Core Development > **PEP Index** > PEP 371 -- Addition of the multiprocessing package to the standard library

PEP:	371
Title:	Addition of the multiprocessing package to the standard library
Version:	70469
Last-Modified:	**2009-03-19 04:35:27 +0100 (Thu, 19 Mar 2009)**
Author:	Jesse Noller <jnoller at gmail.com>, Richard Oudkerk <r.m.oudkerk at googlemail.com>
Status:	Final
Type:	Standards Track
Content-Type:	**text/plain**
Created:	06-May-2008
Python-Version:	2.6 / 3.0
Post-History:	

Abstract


```
This PEP proposes the inclusion of the pyProcessing [1] package
into the Python standard library, renamed to "multiprocessing".

The processing package mimics the standard library threading
module functionality to provide a process-based approach to
threaded programming allowing end-users to dispatch multiple
tasks that effectively side-step the global interpreter lock.

The package also provides server and client functionality
(processing.Manager) to provide remote sharing and management of
objects and tasks so that applications may not only leverage
multiple cores on the local machine, but also distribute objects
and tasks across a cluster of networked machines.

While the distributed capabilities of the package are beneficial,
the primary focus of this PEP is the core threading-like API and
capabilities of the package.
```


Rationale

```
The current CPython interpreter implements the Global Interpreter
Lock (GIL) and barring work in Python 3000 or other versions
currently planned [2], the GIL will remain as-is within the
CPython interpreter for the foreseeable future.  While the GIL
itself enables clean and easy to maintain C code for the
interpreter and extensions base, it is frequently an issue for
those Python programmers who are leveraging multi-core machines.

The GIL itself prevents more than a single thread from running
within the interpreter at any given point in time, effectively
removing Python's ability to take advantage of multi-processor
systems.
```

The pyprocessing package offers a method to side-step the GIL allowing applications within CPython to take advantage of multi-core architectures without asking users to completely change their programming paradigm (i.e.: dropping threaded programming for another "concurrent" approach - Twisted, Actors, etc).

The Processing package offers CPython a "known API" which mirrors albeit in a **PEP 8** compliant manner, that of the threading API, with known semantics and easy scalability.

In the future, the package might not be as relevant should the CPython interpreter enable "true" threading, however for some applications, forking an OS process may sometimes be more desirable than using lightweight threads, especially on those platforms where process creation is fast and optimized.

For example, a simple threaded application:

```
    from threading import Thread as worker

    def afunc(number):
        print number * 3

    t = worker(target=afunc, args=(4,))
    t.start()
    t.join()
```

The pyprocessing package mirrored the API so well, that with a simple change of the import to:

```
    from processing import process as worker
```

The code would now execute through the processing.process class. Obviously, with the renaming of the API to **PEP 8** compliance there would be additional renaming which would need to occur within user applications, however minor.

This type of compatibility means that, with a minor (in most cases) change in code, users' applications will be able to leverage all cores and processors on a given machine for parallel execution. In many cases the pyprocessing package is even faster than the normal threading approach for I/O bound programs. This of course, takes into account that the pyprocessing package is in optimized C code, while the threading module is not.

The "Distributed" Problem

In the discussion on Python-Dev about the inclusion of this package [3] there was confusion about the intentions this PEP with an attempt to solve the "Distributed" problem - frequently comparing the functionality of this package with other solutions like MPI-based communication [4], CORBA, or other distributed object approaches [5].

The "distributed" problem is large and varied. Each programmer working within this domain has either very strong opinions about their favorite module/method or a highly customized problem for which no existing solution works.

The acceptance of this package does not preclude or recommend that programmers working on the "distributed" problem not examine other solutions for their problem domain. The intent of including this package is to provide entry-level capabilities for local concurrency and the basic support to spread that concurrency across a network of machines - although the two are not tightly coupled, the pyprocessing package could in fact, be used in conjunction with any of the other solutions including MPI/etc.

If necessary - it is possible to completely decouple the local concurrency abilities of the package from the network-capable/shared aspects of the package. Without serious concerns or cause however, the author of this PEP does not recommend that approach.

Performance Comparison

As we all know - there are "lies, damned lies, and benchmarks". These speed comparisons, while aimed at showcasing the performance of the pyprocessing package, are by no means comprehensive or applicable to all possible use cases or environments. Especially for those platforms with sluggish process forking timing.

All benchmarks were run using the following:

* 4 Core Intel Xeon CPU @ 3.00GHz
* 16 GB of RAM
* Python 2.5.2 compiled on Gentoo Linux (kernel 2.6.18.6)
* pyProcessing 0.52

All of the code for this can be downloaded from:
http://jessenoller.com/code/bench-src.tgz

The basic method of execution for these benchmarks is in the run_benchmarks.py script, which is simply a wrapper to execute a target function through a single threaded (linear), multi-threaded (via threading), and multi-process (via pyprocessing) function for a static number of iterations with increasing numbers of execution loops and/or threads.

