Python threads: Dive into GIL!

PyCon India 2011
Pune, Sept 16-18

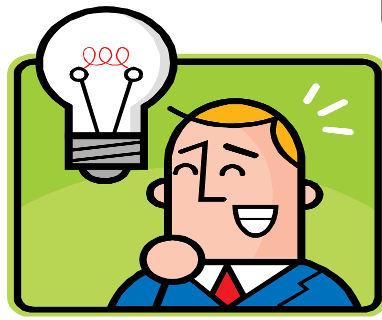
Vishal Kanaujia & Chetan Giridhar

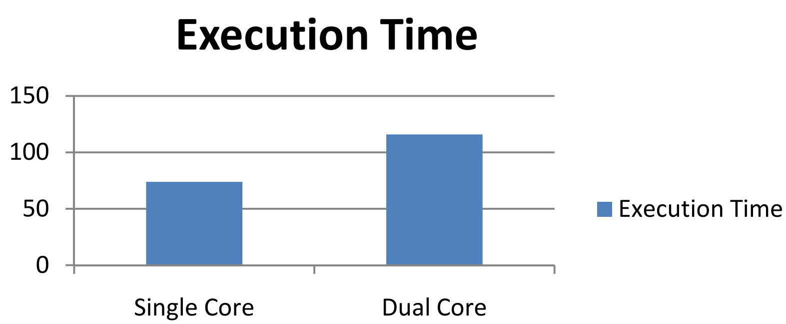
Python: A multithreading example

```
__author__ = "Chetan Giridhar, Vishal Kanaujia"
__date__ = "$Aug 31, 2011 09:37:00$"

from datetime import datetime
import threading
import random

def cpu(n):
    while n>0:
        n-=1
        (random.uniform(1, 10000))/(random.uniform(1, 100000))
        (random.uniform(1, 10000))*(random.uniform(1, 100000))


iterations = 12000000

startTime  = datetime.now()

thread1 = threading.Thread(target=cpu, args=(iterations,))
thread2 = threading.Thread(target=cpu,  args=(iterations,))

thread1.start()
thread2.start()

thread1.join()
thread2.join()

endTime = datetime.now()

diffTime = endTime - startTime
print diffTime
```

Setting up the context!!

Hmm…My threads should be twice as faster on dual cores!





Python v2.7	Execution Time
Single Core	74 s
Dual Core	116 s

57 % dip in Execution Time on dual core!!

Getting into the problem space!!

Python v2.7

GIL

Python Threads

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

What’s wrong with Py threads?

- Each ‘running’ thread requires exclusive access to data structures in Python interpreter
- Global interpreter lock (GIL) provides the synchronization (bookkeeping)
- GIL is necessary, mainly because CPython's memory management is *not* thread-safe.

GIL: Code Details

- A thread create request in Python is just a pthread_create() call
- The function Py_Initialize() creates the GIL
- GIL is simply a synchronization primitive, and can be implemented with a semaphore/ mutex.

  ../Python/ceval.c

  static PyThread_type_lock interpreter_lock = 0; / This is the GIL */*
- A "runnable" thread acquires this lock and start execution

GIL Management

- How does Python manages GIL?
  – Python interpreter *regularly* performs a check on the running thread
  – Accomplishes thread switching and signal handling

