Parallel & Concurrent Processing

Sometimes we have a lot of things to process and not a lot of time to do it. We turn to multithreading and multiprocessing to alleviate this.

Topics:

- Parallelism in Python
- Mulithreading
- Muliprocessing
- The Python Global Interpreter Lock (GIL)

Parallelism in Python

Two methods in Python, processes and threads.

Multiple **threads** are great for:

- I/O bound tasks
- Shared memory for execution
- ie: web scraping, file copying

Multiple **processes** are great for:

- CPU bound-tasks
- Separate memory for execution
- ie: numeric operations, complex algorithms, path finding

Multithreading

In [ ]:
```python
# credit: http://chriskiehl.com/article/parallelism-in-one-line/
import urllib2
from multiprocessing.dummy import Pool as ThreadPool

urls = [
  'http://www.python.org',
  'http://www.python.org/about/',
  'http://www.onlamp.com/pub/a/python/2003/04/17/metaclasses.html',
  'http://planet.python.org/',
  'https://wiki.python.org/moin/LocalUserGroups',
  'http://www.python.org/psf/',
  'http://docs.python.org/devguide/',
  'http://www.python.org/community/awards/'
]

# Make the Pool of workers
nthreads = 4
pool = ThreadPool(nthreads)

# Open the urls in their own threads and return the results
results = pool.map(urllib2.urlopen, urls)

# close the pool and wait for the work to finish
pool.close()
pool.join()
```

Multiprocessing

We'll show an example now of multiprocessing, this time using locks so we can see how to control access to resources across processes.

```
In [8]: # adapted from: https://www.blog.pythonlibrary.org/2016/08/02/python-201
        -a-multiprocessing-tutorial/
        import multiprocessing
        import random
        import time

        def worker(i, lock, stream):
            # sleep for random fractional part of second
            time.sleep(random.random())

            lock.acquire()
            try:
                stream.write('Lock acquired for (%d) directly\n' % i)
            finally:
                lock.release()

        lock = multiprocessing.Lock()

        # start all the processes
        processes = []
        for i in range(10):
            w = multiprocessing.Process(target=worker, args=(i, lock,
        sys.stdout))
            w.start()
            processes.append(w)

        # wait for all to finish
        for p in processes:
            p.join()
```

```
Lock acquired for (1) directly
Lock acquired for (5) directly
Lock acquired for (7) directly
Lock acquired for (0) directly
Lock acquired for (4) directly
Lock acquired for (8) directly
Lock acquired for (6) directly
Lock acquired for (3) directly
Lock acquired for (2) directly
Lock acquired for (9) directly
```

Communication Using a Queue

Sometimes we'd like to dynamically add and remove tasks in a sort of producer/consumer fashion. To do this, we need some shared state that is safe to access from multiple processes and can be shared.

In [18]:
```
# adapted from: https://www.blog.pythonlibrary.org/2016/08/02/python-201
-a-multiprocessing-tutorial/
from multiprocessing import Process, Queue

sentinel = -1  # our "poison pill"

def creator(data, q):
    """
    Creates data to be consumed and waits for the consumer
    to finish processing.
    """
    print('Creating data and putting it on the queue')
    for item in data:
        q.put(item)


def my_consumer(q):
    """
    Consumes some data and performs some work.
    """
    while True:
        data = q.get()  # this is blocking!
        processed = data ** 2
        print "Processed, answer: %d" % processed
        if data is sentinel:
            # our "poison pill"
            break

# create our queue and data
q = Queue()
data = [5, 10, 13, 4, 3, -1]

# start consumer and
creator_process = Process(target=creator, args=(data, q))
consumer_process = Process(target=my_consumer, args=(q,))
creator_process.start()
consumer_process.start()

# wait for everything to finish
q.close()
q.join_thread()
creator_process.join()
consumer_process.join()
```

```
Creating data and putting it on the queue
Processed, answer: 25
Processed, answer: 100
Processed, answer: 169
Processed, answer: 16
Processed, answer: 9
Processed, answer: 1
```

The (In)famous Global Interpreter Lock (GIL)

Python's infamous GIL is a mutex that prevents multiple native threads from executing Python bytecodes at once.

OMG, so is Python really single-threaded? No! Don't worry.

The long running part of most tasks including I/O bound, or matrix crunching, etc happen **outside** the GIL and therefore aren't held up by it.

Certain situations however where bytecode constantly needs to be run do cause significant slowdowns and are the bain of anyone trying to make things highly concurrent in Python.

We can avoid the GIL by using multiprocessing, GIL is only for threads since they share the same memory.

Locks, Semaphores, Conditions

Python has a rich set of tools to allow for parallel and concurrent processing.

- Locks (protect single resource)
- Semaphores (quantifiable access limitations, for example to a database)
- Conditions (Python's way to synchronize workflow so some parts go sequential and others concurrently)

We won't go into depth with these, but there's a few (many) good hours of reading about it here (https://docs.python.org/2/library/multiprocessing.html).

Lab: Coding Excercises

Fill in the method definitions in the file `excercises/parallel.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_parallel.py::ParallelExcercises::<function_name>  # tes
t single function
$ py.test tests/test_parallel.py::ParallelExcercises                   # tes
t all at once
```

Wrap-Up

Topics:

- Parallelism in Python
- Mulithreading vs. Muliprocessing
- The Global Interpreter Lock (GIL)
- Locks, Semaphores