The run_benchmarks.py script executes each function 100 times, picking the best run of that 100 iterations via the timeit module.

First, to identify the overhead of the spawning of the workers, we execute an function which is simply a pass statement (empty):

```
cmd: python run_benchmarks.py empty_func.py
Importing empty_func
Starting tests ...
non_threaded (1 iters)  0.000001 seconds
threaded (1 threads)    0.000796 seconds
processes (1 procs)     0.000714 seconds

non_threaded (2 iters)  0.000002 seconds
threaded (2 threads)    0.001963 seconds
processes (2 procs)     0.001466 seconds

non_threaded (4 iters)  0.000002 seconds
threaded (4 threads)    0.003986 seconds
```

```
    processes (4 procs)     0.002701 seconds

    non_threaded (8 iters)  0.000003 seconds
    threaded (8 threads)    0.007990 seconds
    processes (8 procs)     0.005512 seconds
```

As you can see, process forking via the pyprocessing package is faster than the speed of building and then executing the threaded version of the code.

The second test calculates 50000 Fibonacci numbers inside of each thread (isolated and shared nothing):

```
    cmd: python run_benchmarks.py fibonacci.py
    Importing fibonacci
    Starting tests ...
    non_threaded (1 iters)  0.195548 seconds
    threaded (1 threads)    0.197909 seconds
    processes (1 procs)     0.201175 seconds

    non_threaded (2 iters)  0.397540 seconds
    threaded (2 threads)    0.397637 seconds
    processes (2 procs)     0.204265 seconds

    non_threaded (4 iters)  0.795333 seconds
    threaded (4 threads)    0.797262 seconds
    processes (4 procs)     0.206990 seconds

    non_threaded (8 iters)  1.591680 seconds
    threaded (8 threads)    1.596824 seconds
    processes (8 procs)     0.417899 seconds
```

The third test calculates the sum of all primes below 100000, again sharing nothing.

```
    cmd: run_benchmarks.py crunch_primes.py
    Importing crunch_primes
    Starting tests ...
    non_threaded (1 iters)  0.495157 seconds
    threaded (1 threads)    0.522320 seconds
    processes (1 procs)     0.523757 seconds

    non_threaded (2 iters)  1.052048 seconds
    threaded (2 threads)    1.154726 seconds
    processes (2 procs)     0.524603 seconds

    non_threaded (4 iters)  2.104733 seconds
    threaded (4 threads)    2.455215 seconds
    processes (4 procs)     0.530688 seconds

    non_threaded (8 iters)  4.217455 seconds
    threaded (8 threads)    5.109192 seconds
    processes (8 procs)     1.077939 seconds
```

The reason why tests two and three focused on pure numeric crunching is to showcase how the current threading implementation does hinder non-I/O applications. Obviously, these tests could be improved to use a queue for coordination of results and chunks of work but that is not required to show the performance of the package and core processing.process module.

The next test is an I/O bound test. This is normally where we see a steep improvement in the threading module approach versus a single-threaded approach. In this case, each worker is opening a descriptor to lorem.txt, randomly seeking within it and writing lines to /dev/null:

```
    cmd: python run_benchmarks.py file_io.py
    Importing file_io
    Starting tests ...
    non_threaded (1 iters)  0.057750 seconds
    threaded (1 threads)    0.089992 seconds
    processes (1 procs)     0.090817 seconds

    non_threaded (2 iters)  0.180256 seconds
    threaded (2 threads)    0.329961 seconds
    processes (2 procs)     0.096683 seconds

    non_threaded (4 iters)  0.370841 seconds
    threaded (4 threads)    1.103678 seconds
    processes (4 procs)     0.101535 seconds

    non_threaded (8 iters)  0.749571 seconds
    threaded (8 threads)    2.437204 seconds
    processes (8 procs)     0.203438 seconds
```

As you can see, pyprocessing is still faster on this I/O operation than using multiple threads. And using multiple threads is slower than the single threaded execution itself.

Finally, we will run a socket-based test to show network I/O performance. This function grabs a URL from a server on the LAN that is a simple error page from tomcat. It gets the page 100 times. The network is silent, and a 10G connection:

```
    cmd: python run_benchmarks.py url_get.py
    Importing url_get
    Starting tests ...
    non_threaded (1 iters)  0.124774 seconds
    threaded (1 threads)    0.120478 seconds
    processes (1 procs)     0.121404 seconds

    non_threaded (2 iters)  0.239574 seconds
    threaded (2 threads)    0.146138 seconds
    processes (2 procs)     0.138366 seconds

    non_threaded (4 iters)  0.479159 seconds
    threaded (4 threads)    0.200985 seconds
    processes (4 procs)     0.188847 seconds

    non_threaded (8 iters)  0.960621 seconds
    threaded (8 threads)    0.659298 seconds
    processes (8 procs)     0.298625 seconds
```

We finally see threaded performance surpass that of single-threaded execution, but the pyprocessing package is still faster when increasing the number of workers. If you stay with one or two threads/workers, then the timing between threads and pyprocessing is fairly close.

