Inside the Python GIL

David Beazley
http://www.dabeaz.com

June 11, 2009 @ chipy

Originally presented at my "Python Concurrency Workshop", May 14-15, 2009 (Chicago)

Video Presentation

You can watch the video of this presentation here:

http://blip.tv/file/2232410

It expands upon the slides and is recommended.

Introduction

- As most programmers know, Python has a Global Interpreter Lock (GIL)
- It imposes various restrictions on threads
- Namely, you can't utilize multiple CPUs
- Thus, it's a (frankly) tired subject for flamewars about how Python "sucks" (along with tail-call optimization, lambda, whitespace, etc.)

Disclaimers

- Python's use of a GIL doesn't bother me
- I don't have strong feelings about it either way
- Bias : For parallel computing involving heavy CPU processing, I much prefer message passing and cooperating processes to thread programming (of course, it depends on the problem)
- However, the GIL has some pretty surprising behavior on multicore that interests me

A Performance Experiment

- Consider this trivial CPU-bound function

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

- Run it twice in series

```
count(100000000)
count(100000000)
```

- Now, run it in parallel in two threads

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
t1.join(); t2.join()
```


A Mystery

- Why do I get these performance results on my Dual-Core MacBook?

  Sequential : 24.6s
  Threaded : 45.5s (1.8X slower!)

- And if I disable one of the CPU cores, why does the threaded performance get better?

  Threaded : 38.0s

- Think about that for a minute... Bloody hell!

Overview

- I don't like unexplained mysteries or magic
- As part of a workshop I ran in May, I went digging into the GIL implementation to see if I could figure out exactly why I was getting those performance results
- An exploration that went all the way from Python scripts to the C source code of the pthreads library (yes, I probably need to go outside more often)
- So, let's just jump into it...

What is a Thread?

- Python threads are real system threads
  - POSIX threads (pthreads)
  - Windows threads
- Fully managed by the host operating system
  - All scheduling/thread switching
- Represent threaded execution of the Python interpreter process (written in C)

Thread Creation

- Python threads simply execute a "callable"
- The run() method of Thread (or a function)

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```


Behind the Scenes

- There's not a whole lot going on...
- Here's what happens on thread creation
  - Python creates a small data structure containing some interpreter state
  - A new thread (pthread) is launched
  - The thread calls PyEval_CallObject
- Last step is just a C function call that runs whatever Python callable was specified

Thread-Specific State

- Each thread has its own interpreter specific data structure (PyThreadState)
  - Current stack frame (for Python code)
  - Current recursion depth
  - Thread ID
  - Some per-thread exception information
  - Optional tracing/profiling/debugging hooks
- It's a small C structure (<100 bytes)

PyThreadState Structure

```
typedef struct _ts {
    struct _ts         *next;
    PyInterpreterState *interp;
    struct _frame      *frame;
    int                 recursion_depth;
    int                 tracing;
    int                 use_tracing;
    Py_tracefunc        c_profilefunc;
    Py_tracefunc        c_tracefunc;
    PyObject           *c_profileobj;
    PyObject           *c_traceobj;
    PyObject           *curexc_type;
    PyObject           *curexc_value;
    PyObject           *curexc_traceback;
    PyObject           *exc_type;
    PyObject           *exc_value;
    PyObject           *exc_traceback;
    PyObject           *dict;
    int                 tick_counter;
    int                 gilstate_counter;
    PyObject           *async_exc;
    long                thread_id;
} PyThreadState;
```

Thread Execution

- The interpreter has a global variable that simply points to the ThreadState structure of the currently running thread

