Understanding the Python GIL

David Beazley
http://www.dabeaz.com

Presented at PyCon 2010
Atlanta, Georgia
Introduction

• As a few of you might know, C Python has a Global Interpreter Lock (GIL)

```python
>>> import that
```

The Unwritten Rules of Python

1. You do not talk about the GIL.
2. You do NOT talk about the GIL.
3. Don't even mention the GIL. No seriously.
... 

• It limits thread performance

• Thus, a source of occasional "contention"
An Experiment

- Consider this trivial CPU-bound function

```python
def countdown(n):
    while n > 0:
        n -= 1
```

- Run it once with a lot of work

```python
COUNT = 100000000   # 100 million
countdown(COUNT)
```

- Now, subdivide the work across two threads

```python
t1 = Thread(target=countdown, args=(COUNT//2,))
t2 = Thread(target=countdown, args=(COUNT//2,))
t1.start(); t2.start()
t1.join(); t2.join()
```
A Mystery

• Performance on a quad-core MacPro
  Sequential : 7.8s
  Threaded (2 threads) : 15.4s (2X slower!)

• Performance if work divided across 4 threads
  Threaded (4 threads) : 15.7s (about the same)

• Performance if all but one CPU is disabled
  Threaded (2 threads) : 11.3s (~35% faster than running
  Threaded (4 threads) : 11.6s with all 4 cores)

• Think about it...
This Talk

• An in-depth look at threads and the GIL that will explain that mystery and much more

• Some cool pictures

• A look at the new GIL in Python 3.2
Disclaimers

• I gave an earlier talk on this topic at the Chicago Python Users Group (chipy)
  http://www.dabeaz.com/python/GIL.pdf

• That is a different, but related talk

• I'm going to go pretty fast... please hang on
Part I
Threads and the GIL
Python Threads

- Python threads are real system threads
  - POSIX threads (pthreads)
  - Windows threads
- Fully managed by the host operating system
- Represent threaded execution of the Python interpreter process (written in C)
Alas, the GIL

- Parallel execution is forbidden
- There is a "global interpreter lock"
- The GIL ensures that only one thread runs in the interpreter at once
- Simplifies many low-level details (memory management, callouts to C extensions, etc.)
Thread Execution Model

- With the GIL, you get cooperative multitasking

- When a thread is running, it holds the GIL

- GIL released on I/O (read, write, send, recv, etc.)
CPU Bound Tasks

- CPU-bound threads that never perform I/O are handled as a special case
- A "check" occurs every 100 "ticks"

• Change it using `sys.setcheckinterval()`
What is a "Tick?"

- Ticks **loosely** map to interpreter instructions

```python
def countdown(n):
    while n > 0:
        print n
        n -= 1
```

```python
>>> import dis
>>> dis.dis(countdown)
```

```
0  SETUP_LOOP              33 (to 36)
1  LOAD_FAST                0 (n)
2  LOAD_CONST               1 (0)
5  COMPARE_OP               4 (>)
8  JUMP_IF_FALSE           19 (to 34)
11 POP_TOP
12 JUMP_IF_FALSE 19 (to 34)
15 POP_TOP
16 LOAD_FAST                0 (n)
19 PRINT_ITEM
20 PRINT_NEWLINE
```

- Instructions in the Python VM
- Not related to timing (ticks might be long)
The Periodic "Check"

- The periodic check is really simple
- The currently running thread...
  - Resets the tick counter
  - Runs signal handlers if the main thread
  - Releases the GIL
  - Reacquires the GIL
- That's it
Implementation (C)

Decrement ticks

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

if (--_Py_Ticker < 0) {
    ...
}
```

Reset ticks

```c
_Py_Ticker = _Py_CheckInterval;
```

Run signal handlers

```c
if (things_to_do) {
    if (Py_MakePendingCalls() < 0) {
        ...
    }
}
```

Release and reacquire the GIL

