User space code in Python

```
# Python program to illustrate the concept
# of threading
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))

    # print name of main thread
    print("Main thread name: {}".format(threading.main_thread().name))

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    # starting threads
    t1.start()
    t2.start()

    # wait until all threads finish
    t1.join()
    t2.join()
```

Run on IDE

```
ID of process running main program: 11758
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 11758
Task 2 assigned to thread: t2
ID of process running task 2: 11758
```

Python Interpreter internally calls pthread APIs

https://github.com/enthought/Python-2.7.3/blob/master/Python/thread_pthread.h
https://github.com/python/cpython/blob/master/Python/thread.c

Branch: master ▾ Python-2.7.3 / Python / thread_pthread.h Find file | Copy path

cournape Python 2.7.3. 69fe0ff on 21 Dec 2013

1 contributor

506 lines (420 sloc) | 13 KB Raw | Blame | History

```
1
2   /* Posix threads interface */
3
4   #include <stdlib.h>
5   #include <string.h>
6   #if defined(__APPLE__) || defined(HAVE_PTHREAD_DESTRUCTOR)
7   #define destructor xxdestructor
8   #endif
9   #include <pthread.h>
```

```
33  /* for safety, ensure a viable minimum stacksize */
34  #define THREAD_STACK_MIN        0x8000  /* 32kB */
```

```
159 long
160 PyThread_start_new_thread(void (*func)(void *), void *arg)
161 {
162     pthread_t th;
163     int status;
164 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
165     pthread_attr_t attrs;
166 #endif
167 #if defined(THREAD_STACK_SIZE)
168     size_t      tss;
169 #endif
170
171     dprintf(("PyThread_start_new_thread called\n"));
172     if (!initialized)
173         PyThread_init_thread();
174
175 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
176     if (pthread_attr_init(&attrs) != 0)
177         return -1;
178 #endif
179 #if defined(THREAD_STACK_SIZE)
180     tss = (_pythread_stacksize != 0) ? _pythread_stacksize
181                                      : THREAD_STACK_SIZE;
182     if (tss != 0) {
183         if (pthread_attr_setstacksize(&attrs, tss) != 0) {
184             pthread_attr_destroy(&attrs);
185             return -1;
186         }
187     }
188 #endif
189 #if defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
190     pthread_attr_setscope(&attrs, PTHREAD_SCOPE_SYSTEM);
191 #endif
192
193     status = pthread_create(&th,
194 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
195                             &attrs,
196 #else
197                             (pthread_attr_t*)NULL,
198 #endif
199                             (void* (*)(void *))func,
200                             (void *)arg
201                             );
```

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.

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!)

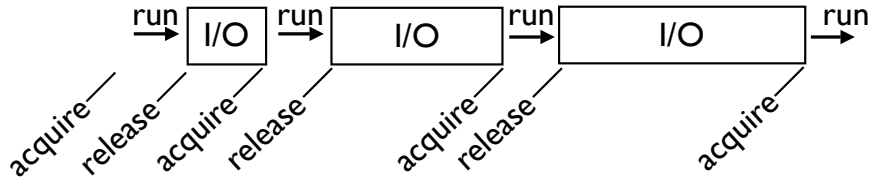
- And if I disable one of the CPU cores, why does the threaded performance get better?

  Threaded : 38.0s

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

GIL Behavior

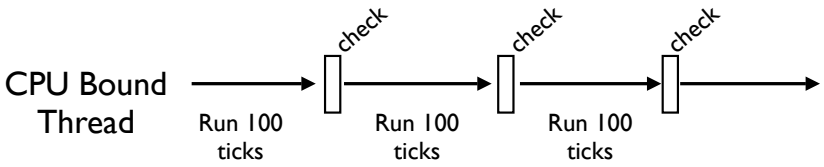
- It's simple : threads hold the GIL when running
- However, they release it when blocking for I/O



- 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

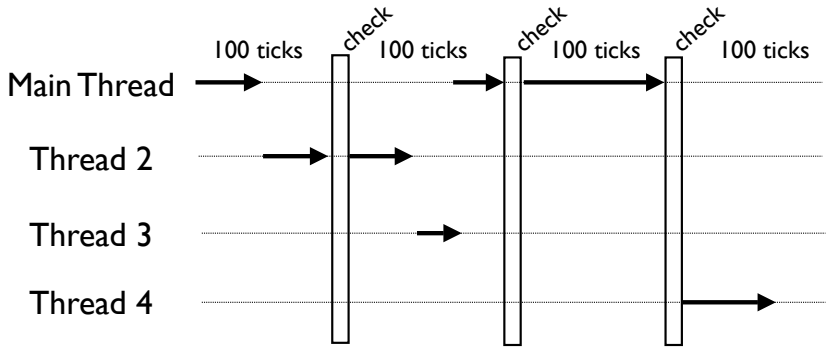
- To deal with CPU-bound threads that never perform any I/O, the interpreter periodically performs a "check"
- By default, every 100 interpreter "ticks"



- sys.setcheckinterval() changes the setting

The Check Interval

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



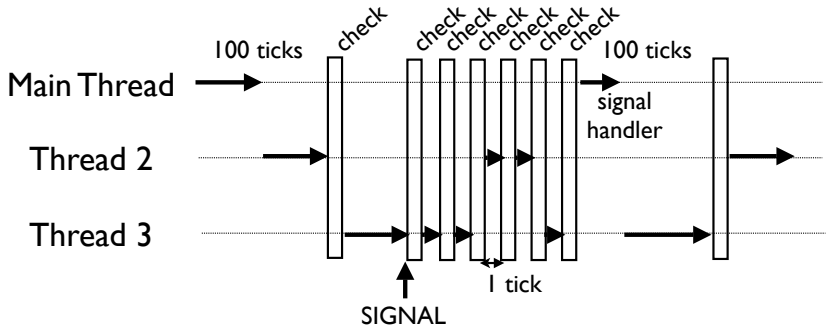
- A "check" is simply made every 100 "ticks"

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).

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)

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

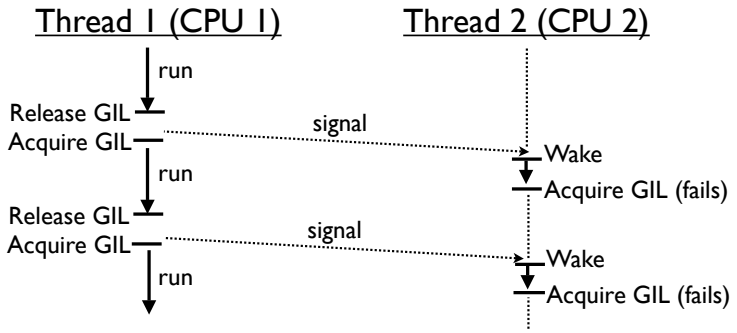
- Sequential Execution (OS-X, 1 CPU)
  - 736 Unix system calls
  - 117 Mach System Calls
- Two CPU-bound threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls
- Yow! Look at that last figure.

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

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)