Global Interpreter Lock

Episode I - Break the Seal

Tzung-Bi Shih
<penvirus@gmail.com>

Introduction

- Global Interpreter Lock[1]
  - giant lock[2]
- GIL in CPython[5] protects:
  - interpreter state, thread state, ...
  - reference count
  - “a guarantee”
- other implementations
  - fine-grained lock[3]
  - lock-free[4]

some CPython features and extensions depend on the agreement

GIL over Multi-Processor[6]

We want to produce efficient program.

To achieve higher throughputs, we usually divide a program into several independent logic segments and execute them simultaneously over MP architecture by leveraging multi-threading technology.

Unfortunately, only one of the threads gets executed at a time if they compete for a same GIL.

Some people are working on how to remove the giant lock which shall be a difficult job[7][8][9]. Before the wonderful world comes, we will need to learn how to live along with GIL well.

Brainless Solution

multi-process

- Embarrassingly parallel[10]
  - no dependency between those parallel tasks
- IPC[11]-required parallel task
  - share states with other peers
- Examples:
  - multiprocessing[12], pp[13], pyCSP[14]

Example[15]

multiprocessing: process pool

```
 1 import os
 2 from multiprocessing import Pool
 3
 4 def worker(i):
 5     print 'pid=%d ppid=%d i=%d' % (os.getpid(), os.getppid(), i)
 6
 7 print 'pid=%d' % os.getpid()
 8 pool = Pool(processes=4)
 9 pool.map(worker, xrange(10))
10 pool.terminate()
```

```
Round 1:
pid=11326
pid=11327 ppid=11326 i=0
pid=11328 ppid=11326 i=1
pid=11328 ppid=11326 i=3
pid=11329 ppid=11326 i=2
pid=11329 ppid=11326 i=5
pid=11329 ppid=11326 i=6
pid=11329 ppid=11326 i=7
pid=11329 ppid=11326 i=8
pid=11327 ppid=11326 i=4
pid=11328 ppid=11326 i=9
```

```
Round 2:
pid=11372
pid=11373 ppid=11372 i=0
pid=11373 ppid=11372 i=2
pid=11374 ppid=11372 i=1
pid=11376 ppid=11372 i=3
pid=11374 ppid=11372 i=4
pid=11374 ppid=11372 i=7
pid=11373 ppid=11372 i=6
pid=11376 ppid=11372 i=8
pid=11375 ppid=11372 i=5
pid=11375 ppid=11372 i=9
```

nondeterministic[16]:
the same input, different output

Example

multiprocessing: further observations (1/2)

- Adopts un-named pipe to handle IPC
- Workers are forked when initializing the pool
  - so that workers can “see” the target function (they will share the same memory copy)

=> What if I create the target function after the pool initialized?

```
 1 import os
 2 from multiprocessing import Pool
 3
 4 print 'pid=%d' % os.getpid()
 5 pool = Pool(processes=4)
 6
 7 def worker(i):
 8     print 'pid=%d ppid=%d i=%d' % (os.getpid(), os.getppid(), i)
 9
10 pool.map(worker, xrange(10))
11 pool.terminate()
```

Example

multiprocessing: further observations (2/2)

```
Output:
pid=12093
Process PoolWorker-1:
Process PoolWorker-2:
Traceback (most recent call last):
Process PoolWorker-3:
Traceback (most recent call last):
  File "/usr/lib/python2.7/multiprocessing/process.py", line 258, in _bootstrap
Traceback (most recent call last):
  File "/usr/lib/python2.7/multiprocessing/process.py", line 258, in _bootstrap
...ignored...
AttributeError: 'module' object has no attribute 'worker'
...ignored...
pid=12101 ppid=12093 i=4
pid=12101 ppid=12093 i=5
pid=12101 ppid=12093 i=6
pid=12101 ppid=12093 i=7
pid=12101 ppid=12093 i=8
pid=12101 ppid=12093 i=9
^CProcess PoolWorker-6:
Traceback (most recent call last):
  File "/usr/lib/python2.7/multiprocessing/process.py", line 258, in _bootstrap
    self.run()
  File "/usr/lib/python2.7/multiprocessing/process.py", line 114, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/lib/python2.7/multiprocessing/pool.py", line 102, in worker
    task = get()
  File "/usr/lib/python2.7/multiprocessing/queues.py", line 374, in get
    racquire()
KeyboardInterrupt
```