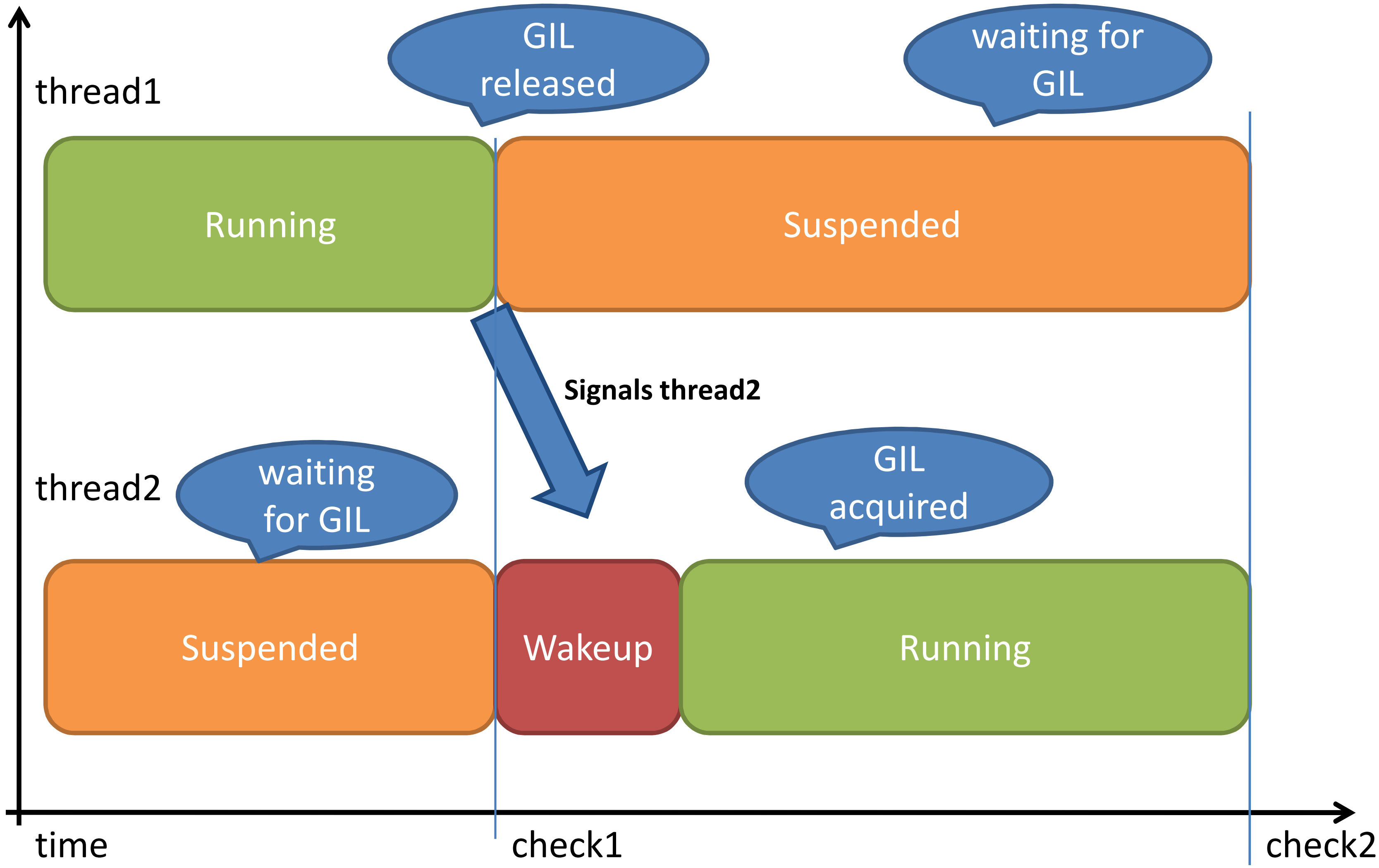
- What is a “check”?
  - A counter of ticks; ticks decrement as a thread runs
  - A tick maps to a Python VM’s byte-code instructions
  - Check interval can be set with sys.setcheckinterval (*interval*)
  - Check dictate CPU time-slice available to a thread