```c
if (interpreter_lock) {
    /* Give another thread a chance */
    PyThread_release_lock(interpreter_lock);

    /* Other threads may run now */
    PyThread_acquire_lock(interpreter_lock, 1);
}
...
```

Note: Each thread is running this same code.
Big Question

• What is the source of that large CPU-bound thread performance penalty?
• There's just not much code to look at
• Is GIL acquire/release solely responsible?
• How would you find out?
Part 2

The GIL and Thread Switching Deconstructed
Python Locks

- The Python interpreter only provides a single lock type (in C) that is used to build all other thread synchronization primitives
- It's *not* a simple mutex lock
- It's a binary semaphore constructed from a pthreads mutex and a condition variable
- The GIL is an instance of this lock
Locks Deconstructed

• Locks consist of three parts

```c
locked = 0                   # Lock status
mutex  = pthreads_mutex()    # Lock for the status
cond   = pthreads_cond()     # Used for waiting/wakeup
```

• Here's how acquire() and release() work

```c
release() {
    mutex.acquire()
    locked = 0
    mutex.release()
    cond.signal()
}
```

```c
acquire() {
    mutex.acquire()
    while (locked) {
        cond.wait(mutex)
    }
    locked = 1
    mutex.release()
}
```

A critical aspect concerns this signaling between threads
Thread Switching

- Suppose you have two threads
  
  Thread 1 : Running
  
  Thread 2 : Ready (Waiting for GIL)
Thread Switching

• Easy case: Thread 1 performs I/O (read/write)

Thread 1: Running

Thread 2: READY

• Thread 1 might block so it releases the GIL
Thread Switching

• Easy case: Thread 1 performs I/O (read/write)

Thread 1

Thread 2

• Release of GIL results in a signaling operation

• Handled by thread library and operating system
Thread Switching

- Tricky case: Thread 1 runs until the check

Thread 1

- 100 ticks
- check
- release
- GIL
- signal
- pthreads/OS

Which thread runs now?

Thread 2

- READY
- ???

- Either thread is able to run

- So, which is it?
• Condition variables have an internal wait queue

- cv.wait() enqueues threads
- cv.signal() dequeues threads

queue of threads waiting on cv (often FIFO)

• Signaling pops a thread off of the queue
• However, what happens after that?
OS Scheduling

- The operating system has a priority queue of threads/processes ready to run
- Signaled threads simply enter that queue
- The operating system then runs the process or thread with the highest priority
- It may or may not be the signaled thread
Thread Switching

• Thread 1 might keep going

Thread 1 (high priority)  
100 ticks  
Running  
release GIL  
acquire GIL  
pthreads/OS

Thread 2 (low priority)  
READY

• Thread 2 moves to the OS "ready" queue and executes at some later time
Thread Switching

• Thread 2 might immediately take over

Thread 1
(low priority)

Thread 2
(high priority)

• Again, highest priority wins
Part 3

What Can Go Wrong?
GIL Instrumentation

• To study thread scheduling in more detail, I instrumented Python with some logging

• Recorded a large trace of all GIL acquisitions, releases, conflicts, retries, etc.

• Goal was to get a better idea of how threads were scheduled, interactions between threads, internal GIL behavior, etc.
GIL Logging

• An extra tick counter was added to record number of cycles of the check interval

• Locks modified to log GIL events (pseudocode)

