Python threads: Dive into GIL!

PyCon 2011

Pune Sept 16-18

Vishal Kanaujia and Chetan Giridhar

Summary

- Benefit of multi-threaded application grows with ubiquity of multi-core architecture that potentially can simultaneously run multiple threads of execution.
- Python supports multi-threaded applications and developers are flocking to realize the assured gain of multiple cores with threaded applications.
- Unfortunately, Python has significant bottleneck for multi-threading.

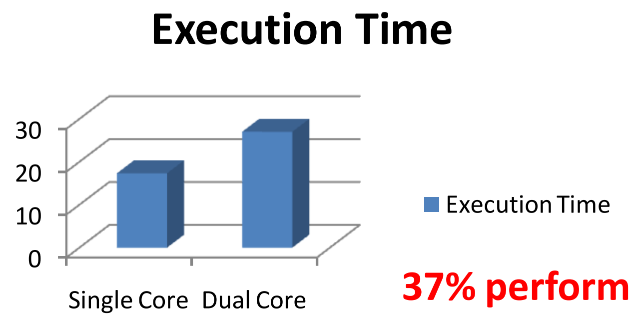
Summary…

- Any thread in CPython interpreter requires a special lock (GIL) which results in serial, rather than parallel execution of multi-threaded applications, irrespective of cores availability and design techniques.
- This talk focuses on the problem, dissects the root cause and its implications.

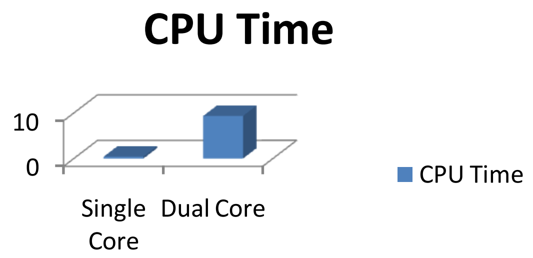
A jaw dropping example!

- A simple python program – two functions()
  - Count(n) -> iterates from n to 0.
  - Add(n) -> adds n, n-1, n-2,..,1
- Call them as two different threads on:
  - Single Core
  - Dual Core

Python v2.7	Execution Time	futex	CPU Time	Total Calls
Single Core	17.337318	7	0.404803	849
Dual Core	27.015506	38	9.344626	879



37% performance dip



Increase in CPU Time by ~9 secs.

Threads: Fundamentals

- Fundamental to a multi-tasking application
- Smallest possible, independent unit of execution
- Light weight processes (resource sharing inclusing address space)
- Concurrent execution
  - Uni-core processor: Single thread at a time; Time division multiplexing
  - Multi-core processor: Threads run at the same time
- CPU bound and I/O bound

Python Threads

- Real system threads (POSIX/ Windows threads)
- Python VM has no intelligence of thread management (priorities, pre-emption, and so on)
- Native operative system supervises thread scheduling
- Python interpreter just does the per-thread bookkeeping.