GIL Management: Implementation

- Involves two global varibales:
  - *PyAPI_DATA(volatile int) _Py_Ticker;*
  - *PyAPI_DATA(int) _Py_CheckInterval;*
- As soon as ticks reach zero:
  - Refill the ticker
  - active thread ***releases** the GIL*
  - *Signals sleeping threads to wake up*
  - *Everyone competes for GIL*

```
File: ../ceval.c

PyEval_EvalFrameEx () {
        --_Py_Ticker;  /* Decrement ticker */
       /* Refill it */
        _Py_Ticker = _Py_CheckInterval;
    if (interpreter_lock) {
       PyThread_release_lock(interpreter_lock);

            /* Other threads may run now */

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

Two CPU bound threads on single core machine

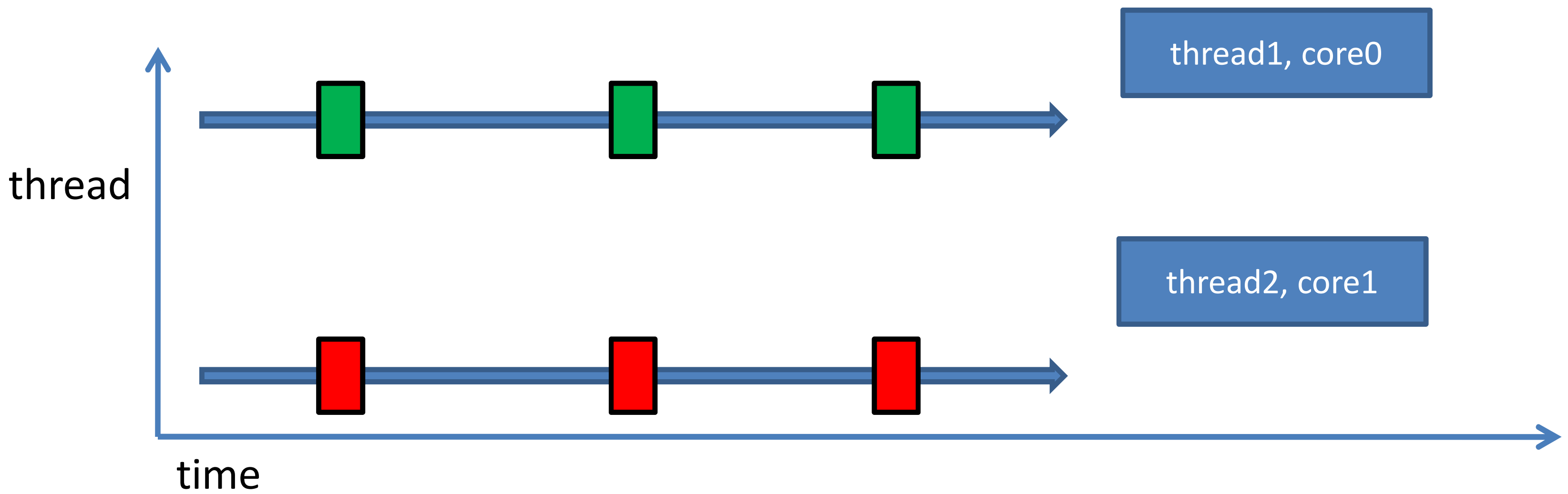


GIL impact

- There is considerable time lag with
  - Communication (Signaling)
  - Thread wake-up
  - GIL acquisition
- GIL Management: Independent of host/native OS scheduling
- **Result**
  - Significant overhead
  - Thread waits if GIL in unavailable
  - Threads run sequentially, rather than concurrently

Try Ctrl + C. Does it stop execution?

Curious case of multicore system



- Conflicting goals of OS scheduler and Python interpreter
- Host OS can schedule threads concurrently on multi-core
- GIL battle

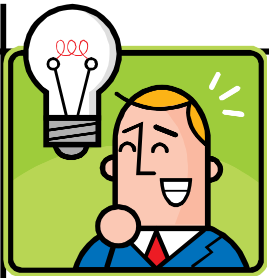
The ‘Priority inversion’