lost 0~3

process hanging

ctrl+c pressed

worker #6

#1~4 were terminated due to the exception
following workers will be forked

Example

overhead of IPC and GIL battle[17] comparison

```python
import time
from multiprocessing import Process
from threading import Thread
from multiprocessing import Queue as MPQ
from Queue import Queue

MAX = 1000000

def test (w class, q class):
    def worker(queue):
        for i in xrange(MAX):
            queue.put(i)

    q = q_class()
    w = w_class(target=worker, args=(q,))

    begin = time.time()
    w.start()
    for i in xrange(MAX):
        q.get()
    w.join()
    end = time.time()

    return end - begin
```

```python
def test_sthread():
    q = Queue()

    begin = time.time()
    for i in xrange(MAX):
        q.put(i)
        q.get()
    end = time.time()

    return end - begin

print 'mprocess: %.6f' % test_(Process, MPQ)
print 'mthread:  %.6f' % test_(Thread, Queue)
print 'sthread:  %.6f' % test_sthread()
```

```
Output:
mprocess: 14.225408
mthread:  7.759567
sthread:  2.743325
```

→ IPC is the most costly

→ overhead of the GIL battle

API of multiprocessing is similar to threading[18]

Example
pp remote node

of workers

listen to wait remote jobs

```
Server:
$ ppserver.py -w 1 -p 10000 &
[1] 16512
$ ppserver.py -w 1 -p 10001 &
[2] 16514
$ ppserver.py -w 1 -p 10002 &
[3] 16516
$ netstat -nlp
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:10000           0.0.0.0:*               LISTEN      16512/python
tcp        0      0 0.0.0.0:10001           0.0.0.0:*               LISTEN      16514/python
tcp        0      0 0.0.0.0:10002           0.0.0.0:*               LISTEN      16516/python
$ pstree -p $$
bash(11971)-+-ppserver.py(16512)---python(16513)
            |-ppserver.py(16514)---python(16515)
            |-ppserver.py(16516)---python(16517)
            `-pstree(16547)
```

workers

Example

pp local node

pp worker collects stdout

# of workers

computed by local node

```
1  import os
2  import pp
3  import time
4  import random
5
6  print 'pid=%d' % os.getpid()
7
8  def worker(i):
9      print 'pid=%d ppid=%d i=%d' % (os.getpid(), os.getppid(), i)
10     time.sleep(random.randint(1, 3))
11
12 servers = ('127.0.0.1:10000', '127.0.0.1:10001', '127.0.0.1:10002')
13 job_server = pp.Server(1, ppservers=servers)
14
15 jobs = list()
16 for i in xrange(10):
17     job = job_server.submit(worker, args=(i,), modules=('time', 'random'))
18     jobs.append(job)
19
20 for job in jobs:
21     job()
```

```
Output:
pid=16633
pid=16634 ppid=16633 i=0
pid=16513 ppid=16512 i=1
pid=16517 ppid=16516 i=2
pid=16515 ppid=16514 i=3
pid=16513 ppid=16512 i=4
pid=16517 ppid=16516 i=5
pid=16515 ppid=16514 i=6
pid=16634 ppid=16633 i=7
pid=16517 ppid=16516 i=8
pid=16513 ppid=16512 i=9
```

determine the result order (deterministic)

accumulative,
beware of RSIZE of remote node

A pp local node is an execution node too. It dispatches jobs to itself first.