Python threads: internals

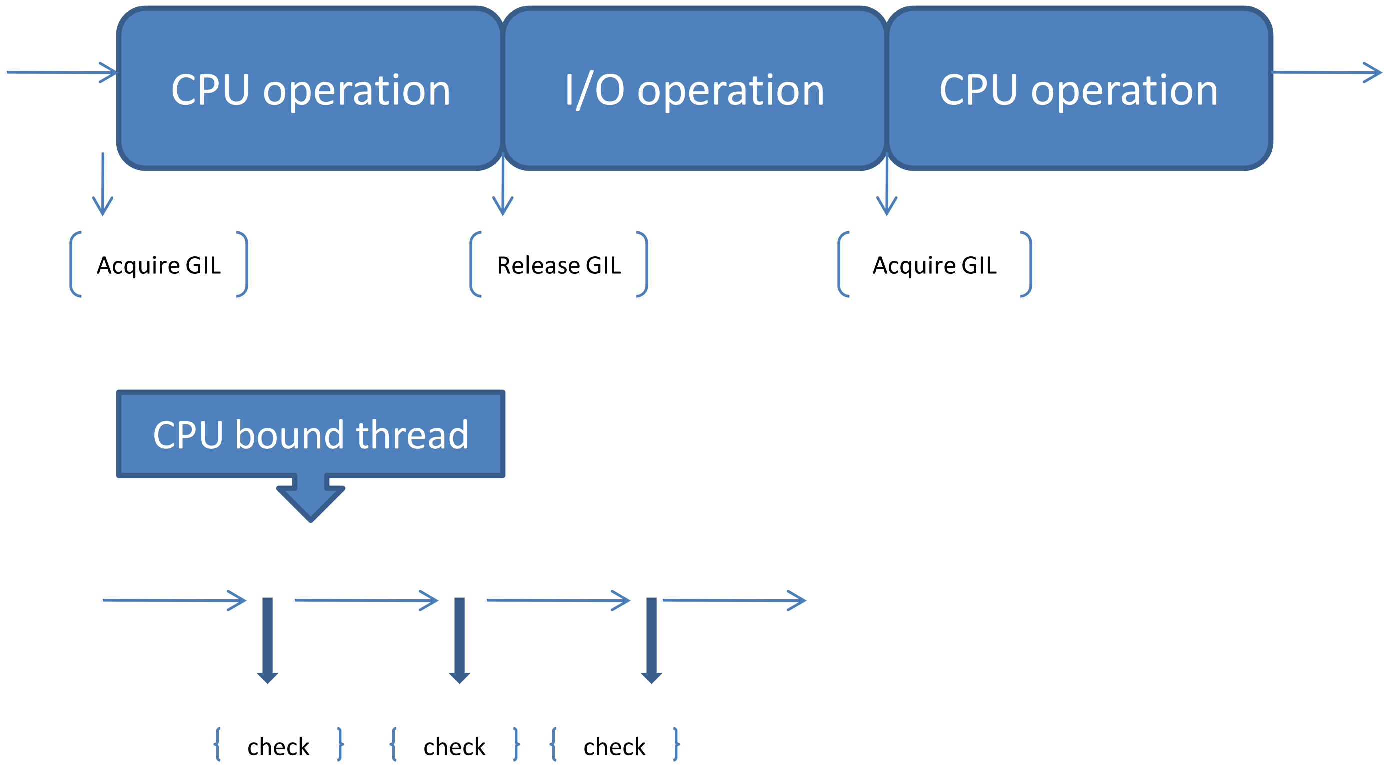
- Only one thread can be active in Python interpreter
- Each ‘running’ thread requires exclusive access to data structures in Python interpreter
- Global interpreter lock (GIL) provides this exclusive synchronization
- This lock is necessary mainly because CPython's memory management is *not* thread-safe.
- **Result**
  - A thread waits if another thread is holding the GIL, even on a multi-core processor! So, threads run sequentially, instead of parallel!

Python threads

- How do Python manages GIL?
  - Python interpreter *regularly* performs a check

- A check is done after ‘n’ ticks.
  - It maps to ‘n’ number of Python VM’s byte-code instructions
  - A global counter; Ticks decrement as a thread executes
- As soon as ticks reach zero:
  - the active thread ***releases and reacquires*** the GIL
  - Signal handling (only in the main thread)
- Effectively, ticks dictate allowed CPU time-slice available to a thread
- Is independent of host/native OS scheduling
- Can be set with sys.setcheckinterval(*interval*)

Python thread: internals



GIL: Details and Bottleneck

- GIL is a conditional variable.
- What goes behind the scene?
  - If GIL is unavailable, a thread goes to sleep and wait.
  - At every ‘check’, a thread release the GIL, and tries to re-acquire
- GIL release is accompanied with a request to host OS to signal all waiting threads
- Regular GIL unlock, thread signaling, wake-up, and GIL relock are an expensive series of operations
- **Threads effectively run in the serial order**