- In a [CPU, I/O]-bound mixed application, I/O bound thread may starve!
- “cache-hotness” may influence the new GIL owner; usually the recent owner!
- Preferring CPU thread over I/O thread
- Python presents a *priority inversion* on multi-core systems.

Understanding the new story!!

Python v3.2

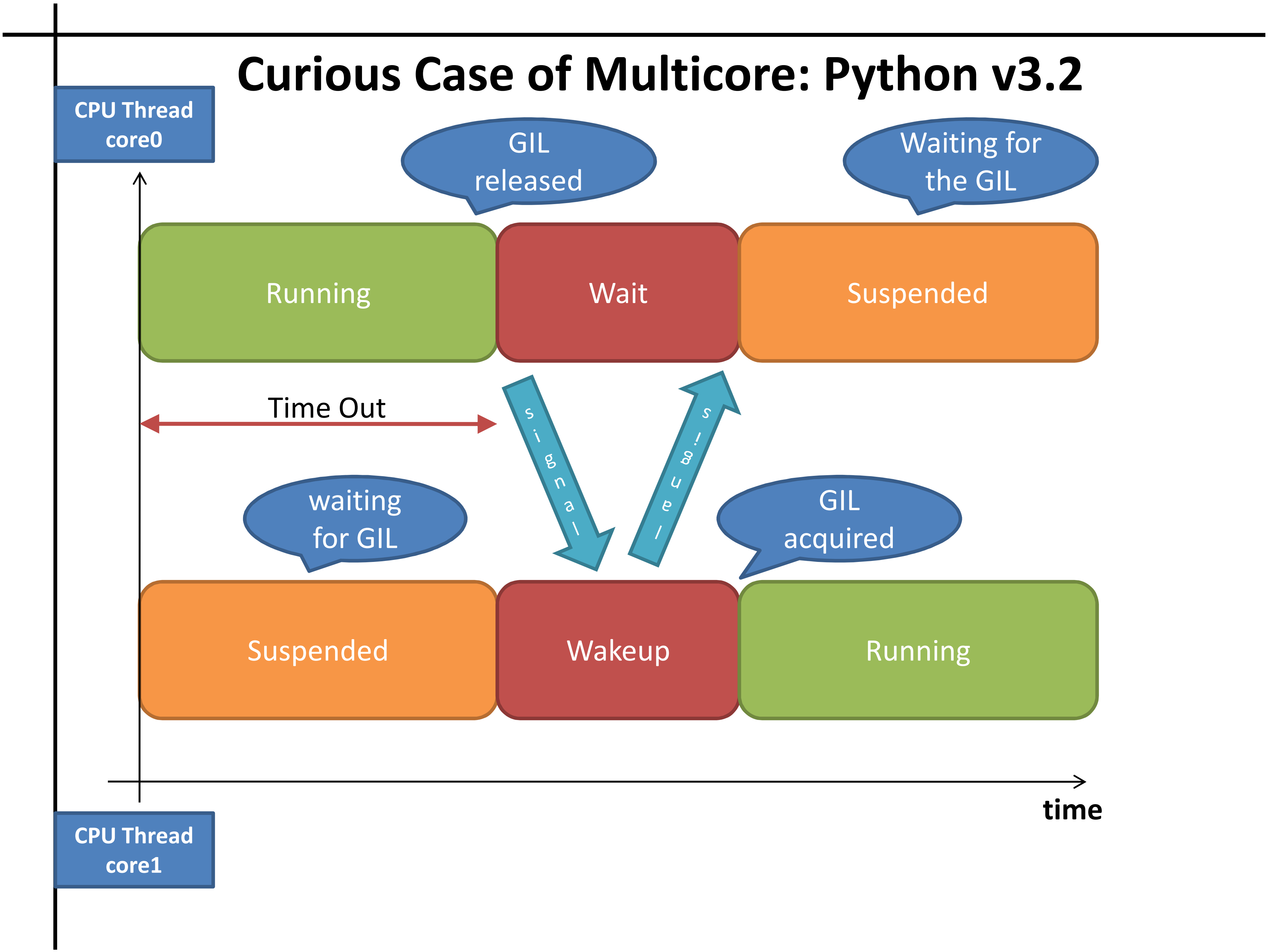
GIL



New GIL: Python v3.2

- Regular “check” are discontinued
- We have new time-out mechanism.
  - Default time-out= 5ms
  - Configurable through sys.setswitchinterval()
- For every time-out, the current GIL holder is forced to release GIL
- It then signals the other waiting threads
- Waits for a signal from new GIL owner (acknowledgement).
- A sleeping thread wakes up, acquires the GIL, and signals the last owner.

Curious Case of Multicore: Python v3.2



Positive impact with new GIL

- Better GIL arbitration
  - Ensures that a thread runs only for 5ms
- Less context switching and fewer signals
- Multicore perspective: GIL battle eliminated!
- More responsive threads (fair scheduling)
- All iz well☺

I/O threads in Python

- An interesting optimization by interpreter
  - I/O calls are assumed blocking
- Python I/O extensively exercise this optimization with file, socket ops (e.g. read, write, send, recv calls)

./Python3.2.1/Include/ceval.h