Example

ppserver.py gives some exceptions

```
Exception:
Exception in thread client_socket:
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 810, in __bootstrap_inner
    self.run()
  File "/usr/lib/python2.7/threading.py", line 763, in run
    self.__target(*self.__args, **self.__kwargs)
  File "/usr/local/bin/ppserver.py", line 176, in crun
    ctype = mysocket.receive()
  File "/usr/local/lib/python2.7/dist-packages/pptransport.py", line 196, in receive
    raise RuntimeError("Socket connection is broken")
RuntimeError: Socket connection is broken
```

Don’t worry. Expected.

Release the GIL

- Especially suitable for processor-bound tasks
- Examples:
  - ctypes[19]
  - Python/C extension[20][21]
  - Cython[22]
  - Pyrex[23]

Example
ctypes (1/2)

```python
duration = 10

def internal_busy():
    import time

    count = 0
    begin = time.time()
    while True:
        if time.time() - begin > duration:
            break
        count += 1

    print 'internal_busy(): count = %u' % count

def external_busy():
    from ctypes import CDLL
    from ctypes import c_uint, c_void_p

    libbusy = CDLL('./busy.so')
    busy_wait = libbusy.busy_wait
    busy_wait.argtypes = [c_uint]
    busy_wait.restype = c_void_p

    busy_wait(duration)

print 'two internal busy threads, CPU utilization cannot over 100%'
t1 = threading.Thread(target=internal_busy); t1.start()
t2 = threading.Thread(target=internal_busy); t2.start()
t1.join(); t2.join()

print 'with one external busy thread, CPU utilization gains to 200%'
t1 = threading.Thread(target=internal_busy); t1.start()
t2 = threading.Thread(target=external_busy); t2.start()
t1.join(); t2.join()
```

```c
void busy_wait(unsigned int duration)
{
 uint64_t count = 0;
 time_t begin = time(NULL);

 while(1) {
     if(time(NULL) - begin > duration)
         break;
     count++;
 }

 printf("busy_wait(): count = %" PRIu64 "\n", count);
}
```

consume CPU resource

specify input/output types
(strongly recommended)

Example
ctypes (2/2)

```
Output:
two internal busy threads, CPU utilization cannot over 100%
internal_busy(): count = 12911610
internal_busy(): count = 16578663
with one external busy thread, CPU utilization gains to 200%
internal_busy(): count = 45320393
busy_wait(): count = 3075909775
```

```
Atop Display:
CPU | sys      46% | user     72% | irq       0% | idle     82% | wait       0% |
cpu | sys      26% | user     39% | irq       1% | idle     35% | cpu001 w   0% |
cpu | sys      20% | user     33% | irq       0% | idle     46% | cpu000 w   1% |
```

```
Atop Display:
CPU | sys       1% | user    199% | irq       0% | idle      0% | wait       0% |
cpu | sys       1% | user     99% | irq       0% | idle      0% | cpu000 w   0% |
cpu | sys       0% | user    100% | irq       0% | idle      0% | cpu001 w   0% |
```

Example

Python/C extension (1/3)

```
20 static PyObject *with_lock(PyObject *self, PyObject *args)
21 {
22   unsigned int duration;
23
24   if(!PyArg_ParseTuple(args, "I", &duration))
25        return NULL;
26
27   busy_wait(duration);
28
29   Py_INCREF(Py_None);
30   return Py_None;
31 }
32
33 static PyObject *without_lock(PyObject *self, PyObject *args)
34 {
35   unsigned int duration;
36
37   if(!PyArg_ParseTuple(args, "I", &duration))
38        return NULL;
39
40   PyThreadState *_save;
41   _save = PyEval_SaveThread();
42   busy_wait(duration);
43   PyEval_RestoreThread(_save);
44
45   Py_INCREF(Py_None);
46   return Py_None;
47 }
48
49 static PyMethodDef busy_methods[] = {
50   {"with_lock", with_lock, METH_VARARGS, "Busy wait for a given duration with GIL"},
51   {"without_lock", without_lock, METH_VARARGS, "Busy wait for a given duration without GIL"},
52   {NULL, NULL, 0, NULL}
53 };
54
55 PyMODINIT_FUNC initbusy(void)
56 {
57   if(Py_InitModule("busy", busy_methods) == NULL)
58        return PyErr_SetString(PyExc_RuntimeError, "failed to Py_InitModule");
59 }
```