```
/* Python/pystate.c */
...
PyThreadState *_PyThreadState_Current = NULL;
```

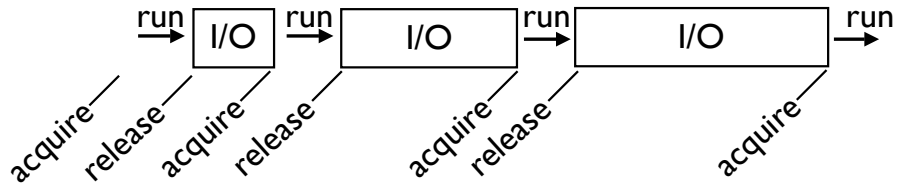
- Operations in the interpreter implicitly depend this variable to know what thread they're currently working with

The Infamous GIL

- Here's the rub...
- Only one Python thread can execute in the interpreter at once
- There is a "global interpreter lock" that carefully controls thread execution
- The GIL ensures that sure each thread gets exclusive access to the interpreter internals when it's running (and that call-outs to C extensions play nice)

GIL Behavior

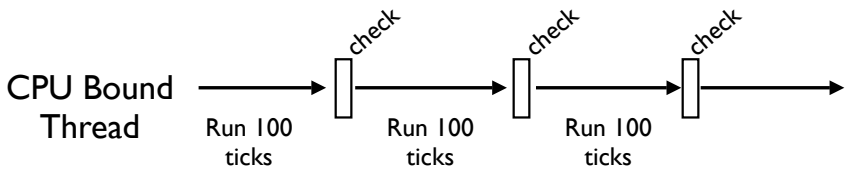
- It's simple : threads hold the GIL when running
- However, they release it when blocking for I/O



- So, any time a thread is forced to wait, other "ready" threads get their chance to run
- Basically a kind of "cooperative" multitasking

CPU Bound Processing

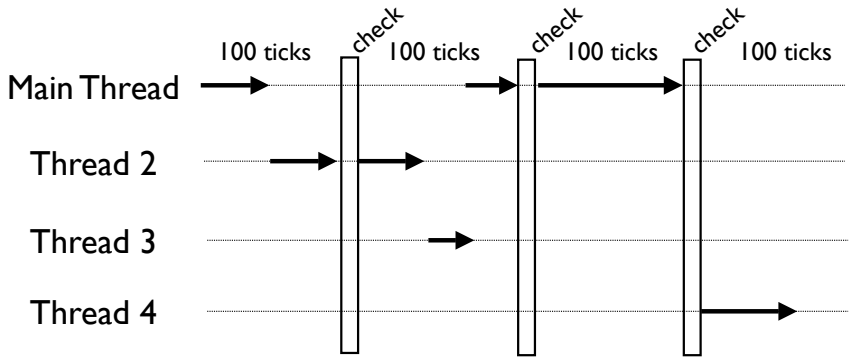
- To deal with CPU-bound threads that never perform any I/O, the interpreter periodically performs a "check"
- By default, every 100 interpreter "ticks"



- sys.setcheckinterval() changes the setting

The Check Interval

- The check interval is a global counter that is completely independent of thread scheduling



- A "check" is simply made every 100 "ticks"

The Periodic Check

- What happens during the periodic check?
  - In the main thread only, signal handlers will execute if there are any pending signals (more shortly)
  - Release and reacquire the GIL
- That last bullet describes how multiple CPU-bound threads get to run (by briefly releasing the GIL, other threads get a chance to run).

ceval.c execution

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

if (--_Py_Ticker < 0) {
    ...
    _Py_Ticker = _Py_CheckInterval;
    ...
    if (things_to_do) {
        if (Py_MakePendingCalls() < 0) {
            ...
        }
    }
    if (interpreter_lock) {
        /* Give another thread a chance */
        ...
        PyThread_release_lock(interpreter_lock);

        /* Other threads may run now */

        PyThread_acquire_lock(interpreter_lock, 1);
        ...
}
```


What is a "Tick?"

- Ticks loosely map to interpreter instructions

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

- Instructions in the Python VM

```
>>> 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
...
```

Tick 1 | Tick 2 | Tick 3 | Tick 4

Tick Execution

- Interpreter ticks are not time-based
- In fact, long operations can block everything

```
>>> nums = xrange(100000000)
>>> -1 in nums
False
>>>
```

→ 1 tick (~ 6.6 seconds)

- Try hitting Ctrl-C (ticks are uninterruptible)

