A Tale of Two Concurrencies (Part 2)

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

In the previous installment [1], we dived into some of the low-level details and problems related to Python threads. As a brief recap, although Python threads are real system threads, there is a global interpreter lock (GIL) that restricts their execution to a single CPU core. Moreover, if your program performs any kind of CPU-intensive processing, the GIL can impose a severe degradation in the responsiveness of other threads that happen to be performing I/O.

In response to some of the perceived limitations of threads, some Python programmers have turned to alternative approaches based on coroutines or green threads. In a nutshell, these approaches rely on implementing concurrency entirely in user space without relying on threads as provided by the operating system. Of course, how one actually goes about doing that often remains a big mystery.

In this installment, we're going to dive under the covers of Python concurrency based on coroutines (or generators). Rather than focusing on the usage of particular libraries, the main goal is to gain a deeper understanding of the underlying implementation to see how it works, performance characteristics, and limitations. As with the previous installment, the examples presented are meant to be tried as experiments. There's a pretty good chance that some of the code presented will bend your brain—it's not often that you get to write a small operating system in the space of an article. Also, certain parts of the code require Python 3. So, with that in mind, let's start!

Threads, What Are They Good For?

Previously, we created a simple multithreaded network service that computed Fibonacci numbers. Here was the code:

```
# 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()
```

```
def fib(n):
    if n <= 2:
       return 1
    else:
       return fib(n-1) + fib(n-2)

def fib_handler(client, address):
    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)
```

When you run the server, you can connect any number of concurrent clients using `nc` or `telnet`, type numbers as input, and get a Fibonacci number returned as a result. For example:

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

If you carefully study this code and think about the role of threads, their primary utility is in handling code that blocks. For example, consider operations such as `sock.accept()` and `client.recv()`. Both of those operations stop progress of the currently executing thread until incoming data is available. That's not a problem, though, when each client is handled by its own thread. If a thread decides to block, the other threads are unaffected and can continue to run. Basically, you just don't have to worry about it, because all of the underlying details of blocking, awaking, and so forth are handled by the operating system and associated thread libraries.

If threads aren't going to be used, then you have to devise some kind of solution that addresses the blocking problem so that multiple clients can concurrently operate. That is the main problem that needs to be addressed.

Enter Generator Functions

In order to implement blocking, you have to figure out some way to temporarily suspend and later resume the execution of a Python function. As it turns out, Python provides a special kind of function that can be used in exactly this way—a generator function. Generator functions are most commonly used to drive iteration. For example, here is a simple generator function:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

Normally, this function would be used to feed a `for-` loop like this:

```
>>> for x in countdown(5):
...     print(x)
...
5
4
3
2
1
>>>
```

Under the covers, the `yield` statement emits values to be consumed by the iteration loop. However, it also causes the generator function to temporarily suspend itself. Here is a low-level view of the mechanics involved.

```
>>> c = countdown(5)
>>> next(c)     # Run to the yield
5
>>> next(c)
4
>>> next(c)
3
...
>>> next(c)
1
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

On each `next()` call, the function runs to the yield, emits a value, and stops. A `StopIteration` exception is raised when the function terminates. The fact that `yield` causes a function to stop is interesting—that's exactly the behavior you need to handle blocking. Perhaps it can be used to do more than simple iteration.

Generators as Tasks

Rather than thinking of generator functions as simply implementing iteration, you can alternatively view them as more generally implementing a task (note: when used in this way, generators are typically called "coroutines," although that term seems to be applied rather loosely in the Python community). If you make a task queue and task scheduler, you can make generators or coroutines look a lot like threads. For example, here's an experiment you can try using the above generator function:

```
from collections import deque

# A task queue
tasks = deque()

# Create some tasks
tasks.append(countdown(10))
tasks.append(countdown(20))
tasks.append(countdown(5))

# Run the tasks
def run():
    while tasks:
        task = tasks.popleft()
        # Run to the yield
        try:
            x = next(task)
            print(x)
            tasks.append(task)   # Reschedule
        except StopIteration:
            print('Task done')

run()
```

In this code, multiple invocations of the countdown() generator are being driven by a simple round-robin scheduler. The output will appear something like this if you run it:

```
10
20
5
9
19
4
8
18
3
7
17
2
...
```

That's interesting, but not very compelling since no one would typically want to run a simple iteration pattern like the countdown() function in this manner.

A much more interesting generator-based task might be a rewritten version of the fib_handler() function from our server. For example:

```
def fib_handler(client, address):
    print('Connection from', address)
    while True:
        yield ('recv', client)    # Added
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield ('send', client)    # Added
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
```

In this new version, yield statements are placed immediately before each socket operation that might block. Each yield indicates both a reason for blocking ('recv' or 'send') and a resource (the socket client) on which blocking might occur.

With the interactive interpreter, let's see how to drive it. First, create a socket and wait for a connection:

```
>>> from socket import *
>>> sock = socket(AF_INET, SOCK_STREAM)
>>> sock.bind(('', 25000))
>>> sock.listen(1)
>>> client, addr = sock.accept()
```

Next, establish a connection using a command such as nc localhost 25000 at the shell. Once you've done this, try these steps:

```
>>> task = fib_handler(client, addr)
>>> task
<generator object fib_handler at 0x10a7c53b8>
>>> reason, resource = next(task)
Connection from ('127.0.0.1', 52474)
>>> reason
'recv'
>>> resource
<socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 25000), raddr=('127.0.0.1', 52474)>
>>>
```

If you carefully study this output, you'll see that the handler task ran to the first yield statement and is now suspended. Before resuming the handler, you need to wait until input is available on the supplied socket (resource). To do that, you can poll the socket using a system call such as select() [2]. For example:

```
>>> from select import select
>>> select([resource], [], [])  # Blocks until data available
```

Go back to the terminal with the connected nc session and type an integer and return. This should force the above select() statement to return. Once it's returned, you can resume the generator by typing the following:

```
>>> reason, resource = next(task)
>>> reason
'send'
>>> resource
<socket.socket fd=4, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 25000), raddr=('127.0.0.1', 52474)>
>>>
```

Now you see that the task has advanced to the next `yield` statement. Use the `select()` statement again to see if it's safe to proceed with sending.

```
>>> select([], [resource], [])
>>> reason, resource = next(task)
>>>
```

In this example, you are using `next()` to drive the generator task forward to the next yield statement. The `select()` call is polling for I/O and is being used to know when it is safe to resume the generator.

A Generator-Based Task Scheduler

Putting the pieces of the last section together, you can make a small generator-based task scheduler like this:

```
from socket import *
from collections import deque
from select import select

tasks = deque()
recv_wait = {}   # sockets -> tasks waiting to receive
send_wait = {}   # sockets -> tasks waiting to send

def run():
    while any([tasks, recv_wait, send_wait]):
        while not tasks:
            can_read, can_send, _ = select(recv_wait, send_wait, [])
            for s in can_read:
                tasks.append(recv_wait.pop(s))
            for s in can_send:
                tasks.append(send_wait.pop(s))
        task = tasks.popleft()
        try:
            reason, resource = next(task)
            if reason == 'recv':
                recv_wait[resource] = task
            elif reason == 'send':
                send_wait[resource] = task
            else:
                raise RuntimeError('Bad reason: %s' % reason)
        except StopIteration:
            print('Task done')
```

The scheduler is essentially a small operating system. There is a queue of ready-to-run tasks (`tasks`) and two waiting areas for tasks that need to perform I/O (`recv_wait` and `send_wait`). The core of the scheduler takes a ready-to-run task and runs it to the next `yield` statement, which acts as a kind of "trap" or "system call." Based on the result of the yield, the task is placed into one of the I/O holding areas. If there are no tasks ready to run, a `select` call is made to wait for I/O and place a previously suspended task back onto the task queue.

To use this scheduler, you take your previous thread-based code and simply instrument it with `yield` calls. For example:

```
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:
        yield 'recv', sock
        client, addr = sock.accept()
        # Create a new handler task and add to the task queue
        tasks.append(handler(client, addr))

def fib(n):
    if n <= 2:
      return 1
    else:
      return fib(n-1) + fib(n-2)

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        yield 'recv', client
        data = client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield 'send', client
        client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()

if __name__ == '__main__':
    tasks.append(tcp_server(('',25000), fib_handler))
    run()
```

This code will require a bit of study, but if you try it out, you'll find that it supports concurrent connections without the slightest hint of a thread—interesting indeed.

Hiding Implementation Details

One complaint about the generator solution is the addition of the extra `yield` statements. Not only do they introduce extra code, they are somewhat low-level, requiring the user to know some details about the underlying scheduling code. However, Python 3.3 introduced the ability to write generator-based subroutines using the `yield from` statement [3]. You can use this to make a wrapper around `socket` objects.