```python
release() {
    mutex.acquire()
    locked = 0
    if gil: log("RELEASE")
    mutex.release()
    cv.signal()
}

acquire() {
    mutex.acquire()
    if locked and gil:
        log("BUSY")
        log("RETRY")
    locked = 1
    if gil: log("ACQUIRE")
    mutex.release()
}
```

Note: Actual code in C, event logs are stored entirely in memory until exit (no I/O)
A Sample Trace

- Trace files were large (>20MB for 1s of running)
Logging Results

- The logs were quite revealing
- Interesting behavior on one CPU
- Diabolical behavior on multiple CPUs
- Will briefly summarize findings followed by an interactive visualization that shows details
Single CPU Threading

- Threads alternate execution, but switch far less frequently than you might imagine

Thread 1

100 ticks → check → 100 ticks → check → ...

signal

READY

Thread 2

READY

Hundreds to thousands of checks might occur before a thread context switch (this is good)
Multicore GIL War

• With multiple cores, runnable threads get scheduled simultaneously (on different cores) and battle over the GIL

• Thread 2 is repeatedly signaled, but when it wakes up, the GIL is already gone (reacquired)
Multicore Event Handling

- CPU-bound threads make GIL acquisition difficult for threads that want to handle events

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

run  signal  Acquire GIL (fails)  run
signal  Acquire GIL (fails)  signal
signal  Acquire GIL (fails)  signal
signal  Acquire GIL (success)  Might repeat 100s-1000s of times
Behavior of I/O Handling

- I/O ops often do not block

Due to buffering, the OS is able to fulfill I/O requests immediately and keep a thread running.

- However, the GIL is always released.

- Results in GIL thrashing under heavy load.
GIL Visualization (Demo)

• Let's look at all of these effects

http://www.dabeaz.com/GIL

• Some facts about the plots:
  • Generated from ~2GB of log data
  • Rendered into ~2 million PNG image tiles
  • Created using custom scripts/tools
  • I used the multiprocessing module
Part 4

A Better GIL?
The New GIL

• Python 3.2 has a new GIL implementation (only available by svn checkout)

• The work of Antoine Pitrou (applause)

• It aims to solve all that GIL thrashing

• It is the first major change to the GIL since the inception of Python threads in 1992

• Let's go take a look
New Thread Switching

• Instead of ticks, there is now a global variable

    /* Python/ceval.c */
    ...
    static volatile int gil_drop_request = 0;

• A thread runs until the value gets set to 1
• At which point, the thread **must** drop the GIL
• Big question: How does that happen?
New GIL Illustrated

• Suppose that there is just one thread

Thread 1 running

• It just runs and runs and runs ...
  • Never releases the GIL
  • Never sends any signals
  • Life is great!
New GIL Illustrated

• Suppose, a second thread appears

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
New GIL Illustrated

- Waiting thread does a timed cv_wait on GIL

Thread 1  \(\rightarrow\)  running

Thread 2  \(\rightarrow\)  SUSPENDED

\[\text{cv\_wait}(\text{gil}, \text{TIMEOUT})\]

By default TIMEOUT is 5 milliseconds, but it can be changed

- The idea: Thread 2 waits to see if the GIL gets released voluntarily by Thread 1 (e.g., if there is I/O or it goes to sleep for some reason)
New GIL Illustrated

- Voluntary GIL release

Thread 1: running -> I/O

Thread 2: Suspend

\[ \text{cv\_wait(gil, TIMEOUT)} \]

- This is the easy case. Second thread is signaled and it grabs the GIL.
New GIL Illustrated

- If timeout, set gil_drop_request

Thread 1

Thread 2

- Thread 2 then repeats its wait on the GIL
New GIL Illustrated

- Thread 1 suspends after current instruction

Thread 1  \[\text{running}\]  \[\text{gil\_drop\_request} = 1\]  \[\text{signal}\]  \[\text{cv\_wait(gil, TIMEOUT)}\]

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

- Signal is sent to indicate release of GIL
New GIL Illustrated

• On a forced release, a thread waits for an ack

Thread 1  running  

Thread 2  SUSPENDED  

• Ack ensures that the other thread successfully got the GIL and is now running

• This eliminates the "GIL Battle"
New GIL Illustrated

• The process now repeats itself for Thread 1

Thread 1

Thread 2

• So, the timeout sequence happens over and over again as CPU-bound threads execute
Does it Work?

- Yes, apparently (4-core MacPro, OS-X 10.6.2)
  
  Sequential : 11.53s
  Threaded (2 threads) : 11.93s
  Threaded (4 threads) : 12.32s

- Keep in mind, Python is still limited by the GIL in all of the usual ways (threads still provide no performance boost)

- But, otherwise, it looks promising!
Part 5

Die GIL Die!!!
Alas, It Doesn't Work

- The New GIL impacts I/O performance
- Here is a fragment of network code

