A Tale of Two Concurrencies (Part 1)

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (www.swig.org) and Python Lex-Yacc (www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Talk to any Python programmer long enough and eventually the topic of concurrent programming will arise—usually followed by some groans, some incoherent mumbling about the dreaded global interpreter lock (GIL), and a request to change the topic. Yet Python continues to be used in a lot of applications that require concurrent operation whether it is a small Web service or full-fledged application. To support concurrency, Python provides both support for threads and coroutines. However, there is often a lot of confusion surrounding both topics. So in the next two installments, we're going to peel back the covers and take a look at the differences and similarities in the two approaches, with an emphasis on their low-level interaction with the system. The goal is simply to better understand how things work in order to make informed decisions about larger libraries and frameworks.

To get the most out of this article, I suggest that you try the examples yourself. I've tried to strip them down to their bare essentials so there's not so much code—the main purpose is to try some simple experiments. The article assumes the use of Python 3.3 or newer.

First, Some Socket Programming

To start our exploration, let's begin with a little bit of network programming. Here's an example of a simple TCP server implemented using Python's low-level `socket` module [1]:

```
# server.py
from socket import *
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        handler(client, addr)
def echo_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        client.send(data)
    print('Connection closed')
    client.close()
if __name__ == '__main__':
    tcp_server(('',25000), echo_handler)
```

Run this program in its own terminal window. Next, try connecting to it using a command such as `nc 127.0.0.1 25000` or `telnet 127.0.0.1 25000`. You should see the server echoing what you type back to you. However, open up another terminal window and try repeating the `nc` or `telnet` command. Now you'll see nothing happening. This is because the server only supports a single client. No support has been added to make it manage multiple simultaneous connections.

Programming with Threads

One way to support concurrency is to use the built-in `threading` library [2]. Simply change the `tcp_server()` function to launch a new thread for each new connection as follows:

```
# server.py
from socket import *
from threading import Thread
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = sock.accept()
        t = Thread(target=handler, args=(client, addr))
        t.daemon=True
        t.start()
...
```

That's it. If you repeat the connection experiment, you'll find that multiple clients work perfectly fine.

Under the covers, a `Thread` represents a concurrently executing Python callable. On UNIX, threads are implemented using POSIX threads and are fully managed by the operating system. A common confusion concerning Python is a belief that it uses some kind of "fake" threading model—not true. Threads in Python are the same first-class citizens as you would find in C, Java, or any number of programming languages with threads. In fact, Python has supported threads for most of its existence (the first implementations provided thread support on Irix and Solaris around 1992). There is also support for various synchronization primitives such as locks, semaphores, events, condition variables, and more. However, the focus of this article is not on concurrent programming algorithms per se, so we're not going to focus further on that.

The Global Interpreter Lock

The biggest downside to using threads in Python is that although the interpreter allows for concurrent execution, it does not support parallel execution of multiple threads on multiple CPU cores. Internally, there is a global interpreter lock (GIL) that synchronizes threads and limits their execution to a single core (see [3] for a detailed description of how it works). There are various reasons for the existence of the GIL, but most of them relate to aspects of the implementation of the Python interpreter itself. For example, the use of reference-counting-based garbage collection is poorly suited for multithreaded execution (since all reference count updates must be locked). Also, Python has historically been used to call out to C extensions that may or may not be thread-safe themselves. So the GIL provides an element of safety.

For tasks that are mostly based on I/O processing, the restriction imposed by the GIL is rarely critical—such programs spend most of their time waiting for I/O events, and waiting works just as well on one CPU as on many. The major concern is in programs that need to perform a significant amount of CPU processing. To see this in action, let's make a slightly modified server that, instead of echoing data, computes Fibonacci numbers using a particularly inefficient algorithm:

```
# server.py
...
def fib(n):
    if n <= 2:
      return 1
    else:
      return fib(n-1) + fib(n-2)
def fib_handler(client, addr):
    print('Connection from', address)
    while True:
       data = client.recv(1000)
       if not data:
          break
       result = fib(int(data))
       client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
if __name__ == '__main__':
    tcp_server(('',25000), fib_handler)
```

If you try this example and connect using `nc` or `telnet`, you should be able to type a number as input and get a Fibonacci number returned as a result. For example:

```
bash % nc 127.0.0.1 25000
10
55
20
6765
```

Larger numbers take longer and longer to compute. For example, if you enter a value such as 40, it might take as long as a minute to finish.

