Let's Love the GIL!

• After blowing up the GIL at PyCon'2010, I thought it needed a little more love

• Hence this talk!

• Let's begin
That is All

• Thanks for listening!
• Hope you learned something new
• Follow me! (@dabeaz)
• P.S. Use multiprocessing, futures
Embracing that the GIL Could be Better

David Beazley
http://www.dabeaz.com

October 6, 2011
PyCodeConf 2011, Miami
No, Seriously

- Let's talk about the GIL
- Apparently, it's an issue for some people
- Always comes up in discussions about Python's future whether warranted or not
- Godwin's law of Python?
My Interest

• Why am I so fixated on the GIL?

• Short answer: It's a fun hard systems problem

• Breaking GILs is my hobby
Premise

Threads are useful

• Yes, yes, lots of people love to hate on threads
• That's only because they're being used!
• Threads make all sorts of great stuff work
• Even if you don't see them directly
Solution: Threads
Solution: Threads

P.S. Come visit me in Chicago
The GIL in a Nutshell

• Python code is compiled into VM instructions

```python
def countdown(n):
    while n > 0:
        print n
    n -= 1
>>> import dis
>>> dis.dis(countdown)
0 SETUP_LOOP              33 (to 36)
3 LOAD_FAST                0 (n)
6 LOAD_CONST               1 (0)
9 COMPARE_OP               4 (>)
12 JUMP_IF_FALSE           19 (to 34)
15 POP_TOP                 
16 LOAD_FAST                0 (n)
19 PRINT_ITEM              
20 PRINT_NEWLINE           
21 LOAD_FAST                0 (n)
24 LOAD_CONST               2 (1)
27 INPLACE_SUBTRACT        
28 STORE_FAST               0 (n)
31 JUMP_ABSOLUTE           3
...
```

• In CPython, it is unsafe to execute instructions concurrently

• Hence: Locking
The GIL in a Nutshell

• Things that the GIL protects
  • Reference count updates
  • Mutable types (lists, dicts, sets, etc.)
  • Some internal bookkeeping
  • Thread safety of C extensions
• Keep in mind: It's all low-level (C)
Major GIL Issues

- Threads using multiple CPUs (for computation)
- Uninterruptible instructions
- Bad behavior of CPU-bound threads
The Challenge

• The GIL is unlikely to go away anytime soon
• However, can it be improved?
• Yes!
• Must embrace the idea that it's possible
• ... and agree that it's worthy goal
• There's been some progress in Python 3
An Experiment: Messaging

- A request/reply server for size-prefixed messages

- Each message: a size header + payload
An Experiment: Messaging

• Why this experiment?
• Messaging comes up in a lot of contexts
• Involves I/O
• Foundation of various techniques for working around the GIL (cooperating processes + IPC)
An Experiment: Messaging

A simple test - message echo (pseudocode)

```python
def client(nummsg, msg):
    while nummsg > 0:
        send(msg)
        resp = recv()
        sleep(0.001)
        nummsg -= 1

def server():
    while True:
        msg = recv()
        send(msg)
```

Copyright (C) 2011, David Beazley, http://www.dabeaz.com
An Experiment: Messaging

- A simple test - message echo (pseudocode)

```python
def client(nummsg, msg):
    while nummsg > 0:
        send(msg)
        resp = recv()
        sleep(0.001)
        nummsg -= 1

def server():
    while True:
        msg = recv()
        send(msg)
```

- To be less evil, it's throttled (<1000 msg/sec)

- Hardly a messaging stress test
An Experiment: Messaging

• Five server implementations
  • C with ZeroMQ (no Python)
  • Python with ZeroMQ (C extension)
  • Python with multiprocessing
  • Python with blocking sockets
  • Python with nonblocking sockets, coroutines

• Reminder: Not a messaging stress test
An Experiment: Messaging

- Hardware setup
- 8-CPU Amazon EC2 (c1.xlarge) instance
  - Linux
  - 64 bit
  - 7 GB RAM
  - High I/O performance
- In other words, not my laptop
An Experiment: Messaging

- The test
  - Send/receive 10000 8K messages (echo)
  - 1ms delay after each message
- Emphasis: Not a messaging stress test
An Experiment: Messaging

- Scenario 1: Unloaded server

  ![Diagram of Client and Server](image)

  Time to send/receive 10000 8k messages (Py3.2)

  - Question: What do you expect?
  - 10000 messages w/ 1ms delay = ~10sec
An Experiment: Messaging

- Scenario 1: Unloaded server

  Time to send/receive 10000 8k messages (Py3.2)

  - C + ZeroMQ: 12.8s
  - Python + ZeroMQ: 13.0s
  - Python + multiprocessing: 11.6s
  - Python + blocking sockets: 11.8s
  - Python + nonblocking sockets: 12.2s

- Runs at about 10-20% CPU load
An Experiment: Messaging

- Scenario 2: Server competes with one CPU-thread

  Imagine it's computing something very important
  Like the 200th Fibonacci number via recursion
An Experiment: Messaging

• Scenario 2: Server competes with one CPU-thread

Time to send/receive 10000 8k messages (Py3.2)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>C + ZeroMQ</td>
<td>12.6s</td>
<td>(same)</td>
</tr>
<tr>
<td>Python + ZeroMQ</td>
<td>91.6s</td>
<td>(7.0x slower)</td>
</tr>
<tr>
<td>Python + multiprocessing</td>
<td>103.3s</td>
<td>(8.9x slower)</td>
</tr>
<tr>
<td>Python-Blocking</td>
<td>142.7s</td>
<td>(12.1x slower)</td>
</tr>
<tr>
<td>Python-Nonblocking</td>
<td>126.2s</td>
<td>(10.3x slower)</td>
</tr>
</tbody>
</table>
Commentary

