:
            try:
                empty_bike = empty_queue.get(True, 1)
            except Queue.Empty:
                continue

            sleep(5)
            ready_queue.put(empty_bike, True, 1)

    while True: ←
        try:
            beer_order = order_queue.get()
        except Queue.Empty:
            sleep(5)
            continue

    process_order(beer_order)
```

```
def messenger_worker(full_queue, empty_queue, order_queue, n):
    while True:
        try:
            loaded_bike = full_queue.get(True, 1)
        except Queue.Empty:
            continue

        sleep(10)
        empty_queue.put(loaded_bike, True, 1)
        order_queue.task_done()
```

Conditional wait() over while True:

```
#order picker
ready_cv.acquire()
orders.put(ready_order)
ready_cv.notify()
ready_cv.release()

#messenger
ready_cv.acquire()
while True:
     order = orders.get()
     if order:
          break
     ready_cv.wait()
ready_cv.release()
do_stuff_with_order()
```

Greenlets instead of Threads

```
import gevent
from gevent.queue import Queue #similar to Queue, tuned for greenlets

gevent.spawn(messenger) #start as greenlet thread
…
gevent.spawn(order_picker)
…
gevent.sleep(0)
…
```

The GIL holds us back

- The more resources, the slower we get (the dispatcher spends most of her time switching between messengers)
- Messengers are mostly CPU bound, not I/O. The GIL does not release often
- We're only ever utilising one core (dispatcher); how can we have multiple messengers delivering crates in parallel?

Enter multiprocessing!

Difficulties within parallelism

- Amdahl's law

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Parallelising any problem by adding more processes to it is subject to the law of diminishing returns.

Amdahl's law

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

S = speedup

N = execution threads

P = parallelisation factor

S(320) = 1 / (1-0.8) + 0.8 / 320 = 5.0025

S(160) = 1 / (1-0.8) + 0.8 / 160 = 5.005

S(80) = 1 / (1-0.8) + 0.8 / 80 = 5.01

S(40) = 1 / (1-0.8) + 0.8 / 40 = 5.02

S(20) = 1 / (1-0.8) + 0.8 / 20 = 5.04

S(15) = 1 / (0.8) + 0.8 / 15 = 5.05

S(10) = 1 / (1-0.8) + 0.8 / 10 = 5.08

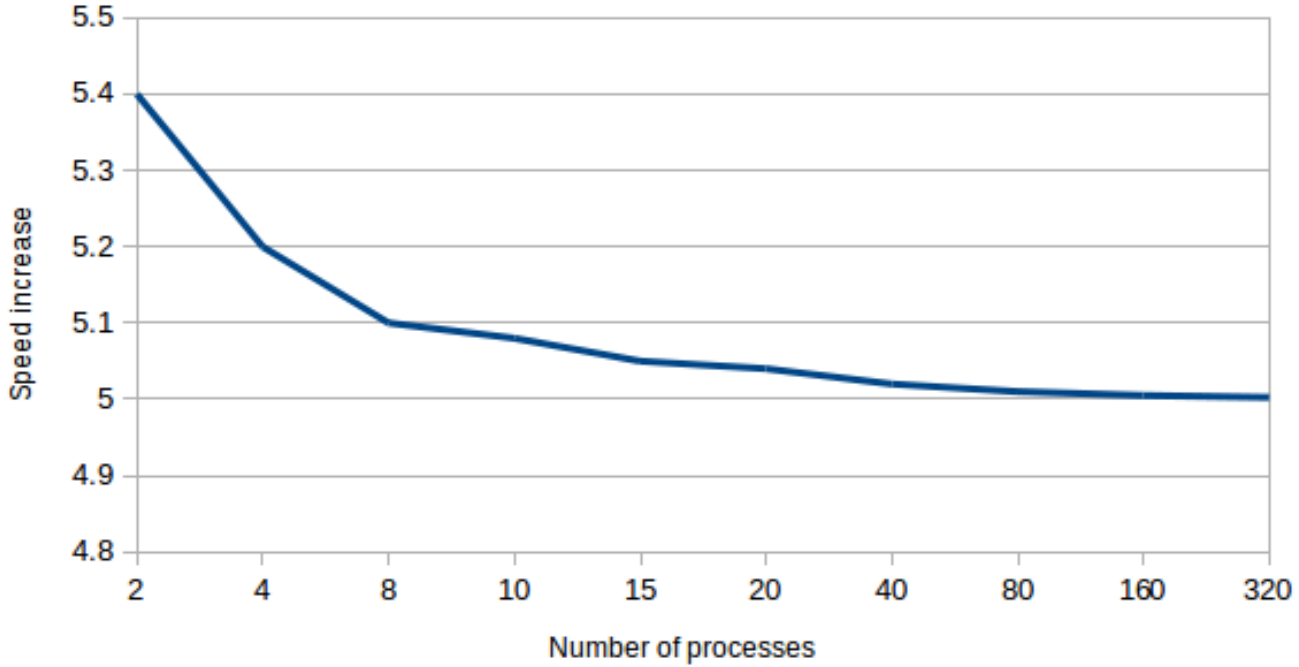
S(8) = 1 / (1-0.8) + 0.8 / 8 = 5.1

S(4) = 1 / (1-0.8) + 0.8 / 4 = 5.2

S(2) = 1 / (1-0.8) + 0.8/ 2 = 5.4

Amdahl's law

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$



Multiprocessing

- Spawn/fork processes similarly to thread
- The OS takes care of scheduling the process
- Each has their own Python Interpreter and GIL and can spawn threads
- Process can communicate using Queues, Pipes and Memory maps (shared memory)
    - Or using a different synchronisable medium, Cassandra, SQlite, you name it
- Process isolation means no locking
- Python data can be passed around using pickle, or the faster cpickle

Creating multiple processes

```
import multiprocessing as mp

manager = mp.Manager()
queue_1 = manager.Queue()
…
process = mp.Process(target=messenger_worker, args=(queue_1, queue_2…)
process.start()
…
```

Communication between processes

```
import multiprocessing as mp

manager = mp.Manager()
queue_1 = manager.Queue()
…
process = mp.Process(target=messenger_worker, args=(queue_1, queue_2…)
process.start()
…
class Messenger(mp.Process)
…
```

Communication between processes

```
import multiprocessing as mp

current_p, new_p = Pipe() # is actually mp.Connection instances
process = mp.Process(target=messenger_worker, args=(new_p)
process.start()
new_p.send(some_obj)
current_p.rcv() # some_obj
```

- mp.Connection pickles for you (so does Queue) - this is overhead

Wrapping up

- CPython is multithreading limited b/c GIL
  - GIL can only be lifted (in CPy) using C extensions
- Multithreading with shared state requires synchronisation
- Multiprocessing allows circumvention of GIL
  - multiprocessing library provides many helpful synchronisation and communication mechanisms
- Consider threading when using lots of I/O
- Consider async (event loop) when using lots of I/O
- Consider MP when using lots of CPU
- Concurrency is fun!

Thanks!

Questions?