GIL: Battle in multi-cores

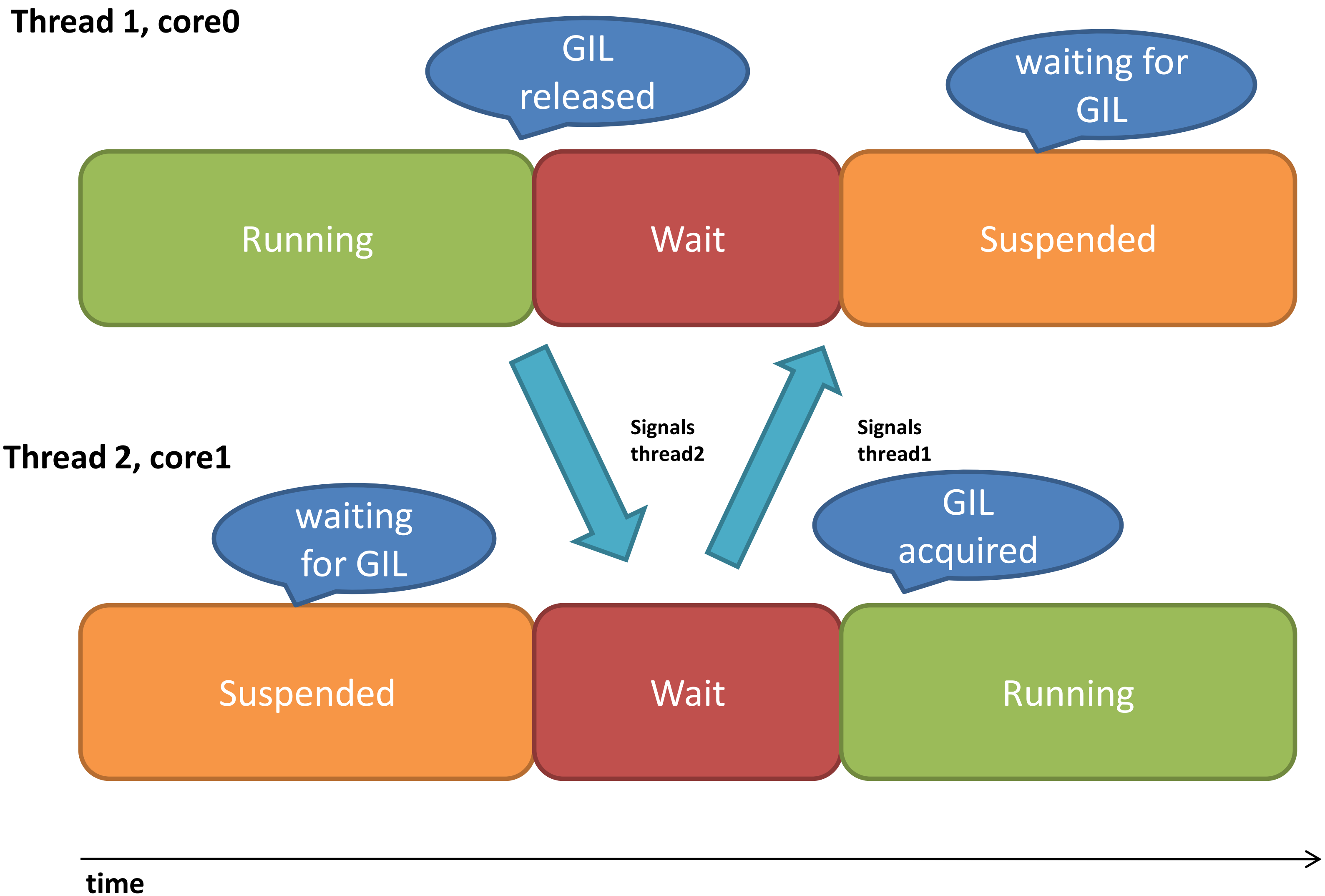
- Unlike single core, multiple cores allows the host OS to schedule many threads concurrently
- A thread that had just released the GIL, will send a signal to waiting threads (through host OS) and is ready to acquire the GIL again!
- This is a GIL contention among all threads

