Introduction to Multiprocessing

... In Python.

Jesse Noller

PyCon 2009 Chicago

30,000 Foot View

- Introduction
- Multiprocessing vs. Threads
- How it works
- Usage/API
- Gotchas
- Summary
- Questions

Hello there!

- Who am I?
    - Authored PEP 371, now maintainer
- Why am I doing this?
- Email: jnoller@gmail.com
- Blog - http://www.jessenoller.com
- Pycon - http://jessenoller.com/category/pycon-2009/

What is multiprocessing?

A package which mimics the **threading API** but uses **processes** and interprocess communication underneath to provide true **parallelism** in a **simple** way.

Threads

- **Share memory** and state with each other and the parent
- When discussed, most people are talking about **posix-threads**
- Threads are **cheap** (quick to spawn, low memory)

Processes

- Processes are **share-nothing**
- **Heavyweight** (each is it’s own app instance)
- Must use **interprocess communication** to share information

Threads in CPython

- Threads in Python are **real** Posix threads
- Have all the attributes of pthreads, low footprint, quick to spawn
- Hampered by the global interpreter lock

The Global Interpreter Lock

- The GIL **prevents** true parallelism
- C Extensions side-step this effect
- It's not entirely evil
  - Makes the interpreter **easier to maintain**
  - Makes C extensions **easier to write**

Why multiprocessing?

It has an API!

- Mimics the threading and queue APIs
- Additional APIs for network-data sharing
- Communication Pipes
- Pool (map, apply, imap) functions
- Abstracts the fork/IPC mechanism **away** from users.
- Acts as a "drop-in" replacement for threading
- Side-steps the GIL

Faster than threads!*

- Speed depends on the problem being addressed
  - Pure math? Faster.
  - Map-Reduce? Faster.
  - Anywhere where you have pure Python code that can run in parallel and not contend over a single resource

* lies, damned lies and benchmarks

How much faster?

- Simple problem - calculate the sum of all primes between 1,000,000 and 5,000,000
- Executed on an 8 core Mac Pro, 8GB of RAM, completely idle except for this test
- **Yes**, this is contrived, and shared-nothing

the functions

```python
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)])
```

single threaded

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

multithreaded

```python
# 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

```
# 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()
```

Changed import

