PEP: 371 - Birth of the Multiprocessing in Python

Edward Zhu, Steven Tang, John Zavidniak

December 10, 2014

Python before MultiProcessing

- Python interpretor required thread safe mechanism
- Global Interpreter Lock.
- Lock only enabled single statement executions (single threading)

Python's MultiProcessing Package

- MultiProcessing introduced in version 2.6.
- Able to run multiple tasks w/ either local or remote concurrency, side-steping the global interpreter lock.

MultiThreading vs. MultiProcessing P1

- Multithreading - exists one process, and each thread exists in this process as well.
- Bad: Could lead to race conditions, synchronization errors.

MultiThreading vs. MultiProcessing P2

- MultiProcessing - each process is separate is completely independent.
- No memory shared, so no race conditions.

Basics of Multprocessing

- Usage or two or more central processing units within a single system's systems
- Parallel execution of multiple processors.

Why Use Multiprocessing

- Improves performance.
- Running multiple tasks at the same time.

Benchmark Results

- Results show an increase of up to three times speed in programs
- Speed really shows as processes/iterations increase.

Basic Classes in Package

- Comes with many built in options for creating a parallel application.
- Process, Queue, Pool Classes.

Process Class

- In Multiprocessing, processes are spawned by created a Process Object.
- Once the process object has been instantiated, a start method must be called.
- The Start method is called once per process object and start's the process's activity.

Example - Process Class Usage

Edward Zhu, Steven Tang, John Zavidniak

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Queue Class

- Queues are thread and process safe.
- Essentially like a queue data structure - FIFO implementation (first tasks added are first retrieved).
- Allows for communication between processes.

Example - Queue Class Usage

```python
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

Pool Class

- Manages parallel-processing tasks.
- Instantiated with certain # of tasks, all operations limited to that #.

Example - Pool Class Usage

```python
def cube(x):
    return x**3
pool = mp.Pool(processes=6)
results = pool.map(cube, range(1,7))
print(results)
```

```
[1, 8, 27, 64, 125, 216]
```

Drawbacks

- Sharing data is tedious.
- Must copy every message to send; retrieval is costly.

Drawback Work Arounds 1

- Workaround: shared memory and server processing.
- Shared memory: Uses Value and Array Classes.

Drawback Work Arounds 2

- Server Processes.
- Uses Manager Class, hold and manipulates objects using proxies.
- Slower than shared memory.

Edward Zhu, Steven Tang, John Zavidniak

Open Issues

- No remote connection capabilities.
- Must enable the remote security mechanisms.
- Some minor APIs have not been implemented.

OS Restictions

- Windows lacks os.fork().
- Users must protect “entry point” of the program processes.

```
__name__ == ‘__main__’
```

- Above code allows safe importing of modules and inner function.

Conclusion

- MultiProcessing is a crucial addition to the standard library.
- Dramatically increased speed in programs.
- Spawned many other innovations for Python.