GIL: Battle continues…

- There is considerable time lag of
  - Communication
  - Signal-handling
  - Thread wake-up
  - and acquire GIL
- These factors along with cache-hotness of influence new GIL owner which is usually the recent owner!
- In a [CPU,I/O]-bound mixed application, if the previous owner happens to be a CPU-bound thread, I/O bound thread starves!
  - Since I/O bound threads are preferred by OS over CPU-bound thread; Python presents a priority inversion on multi-core systems.

New GIL: Python 3.2

- Tries to avoid GIL battle. How?
- Regular “check” are discontinued and replaced with a time-out.
    - Default time-out= 5ms
    - Configurable through sys.setswitchinterval()
- For every time-out, current GIL holder, is forced to release it, signals the waiting threads and, *waits for a signal from the new owner of GIL*.
    - A thread does not compete for GIL in succession
- A sleeping thread wakes up, acquires the GIL, and signals the last owner.
- New GIL ensures that every thread gets a chance to run (on a multi-core system)

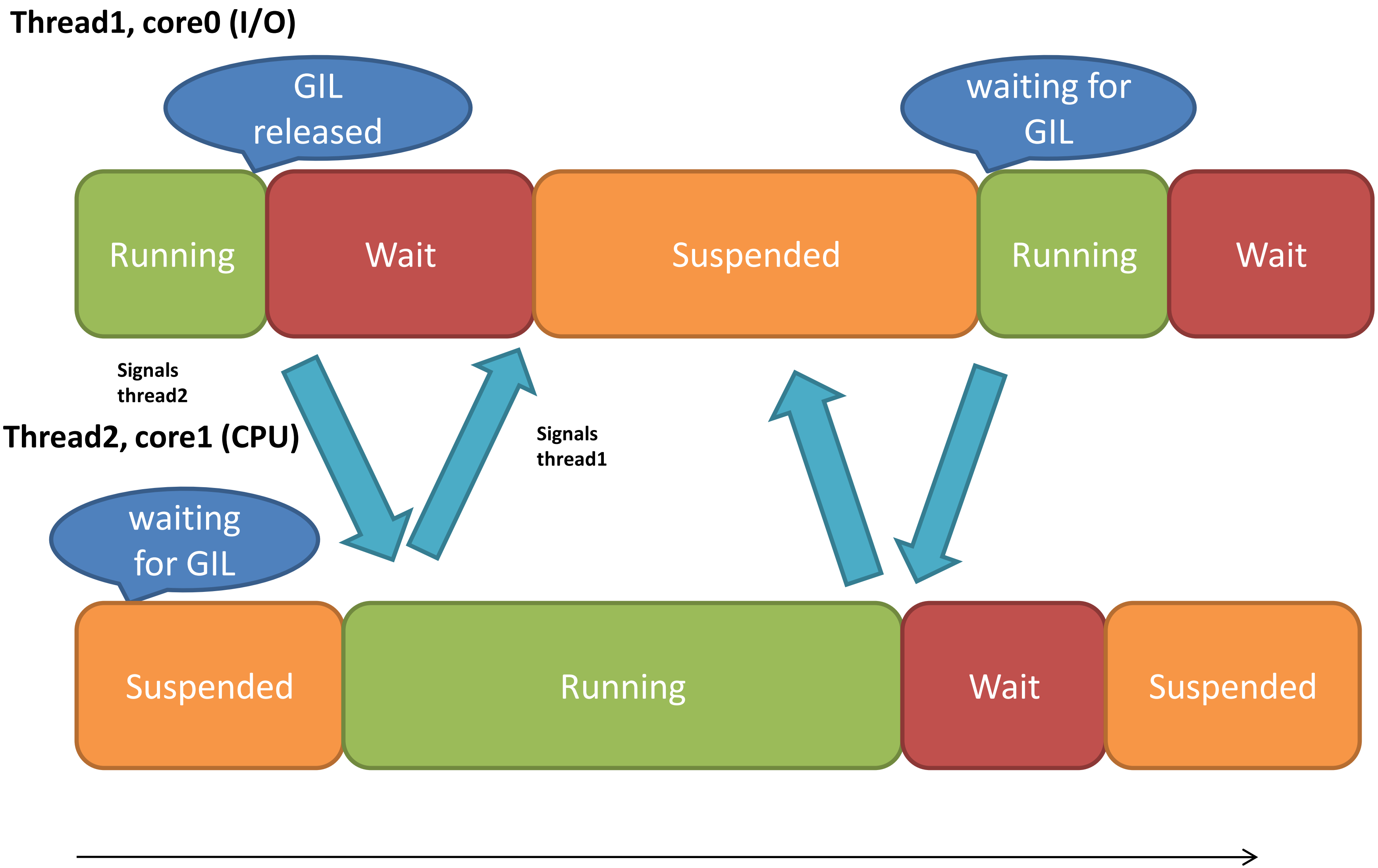

Thread 1, core0
GIL released
waiting for GIL
Running
Wait
Suspended
Signals thread2
Signals thread1
Thread 2, core1
waiting for GIL
GIL acquired
Suspended
Wait
Running
time

Python v3.2: What's good?

- More responsive threads
- Less overhead, lower lock contention
- No GIL battle
- All iz well☺

New GIL: All is not well

