Getting started with Concurrency

...using Multiprocessing and Threading

PyWorks, Atlanta 2008
Jesse Noller

Who am I?

- Just another guy.
- Wrote PEP 371- "Addition of the multiprocessing package"
- Did a lot of the integration, now the primary point of contact.
    - This means I talk a lot on the internet.
- Test Engineering is my focus - many of those are distributed and/or concurrent.
    - This stuff is not my full time job.

What is concurrency?

- **Simultaneous** execution
- Potentially **interacting** tasks
- Uses **multi-core** hardware
- Includes **Parallelism**.

What is a thread?

- Share the **memory** and **state** of the parent.
- Are "**light weight**"
- Each gets its **own stack**
- **Do not** use Inter-Process Communication or messaging.
- POSIX "Threads" - pthreads.

What are they good for?

- Adding throughput and reduce latency within most applications.
  - **Throughput**: adding threads allows you to process more information **faster**.
  - **Latency**: adding threads to make the application react **faster**, such as GUI actions.
- Algorithms which rely on shared data/state

What is a process?

- An **independent** process-of-control.
- Processes are "**share nothing**".
- Must use some form of **Inter-Process Communication** to communicate/coordinate.
- Processes are "**big**".

Uses for Processes.

- When you don't need to share lots of state and want a large amount of throughput.
- Shared-Nothing-or-Little is "**safer**" then shared-everything.
- Processes automatically run on multiple cores.
- Easier to turn into a **distributed** application.

The Difference

- Threads are implicitly "share everything" - this makes the programmer have to protect (lock) anything which will be shared between threads.

- Processes are "share nothing" - programmers must explicitly share any data/state - this means that the programmer is forced to **think** about what is being shared

- Explicit is better than Implicit

Python Threads

- Python has threads, they are real, OS/Kernel level POSIX (p) threads.
- When you use threading.Thread, you get a **pthread**.

2.6 changes

- camelCase method names are now foo_bar() style, e.g.: **active_count**, **current_thread**, **is_alive**, etc.
- Attributes of threads and processes have been turned into properties.
  - E.g.: daemon is now **Thread.daemon** = <bool>
- For threading: these changes are optional. The old methods still exist.

...

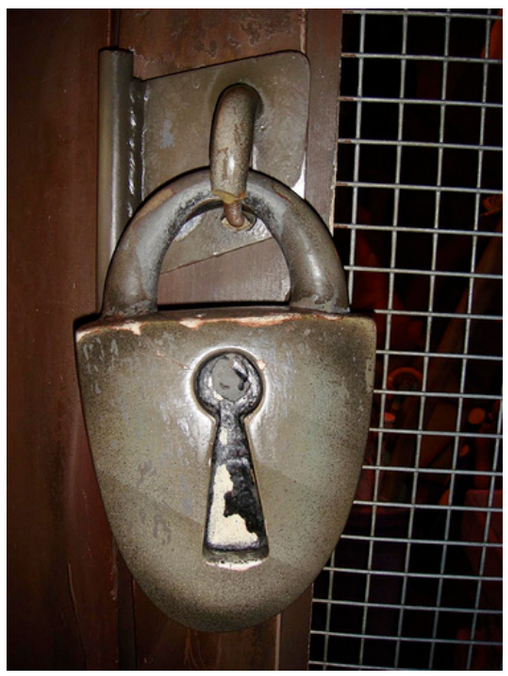
Python does not use “green threads”. It has real threads. OS Ones. Stop saying it doesn’t.

...

But: Python only allows a single thread to be executing within the interpreter at once. This restriction is enforced by the **GIL**.

The GIL

- GIL: "Global Interpreter Lock" - this is a lock which must be acquired for a thread to enter the interpreter's space.
- Only one thread may be executing within the Python interpreter at once.



Yeah but...

- No, it is not a bug.
- It is an implementation detail of CPython interpreter
- Interpreter maintenance is easier.
- Creation of new C extension modules easier.
- (mostly) sidestepped if the app is I/O (file, socket) bound.
- A threaded app which makes heavy use of sockets, won't see a a huge GIL penalty: it is still there though.
- Not going away right now.

The other guys

- Jython: No GIL, allows “free” threading by using the underlying Java threading system.
- IronPython: No GIL - uses the underlying CLR, threads all run concurrently.
- Stackless: Has a GIL. But has micro threads.
- PyPy: Has a GIL... *for now* (dun dun dun)

Enter Multiprocessing

What is multiprocessing?