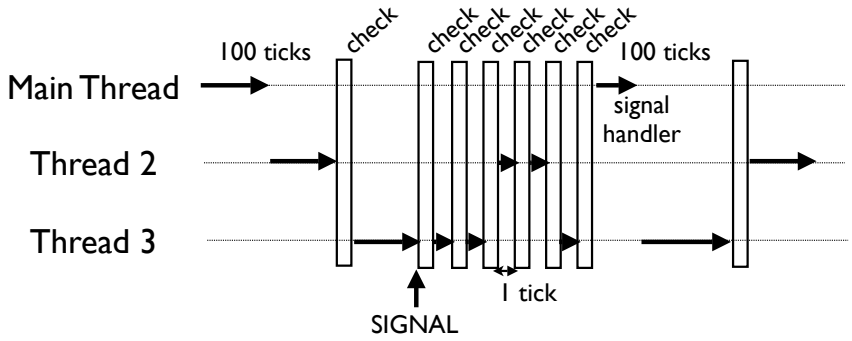
```
>>> nums = xrange(100000000)
>>> -1 in nums
^C^C^C   (nothing happens, long pause)
...
KeyboardInterrupt
>>>
```


Interlude : Signals

- Let's briefly talk about Ctrl-C
- A very common problem encountered with Python thread programming is that threaded programs can no longer be killed with the keyboard interrupt
- It is EXTREMELY ANNOYING (you have to use kill -9 in a separate window)
- Ever wonder why it doesn't work?

Signal Handling

- If a signal arrives, the interpreter runs the "check" after every tick until the main thread runs



- Since signal handlers can only run in the main thread, the interpreter quickly acquires/releases the GIL after every tick until it gets scheduled

Thread Scheduling

- Python does not have a thread scheduler
- There is no notion of thread priorities, preemption, round-robin scheduling, etc.
- All thread scheduling is left to the host operating system (e.g., Linux, Windows, etc.)
- This is partly why signals get so weird (the interpreter has no control over scheduling so it just attempts to thread switch as fast as possible with the hope that main will run)

Frozen Signals

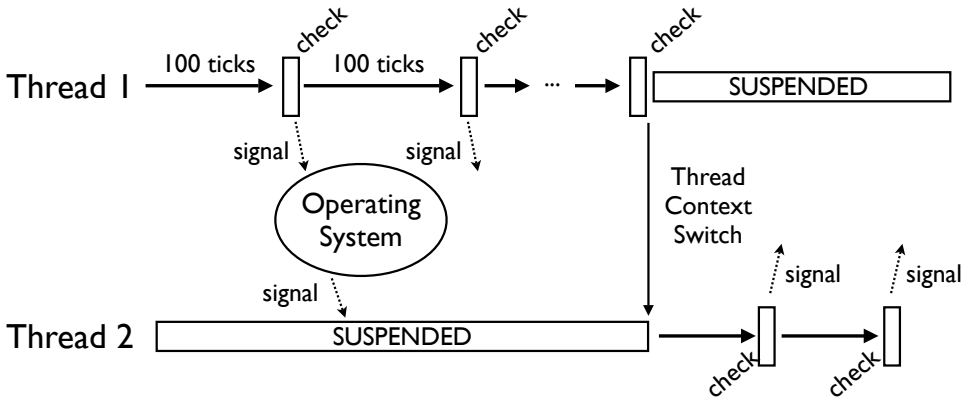
- The reason Ctrl-C doesn't work with threaded programs is that the main thread is often blocked on an uninterruptible thread-join or lock
- Since it's blocked, it never gets scheduled to run any kind of signal handler for it
- And as an extra little bonus, the interpreter is left in a state where it tries to thread-switch after every tick (so not only can you not interrupt your program, it runs slow as hell!)

GIL Implementation

- The GIL is not a simple mutex lock
- The implementation (Unix) is either...
  - A POSIX unnamed semaphore
  - Or a pthreads condition variable
- All interpreter locking is based on signaling
  - To acquire the GIL, check if it's free. If not, go to sleep and wait for a signal
  - To release the GIL, free it and signal

Thread Scheduling

- Thread switching is far more subtle than most programmers realize



- The lag between signaling and execution may be significant (depends on the OS)

Thread Scheduling

- The OS is just going to schedule whichever thread has the highest execution "priority"
  - CPU-bound : low priority
  - I/O bound : high priority
- If a signal is sent to a thread with low priority and the CPUs are busy with higher priority tasks, it won't run until some later point
- Read an OS textbook for details

CPU-Bound Threads

- As we saw earlier, CPU-bound threads have horrible performance properties
- Far worse than simple sequential execution
  - 24.6 seconds (sequential)
  - 45.5 seconds (2 threads)
- A big question : Why?
  - What is the source of that overhead?