```python
# 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()
```

Process() has same signature as Thread

“Results”

wtf?

- Single Threaded: **41 minutes**
- Multithreaded (8 threads): **106 minutes**
- Multiprocessing (8 procs): **6 minutes**

Why (not) multiprocessing?

Premature Optimization

There's always a catch

- Processes are **expensive** - in both memory footprint and time-to-spawn
- IPC requires that object be able to be serialized and sent back and forth (serialize, de-serialize)
  - **IPC is not cheap**
  - Some objects can **not** be shared
    - For example, som GUI objects

Operating systems screw you

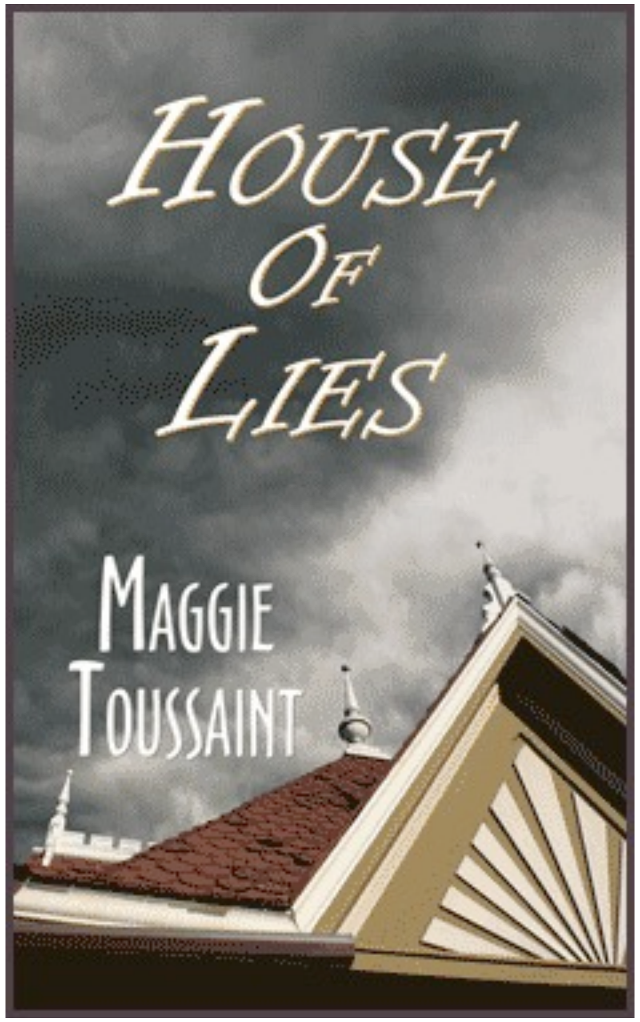
- Sketchy support on *BSD for example
- Requires named semaphore support as well as other OS-level libraries (fork!)
- Spawning a process on windows is slow(est)
- Bizarre bugs show up on different platforms

Threads aren’t evil

- Threads in general are **useful**
- Threads in Python are **functional**
- Threads work well for problems which **need to share**
- Threads are **not** impossible to “get right” - but easy to get wrong.
- Avoid **unconstrained** shared data!

Remember that benchmark?

- Let's see another one, this time - thread/Queue vs. processing passing objects via a queue
- Write 20,000 things to the queue
  - dict.fromkeys(range(10), str(i) * 100)
- Everything is serialized for mp



“Results”

non_threaded (1 iters)	0.064284 seconds
threaded (1 threads)	0.793872 seconds
processes (1 procs)	1.094208 seconds
non_threaded (2 iters)	0.134313 seconds
threaded (2 threads)	**1.339949** seconds
processes (2 procs)	**1.544650** seconds

* fyi, unladen-swallow speeds this up, ~7%

How does it work?

if sys.platform != 'win32':

Calls os.fork() in _bootstrap, passing it a process object

else:

creates pipe to communicate to child

calls custom _subprocess/Popen function

pickles the current process (+state) and passes it to the child

Usage

Simple: Drop it in

- Using Christopher Arndt's wonderful module
  - http://www.chrisarndt.de/projects/threadpool/

```
54a55
> import multiprocessing
113c114
< class WorkerThread(threading.Thread):
---
> class WorkerThread(multiprocessing.Process):
129,130c130,131
<         threading.Thread.__init__(self, **kwds)
<         self.setDaemon(1)
---
>         multiprocessing.Process.__init__(self, **kwds)
>         self.daemon = True
250,251c251,252
<         self._requests_queue = Queue.Queue(q_size)
<         self._results_queue = Queue.Queue(resq_size)
---
>         self._requests_queue = multiprocessing.Queue(q_size)
>         self._results_queue = multiprocessing.Queue(resq_size)
```

Bits of the API

- **Queues**
  - 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

Bits of the API

- **Communication**
  - 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
- **Locks**
  - Multiprocessing has clones of all of the threading modules lock/**RLock**, **Event**, **Condition** and semaphore objects

Bits of the API

- **Pool Objects**
  - 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

Bits of the API

Managers

- Managers are a network and process-based way of sharing data between processes (and machines)
- The primary manager type is the BaseManager
- 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

Gotchas

I'd like not to have such a newbie trap lying around. -GvR

“Why it not go faster?”

- Adding parallelism to an application locked on a single resource means you just added contention
  - $N processes reading files from disk
  - $N processes accessing the same locked resource
- Are you spawning those processes up front?
- Sharing lots of data means **lots** of serialization cost
- The same rules for threaded apps apply here

Avoid

- Nested functions, stick within the limitations of pickle for all objects being shared/transmitted
- Avoid passing lots of state in the constructor, stick to queues and pipes
- Calling .terminate() - you will corrupt something
- Globals bad; pass objects to be shared to the child

(this way they don't get gc'ed)

Do

- Spawn the processes as far in advance as possible
- Be mindful that having more processes than processors doesn't make sense
- Use *cancel_join_thread* or drain the queue that processes write to prior to join
- Use pipes and queues to share data between processes

Summary

- Multiprocessing is complimentary to threads
- Multiprocessing has a simple API
  - This lowers the barrier **significantly**
- Has the start of distributed/grid system tools
- **Do realize**: not on all platforms, and does have innate limitations

Questions?


Fin.