- Follows the threading API closely but uses Processes and inter-process communication under the hood
- Also offers distributed-computing faculties as well.
- Allows the side-stepping of the GIL for CPU bound applications.
- Allows for data/memory sharing.
- CPython only.

Why include it?

- Covered in PEP 371, we wanted to have something which was fast and "freed" many users from the GIL restrictions.
- Wanted to add to the concurrency toolbox for Python as a whole.
    - It is not the "final" answer. Nor is it "feature complete"
- Oh, and it beats the threading module in speed.*

*** lies, damned lies and benchmarks**

How much faster?

- It depends on the problem.
- For example: number crunching, it's significantly faster then adding threads.
- Also faster in wide-finder/map-reduce situations
- Process creation **can** be sluggish: create the workers up front.

Example: Crunching Primes

- **Yes**, I picked something embarrassingly parallel.
- Sum all of the primes in a range of integers starting from 1,000,000 and going to 5,000,000.
- Run on an 8 Core Mac Pro with 8 GB of ram with Python 2.6, completely idle, except for iTunes.
- The single threaded version took so long I needed music.

```
# Single threaded version
import math

def isprime(n):
    """Returns True if n is prime and False otherwise"""
    if not isinstance(n, int):
        raise TypeError("argument passed to is_prime is not of 'int' type")
    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= max:
        if n % i == 0:
            return False
        i += 1
    return True

def sum_primes(n):
    """Calculates sum of all primes below given integer n"""
    return sum([x for x in xrange(2, n) if isprime(x)])

if __name__ == "__main__":
    for i in xrange(100000, 5000000, 100000):
        print sum_primes(i)
```

```
# Multi Threaded version
from threading import Thread
from Queue import Queue, Empty
...
def do_work(q):
    while True:
        try:
            x = q.get(block=False)
            print sum_primes(x)
        except Empty:
            break

if __name__ == "__main__":
    work_queue = Queue()
    for i in xrange(100000, 5000000, 100000):
        work_queue.put(i)

    threads = [Thread(target=do_work, args=(work_queue,)) for i in range(8)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

```
# Multiprocessing version
from multiprocessing import Process, Queue
from Queue import Empty
...

if __name__ == "__main__":
    work_queue = Queue()
    for i in xrange(100000, 5000000, 100000):
        work_queue.put(i)

    processes = [Process(target=do_work, args=(work_queue,)) for i in range(8)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

Results

- All results are in wall-clock time.
- Single Threaded: **41 minutes, 57 seconds**
- Multi Threaded (8 threads): **106 minutes, 29 seconds**
- MultiProcessing (8 Processes): **6 minutes, 22 seconds**
- This is a trivial example. More benchmarks/data were included in the PEP.

The catch.

- Objects that are shared between processes must be serialize-able (pickle).
  - 40921 object/sec versus 24989 objects a second.
- Processes are “heavy-weight”.
- Processes can be slow to start. (on windows)
- Supported on Linux, Solaris, Windows, OS/X - but not *BSD, and possibly others.
- If you are creating and destroying lots of threads - processes are a significant impact.

API Time

"Now! *That* should clear up a few things around here!"

It starts with a Process

- Exactly like threading:
  - **Thread(target=func, args=(args,)).start()**
  - **Process(target=func, args=(args,)).start()**
- You can subclass multiprocessing.Process exactly as you would with threading.Thread.

```
from threading import Thread
threads = [Thread(target=do_work, args=(q,)) for i in range(8)]

from multiprocessing import Process
processes = [Process(target=do_work, args=(q,)) for i in range(8)]
```

```python
# Multiprocessing version
from multiprocessing import Process

class MyProcess(Process):
    def __init__(self):
        Process.__init__(self)
    def run(self):
        a, b = 0, 1
        for i in range(100000):
            a, b = b, a + b

if __name__ == "__main__":
    p = MyProcess()
    p.start()
    print p.pid
    p.join()
    print p.exitcode
```

```python
# Threading version
from threading import Thread

class MyThread(Thread):
    def __init__(self):
        Thread.__init__(self)
    def run(self):
        a, b = 0, 1
        for i in range(100000):
            a, b = b, a + b

if __name__ == "__main__":
    t = MyThread()
    t.start()
    t.join()
```

Queues

- multiprocessing includes 2 Queue implementations - Queue and JoinableQueue.
- Queue is modeled after Queue.Queue but uses pipes underneath to transmit the data.
- JoinableQueue is the same as Queue except it adds a .join() method and .task_done() ala Queue.Queue in python 2.5.

