Inside the New GIL

David M. Beazley
http://www.dabeaz.com

January 14, 2010
@chipy

What Happens at Chipy...

- ... gets people to go change Python
- In June, 2009, I gave that "Mindblowing GIL" presentation and said it would be cool for someone to hack on the problem
- Python 3.2 has a brand new GIL (implemented by Antoine Pitrou)
- Yay!
This Talk

- A very brief refresher on the old GIL
- An overview of the new one
- If you didn't see the previous talk, go to http://www.dabeaz.com/python/GIL.pdf

Disclaimer

- All of this is pretty bleeding edge
- I'm still working on a bunch of updated GIL benchmarks and other results in preparation for PyCON'2010
- So, this talk is rather preliminary... a preview perhaps.
Memory Refresh

- Python has the Global Interpreter Lock (GIL)
- It prevents more than one thread from running simultaneously in the interpreter
- On multicore, it has diabolical behavior
- Not only kills the performance of Python, but affects the performance of the whole machine due to all sorts of crazy system thrashing.

A Performance Test

- Consider this CPU-bound function
  ```python
def count(n):
    while n > 0:
      n -= 1
  ```
- Sequential Execution:
  ```
count(100000000)
count(100000000)
```
- Threaded execution
  ```python
t1 = Thread(target=count, args=(100000000,))
t1.start()
t2 = Thread(target=count, args=(100000000,))
t2.start()
```
Bizarre Results

• Performance comparison (Dual-Core 2Ghz Macbook, OS-X 10.5.6)
  
  Sequential : 24.6s  
  Threaded : 45.5s (1.8X slower!)

• If you disable one of the CPU cores...
  
  Threaded : 38.0s

• Insanely horrible performance. Better performance with fewer CPU cores? It makes no sense.

---

Thread Scheduling

• The old GIL was entirely based on interpreter ticks and repeated signaling on a cond. var.

[Diagram showing thread scheduling]

• All of that signaling is what kills performance
Multicore GIL Battle

- With multiple cores, CPU-bound threads get scheduled simultaneously (on different processors) and then fight it out

Thread 1 (CPU 1)  Thread 2 (CPU 2)

- Release GIL
- Acquire GIL
- Run
- Release GIL
- Acquire GIL
- Run
- Signal
- Wake
- Acquire GIL (fails)
- Wake
- Acquire GIL (fails)

- The waiting thread (T2) may make 100s of failed GIL acquisitions before any success

GIL Battle (In Pictures)

Commentary: Even hard-core Python developers had no idea that this was going on with multicore
The New GIL

- First things first: The new GIL does not eliminate the GIL--it makes it better
- New implementation aims to provide more consistent runtime behavior of threads
- Namely, a significant reduction in all of that thrashing and extra signaling overhead

New GIL Explained

- The new GIL is still based on condition variables and signaling
- However, it's put together in an entirely different way
- Let's take a look
Interpreter Ticks - Gone

- Past versions of Python kept track of interpreter instructions and "ticks"
- Once a certain number of ticks had executed, a thread-switch signal was sent
- This is gone. There are no more ticks.
- sys.setcheckinterval() is gone too
- New GIL is time-based (more in a second)

New Thread Switching

- Decision to thread switch tied to a global var

```c
/* Python/ceval.c */

... static volatile int gil_drop_request = 0;
```

- A thread runs forever in the interpreter until the value of this variable gets set to 1
- At which point, the thread must drop the GIL
- Big question: How does that happen?
New GIL Illustrated

• In the beginning, there is one thread

  Thread 1  running

  • It runs forever
    • Never releases the GIL
    • Never sends any signals
    • Life is good

New GIL Illustrated

• Now, a second thread makes an appearance...