```
    Py_BEGIN_ALLOW_THREADS
        Do some blocking I/O operation ...
    Py_END_ALLOW_THREADS
```

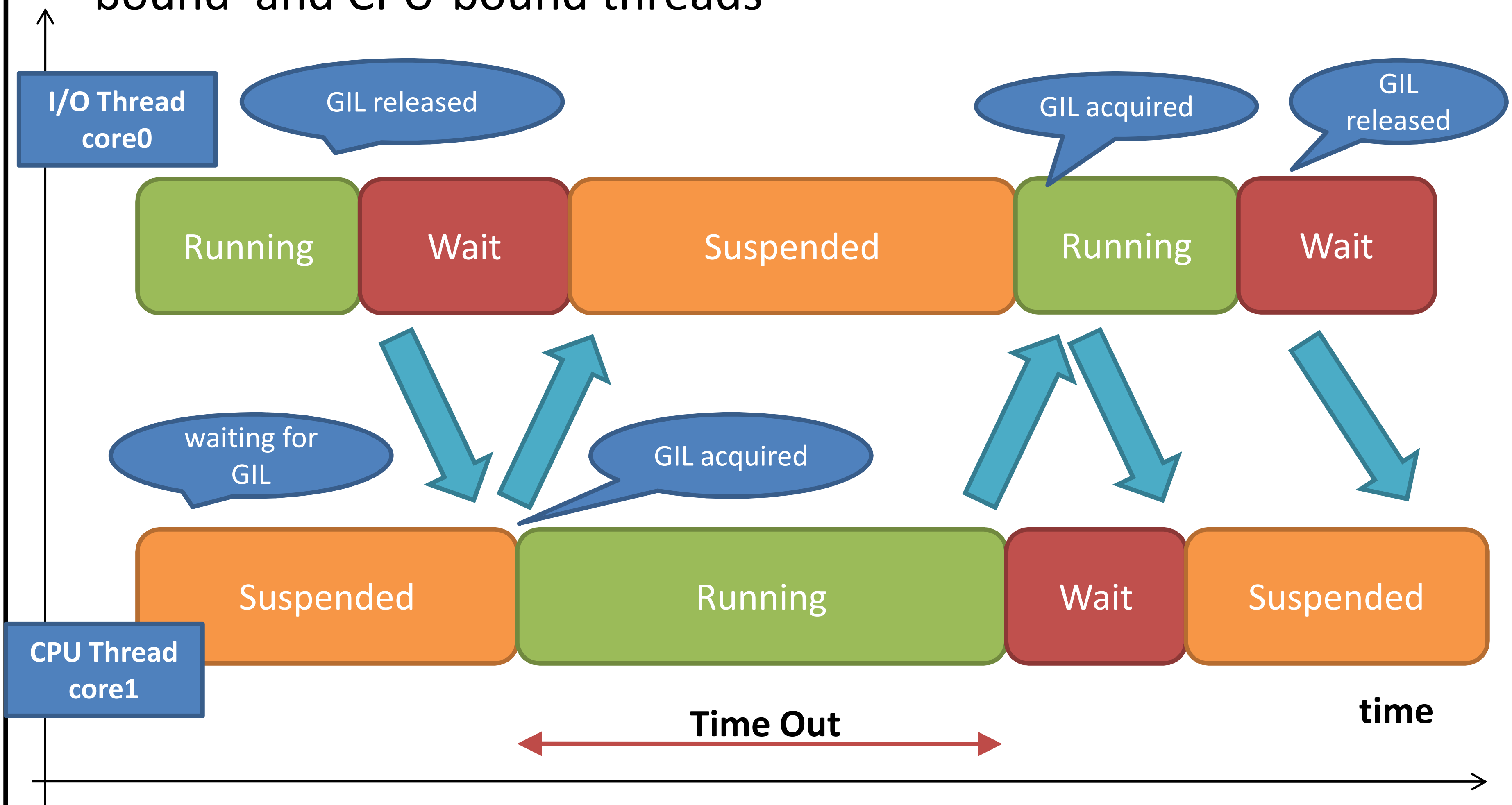
- I/O thread always releases the GIL

Convoy effect: Fallout of I/O optimization

- 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 (default: 5ms)!
- Once CPU thread releases GIL, I/O thread acquires and releases it again
- This cycle goes on => performance suffers ☹

Convoy "in" effect

Convoy effect- observed in an application comprising I/O-bound and CPU-bound threads



Performance measurements!

- Curious to know how convoy effect translates into performance numbers
- We performed following tests with Python3.2:
  - An application with a CPU and a I/O thread
  - Executed on a dual core machine
  - CPU thread spends less than few seconds (<10s)!

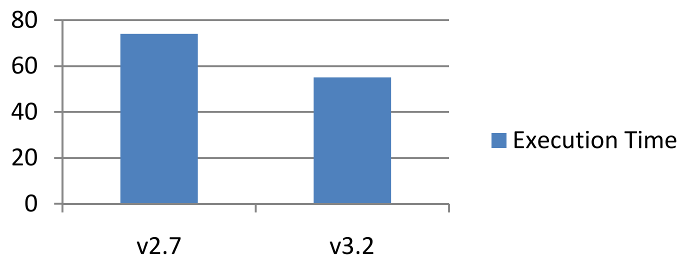
I/O thread with CPU thread	I/O thread without CPU thread
97 seconds	23 seconds

Comparing: Python 2.7 & Python 3.2

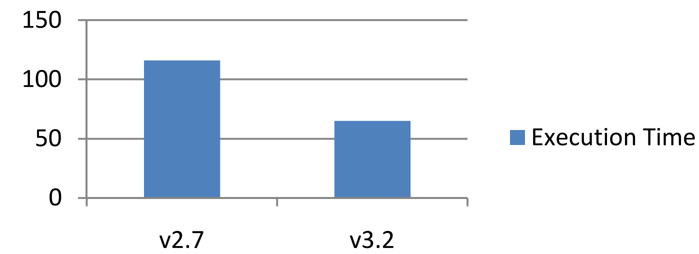
Python v2.7	Execution Time
Single Core	74 s
