Multicore Programming
Spring 2010
Final Report

Erik Froese and Antony Kaplan

"Just Say No to the combined evils of locking, deadlocks, lock granularity, livelocks, nondeterminism and race conditions." - Guido Van Rossum

**Abstract**

The CPython interpreter's global interpreter lock (GIL) introduces a big speed penalty for multi-threaded Python programs. The GIL has been around as long as the interpreter and has survived several attempts to remove it as well as the ire of a lot of Python programmers. The GIL serializes access to the Python interpreter, effectively removing the benefit of threading for CPU-bound programs. Because of some design choices in implementing the interpreter the performance of a multi-threaded program can be significantly worse than a serial version when run on multiple CPUs or cores.

# The Python GIL

## Introduction

Python is an interpreted, dynamically typed, object-oriented programming language. The Python interpreter is written in C/ASM and is known as "CPython" (to differentiate it from other Python implementations). It first translates Python source code into an intermediate language known as "Python bytecode"which is roughly analogous to assembly language but contains higher-level instructions. When one runs a Python program they are invoking the interpreter program which then translates bytecode into machine-specific operations in what is known as the "evaluation loop". One can think of the interpretation process as going back and forth between the programmer's code and the interpreter itself. This is of course not completely true as the interpreter is in control the entire time but it's important to keep this in mind as we investigate the behavior of the Python Global Interpreter Lock or "GIL".

Using an interpreter has its advantages but also incurs overhead. To that end people have been looking for ways to speed up Python programs. With CPU-intensive code for instance, one can write modules in C and call them directly from Python. Another solution is Psyco, a "Just in Time" compiler for Python that emits efficient machine code on the fly. It claims 2x to 100x speedups over interpreted Python. What about "multi-threading"? Python provides two libraries for programs that could benefit from parallelism, threads and multiprocessing. However, there's a big catch when it comes to using threads in Python; the CPython interpreter has a global lock known as the GIL.

## What is the "GIL"?

Before approaching the GIL it would be wise to talk about threading in Python. There's no special "Python Threads"- that is, threads in python are really just native operating system threads. On Windows systems these are native threads and on Mac and *Nix systems these are pthreads.

When spawning a new thread in Python, a native operating system thread and a small C data structure called PyThreadState are created. As implied by the name, the data structure holds some of the internal state of the interpreter relevant to the currently executing thread. Python threads are just threads of execution of the interpreter itself- that is in execution, each thread goes through this aforementioned "back and forth" process between executing interpreter code and user written code. It is clear to see how allowing multiple threads to simultaneously execute in the interpreter can become very messy very quickly. Hence, the interpreter does not execute more than one thread at any one time- it's serialized by the GIL. Re-stated, only one stream of Python bytecode is ever being executed at a time. The GIL actually just protects a pointer called _PyThreadState_Current, which points to the state of the currently executing thread.

The interpreter has a GIL for a couple of reasons- the most obvious may be that it's easier to implement an interpreter with one coarse lock. Anyone who has ever written a multi-threaded program knows that reasoning about the code becomes a lot more complicated as the number of locks goes up. The GIL protects the internal state of the interpreter. It makes actions like updating the reference counts of objects thread-safe. It also makes the lives of module developers easier. One doesn't have to worry about their module being thread-safe or not- it's protected by the GIL.

So why even have threads in Python? The short answer is I/O. Any Python thread that executes an I/O instruction has the potential to block, so it releases the GIL. If you have some mix of I/O and CPU-bound processing you can benefit from threading by doing work while you wait for I/O to complete. You could also use multiple threads for GUI programs or other cases where you want to remain responsive to user input while doing something else.

## Thread Scheduling

There is no notion of threads scheduling in the Python interpreter. All scheduling decisions are delegated to the operating system itself. This is usually viewed as a wise choice- operating system schedulers are extremely complicated, and hence rewriting one from scratch is a formidable task, to be attempted only if necessary. However, we have discovered that this delegation of scheduling to the operating system produces some interesting (and unfortunate) consequences, ultimately resulting in extremely poor performance of multi-threaded code in CPython.