- Convoy effect- observed in an application comprising I/O-bound and CPU-bound threads
- A side-effect of an optimization in Python interpreter
  - Release the GIL before executing an I/O service (read, write, send, recv calls)
- When an I/O thread releases the GIL, another ‘runnable’ CPU bound thread can acquire it (remember we are on multiple cores).
- It leaves the I/O thread waiting for another time-out (5ms)!
- Once CPU thread releases GIL, I/O thread acquires and release it again
- This cycle goes on => performance suffers ☹


Thread1, core0 (I/O)
GIL released
waiting for GIL
Running
Wait
Suspended
Running
Wait
Signals thread2
Thread2, core1 (CPU)
Signals thread1
waiting for GIL
Suspended
Running
Wait
Suspended

Convoy effect

- Adversely impacts an I/O thread, if application has a CPU thread(s)
- Voluntary relinquish of GIL proves fatal for I/O thread’s performance
- We performed following tests with Python3.2:
    - CPU thread spends less than few seconds (<10s)!

I/O thread with CPU thread	I/O thread without CPU thread
97 seconds	23 seconds

Convoy effect: Python v2?

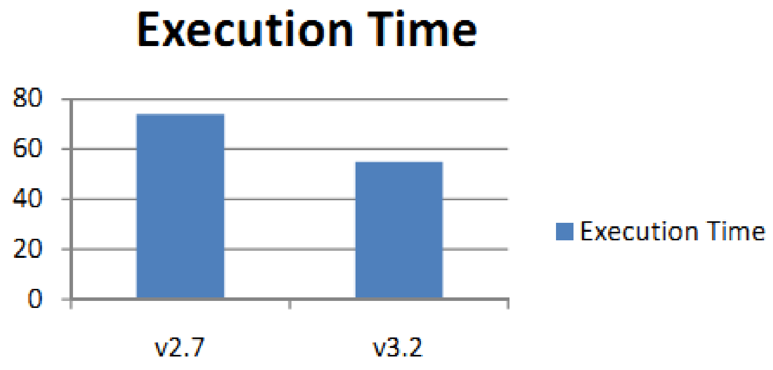
- Convoy effect holds true for Python v2 also
- The smaller interval of ‘check’ saves the day!
  - I/O threads don’t have to wait for a longer time (5 m) for CPU threads to finish
  - Should choose the setswitchtime() interval wisely
- The effect is not so visible in Python v2.0

