sendwithus

Dyspatch

Asynchronous Python

I am:

Chris Pouliot

Platform Developer @ Sendwithus

WHO AM I

Teach thingies

Backend thingies

Sendwithus, Lighthouse Labs, LLC

UVic

Moxuz
@christophepoug
me@chrispouliot.codes

Dyspatch

sendwithus

founded

Jan 2013

25,000+

Requests/Min

~1B

emails per month

Dyspatch

https://github.com/moxuz/startupslam2018-performant-python

CAT

QUESTION TIME

Tell me what a Generator is!

Agenda

Generators

What is the GIL

Concurrency

Multiprocessing

Live demo

Generators

Let’s you return a value to the caller without exiting

Can be used iteratively

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


count = countdown(10)
for n in count:
    print(n)

# 10
# 9
# ...
```

Generators

Suspends the code until you call next()

Keeps function state, unlike a return

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


count = countdown(10)
print(next(count))
print(next(count))
print(next(count))

# 10
# 9
# ...
```

Generators

Ability to send data back into a generator

```
def mult(m):
    while True:
        x = yield
        yield m * x


multiplier = mult(2)
next(multiplier)
print(multiplier.send(1))
next(multiplier)
print(multiplier.send(2))

# 2
# 4
```

Generators

Context switching

```
def generator():
    for i in range(1, 10):
        print("From generator {}".format((yield i)))


c = generator()
c.send(None)  # Prime the generator. Can't send normal data at first

while True:
    print("From user {}".format(c.send(1)))

# From generator 1
# From user 2
# From generator 1
# From user 3
# ...
```

Generators

```python
def reader():
    # Read from a file, a socket, a database
    for i in range(4):
        yield '<< %s' % i


def reader_wrapper(g):
    for v in g:
        # Do some other work
        yield v


wrap = reader_wrapper(reader())
for i in wrap:
    print(i)

# << 0
# << 1
# ...
```

```python
def reader():
    # Read from a file, a socket, a database
    for i in range(4):
        yield '<< %s' % i


def reader_wrapper(g):
    yield from g


wrap = reader_wrapper(reader())
for i in wrap:
    print(i)

# << 0
# << 1
# ...
```

Generators

```
def writer():
    # Write to a file, a socket, a database
    while True:
        w = (yield)
        print('>> ', w)


def writer_wrapper(g):
    g.send(None)
    while True:
        try:
            x = (yield)  # Capture the value that's sent
            g.send(x)  # and pass it to the writer
        except StopIteration:
            pass

# >>  0
# >>  1
```

```
def better_writer_wrapper(g):
    yield from g

# >>  0
# >>  1
```

Concurrency

Ability to perform multiple tasks at the same time

Multiple threads, multiple processes

How does Python handle this?

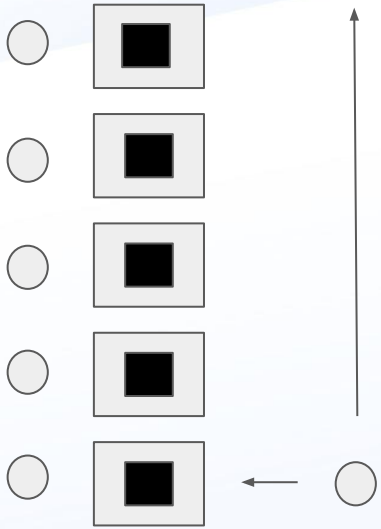
Single threaded

Concurrency

Imagine a chess exposition

Can play one game at a time

Can play all games at a time



Global Interpreter Lock (GIL)

Python created early 1990s

Multi cores / multi threads almost non-existent

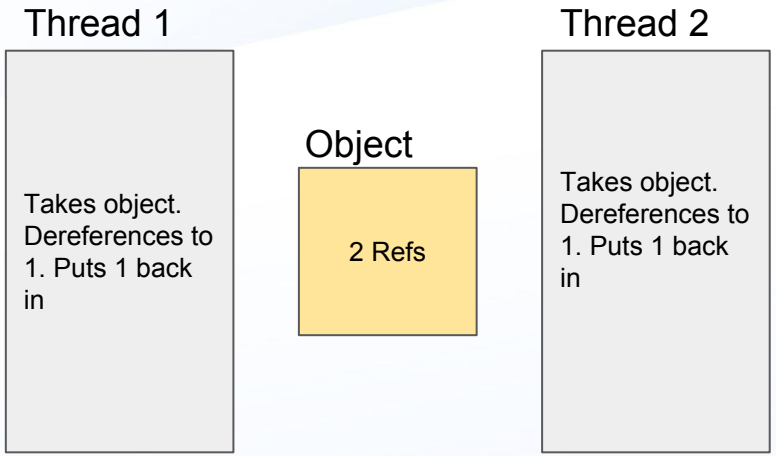
1992 multi threading is considered

How can you manage global variables with threads?