The GIL is an object that contains a mutex and a condition variable. A thread that holds the GIL mutex is allowed to proceed interpreting bytecode. Other threads that try to acquire the GIL and fail issue a wait() on the condition variable and go to sleep. When the thread that holds the GIL releases the mutex it signals the condition variable instructing the operating system to wake up one or more threads that are sleeping waiting for the GIL. It is important to note that in the contended case most of these actions require system calls which introduce even more overhead.

If only one thread can execute at a time and it must have the GIL to do so, the interpreter must do something to avoid one thread holding the GIL and never releasing it. The CPython implementation approaches this problem by forcing the interpreter to give up the GIL periodically. The interpreter releases the GIL every 100 "ticks". The concept of a tick is a little tricky. Roughly stated, a tick corresponds to a Python bytecode operation. For the most part that's true, however there are certain bytecode instructions that do not qualify as whole ticks. Furthermore, ticks are uninterruptible. The interpreter will not thread switch in the middle of a tick.  It is possible to construct short pieces of python code that execute forever without "ticking", and hence are uninterruptible! In normal python code the interpreter counts down from 100 as it goes through the evaluation loop and calls a check() function when the tick counter gets to 0.

So how does the interpreter switch between threads? As we said before, thread scheduling is delegated to the operating system. After the tick counter gets to 0, the executing thread stops executing user code, updates its state, and then releases the GIL. The next bit of code that any thread will execute if scheduled is to grab the GIL. So if in this short interval (when the GIL is released), the operating system decides to initiate a thread switch, then this new thread will start executing in Python. If not, the old thread will reacquire the GIL and continue its execution. In this way, the operating system is almost entirely responsible for thread scheduling. The one thing that you can control in the Python configuration is how often this interpreter "check" occurs (100 ticks by default). This however does not directly correlate to how often thread switching occurs, just how often the operating system is given the opportunity to thread switch.

On multi-core and multi-processor machines, the scheme above proves to be problematic. One a single core machine, when an executing thread releases the GIL, the operating system is given the opportunity to suspend the currently executing thread, and schedule another thread in the ready queue to run. On a multi-core system, the operating system sees no need in suspending the original thread, and just schedules a new thread on a different physical core. However, looking at the code in "ceval.c" which contains CPython's evaluation loop, we see that the two lines of code releasing and reacquiring the GIL are next to each other- that is, an executing thread releases the GIL and then immediately tries to reacquire it. On a multi-core machine, the operating system does not suspend the thread which releases the GIL, and hence this thread immediately reacquires it, before any other thread has a chance to even be scheduled. This is what David Beazely calls a "GIL battle". When an executing thread releases the GIL, it signals all other threads waiting on the condition variable, and hence a new thread is scheduled, attempts to grab the GIL but finds that it has already been reacquired by the original executing thread. There could be upwards of a hundred thousand failed attempts before a thread switch actually occurs.

## GIL Performance Problems

How does this translate to Python's performance? For a simple program like "countdown", which is just a while loop in which we decrement a counter until it gets to 0, a multithreaded implementation (where two threads just execute "countdown" each on half the original count) is almost twice as slow as a serial implementation!

Since the interpreter threads are serialized you don't get any performance increase dividing cpu-bound work between python threads. Furthermore you incur the overhead of system calls to coordinate the threads using lock, unlock, wait, and signal calls on the GIL mutex and condition variable. The problem is worse on computers with more than one core or CPU; the operating system can be simultaneously running many python threads exacerbating this "GIL Battle". One thread will be running, rapidly releasing and re-acquiring the GIL. Other threads will wake on the condition variable and try to acquire the GIL mutex. Unfortunately for the waiting threads, the cpu hungry thread will re-acquire the GIL too quickly. This leads to a lot of time being sent signaling, waking up threads, and failing to acquire locks. Using the simple unix time utility, you will see a lot of time spent in the system area with a muti-threaded python application. This is all of those signaling and and failed acquisition calls.