Now, let's write a little performance test. The purpose of this test is to repeatedly hit the server with a CPU-intensive request and to measure its response time.

```
# perf1.py
from socket import *
import time
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))
while True:
    start = time.time()
    sock.send(b'30')
    resp = sock.recv(100)
    end = time.time()
    print(end-start)
```

Try running this program. It should start producing a series of timing measurements such as this:

```
bash % python3 perf1.py
0.6157200336456299
0.6006970405578613
0.6721141338348389
0.7784650325775146
0.5988950729370117
...
```

Go to a different terminal window and run the same performance test as the first one runs (you have two clients making requests to the server). Carefully watch the reported times. You'll see them suddenly double like this:

```
0.6514902114868164
0.629213809967041
1.2837769985198975          # 2nd test started
1.4181411266326904
1.3628699779510498
...
```

This happens even if you're on a machine with multiple CPU cores. The reason? The global interpreter lock. Python is limited to one CPU, so both clients are forced to share cycles.

Lesser Known Evils of the GIL

The restriction of execution to a single CPU core is the most widely known evil of the GIL. However, there are two other lesser known facets to it that are worth considering. The first concerns the instructions that Python executes under the covers. Consider a simple Python function:

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
```

To execute the function, it is compiled down to a Python-specific machine code that you can view using the `dis.dis()` function:

```
>>> import dis
>>> dis.dis(countdown)
  2       0 SETUP_LOOP              39 (to 42)
    >>    3 LOAD_FAST                0 (n)
          6 LOAD_CONST               1 (0)
          9 COMPARE_OP               4 (>)
         12 POP_JUMP_IF_FALSE       41
  3      15 LOAD_GLOBAL              0 (print)
         18 LOAD_CONST               2 ('T-minus')
         21 LOAD_FAST                0 (n)
         24 CALL_FUNCTION            2 (2 positional, 0 keyword pair)
         27 POP_TOP
  4      28 LOAD_FAST                0 (n)
         31 LOAD_CONST               3 (1)
         34 INPLACE_SUBTRACT
         35 STORE_FAST               0 (n)
         38 JUMP_ABSOLUTE            3
    >>   41 POP_BLOCK
    >>   42 LOAD_CONST               0 (None)
         45 RETURN_VALUE
```

In this code, each low-level instruction executes atomically. That is, each instruction is uninterruptible and can't be preempted. Although most instructions execute very quickly, there are edge cases where a single instruction might take a very long time to execute. To see that, try this experiment where you first launch a simple thread that simply prints a message every few seconds:

```
>>> import time
>>> def hello():
...     while True:
...         print('Hello')
...         time.sleep(5)
...
>>> import threading
>>> t = threading.Thread(target=hello)
>>> t.daemon=True
>>> t.start()
Hello
Hello
... repeats every 5 seconds
```

Now, while that thread is running, type the following commands into the interactive interpreter (note: it might be a bit weird since "Hello" is being printed at the same time).

```
>>> nums = range(1000000000) # Use xrange on Python 2
>>> 'spam' in nums
```

At this point, the Python interpreter will go completely silent. You'll see no output from the thread. You'll also find that not even the Control-C works to interrupt the program. The reason is that

the "in" operator in this example is executing as a single interpreter instruction—it just happens to be taking a very long time to execute due to the size of the data. In practice, it's actually quite difficult to stall pure-Python code in this way. Ironically, it's most likely to occur in code that calls out to long-running operations implemented in C extension modules. Unless the author of an extension module has programmed it to explicitly release the GIL, long operations will stall everything else until they complete (see [4] for information on avoiding this).

The second subtle problem with the GIL is that it makes Python prioritize long-running CPU-bound tasks over short-running I/O-bound tasks. To see this, type in the following test program, which measures how many requests are made per second:

```
# perf2.py
import threading
import time
from socket import *
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('127.0.0.1', 25000))
N = 0
def monitor():
   global N
   while True:
      time.sleep(1)
      print(N, 'requests/second')
      N = 0
t = threading.Thread(target=monitor)
t.daemon=True
t.start()
while True:
    sock.send(b'1')
    resp = sock.recv(100)
    N += 1
```

This program hits the server with a rapid-fire stream of requests. Start your server and run it; you should see output such as this:

```
bash % python3 perf2.py
22114 requests/second
21874 requests/second
21734 requests/second
21137 requests/second
21866 requests/second
...
```

While that is running, initiate a separate session and try computing a large Fibonacci number:

```
bash % nc 127.0.0.1 25000
40
102334155     (takes awhile to appear)
```

When you do this, the `perf2.py` program will have its request rate drop precipitously like this:

```
21451 requests/second
21913 requests/second
6942 requests/second
99 requests/second
103 requests/second
101 requests/second
99 requests/second
...
```

This more than 99% drop in the request rate is due to the fact that if any thread wants to execute, it waits as long as 5 ms before trying to preempt the currently executing thread. However, if any computationally intensive task is running, it will almost always be holding the CPU for the entire 5 ms period and stall progress on short I/O intensive tasks that want to run.

Both of these problems, uninterruptible instructions and prioritization of CPU-bound work, would manifest themselves in an application as a kind of performance "glitch" or a kind of sluggishness. For example, suppose that this service was implementing a backend service for a Web site. Maybe most of the operations are fast-running data queries, but suppose that there were a few corner cases where the service had to perform a significant amount of CPU processing. For those cases, you would find that the responsiveness of the service would degrade significantly as those CPU-intensive tasks are carried out.

Personally, I think the inversion of priority of CPU-bound threads over I/O-bound threads might be the most serious problem with using Python threads—more so than the limitation of execution to a single CPU core. This preference for CPU-bound tasks is exactly the opposite of how operating systems typically prioritize processes (short-running interactive processes usually get priority). In most applications, it's almost always better to maintain a quick response time even if it means certain long-running operations take a slight bit longer to complete than they already do.

Using Subprocesses

Although you might look in dismay at the performance of Python threads, keep in mind that their primary limitation concerns long-running CPU-bound tasks. If this applies, you'll want to seek some other subdivision of tasks in your system. A typical solution is to run multiple instances of the Python interpreter, either through process forking or the use of a process pool.

For example, to use a process pool, you can modify the server to use the `concurrent.futures` module as follows:

```
# server.py
from concurrent.futures import ProcessPoolExecutor as Pool
NPROCS = 4
pool = Pool(NPROCS)
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        future = pool.submit(fib, int(data))
        result = future.result()
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

If you make this simple change, you'll find that the first performance test (perf1.py) now nicely scales to use all available CPU cores.

However, using a process Pool for short-running tasks is probably not the best approach. If you run the second performance test (perf2.py), you'll see about a 95% reduction in the request rate such as this:

```
bash % python3 perf2.py
1319 reqs/sec
1313 reqs/sec
1315 reqs/sec
1308 reqs/sec
```

This performance reduction is solely due to all of the extra overhead associated with sending the request to another process, serializing data, and so forth. If there's any bright spot, it's that this request rate will now remain constant even if another client starts performing a long-running task (so, at the very least, the performance will simply be consistently and predictably bad). A smarter approach might involve a threshold that only kicks work out to a pool if it's known in advance that it will take a long time to compute. For example:

```
# server.py
...
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = client.recv(1000)
        if not data:
            break
        n = int(data)
        if n > 15:
            future = pool.submit(fib, n)
            result = future.result()
        else:
            result = fib(n)
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
...
```

Using a process pool is not the only approach. For example, an alternative approach might involve a pre-forked server like this:

```
# server.py
from socket import *
import os
NPROCS = 8
def tcp_server(address, handler):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    # Fork copies of the server
    for n in range(NPROCS):
        if os.fork() == 0:
            break
    while True:
        client, addr = sock.accept()
        handler(client, addr)
...
```

In this case, there is no communication overhead associated with submitting work out to a pool. However, you're also strictly limiting your concurrency to the number of processes spawned. So if a large number of long-running requests were made, they might lock everything out of the server until they finish.