require an unsigned integer argument (busy duration)

return None

Compilation:

```
$ cat Makefile
busy.so: busy.c
	$(CC) -o $@ -fPIC -shared -I/usr/include/python2.7 busy.c
$ make
```

release the GIL before being busy

exported symbol name

accept positional args.

module name

Example
Python/C extension (2/3)

```
 1 import threading
 2
 3 duration = 10
 4
 5 def internal_busy():
 6     import time
 7
 8     count = 0
 9     begin = time.time()
10     while True:
11         if time.time() - begin > duration:
12             break
13         count += 1
14
15     print 'internal_busy(): count = %u' % count
16
17 def external_busy_with_lock():
18     from busy import with_lock
19
20     with_lock(duration)
21
22 def external_busy_without_lock():
23     from busy import without_lock
24
25     without_lock(duration)
26
27 print 'two busy threads compete for GIL, CPU utilization cannot over 100%'
28 t1 = threading.Thread(target=internal_busy); t1.start()
30 t2 = threading.Thread(target=external_busy_with_lock); t2.start()
32 t1.join(); t2.join()
34
35 print 'with one busy thread released GIL, CPU utilization gains to 200%'
36 t1 = threading.Thread(target=internal_busy); t1.start()
38 t2 = threading.Thread(target=external_busy_without_lock); t2.start()
40 t1.join(); t2.join()
```

linking to the busy.so extension

Example

Python/C extension (3/3)

```
Output:
two busy threads compete for GIL, CPU utilization cannot over 100%
busy_wait(): count = 3257960533
internal_busy(): count = 45524
with one busy thread released GIL, CPU utilization gains to 200%
internal_busy(): count = 48049276
busy_wait(): count = 3271300229
```

```
Atop Display:
CPU | sys       2% | user    100% | irq       0% | idle     99% | wait      0% |
cpu | sys       0% | user    100% | irq       0% | idle      0% | cpu001 w  0% |
cpu | sys       1% | user      0% | irq       0% | idle     99% | cpu000 w  0% |
```

```
Atop Display:
CPU | sys       2% | user    198% | irq       0% | idle      0% | wait      0% |
cpu | sys       0% | user    100% | irq       0% | idle      0% | cpu000 w  0% |
cpu | sys       1% | user     98% | irq       0% | idle      0% | cpu001 w  0% |
```

Cooperative Multitasking

- Only applicable to IO-bound tasks
- Single process, single thread
  - no other thread, no GIL battle
- Executing the code when exactly needed
- Examples:
  - generator[24]
  - pyev[25]
  - gevent[26]

Example

pyev

```
1  import pyev
2  import signal
3  import sys
4
5  def alarm_handler(watcher, revents):
6      sys.stdout.write('.')
7      sys.stdout.flush()
8
9  def timeout_handler(watcher, revents):
10     loop = watcher.loop
11     loop.stop()
12
13 def int_handler(watcher, revents):
14     loop = watcher.loop
15     loop.stop()
16
17 if __name__ == '__main__':
18     loop = pyev.Loop()
19
20     alarm = loop.timer(0.0, 1.0, alarm_handler)
21     alarm.start()
22
23     timeout = loop.timer(10.0, 0.0, timeout_handler)
24     timeout.start()
25
26     sigint = loop.signal(signal.SIGINT, int_handler)
27     sigint.start()
28
29     loop.start()
```

```
Case 1 Output:
...........
```

11 dots

```
Case 2 Output:
..^C
```

```
libev Timer:
(after)|(repeat)|(repeat)|(repeat)|...
```

interval event raised

the example:
after 0.0 second, raise
every 1.0 second, raise
raises 11 times in total