• This aggression will not stand.
• Surely it can be better
• We're not talking about micro-optimization
• Reminder: Not a messaging stress test
Thought: Try PyPy

- Scenario 2: Server competes with one CPU-thread

Time to send/receive 10000 8k messages (pypy-1.6)

.... wait for it (drumroll)
Thought: Try PyPy

• Scenario 2: Server competes with one CPU-thread

```
Client                   CPU-Thread
                    -\           -\
                     |        |   |
          \-----\       |       |   |
          \       \     |       |   |
           \     \    |       |   |
            \   \   |       |   |
             \ \ \  |       |   |
               \ \ \ |       |   |
                \ \ \|       |   |
```

Time to send/receive 10000 8k messages (pypy-1.6)

- Python-Blocking: 6689.2s (567x slower)
- Python-Nonblocking: 4975.0s (408x slower)

• To be fair--there was a bug (already fixed)
Thought : Try Python 2.7

- Scenario 2: Server competes with one CPU-thread

Time to send/receive 10000 8k messages (Py2.7)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>C + ZeroMQ</td>
<td>12.6 (same)</td>
<td></td>
</tr>
<tr>
<td>Python + ZeroMQ</td>
<td>27.7 (2.1x slower)</td>
<td></td>
</tr>
<tr>
<td>Python + multiprocessing</td>
<td>15.0 (1.3x slower)</td>
<td></td>
</tr>
<tr>
<td>Python-Blocking</td>
<td>15.6 (1.3x slower)</td>
<td></td>
</tr>
<tr>
<td>Python-Nonblocking</td>
<td>18.1 (1.5x slower)</td>
<td></td>
</tr>
</tbody>
</table>
Try This At Home

• Not just networks: Try this GUI experiment

```python
# badidle.py

import threading
def spin():
    while True:
        pass

t = threading.Thread(target=spin)
t.daemon=True
t.start()

import idlelib.idle
```

• GUI is completely unusable!
Thread Switching

- The performance problems are related to the mechanism used to switch threads
- In particular, the preemption mechanism and lack of thread priorities
- Py3.2 GIL severely penalizes response-time
GIL Acquisition Sequence

- GIL acquisition based on timeouts

- Any thread that wants the GIL must wait 5ms
Problem: GIL Release

- CPU-bound threads significantly degrade I/O

Thread 1

- Each I/O call drops the GIL and might restart the CPU bound thread

Thread 2

- If it happens, need 5ms to get the GIL back
Performance Explained

• Go back to the server

```python
def server():
    while True:
        msg = recv()
        send(msg)
```
Performance Explained

- What's really happening

```python
def server():
    while True:
        <release GIL>
        msg = recv()
        <acquire GIL>
        <release GIL>
        send(msg)
        <acquire GIL>
```
Performance Explained

• Actually, it's just a bit worse...

def server():
    while True:
        <release GIL>
        msgsize = recv(headersize)
        <acquire GIL>
        <release GIL>
        msgbody = recv(msgsize)
        <acquire GIL>
        <release GIL>
        send(msg)
        <acquire GIL>

• 10000 messages x 15ms = 150s (worst case)
Thread Priorities

• To fix, you need priorities

Thread 1
(low priority)

Thread 2
(high priority)

• The original "New GIL" patch had priorities

• That should be revisited
An Experiment

- I have an experimental Python3.2 w/ priorities
- Extremely minimal
  - Manual priority adjustment (sys.setpriority)
  - Highest priority thread always runs
- Probably too minimal for real (just for research)
Example: Priorities

• Setting a thread's priority

```python
import sys
import threading

def cpushortread():
    sys.setpriority(-1)  # Lower my priority
...

t = threading.Thread(target=cpushortread)
t.start()
```
Messaging + Priorities

- Scenario 2: Server competes with one CPU-thread

Send/receive 10000 8k messages (Py3.2+priorities)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (s)</th>
<th>Speed</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C + ZeroMQ</td>
<td>12.6</td>
<td>same</td>
<td></td>
</tr>
<tr>
<td>Python + ZeroMQ</td>
<td>17.6</td>
<td>1.3x</td>
<td>slower</td>
</tr>
<tr>
<td>Python + multiprocessing</td>
<td>14.2</td>
<td>1.2x</td>
<td>slower</td>
</tr>
<tr>
<td>Python-Blocking</td>
<td>13.0</td>
<td>1.1x</td>
<td>slower</td>
</tr>
<tr>
<td>Python-Nonblocking</td>
<td>14.0</td>
<td>1.1x</td>
<td>slower</td>
</tr>
</tbody>
</table>
GUI Revisited

• Try this variant with priorities

```python
# badidie.py

import sys
import threading
def spin():
    sys.setpriority(-1)
    while True:
        pass

t = threading.Thread(target=spin)
t.daemon=True
t.start()

import idlelib.idle
```

• GUI is completely usable (barely notice)
Some Thoughts

• A huge boost in performance with very few modifications to Python (only a few files)

• Is this the only possible GIL improvement?

• Answer: No

• Example: Should the GIL be released on non-blocking I/O operations? (think about it)
Wrapping Up

• I think all Python programmers should be interested in having a better GIL

• Improving it doesn't necessarily mean huge patches to the Python core

• You (probably) don't have to write an OS

• Incremental improvements can be made
Final Words

- Code and resources
  
  http://www.dabeaz.com/talks/EmbraceGIL/

- All code available under version control
- Hope you enjoyed the talk!
- Follow me on Twitter (@dabeaz)