  Thread 1  running

  Thread 2  SUSPENDED

  • It is suspended because it doesn't have the GIL
  • Somehow, it has to get it from Thread 1
• Second thread does a timed `cv_wait` on GIL

```
Thread 1  running
```

```
Thread 2  SUSPENDED
```

```
cv_wait(gil, TIMEOUT)
```

• The idea: Thread 2 will wait to see if the GIL gets released voluntarily by Thread 1 (e.g., if Thread 1 performs I/O or goes to sleep)

---

• Voluntary GIL release

```
Thread 1  running  I/O wait
```

```
Thread 2  SUSPENDED
```

```
cv_wait(gil, TIMEOUT)
```

• This is the easy case. Second thread gets signaled when Thread 1 sleeps. It runs
New GIL Illustrated

• Timeout causes gil_drop_request to be set

\[
\text{Thread 1} \quad \text{running} \quad \rightarrow \\
\text{gil\_drop\_request} = 1
\]

\[
\text{Thread 2} \quad \text{SUSPENDED} \quad \rightarrow \\
\text{cv\_wait(gil,TIMEOUT)} \quad \text{cv\_wait(gil,TIMEOUT)}
\]

• After setting gil_drop_request, Thread 2 repeats its wait request on the GIL

New GIL Illustrated

• Thread 1 is forced to give up the GIL

\[
\text{Thread 1} \quad \text{running} \quad \rightarrow \\
\text{gil\_drop\_request} = 1 \quad \text{signal} \quad \text{running}
\]

\[
\text{Thread 2} \quad \text{SUSPENDED} \quad \rightarrow \\
\text{cv\_wait(gil,TIMEOUT)} \quad \text{cv\_wait(gil,TIMEOUT)}
\]

• It will finish its current instruction, drop the GIL and signal that it has released it
New GIL Illustrated

- On GIL release, Thread 1 waits for a signal

On GIL release, Thread 1 waits for a signal

Thread 1

- The process now repeats itself for Thread 1

Thread 1

- Signal indicates that the other thread successfully got the GIL and is now running

Thread 2

- This eliminates the "GIL Battle"

Thread 2

- So, the sequence you see above happens over and over again as CPU-bound threads execute

Thread 2
Default Timeout

• Default timeout for thread switching is 5 milliseconds (0.005s)
• By comparison, default context-switching interval on most systems is 10 milliseconds
• Adjust with sys.setswitchinterval()

Multiple Thread Handling

• On GIL timeout, a thread only sets gil_drop_request=1 if no thread switches of any kind have occurred in that period
• It's subtle, but if there are a lot of threads competing, gil_drop_request only gets set once per "time interval"
• You want this
Multiple Threads

Thread 1  running  SUSPENDED

Thread 2  SUSPENDED  running  SUSPENDED

gil_drop_request = 1

Thread 3  SUSPENDED  SUSPENDED

Thread 4  SUSPENDED  SUSPENDED

These timeouts do not cause the just started Thread 2 to drop the GIL

First thread to timeout after Thread 2 starts makes the drop request

Multiple Thread Handling

- The thread that makes the request to drop the GIL is not necessarily the one that runs
- This is determined largely by OS priorities
Multiple Threads

Thread 1 running

gil_drop_request = 1

Thread 2 SUSPENDED

Thread 3 SUSPENDED

Thread 4 SUSPENDED

• Here, Thread 2 made Thread 1 drop the GIL, but Thread 3 starts running (up to OS)

Does it Work?

• Yes, it's better (4-core MacPro, OS-X 10.6.2)
  Sequential : 23.5s
  Threaded : 24.0 (2 threads)

• Still working on some other tests (in preparation for PyCON), but it seems to be much better behaved--even if creating 100s of CPU-bound threads
Interesting Features

• The new GIL allows a thread to run for 5ms regardless of other threads or I/O priorities
• So, a CPU-bound thread might block an I/O bound thread for that amount of time
• This is probably what you want to avoid excessive thrashing/context switching
• Be aware that it might impact response time (so you may want to adjust the interval)

Interesting Features

• Long running calculations and C/C++ extensions may block thread switching
• Thread switching is not preemptive
• So, if an operation in an C extension takes 5 seconds to run, you will have to wait that long before the GIL gets released (same was true of old GIL)
Final Comments

• New GIL probably needs further study

• Seems good. Need to investigate behavior under heavy I/O processing

• Again, only implemented in Python 3.2 which is only available via svn checkout

• Backport to Python 2.7? (Don't know)