Example

pyev: further observations

```
20      loop.timer(0.0, 1.0, alarm_handler).start()
21
22      loop.start()
```

Output:

```
Exception SystemError: 'null argument to internal routine' in Segmentation fault (core dumped)
```

```
20      timeout = loop.timer(0.0, 1.0, alarm_handler)
21      timeout.start()
22
23      timeout = loop.timer(10.0, 0.0, timeout_handler)
24      timeout.start()
25
26      loop.start()
```

manual of ev[27]:
you are responsible for allocating the memory for your watcher structures

```
20      alarm = loop.timer(0.0, 1.0, alarm_handler)
21      alarm.start()
22      sigint = loop.timer(10.0, 0.0, timeout_handler)
23      sigint.start()
24      sigint = loop.signal(signal.SIGINT, int_handler)
25      sigint.start()
26      loop.start()
```

Output:

```
..........Exception SystemError: 'null argument to internal routine' in Segmentation fault (core dumped)
```

Example
gevent

```
1  import gevent
2  from gevent import signal
3  import signal as o_signal
4  import sys
5
6  if __name__ == '__main__':
7      ctx = dict(stop_flag=False)
8
9      def int_handler():
10         ctx['stop_flag'] = True
11     gevent.signal(o_signal.SIGINT, int_handler)
12
13     count = 0
14     while not ctx['stop_flag']:
15         sys.stdout.write('.')
16         sys.stdout.flush()
17
18         gevent.sleep(1)
19
20         count += 1
21         if count > 10:
22             break
```

```
Case 1 Output:
..........
```

```
Case 2 Output:
..^C
```

Interpreter as an Instance

- Rough idea, not a concrete solution yet
- C program, single process, multi-thread
  - still can share states with relatively low penalty
- Allocate memory space for interpreter context
  - that is, accept an address to put instance context in Py_Initialize()

Conclusion

- How to live along with GIL well?
  - Multi-process
  - Release the GIL
  - Cooperative Multitasking
  - Perhaps, Interpreter as an Instance

References

[1]: http://en.wikipedia.org/wiki/Global_Interpreter_Lock
[2]: http://en.wikipedia.org/wiki/Giant_lock
[3]: http://en.wikipedia.org/wiki/Fine-grained_locking
[4]: http://en.wikipedia.org/wiki/Non-blocking_algorithm
[5]: https://wiki.python.org/moin/GlobalInterpreterLock
[6]: http://en.wikipedia.org/wiki/Multiprocessing
[7]: https://docs.python.org/2/faq/library.html#can-t-we-get-rid-of-the-global-interpreter-lock
[8]: http://www.artima.com/weblogs/viewpost.jsp?thread=214235
[9]: http://dabeaz.blogspot.tw/2011/08/inside-look-at-gil-removal-patch-of.html
[10]: http://en.wikipedia.org/wiki/Embarrassingly_parallel
[11]: http://en.wikipedia.org/wiki/Inter-process_communication
[12]: https://docs.python.org/2/library/multiprocessing.html
[13]: http://www.parallelpython.com/
[14]: https://code.google.com/p/pycsp/
[15]: https://github.com/penvirus/gil1
[16]: http://en.wikipedia.org/wiki/Nondeterministic_algorithm
[17]: http://www.dabeaz.com/python/GIL.pdf
[18]: https://docs.python.org/2/library/threading.html
[19]: https://docs.python.org/2/library/ctypes.html
[20]: https://docs.python.org/2/c-api/
[21]: https://docs.python.org/2/c-api/init.html#releasing-the-gil-from-extension-code
[22]: http://cython.org/
[23]: http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
[24]: http://www.dabeaz.com/coroutines/Coroutines.pdf
[25]: http://pythonhosted.org/pyev/
[26]: http://www.gevent.org/
[27]: http://linux.die.net/man/3/ev