Signaling Overhead

- GIL thread signaling is the source of that
- After every 100 ticks, the interpreter
  - Locks a mutex
  - Signals on a condition variable/semaphore where another thread is always waiting
  - Because another thread is waiting, extra pthreads processing and system calls get triggered to deliver the signal

A Rough Measurement

- Sequential Execution (OS-X, 1 CPU)
  - 736 Unix system calls
  - 117 Mach System Calls
- Two CPU-bound threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls
- Yow! Look at that last figure.

Multiple CPU Cores

- The penalty gets far worse on multiple cores
- Two CPU-bound threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~3.3 Million Mach System Calls
- Two CPU-bound threads (OS-X, 2 CPUs)
  - 1149 Unix system calls
  - ~9.5 Million Mach System calls

An Experiment

- I did some instrumentation of the Python interpreter to look into this a little deeper
- Recorded a real-time trace of all GIL acquisitions, releases, conflicts, retries, etc.
- Trying to get an idea of what the interpreter is doing, what different threads are doing, interactions between threads and the GIL, and the overall sequencing of events

A Sample Trace

```
t2 100 5351 ENTRY
t2 100 5351 ACQUIRE
t2 100 5352 RELEASE
t2 100 5352 ENTRY
t2 100 5352 ACQUIRE
t2 100 5353 RELEASE
t1 100 5353 ACQUIRE
t2 100 5353 ENTRY
t2  38 5353 BUSY
t1 100 5354 RELEASE
t1 100 5354 ENTRY
t1 100 5354 ACQUIRE
t2  79 5354 RETRY
t1 100 5355 RELEASE
t1 100 5355 ENTRY
t1 100 5355 ACQUIRE
t2  73 5355 RETRY
t1 100 5356 RELEASE
t2 100 5356 ACQUIRE
t1 100 5356 ENTRY
t1  24 5356 BUSY
t2 100 5357 RELEASE
```

thread id

tick countdown

total number of "checks" executed

ENTRY : Entering GIL critical section

ACQUIRE : GIL acquired

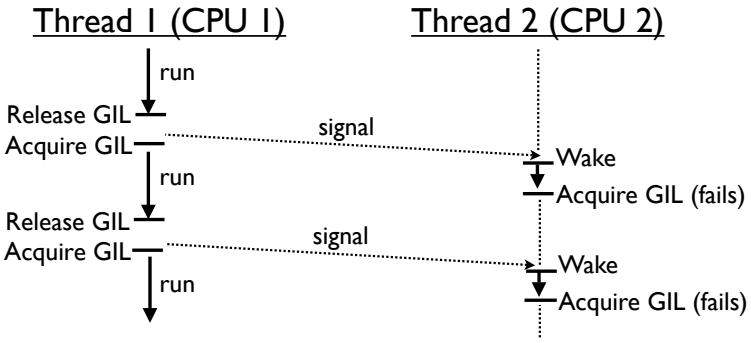
RELEASE : GIL released

BUSY : Attempted to acquire GIL, but it was already in use

RETRY : Repeated attempt to acquire the GIL, but it was still in use

Multicore GIL Contention

- With multiple cores, CPU-bound threads get scheduled simultaneously (on different cores) and then have a GIL battle



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

The GIL Battle (Traced)

```
t2 100 5392 ENTRY
t2 100 5392 ACQUIRE
t2 100 5393 RELEASE
t1 100 5393 ACQUIRE
t2 100 5393 ENTRY
t2  27 5393 BUSY
t1 100 5394 RELEASE
t1 100 5394 ENTRY
t1 100 5394 ACQUIRE
t2  74 5394 RETRY
t1 100 5395 RELEASE
t1 100 5395 ENTRY
t1 100 5395 ACQUIRE
t2  83 5395 RETRY
t1 100 5396 RELEASE
t1 100 5396 ENTRY
t1 100 5396 ACQUIRE
t2  80 5396 RETRY
t1 100 5397 RELEASE
t1 100 5397 ENTRY
t1 100 5397 ACQUIRE
t2  79 5397 RETRY
...
```

signal

signal

signal

signal

A thread switch

t2 tries to keep running, but immediately has to block because t1 acquired the GIL