Comparing : Python2.7 and Python3.2

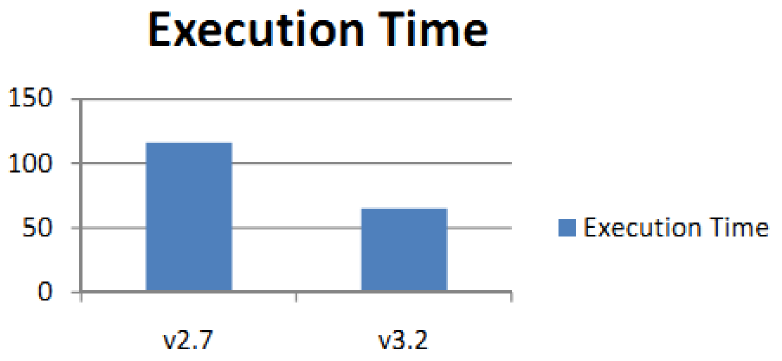
Python 2.7	Execution Time
Single Core	74
Dual Core	116

Python 3.2	Execution Time
Single Core	55
Dual Core	65

On Single Core



On Dual Core



Solving GIL problems

- Thought #1: reduce the waiting time interval between threads.
  - Caveat: increases the overhead of context switching between threads
- Thought #2: implement GIL with C API extensions
  - Caveat: Lot of rework involved
- Thought #3: allow running of I/O threads with GIL if they are not blocking other threads.
  - Caveat: to be analyzed

Jython: GIL

- Jython is free of GIL
- It can fully exploit multiple cores, as per our experiments
- Experiments with Jython2.5
  – Run with two CPU thread in tandem

Jython2.5	Execution time	User time
Single core	38	0.652
Dual core	32	1.524

- Experiment shows positive performance improvement on multi-core system

Conclusion

- Multi-core systems are becoming ubiquitous
- Python application should exploit this abundant power
- Python inherently suffers the GIL limitation
- An intelligent awareness of Python interpreter behavior is helpful in developing multi-threaded applications
- Understand and use ☺

References

- http://wiki.python.org/moin/GlobalInterpreterLock
- http://docs.python.org/c-api/init.html#threads
- http://docs.python.org

Backup slides

Python: GIL

- A thread needs GIL before updating Python objects, calling C/Python API functions
- Concurrency is emulated with regular ‘checks’ to switch threads
- Applicable to only CPU bound thread
- A blocking I/O operation implies relinquishing the GIL
  - ./Python2.7.5/Include/ceval.h

    Py_BEGIN_ALLOW_THREADS

    Do some blocking I/O operation ...

    Py_END_ALLOW_THREADS
- Python file I/O extensively exercise this optimization

GIL: Internals

- The function Py_Initialize() creates the GIL
- A thread create request in Python is just a pthread_create() call
- ../Python/ceval.c
- static PyThread_type_lock interpreter_lock = 0; /* This is the GIL */
- o) thread_PyThread_start_new_thread: we call it for "each" user defined thread.
- calls PyEval_InitThreads() -> PyThread_acquire_lock() {}

GIL: in action

- Each CPU bound thread requires GIL
- ‘ticks count’ determine duration of GIL hold
- new_threadstate() -> tick_counter
- We keep a list of Python threads and each thread-state has its tick_counter value
- As soon as tick decrements to zero, the thread release the GIL.

GIL: Details

- thread_PyThread_start_new_thread() ->
- void PyEval_InitThreads(void)
- {
-     if (interpreter_lock)
-         return;
-     interpreter_lock = PyThread_allocate_lock();
-     PyThread_acquire_lock(interpreter_lock, 1);
-     main_thread = PyThread_get_thread_ident();
- }