Queue.warning

- The first is that if you call .terminate or kill a process which is currently accessing a queue: *that queue may become corrupted*.
- The second is that any Queue that a Process has put data on must be drained prior to joining the processes which have put data there: otherwise, you'll get a deadlock.
    - Avoid this by calling Queue**.cancel_join_thread**() in the child process.
    - Or just eat everything on the results pipe before calling join (e.g. work_queue, results_queue).

Pipes and Locks

- Multiprocessing supports communication primitives.
  - multiprocessing.Pipe(), which returns a pair of Connection objects which represent the ends of the pipe.
    - The data sent on the connection must be pickle-able.
- Multiprocessing has clones of all of the threading modules lock/**RLock**, **Event**, **Condition** and semaphore objects.
  - Most of these support timeout arguments, too!

Shared Memory

- Multiprocessing has a sharedctypes module.
- This module allows you to create a ctypes object in shared memory and share it with other processes.
- The sharedctypes module offers some safety through the use/ allocation of locks which prevent simultaneous accessing/ modification of the shared objects.

```
from multiprocessing import Process
from multiprocessing.sharedctypes import Value
from ctypes import c_int

def modify(x):
    x.value += 1

x = Value(ctypes.c_int, 7)
p = Process(target=modify, args=(x))
p.start()
p.join()

print x.value
```

Pools

- One of the big "ugh" moments using threading is when you have a simple problem you simply want to pass to a pool of workers to hammer out.
- **Fact**: There's more thread pool implementations out there then stray cats in my neighborhood.

Process Pools!

- Multiprocessing has the Pool object. This supports the up-front creation of a number of processes and a number of methods of passing work to the workers.
- Pool.apply() - this is a clone of builtin apply() function.
    - Pool.apply_async() - which can call a callback for you when the result is available.
- Pool.map() - again, a parallel clone of the built in function.
    - Pool.map_async() method, which can also get a callback to ring up when the results are done.
- **Fact**: Functional programming people love this!

Pools raise insurance rates!

```
from multiprocessing import Pool

def f(x):
    return x*x
```

```
if __name__ == '__main__':
    pool = Pool(processes=2)
    result = pool.apply_async(f, (10,))
    print result.get()
```

The output is: 100, note that the result returned is a **AsyncResult** type.

Managers

- Managers are a network and process-based way of sharing data between processes (and machines).
- The primary manager type is the **BaseManager** - this is the basic Manager object, and can easily be subclassed to share data remotely
- A **proxy object** is the type returned when accessing a shared object - this is a reference to the actual object being exported by the manager.

Sharing a queue (server)

```
# Manager Server
from Queue import Empty
from multiprocessing import managers, Queue

_queue = Queue()
def get_queue():
    return _queue

class QueueManager(managers.BaseManager): pass

QueueManager.register('get_queue', callable=get_queue)

m = QueueManager(address=('127.0.0.1', 8081), authkey="lol")
_queue.put('What's up remote process')

s = m.get_server()
s.serve_forever()
```

Sharing a queue (client)

```
# Manager Client
from multiprocessing import managers

class QueueManager(managers.BaseManager): pass

QueueManager.register('get_queue')

m = QueueManager(address=('127.0.0.1', 8081), authkey="lol")
m.connect()
remote_queue = m.get_queue()
print remote_queue.get()
```

Gotchas

- Processes which feed into a multiprocessing.Queue will block waiting for all the objects it put there to be removed.
- Data must be pickle-able: this means some objects (for instance, GUI ones) can not be shared.
- Arguments to Proxies (managers) methods must be pickle-able as well.
- While it supports locking/semaphores: using those means you're sharing something you may not need to be sharing.

In Closing

- Multiple processes are not mutually exclusive with using Threads
- Multiprocessing offers a simple and known API
    - This lowers the barrier of entry significantly
    - Side steps the GIL
- In addition to “just processes” multiprocessing offers the start of grid-computing utilities

Questions?