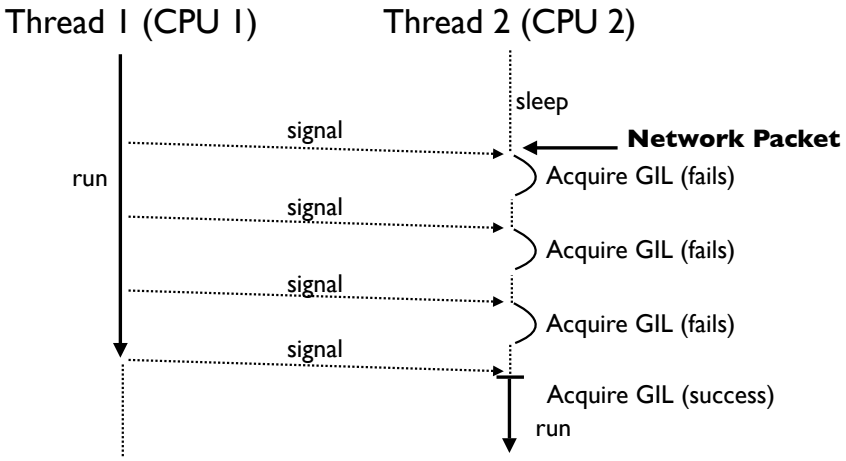
Here, the GIL battle begins. Every RELEASE of the GIL signals t2. Since there are two cores, the OS schedules t2, but leaves t1 running on the other core. Since t1 is left running, it immediately reacquires the GIL before t2 can get to it (so, t2 wakes up, finds the GIL is in use, and blocks again)

A Scheduler Conflict

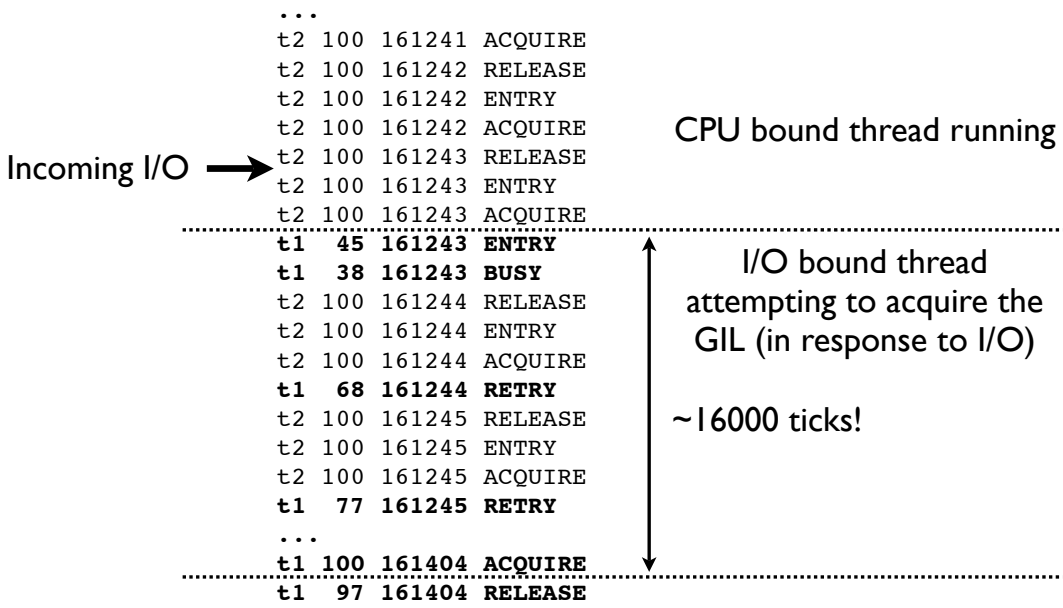
- What's happening here is that you're seeing a battle between two competing (and ultimately incompatible) goals
  - Python - only wants to run single-threaded, but doesn't want anything to do with thread scheduling (up to OS)
  - OS - "Oooh. Multiple cores." Freely schedules processes/threads to take advantage of as many cores as possible