One item of note however, is that there is an implicit overhead within the pyprocessing package's Queue implementation due to the

```
object serialization.

Alec Thomas provided a short example based on the
run_benchmarks.py script to demonstrate this overhead versus the
default Queue implementation:

    cmd: run_bench_queue.py
    non_threaded (1 iters)  0.010546 seconds
    threaded (1 threads)    0.015164 seconds
    processes (1 procs)     0.066167 seconds

    non_threaded (2 iters)  0.020768 seconds
    threaded (2 threads)    0.041635 seconds
    processes (2 procs)     0.084270 seconds

    non_threaded (4 iters)  0.041718 seconds
    threaded (4 threads)    0.086394 seconds
    processes (4 procs)     0.144176 seconds

    non_threaded (8 iters)  0.083488 seconds
    threaded (8 threads)    0.184254 seconds
    processes (8 procs)     0.302999 seconds

Additional benchmarks can be found in the pyprocessing package's
source distribution's examples/ directory.  The examples will be
included in the package's documentation.
```

Maintenance

```
Richard M. Oudkerk - the author of the pyprocessing package has
agreed to maintain the package within Python SVN.  Jesse Noller
has volunteered to also help maintain/document and test the
package.
```

API Naming

```
While the aim of the package's API is designed to closely mimic that of
the threading and Queue modules as of python 2.x, those modules are not
PEP 8 compliant. It has been decided that instead of adding the package
"as is" and therefore perpetuating the non-PEP 8 compliant naming, we
will rename all APIs, classes, etc to be fully PEP 8 compliant.

This change does affect the ease-of-drop in replacement for those using
the threading module, but that is an acceptable side-effect in the view
of the authors, especially given that the threading module's own API
will change.

Issue 3042 in the tracker proposes that for Python 2.6 there will be
two APIs for the threading module - the current one, and the PEP 8
compliant one. Warnings about the upcoming removal of the original
java-style API will be issued when -3 is invoked.

In Python 3000, the threading API will become PEP 8 compliant, which
means that the multiprocessing module and the threading module will
again have matching APIs.
```

Timing/Schedule

Some concerns have been raised about the timing/lateness of this PEP for the 2.6 and 3.0 releases this year, however it is felt by both the authors and others that the functionality this package offers surpasses the risk of inclusion.

However, taking into account the desire not to destabilize Python-core, some refactoring of pyprocessing's code "into" Python-core can be withheld until the next 2.x/3.x releases. This means that the actual risk to Python-core is minimal, and largely constrained to the actual package itself.

Open Issues

* Confirm no "default" remote connection capabilities, if needed enable the remote security mechanisms by default for those classes which offer remote capabilities.

* Some of the API (Queue methods qsize(), task_done() and join()) either need to be added, or the reason for their exclusion needs to be identified and documented clearly.

Closed Issues

* The PyGILState bug patch submitted in issue 1683 by roudkerk must be applied for the package unit tests to work.

* Existing documentation has to be moved to ReST formatting.

* Reliance on ctypes: The pyprocessing package's reliance on ctypes prevents the package from functioning on platforms where ctypes is not supported. This is not a restriction of this package, but rather of ctypes.

* DONE: Rename top-level package from "pyprocessing" to "multiprocessing".

* DONE: Also note that the default behavior of process spawning does not make it compatible with use within IDLE as-is, this will be examined as a bug-fix or "setExecutable" enhancement.

* DONE: Add in "multiprocessing.setExecutable()" method to override the default behavior of the package to spawn processes using the current executable name rather than the Python interpreter. Note that Mark Hammond has suggested a factory-style interface for this[7].

References

[1] PyProcessing home page
http://pyprocessing.berlios.de/

[2] See Adam Olsen's "safe threading" project
http://code.google.com/p/python-safethread/

[3] See: Addition of "pyprocessing" module to standard lib.
http://mail.python.org/pipermail/python-dev/2008-May/079417.html

[4] **http://mpi4py.scipy.org/**

[5] See "Cluster Computing"
http://wiki.python.org/moin/ParallelProcessing

[6] The original run_benchmark.py code was published in Python Magazine in December 2007: "Python Threads and the Global Interpreter Lock" by Jesse Noller. It has been modified for this PEP.

[7] **http://groups.google.com/group/python-dev2/msg/54cf06d15cbcbc34**

[8] Addition Python-Dev discussion
http://mail.python.org/pipermail/python-dev/2008-June/080011.html

Copyright