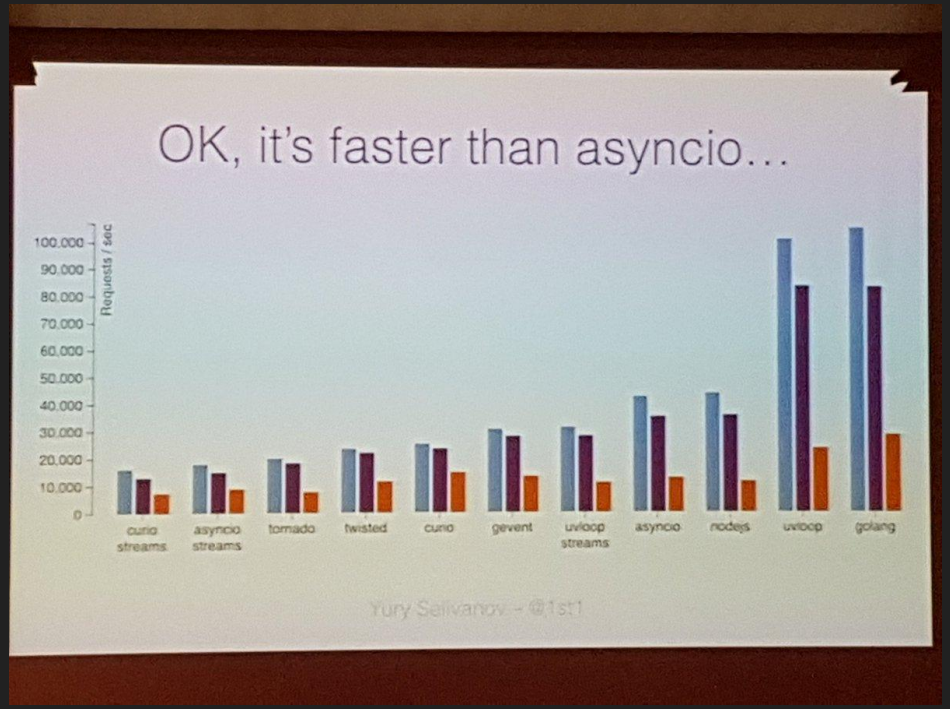
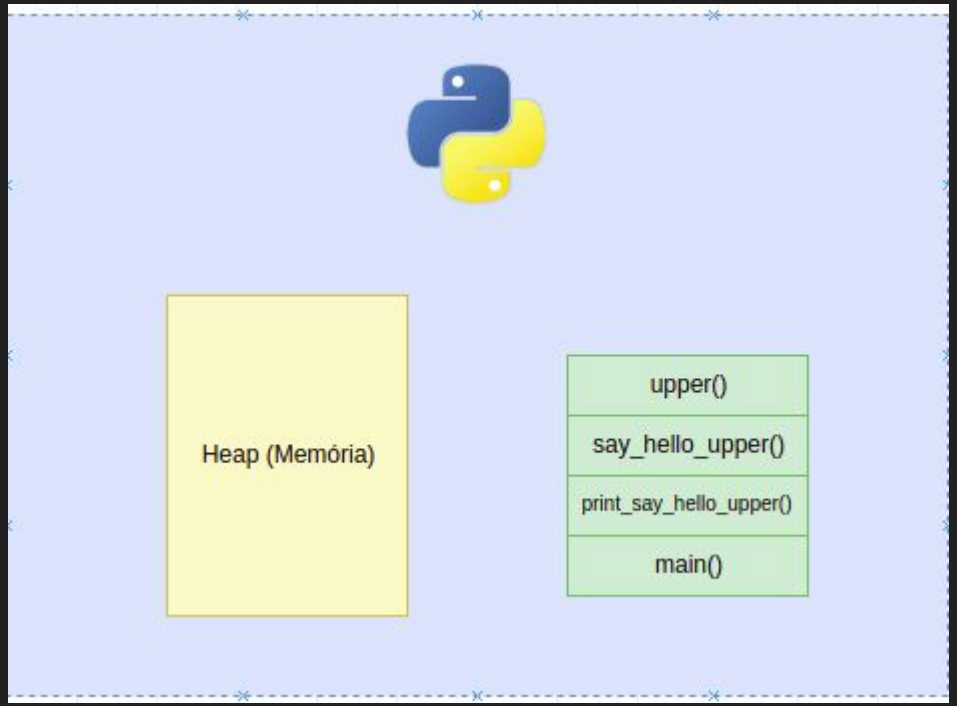
# Python I/O Performance: Asyncio + Uvloop



# Programa Python

```
def say_hello_upper(name):
    return ('Hello ' + name).upper()


def print_say_hello_upper():
    print(say_hello_upper('foo'))


def main():
    print_say_hello_upper()


main()
```



# Problema: Healthcheck Counter

Imagine que precisamos fazer um script que faz uma contagem de quantos status tiveram sucesso de uma lista de URLs e devolver o total de status com sucesso em formato JSON;
Vamos verificar algumas abordagens.