Global Interpreter Lock

Scope dictates if a var is in use

Thread race condition with Python's "garbage collection"

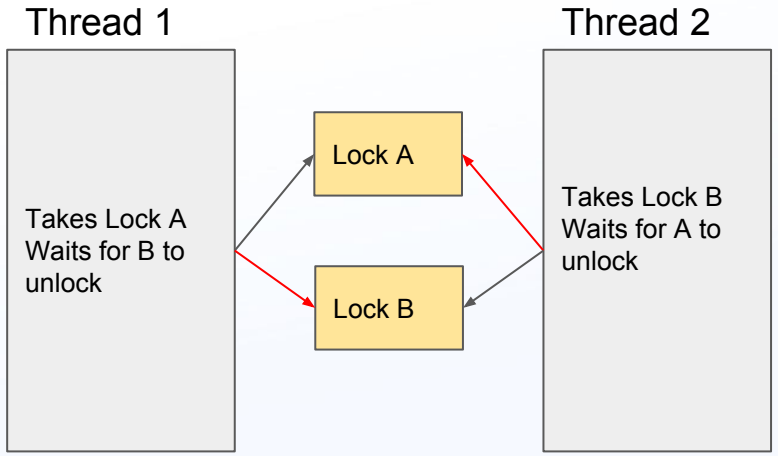


Global Interpreter Lock

Could solve by adding locks to the objects.

Lock object. Take references. Decrement. Update. Unlock

Problem with “deadlocking”



Global Interpreter Lock

To solve this, the Global Interpreter Lock (GIL) is added

Just a global lock

Must have lock to execute Python code

Very easy to understand. Big reason Python became so popular due to the ease of writing C extensions

Global Interpreter Lock

I/O Bound programs: totally cool

CPU Bound programs: uh oh

Long computations release GIL

Turns out the GIL had an issue (Fixed in Python 3.2)

Global Interpreter Lock

TLDR: Python created when multicores weren’t a thing

Why can’t we just remove it? 4-7x slower single threaded

Global Interpreter Lock

Other Python implementations don’t use GIL
Jython, Ironpython, PyPy

Can’t add to CPython without breaking C Extensions

Why don’t Java, Go, Rust, C++ have a GIL?

Removing the GIL would probably mean removing the C API

CAT TAX

Concurrency

How can we manage async Python with the GIL?

Concurrency still using a single thread at a time

Context switching: Overhead/loss of CPU caching

Concurrency

Switching to another threads work when the current thread blocks: Event Loop

Python async functions return immediately: futures

Event fires when work is completed

Context switching is slow, but faster than IO wait (<1ms vs multiple seconds!)

Concurrency

Event Loops can take two forms: Callbacks, futures

Callbacks: functions passed as arguments, data passed with function calls

Futures: immediate return but with promise of future value

Concurrency

Coroutines can be implemented using generators

Recall the generator syntax we went over earlier

Generators yielding: exactly an Event Loop

How can we do this before Python 3?

```
@coroutine
def example():
    data = yield from slow()
    print(data)
```

Concurrency: gevent

Enter gevent

Monkey patches standard lib to be asynchronous so your code stays the same

Gevent provides “greenlets”

Greenlets run on one single thread: swaps between them on IO wait

Spawn greenlets in order to run non-blocking IO

Concurrency: gevent

Notice how nothing starts until we join

Spawning a gevent greenlet does not start work

gevent sleep pauses execution of greenlet

```python
import gevent


def foo():
    print('Im in foo!')
    gevent.sleep(0)
    print('End of foo')


def bar():
    print('Im in bar')
    gevent.sleep(0)
    print('End of bar')


gevent.joinall([
    gevent.spawn(foo),  # Run foo in a greenlet
    gevent.spawn(bar),  # Run bar in a greenlet
])

# Im in foo!
# Im in bar
# End of foo
# End of bar
```

Concurrency: gevent

Remember back to “context switching”: loss of CPU caching

Let’s just send 2000 asynchronous requests at once

We should restrict how many requests are actually running

Semaphores!