Memory Overhead of Threads

A common complaint lodged against threads is that they impose a steep memory overhead. To be sure, if you create 1000 threads, each thread requires some dedicated memory to use as a call stack and some data structures for management in the operating system. However, this overhead is intrinsic to POSIX threads and not to Python specifically. The Python-specific overhead is actually quite small. Internally, each Python thread is represented by a small C data structure that is less than 200 bytes in size. Beyond that, the only additional overhead are the stack frames used to make function calls in the thread.

Even the apparent memory overhead of threads can be deceptive. For example, by default, each thread might be given 8 MB of memory for its stack (thus, in a program with 1000 threads, it will appear that more than 8 GB of RAM is being used). However, it's important to realize that this memory allocation is typically just a maximum reservation of virtual memory, not actual physical RAM. The operating system will allocate page-sized chunks

of memory (typically 4096 bytes) to this space as it's needed during execution but leave the unused space unmapped. Many operating systems (e.g., Linux) take it a step further and won't even reserve space on the swap disk for thread stacks unless specifically configured [5]. So the actual memory overhead of threads is far less than it might seem at first glance. (Note: the lack of a swap allocation for threads presents a possible danger to production systems—if memory is ever exhausted, the system might start randomly killing processes to make space!)

As far as Python is concerned, the main memory overhead risk is in code based on deeply recursive algorithms because this would create the potential to use up all of the thread stack space. However, most Python programmers just don't write code like this. In fact, if you blow up the stack on purpose, you'll find that it takes nearly 15,000 levels of recursive function calls to do it. The bottom line: it's unlikely that you would need to worry about this in normal code. In addition, if you're really concerned, you can set a much smaller thread stack size using the `threading.stack_size(nbytes)` function.

All of this said, the overhead of threads is still a real concern. Many systems place an upper limit on the number of threads a single process or user can create. There are also certain kinds of applications where the degree of concurrency is so high that threads simply aren't practical. For example, if you're writing a server to support something like WebSockets, you might have a scenario where the system needs to maintain tens of thousands of open socket connections simultaneously. In that case, you probably don't want to manage each socket in its own thread (we'll address this in the next installment).

Everything Is Terrible, Well, Only Sometimes. Maybe.

If you've made it this far, you might be inclined to think that just about everything with Python threads is terrible. To be sure, threads are probably not the best way to handle CPU-intensive work in Python, and they might not be appropriate if your problem involves managing 30,000 open socket connections. However, for everything else in the middle, they offer a sensible choice.

For one, threads work great with programs that are primarily performing I/O. For example, if you're simply moving data around on network connections, manipulating files, or interacting with a database, most of your program's time is going to be spent waiting around. So you're unlikely to see the worst effects of the GIL. Second, threads offer a relatively simple programming model. Launching a thread is easy: they are generally compatible with most Python code that you're likely to write, and they're likely to work with most Python extensions (caveat: if you're manipulating shared state, be prepared to add locks). Finally, Python provides libraries for moving work off to separate processes if you need to.

In the next installment, we'll look at an alternative to programming with threads based on Python coroutines. Coroutines are the basis of Python's new `asyncio` module, and the techniques are being used a variety of other programming languages as well.

References

[1] Python3 socket module: https://docs.python.org/3/library/socket.html.

[2] Python3 threading module: https://docs.python.org/3/library/threading.html.

[3] Dave Beazley, "Understanding the Python GIL": http:/www.dabeaz.com/GIL.

[4] Thread state and the Global Interpreter Lock: https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[5] R. Love, "Linux System Programming," 2nd ed. (O'Reilly Media, Inc., 2013); see the section on "Opportunistic Allocation" in Chapter 9.