This problem also exists for I/O bound programs! Threads that block frequently for I/O can experience increased latency and reduced throughput because of the behavior of multiple python threads on multicore machine. A blocked I/O thread will be woken up by the operating system to complete an I/O but it can take hundreds of thousands of failed GIL acquisitions to finally acquire it and make progress.

## Signal handling messiness

The CPython interpreter can only handle signals in the main thread of execution. If a signal arrives, the interpreter will try to get the main thread scheduled by running a check every tick until operating system schedules main execution thread. If the main thread is blocked somewhere, say on an uninterruptible thread-join, the program becomes uninterruptible and performance degrades significantly as more and more signaling and rescheduling occurs. This is why sometimes in threaded programs entering the "Cntrl+c" break does not work and the program seems to hang. In reality the threads are making progress very slowly and the operating system is hard at work delivering signals and rescheduling your Python threads.

## The New GIL in 3.2

There is a new implementation of the GIL by Antoine Pitrou in the upcoming Python 3.2 release. The GIL is still a mutex and a condition variable. The interpreter no longer releases the GIL every hundred ticks. In fact ticks are gone altogether. Interpreter threads now issue a timed wait for the GIL instead of a simple wait. If a waiting thread is unable to acquire the GIL twice it sets a global variable drop_gil_request to true and waits again.

The interpreter is constantly monitoring the status of the drop_gil_request variable. If it detects that drop_gil_request is true it immediately releases the GIL and issues a wait on a condition variable gil_acquired. When that second thread acquires the GIL it signals on the gil_acquired variable, waking up the first thread and causing it to then issue a timed wait on the GIL. That second variable is an acknowledgement that another thread has acquired the GIL.

So it goes like this:

1. A has the GIL and is running

2. B is spawned. timed_wait(GIL, DEFAULT_WAIT).

3. B wants the GIL and can not acquire it. drop_gil_request=true
```
    timed_wait(GIL.cv, DEFAULT_WAIT).
```

4. A has the GIL, detects the drop_gil_request is true
```
    release(GIL.mutex)
    signal(GIL.cv).
    wait(gil_acquired)
```

5. B receives the signal on GIL.cv, wants the GIL, and it's free. It acquires it and starts running.
```
    acquire(GIL.mutex)
    signal(gil_acquired)
    drop_gil_request=false
```

6. A receives the signal on gil_acquired and waits for the GIL, timed_wait(GIL, DEFAULT_WAIT)

One of the benefits of the new GIL is that it avoids a series of expensive and unnecessary system calls on systems with multiple CPUs. It does still have the affect of possibly starving I/O bound threads in the presence of CPU bounce threads but that can be tuned by reducing the default time to wait on the GIL condition variable.

## References

**Dave Beazely's GIL homepage**
http://www.dabeaz.com/GIL/
**Video of the New GIL presentation**
http://python.mirocommunity.org/video/1226/changes-to-the-gil-in-python-3
**Inside the GIL presentation - Good intro to the CPython Interpreter**
**http://dabeaz.blogspot.com/2009/08/inside-inside-python-gil-presentation.html**
**Reference counting, atomic instructions, and strategies to improve python without removing the GIL**
http://renesd.blogspot.com/2009/12/python-gil-unladen-swallow-reference.html
**Discussion of the original attempt to remove the python GIL**
**http://mail.python.org/pipermail/python-dev/2001-August/017099.html**

**Python 2.6.4 source**
http://www.python.org/ftp/python/2.6.4/Python-2.6.4.tar.bz2
**New GIL**
svn co http://svn.python.org/projects/sandbox/trunk/newgil/
http://mail.python.org/pipermail/python-dev/2009-October/093321.html
http://www.mail-archive.com/python-dev@python.org/msg46351.html