LIVE DEMO

Concurrency: gevent

Notice the patch_all() call

Requests to a server that sleeps input int time

Gets slower as more requests run at the same time

```python
from random import randint
import time
import urllib2

from gevent import monkey
import gevent
monkey.patch_all()


url = "http://209.97.148.110"


def download(url):
    return urllib2.urlopen(url).read()


def build_urls(url, count):
    for i in range(count):
        yield "{}/{}".format(url, randint(1, 1000))


def main():
    requests = [gevent.spawn(download, u) for u in build_urls(url, 200)]  # Get's slower as we add more threads (200..300..400)
    gevent.joinall(requests)


if __name__ == '__main__':
    start = time.time()
    main()
    print("Elapsed: {}s".format(time.time() - start))
```

Concurrency: gevent

We are sending 500 requests

100 requests active at a time

Even ignoring context switching, 1000 threads at once could still be slow

```python
from random import randint
import time
import urllib2

from gevent import monkey, pool
import gevent
monkey.patch_all()


url = "http://209.97.148.110"


def download(url):
    urllib2.urlopen(url).read()


def build_urls(url, count):
    for i in range(count):
        yield "{}/{}".format(url, randint(1, 1000))


def main():
    p = pool.Pool(100)
    requests = [p.spawn(download, u) for u in build_urls(url, 500)]
    gevent.joinall(requests)


if __name__ == '__main__':
    start = time.time()
    main()
    print("Elapsed: {}s".format(time.time() - start))
```

Concurrency

More than one way to do concurrency (futures, callbacks)

Let’s take a look at some other libraries

Concurrency: Tornado

Created by Facebook

Uses callback-style concurrent programming

Recall: generators yield a value. How do we tell Tornado our func is done?

Need to raise an error containing data!

Concurrency: Tornado

Event Loop runs entire app

Notice the callback funcs

```
AsyncHTTPClient.configure("tornado.curl_httpclient.curlAsyncHTTPClient", max_clients=100)
url = "http://209.97.148.110"


def build_urls(url, count):
    for i in range(count):
        yield "{}/{}".format(url, randint(1, 1000))


@gen.coroutine
def main(base_url):
    http_client = AsyncHTTPClient()
    urls = build_urls(url, 100)
    responses = yield [http_client.fetch(u) for u in urls]
    raise gen.Return(value=responses)  # We cant return from a generator, we have to raise an exception!


if __name__ == "__main__":
    _ioloop = ioloop.IOLoop.instance()
    run_func = partial(main, url)
    result = _ioloop.run_sync(run_func)
    print(result)
```

Concurrency: Gevent vs Tornado

Tornado runs your entire app

Gevent better for cpu-bound, Tornado better for io-bound.

Tornado magic

Concurrency: Tornado

Event Loop runs entire app

Notice the callback funcs

```
AsyncHTTPClient.configure("tornado.curl_httpclient.curlAsyncHTTPClient", max_clients=100)
url = "http://209.97.148.110"


def build_urls(url, count):
    for i in range(count):
        yield "{}/{}".format(url, randint(1, 1000))


@gen.coroutine
def main(base_url):
    http_client = AsyncHTTPClient()
    urls = build_urls(url, 100)
    responses = yield [http_client.fetch(u) for u in urls]
    raise gen.Return(value=responses)  # We cant return from a generator, we have to raise an exception!


if __name__ == "__main__":
    _ioloop = ioloop.IOLoop.instance()
    run_func = partial(main, url)
    result = _ioloop.run_sync(run_func)
    print(result)
```

Concurrency: AsyncIO

Introduced Python 3.4

Influenced from gevent/tornado's use of generators

Like tornado event loop runs entire app

Lower level (gevent/tornado in Python 3.4+ use asyncIO under the hood)

Concurrency: AsyncIO

Event Loop runs entire app

Notice semaphore and coroutine

Notice “yield from”

```
url = "http://209.97.148.110"


def build_urls(url, count):
    for i in range(count):
        yield "{}/{}".format(url, randint(1, 1000))


def chunked_http_client():
    semaphore = asyncio.Semaphore(100)  # 100 active threads

    @asyncio.coroutine
    def http_get(url):
        nonlocal semaphore  # enclosed scope variable
        with (yield from semaphore):
            r = yield from aiohttp.request('GET', url)
            data = yield from r.content.read()
            yield from r.wait_for_close()
        return data

    return http_get


def main(url):
    urls = build_urls(url, 500)
    http_client = chunked_http_client()
    tasks = [http_client(u) for u in urls]
    for future in asyncio.as_completed(tasks):
        data = yield from future
        print(data)
    return


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    start = time.time()
    loop.run_until_complete(main(url))
    print("Elapsed: {}s".format(time.time() - start))
```

Concurrency: AsyncIO

“yield from” a bit verbose..