```
class GenSocket(object):
    def __init__(self, sock):
        self.sock = sock

    def accept(self):
        yield 'recv', self.sock
        client, addr = self.sock.accept()
        return GenSocket(client), addr

    def recv(self, maxbytes):
        yield 'recv', self.sock
        return self.sock.recv(maxbytes)

    def send(self, data):
        yield 'send', self.sock
        return self.sock.send(data)

    def __getattr__(self, name):
        return getattr(self.sock, name)
```

This wrapper class merely combines the appropriate `yield` statement with the subsequent socket operation. Here is a modified server that uses the wrapper:

```
def tcp_server(address, handler):
    sock = GenSocket(socket(AF_INET, SOCK_STREAM))
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    while True:
        client, addr = yield from sock.accept()
        # Create a new handler task and add to the task queue
        tasks.append(handler(client, addr))

def fib_handler(client, address):
    print('Connection from', address)
    while True:
        data = yield from client.recv(1000)
        if not data:
            break
        result = fib(int(data))
        yield from client.send(str(result).encode('ascii')+b'\n')
    print('Connection closed')
    client.close()
```

In this version, blocking calls such as `client.recv()` are replaced by calls of the form `yield from client.recv()`. Other than that, the code looks virtually identical to the threaded version. Moreover, details of the underlying task scheduler are now hidden. Again, keep in mind that no threads are in use.

Studying the Performance

Previously, two performance tests were performed. The first test simply measured the performance of the server on CPU-bound work:

```
# 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)
```

If you run this program, it will start producing a series of timing measurements that are essentially the same as the threaded version of code. If you run multiple clients, however, you'll find that the server is limited to using a single CPU core as before. There's no global interpreter lock in play, but since the entire server executes within a single execution thread, there's no way for it to take advantage of multiple CPU cores either. That's one important lesson—using coroutines is not a technique that can be used to make code scale to multiple processors.

The second performance test measured the performance on a rapid-fire series of fast-running operations. Here it is again:

```
# 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
```

If you run the program, you'll see output similar to the following:

```
bash % python3 perf2.py
16121 requests/second
16245 requests/second
16179 requests/second
16305 requests/second
16210 requests/second
...
```

The initial request rate will be lower than that reported with the examples involving threads in the previous article. There is simply more overhead in managing the various generator functions, invoking `select()`, and so forth. While the test is running, computing a large Fibonacci number from a separate connection produces:

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

After you do this, the `perf2.py` will stop responding entirely. For example:

```
16151 requests/second
16265 requests/second
0 requests/second
0 requests/second
0 requests/second
...
```

This will continue until the large request completes entirely. Since there are no threads at work, there is no notion of preemption or parallelism. In fact, any operation that decides to block or take a lot of compute cycles will block the progress of everything else.

Back to Subprocesses

As it turns out, problems with performance and blocking have to be solved in the same manner as with threads. Specifically, you have to use threads or process pools to carry out such calculations outside of the task scheduler. For example, you might rewrite the `fib_handler()` function using `concurrent.futures` exactly as you did before with threads:

```
from concurrent.futures import ProcessPoolExecutor as Pool

NPROCS = 4
pool = Pool(NPROCS)

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

The only catch is that even if you make this change, you'll find that it still doesn't work. The problem here is that the `future.result()` operation blocks, waiting for the result to come back. By blocking, it stalls the entire task scheduler. In fact, this will happen for any operation at all that might block (e.g., resolving a domain name, accessing a database, etc.).

Generators: It's All In

In order for a generator-based solution to work, every blocking operation has to be written to work with the task loop. In the previous example, attempts to use a process pool are unsuccessful since calls to obtain the result block. To make it work, you need to write additional supporting code to turn blocking operations into something that can yield to the task loop. The following code gives an idea of how you might do it.

The first step is to write a wrapper around the `Future` object's `result()` method to make it use `yield`. For example:

```
class GenFuture(object):
    def __init__(self, future):
        self.future = future

    def result(self):
        yield 'future', self.future
        return self.future.result()

    def __getattr__(self, name):
        return getattr(self.future, name)
```

Next, you might create a wrapper around pools to adjust the output of the `pool.submit()` to return a `GenFuture` object:

```
class GenPool(object):
    def __init__(self, pool):
        self.pool = pool


    def submit(self, func, *args, **kwargs):
        f = self.pool.submit(func, *args, **kwargs)
        return GenFuture(f)

    def __getattr__(self, name):
        return getattr(self.pool, name)
```

