Multiprocessing

Reuven M. Lerner, PhD
reuven@lerner.co.il

Threading

- Thread programming is difficult
  - (That’s why people like me avoid it!)
- But it can be useful
- Python threads are native threads
  - But only one runs at a time
  - So that might not be good for your needs

GIL

- “Global Interpreter Lock”
- Ensures that only one thread runs at a time
- The bane of Python threads
- Starting with Python 3.2, GIL scope was reduced a lot
    - But it’s still something to keep in mind

GIL? Who cares?

- The GIL means that only one thread runs at a time
  - Cannot use multiple CPUs!
  - Less chance of corruption in shared data structures
- Only in CPython — Jython and IronPython don’t have a GIL

Why not remove it?

- Speed — replacements cannot make single-threaded programs slower
- Compatibility — must handle things like __del__ and weak refs
- Ordered destruction
- API compatibility — remain compatible with today's systems

Threading in Python

- Only one thread executes at a time
- Used to be based on interpreter ticks
- Now it's based on a combination of time and requests for resources
- Every 10 bytecodes, or every I/O request, a thread gives up the CPU

Multicore

- Each thread waits to acquires the GIL
- So while you might have 10 threads, each on a separate core — but only one is executing
- Waiting threads try again and again!

Scheduling

- Python doesn't schedule threads!
- Rather, the OS is responsible
- No priorities, pre-emption, etc.

Releasing the GIL

- Voluntary releases if a thread:
  - Sleeps
  - Does I/O
- Involuntary release if a thread sets gil_drop_request and threshold has passed
- sys.getswitchinterval / sys.setswitchinterval

thread module

- Called “_thread” in Python 3
- Lower level, and is discouraged by many

Using thread

```
import thread
def child(tid):
    print('Hello from thread', tid)
def parent():
    i= 0
    while True:
        i += 1
        thread.start_new_thread(child, (i,))
        if raw_input() == 'q':
            break
parent()
```

Starting threads

- thread.start_new_thread
    - Takes a callable and a tuple of parameters
    - Doesn’t return anything useful
- If an exception occurs in the child, a stack trace is printed — but execution of the main thread continues

threading module

- Higher level, encouraged as “your first threading module”
- Works with objects and classes
- Have your class inherit from threading.Thread, write a “run” method
  - Invoke start(), and run() executes in a thread

Using threading

```
#!/usr/bin/env python
import threading
class DoIt(threading.Thread):
    def run(self):
        print "In the thread"
DoIt().start()
DoIt().start()
DoIt().start()
```

Threading information

- `threading.current_thread()`
- `threading.current_thread().name`
- `threading.active_count()`
- `threading.enumerate()`

Daemon threads

- The Python process exits when all threads are done
- Sometimes, you want a thread that doesn't hang the system
- Daemon threads!
- These threads are stopped abruptly at shutdown

Daemon threads

- Before start() is invoked, you set the thread’s “daemon” attribute to True
- In other words, set it in __init__
- Then the program will exit when all threads terminate, except for daemon threads

Locking/mutexes

```
mutex = threading.Lock()

mutex.acquire() # grab the mutex

mutex.release() # release the mutex

mutex.locked()  # check the state
```

Example with threading

```
class Mythread(threading.Thread):
    def __init__(self, i):
        self.i = i
        threading.Thread.__init__(self)
    def run(self):
        print(self.i ** 32)

Mythread(2).start()
```

Queues to share data

```
jobs = Queue.Queue(5) # max

jobs = Queue.Queue(-1)

jobs.put(ITEM) # Blocks!

jobs.put(ITEM, block=False)

jobs.put(ITEM, block=False, timeout=10)
```

Get from queue

```
jobs.get() # Blocks!

jobs.get(block=False)

jobs.get(block=False, timeout=10)
```

Exceptions

- If you try to get from an empty Queue, you get an exception: Queue.Empty
- Try to put into a full Queue, and you get another exception: Queue.Full

Wait for threads

- Each thread can indicate it’s done processing

```
jobs.task_done()
```

- Main thread waits for all to signal that they’re done processing

```
jobs.join()
```

join?

- Yes, “join” is a terrible name. It really should be called something like “wait_to_finish”.
- Invoice join() on a thread, and the calling thread blocks until the joined thread returns/exits

Guido on GIL, threads

… the GIL is not as bad as you would initially think: you just have to undo the brainwashing you got from Windows and Java proponents who seem to consider threads as the only way to approach concurrent activities.

Just Say No to the combined evils of locking, deadlocks, lock granularity, livelocks, nondeterminism and race conditions.

So, what should we do?

- Currently, the "multiprocessing" module is recommended (similar API to threading)
- This module spawns processes, rather than threads
- You can use multiprocessing.Queue, which is almost identical to Queue.Queue — but works between processes!

Simple start

```
from multiprocessing import Process


def f(name):

    print 'hello', name


if __name__ == '__main__':

    p = Process(target=f, args=('bob',))

    p.start()

    p.join()
```

Queue

```
from multiprocessing import Process, Queue

def f(q):

    q.put([42, None, 'hello'])

if __name__ == '__main__':

    q = Queue()

    p = Process(target=f, args=(q,))

    p.start()

    print q.get()    # prints "[42, None, 'hello']"

    p.join()
```

Pipes

```
from multiprocessing import Process, Pipe

def f(conn):

    conn.send([42, None, 'hello'])

    conn.close()

if __name__ == '__main__':

    parent_conn, child_conn = Pipe()

    p = Process(target=f, args=(child_conn,))

    p.start()

    print parent_conn.recv()   # prints "[42, None, 'hello']"

    p.join()
```

Synchronization

```
from multiprocessing import Process, Lock

def f(l, i):

    l.acquire()

    print 'hello world', i

    l.release()

if __name__ == '__main__':

    lock = Lock()

    for num in range(10):

        Process(target=f, args=(lock, num)).start()
```

Shared memory

```
from multiprocessing import Process, Value, Array

def f(n, a):

    n.value = 3.1415927

    for i in range(len(a)):

        a[i] = -a[i]

if __name__ == '__main__':

    num = Value('d', 0.0)

    arr = Array('i', range(10))

   p = Process(target=f, args=(num, arr))

    p.start()

    p.join()

    print num.value

    print arr[:]
```

Process pooling

```
from multiprocessing import Pool

def f(x):

    return x*x

if __name__ == '__main__':

    pool = Pool(processes=4)              # start 4 workers

    result = pool.apply_async(f, [10]) # run "f(10)"

    print result.get(timeout=1)        # print "100"

    print pool.map(f, range(10))       # prints numbers
```