Python 3.5 -> async/await

```
import asyncio
import aiohttp

url = "http://209.97.148.110"
loop = asyncio.get_event_loop()
client = aiohttp.ClientSession(loop=loop)


async def make_request(client, url):
    async with client.get(url) as response:
        print(await response.read())
        return

asyncio.ensure_future(make_request(client, url + "/100"))
loop.run_forever()
```

Multiprocessing

We've looked at IO-bound applications, what about CPU-bound

Before Python 3.2 Threads actually make CPU-bound problems slower

Can fork a new process running its own Python interpreter, no GIL issues

Watch for ram usage

Multiprocessing

Very easy to spawn a new Process

Ability to pass arguments

Now have Python code running in **parallel**

```
from multiprocessing import Process


def f(name):
    print('hello', name)


if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Multiprocessing

You can use a queue for IPC

Join call to make sure program waits

Thread safe and process safe

Can have multiple consumers

Queues are not persistent

Watch out for pickling overhead

```python
from multiprocessing import Process, Queue


def f(q):
    q.put(["so", "many", "strings!"])


if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Multiprocessing

Let’s take a look at finding primes

Basic serial version

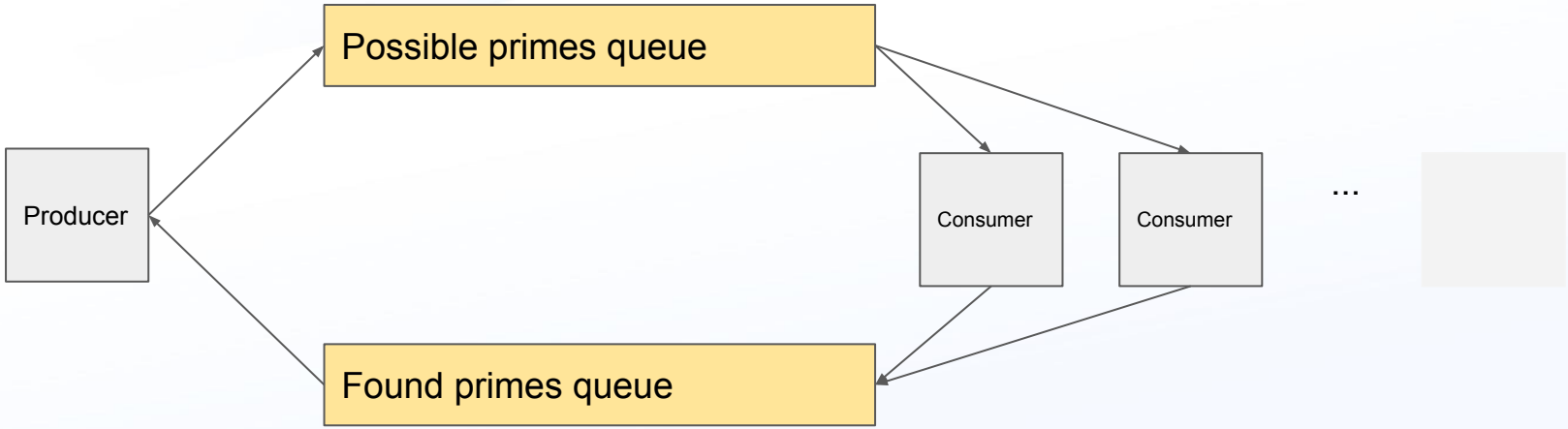
```python
import math


def is_prime(n):
    if not n % 2:
        return False
    start = 3
    to = math.sqrt(n) + 1
    for i in range(start, int(to), 2):
        if not n % i:
            return False
    return True


print(is_prime(5))  # True
print(is_prime(6))  # False
print(is_prime(7))  # True
print(is_prime(8))  # False
```

Multiprocessing

Let’s live-code a multi-process prime checker



Multiprocessing

Questions?