```python
# Thread 1
def spin():
    while True:
        # some work
        pass

# Thread 2
def echo_server(s):
    while True:
        data = s.recv(8192)
        if not data:
            break
        s.sendall(data)
```

- One thread is working (CPU-bound)
- One thread receives and echos data on a socket
Response Time

• New GIL increases response time

Thread 1

gil_drop_request = 1

Thread 2

data arrives

cv_wait(gil, TIMEOUT)

cv_wait(gil, TIMEOUT)

• To handle I/O, a thread must go through the entire timeout sequence to get control

• Ignores the high priority of I/O or events
Unfair Wakeup/Starvation

- Most deserving thread may not get the GIL

- Caused by internal condition variable queuing

- Further increases the response time
Convoy Effect

- I/O operations that don't block cause stalls

  Thread 1
  ▶️ running
  ▶️ timeout
  ▶️ sig
  ▶️ timeout
  ▶️ sig
  ▶️ timeout

  Thread 2
  ▶️READY
  ▶️ data arrives
  ▶️ send (executes immediately)
  ▶️ sig
  ▶️ send (executes immediately)
  ▶️ sig
  ▶️ timeout

- Since I/O operations always release the GIL, CPU-bound threads will always try to restart

- On I/O completion (almost immediately), the GIL is gone so the timeout has to repeat
An Experiment

• Send 10MB of data to an echo server thread that's competing with a CPU-bound thread

  Python 2.6.4 (2 CPU) : 0.57s  (10 sample average)
  Python 3.2 (2 CPU)  : 12.4s  (20x slower)

• What if echo competes with 2 CPU threads?

  Python 2.6.4 (2 CPU) : 0.25s  (Better performance?)
  Python 3.2 (2 CPU)  : 46.9s  (4x slower than before)
  Python 3.2 (1 CPU)  : 0.14s  (330x faster than 2 cores?)

• Arg! Enough already!
Part 6

Score: Multicore 2, GIL 0
Fixing the GIL

• Can the GIL's erratic behavior be fixed?
• My opinion: Yes, maybe.
• The new GIL is already 90% there
• It just needs a few extra bits
The Missing Bits

- **Priorities**: There must be some way to separate CPU-bound (low priority) and I/O bound (high priority) threads

- **Preemption**: High priority threads must be able to immediately preempt low-priority threads
A Possible Solution

- Operating systems use timeouts to automatically adjust task priorities (multilevel feedback queuing)
  - If a thread is preempted by a timeout, it is penalized with lowered priority (bad thread)
  - If a thread suspends early, it is rewarded with raised priority (good thread)
  - High priority threads always preempt low priority threads
- Maybe it could be applied to the new GIL?
Remove the GIL?

• This entire talk has been about the problem of implementing one tiny little itty bitty lock

• Fixing Python to remove the GIL entirely is an exponentially more difficult project

• If there is one thing to take away, there are practical reasons why the GIL remains
Final Thoughts

• Don't use this talk to justify not using threads
• Threads are a very useful programming tool for many kinds of concurrency problems
• Threads can also offer excellent performance even with the GIL (you need to study it)
• However, you should know about the tricky corner cases
Final Thoughts

• Improving the GIL is something that all Python programmers should care about

• Multicore is not going away

• You might not use threads yourself, but they are used for a variety of low-level purposes in frameworks and libraries you might be using

• More predictable thread behavior is good
Thread Terminated

- That's it!
- Thanks for listening!
- Questions