Dual Core	116 s

Python v3.2	Execution Time
Single Core	55 s
Dual Core	65 s

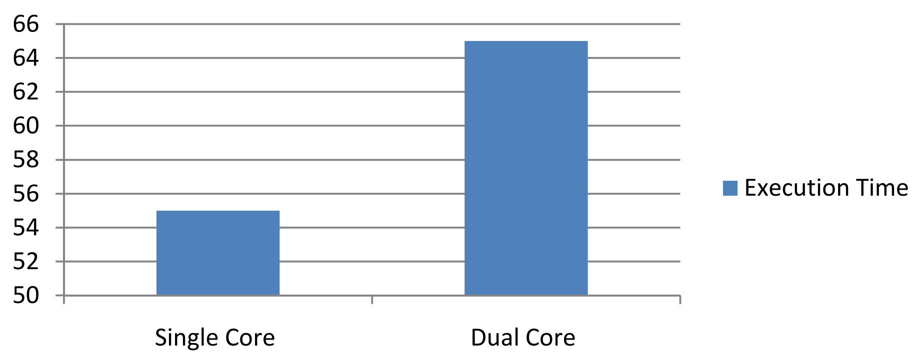
Execution Time – Single Core



Execution Time – Dual Core



Execution Time – Python v3.2



Performance dip still observed in dual cores ☹

Getting into the problem space!!

Python
v2.7 / v3.2

GIL

Solution
space!!

GIL free world: Jython

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

Jython2.5	Execution time
Single core	44 s
Dual core	25 s

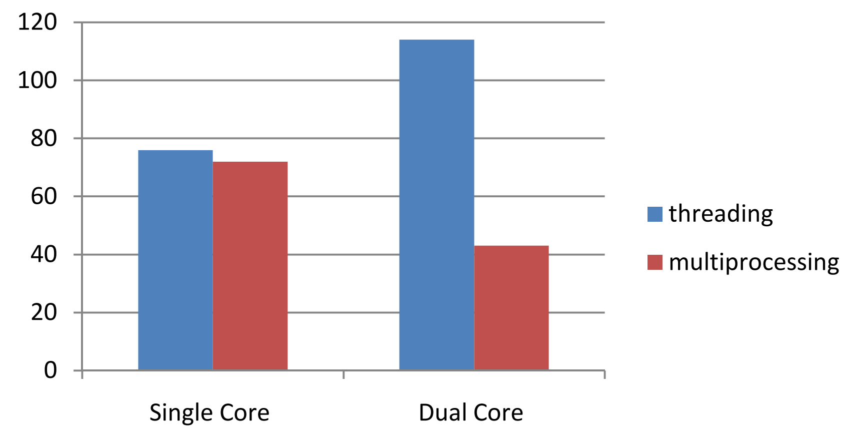
- Experiment shows performance improvement on a multi-core system

Avoiding GIL impact with multiprocessing

- multiprocessing — Process-based “threading” interface
- “multiprocessing” module spawns a new Python interpreter instance for a process.
- Each process is independent, and GIL is irrelevant;
  - Utilizes multiple cores better than threads.
  - Shares API with “threading” module.

Python v2.7	Single Core	Dual Core
threading	76 s	114 s
multiprocessing	72 s	43 s

Cool! 40 % improvement in Execution Time on dual core!! ☺



Conclusion

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

Questions

Thank you for your time and attention ☺

- Please share your feedback/ comments/ suggestions to us at:
- cjgiridhar@gmail.com , http://technobeans.com
- vishalkanaujia@gmail.com, http://freethreads.wordpress.com

References

- Understanding the Python GIL, http://dabeaz.com/talks.html
- GlobalInterpreterLock, http://wiki.python.org/moin/GlobalInterpreterLock
- Thread State and the Global Interpreter Lock, http://docs.python.org/c-api/init.html#threads
- Python v3.2.2 and v2.7.2 documentation, http://docs.python.org/
- Concurrency and Python, http://drdobbs.com/open-source/206103078?pgno=3

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();
}
```

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 setswitchinterval() wisely
- The effect is not so visible in Python v2.0

Stackless Python

- A different way of creating threads: Microthreads!
- No improvement from multi-core perspective
- Round-robin scheduling for “tasklets”
- Sequential execution ☹