The main goal of these classes is to preserve the programming interface of the blocking code. In fact, you will only make a slight change to the `fib_handler()` code as shown here:

```
from concurrent.futures import ProcessPoolExecutor as Pool

NPROCS = 4
pool = GenPool(Pool(NPROCS))     # Note: Use GenPool

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

Carefully observe how all blocking operations are now preceded by a `yield from` declaration. The only remaining task is to modify the task scheduler to support futures. Here is that code:

```
from socket import socketpair

tasks = deque()
recv_wait = {}
send_wait = {}
future_wait = {}

# Callback triggered on future completion
def _future_callback(future):
    tasks.append(future_wait.pop(future))
    _loop_wake()

# Sockets to allow waking of the I/O loop
_loop_notify_socket, _loop_wait_socket = socketpair()

# Function to wake the task loop when blocked on select()
def _loop_wake():
    _loop_notify_socket.send(b'x')

# Dummy task that allows select() to wake
def _loop_sleeper():
    while True:
        yield 'recv', _loop_wait_socket
        _loop_wait_socket.recv(1000)

tasks.append(_loop_sleeper())

def run():
    while any([tasks, recv_wait, send_wait, future_wait]):
        while not tasks:
            can_read, can_send, _ = select(recv_wait, send_wait, [])
            for s in can_read:
                tasks.append(recv_wait.pop(s))
            for s in can_send:
                tasks.append(send_wait.pop(s))
        task = tasks.popleft()
        try:
            reason, resource = next(task)
            if reason == 'recv':
                recv_wait[resource] = task
            elif reason == 'send':
                send_wait[resource] = task
            elif reason == 'future':
                future_wait[resource] = task
                resource.add_done_callback(_future_callback)
            else:
                raise RuntimeError('Bad reason: %s' % reason)
        except StopIteration:
            print('Task done')
```

Whew! There are a lot of moving parts, but the general idea is as follows. For futures, the task is placed into a waiting area as before (`future_wait`). A callback function (`_future_callback`) is then attached to the future to be triggered upon completion. When results return, the callback function puts the task back onto the `tasks` queue. A byte of I/O is then written to a special loopback socket (`_loop_notify_socket`). A separate task (`_loop_sleeper`) constantly monitors this socket and wakes to read the byte. (The main purpose of this special task is really just to get the task loop to wake from the `select()` call to allow ready tasks to run again.)

This Is Crazy (But Most Things Are When You Think About It)

Needless to say, if you're going to abandon threads for concurrency, you're going to have to do more work to make it work. If you get down to it, the code involving generators is actually a lot like a small user-level operating system, with all of the underlying task scheduling, I/O polling, and so forth. At first glance, the whole approach might seem crazy. However, keep in mind that it would rarely be necessary to write such code yourself. Instead, you would use an existing library such as the new `asyncio` module [4].

Even if you use a library, you still have to know what you're doing. Specifically, you need to be fully aware of places where your code might block and stall the task scheduler. Coroutines also do not free you from limitations such as Python's GIL—you should still be prepared to execute work in thread or process pools as appropriate.

At this point, you might be seeking some kind of sage advice on how to proceed with Python concurrency. Should you use threads? Should you use coroutines? Unfortunately, I can't offer anything more than it depends a lot on the problem that you are trying to solve. Python provides a wide variety of tools for addressing the concurrency problem. All of those tools have various tradeoffs and limitations. As such, anyone expecting a kind of "magic" solution that solves every possible problem will likely be disappointed. Again, some thinking is required—in the end, it really helps to understand what you're doing and how things work.

Postscript

The code examples in this article were the foundation of a PyCon 2015 talk I gave on concurrency. If you're interested in seeing the code work with a live coding demonstration, the talk video can be found online [5].

References

[1] D. Beazley, "A Tale of Two Concurrencies (Part 1)," *;login:*, vol. 40, no. 3, June 2015: https://www.usenix.org/publications/login/june15/beazley.

[2] "`select`—Waiting for I/O Completion": https://docs.python.org/3/library/select.html (select module).

[3] "PEP 380: Syntax for Delegating to a Subgenerator": https://www.python.org/dev/peps/pep-0380/.

[4] "asyncio—Asynchronous I/O, event loop, coroutines and tasks": https://docs.python.org/3/library/asyncio.html (asyncio module).

[5] PyCon 2015 presentation on concurrency: http://pyvideo.org/video/3432/python-concurrency-from-the-ground-up-live.