Multicore GIL Contention

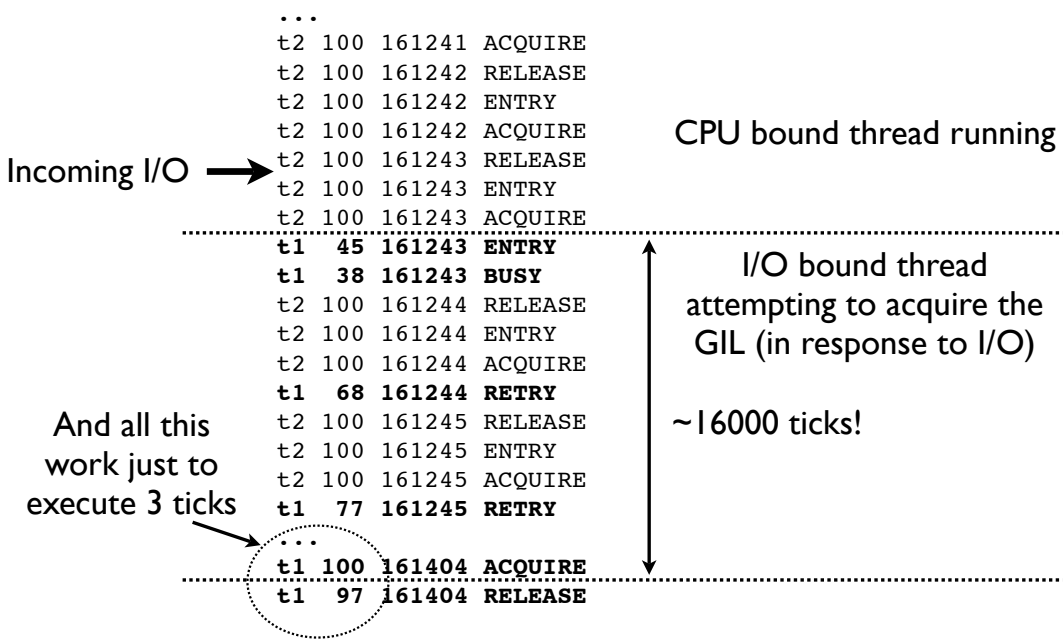
- Even 1 CPU-bound thread causes problems
- It degrades response time of I/O-bound threads



An I/O Bound Trace



An I/O Bound Trace



Priority Inversion

- This last scenario is a bizarre sort of "priority inversion" problem
- A CPU-bound thread (low priority) is blocking the execution of an I/O-bound thread (high priority)
- It occurs because the I/O thread can't wake up fast enough to acquire the GIL before the CPU-bound thread reacquires it
- And it only happens on multicore...

Comments

- As far as I can tell, the Python GIL implementation has not changed much (if at all) in the last 10 years
- The GIL code in Python 1.5.2 looks almost identical to the code in Python 3.0
- I don't know whether it's even been studied all that much (especially on multicore)
- There is more interest in removing the GIL than simply changing the GIL

Comments

- I think this deserves further study
  - There is a pretty severe performance penalty for using threads on multicore
  - The priority inversion for I/O-bound processing is somewhat disturbing
- Probably worth fixing--especially if the GIL is going to stick around

Open Questions

- How in the hell would you fix this?
- I have some vague ideas, but they're all "hard"
- Require Python to do its own form of thread scheduling (or at least cooperate with the OS)
- Would involve a non-trivial interaction between the interpreter implementation, the operating system scheduler, the thread library, and C extension modules (egad!)

Is it Worth It?

- If you could fix it, it would make thread execution (even with the GIL) more predictable and less resource intensive
- Might improve performance/responsiveness of applications that have a mix of CPU and I/O-bound processing
- Probably good for libraries that use threads in the background (e.g., multiprocessing)
- Might be able to do it without rewriting the whole interpreter.

That's All Folks

- I'm not actively working on any patches or code related to this presentation
- However, the problem interests me
- If it interests you and you want to hack on any of my code or examples, send me an email (dave@dabeaz.com)