# Single Thread (Sem paralelismo)

```python
def website_statuses(websites):
    statuses = {}
    for website in websites:
        response = requests.get(website)
        status = response.status_code
        if not statuses.get(status):
            statuses[status] = 0
        statuses[status] += 1
    return statuses


if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = [url for url in f.read().split('\n') if url != ""]
    t0 = time.time()
    print(json.dumps(website_statuses(websites)))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

```
$ python3 naive-checker.py list.txt
{"200": 31}
getting website statuses took 35.1 seconds
```

Fonte: https://www.youtube.com/watch?v=qfY2cqjJMdw

# Problemas

O script levou muito tempo para rodar;
É necessário cada requisição ser feita e terminar de receber os dados para ir para a próxima.

# Vantagens

Simplicidade do código

# Multi Processing

Multi processos : Podemos abrir vários processos assim cada requisição rodaria de forma paralela.

```python
if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = [url for url in f.read().split('\n') if url != ""]
    number_of_processes = int(sys.argv[2])
    per_process = math.ceil(len(websites) / number_of_processes)
    # split up the work based on number of processes
    …
    t0 = time.time()
    processes = []
    for i in range(number_of_processes):
        p = subprocess.Popen(
            ["python3", "naive-checker.py", "/tmp/list-{}.txt".format(i)],
            stdout=subprocess.PIPE)
        processes.append(p)
    # gather the results
    …
    print(combined)
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

```
$ python3 subprocess-checker.py list.txt 3
{'200': 31}
getting website statuses took 9.6 seconds
```

Fonte: https://www.youtube.com/watch?v=qfY2cqjJMdw

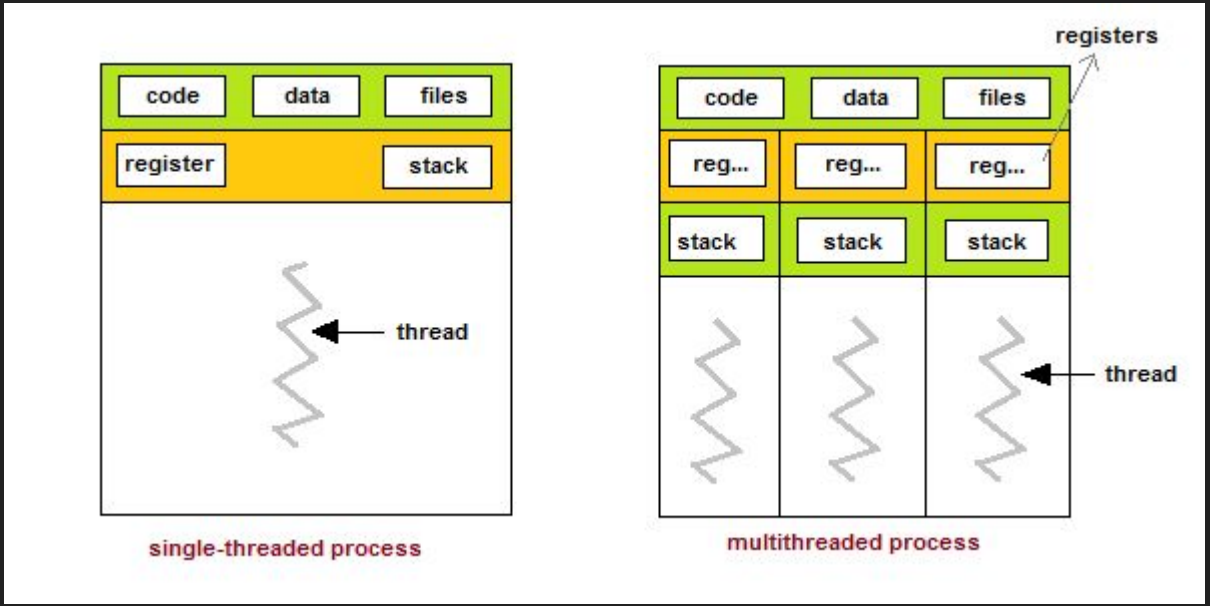
# Problemas

Memória não é compartilhada entre os processos;
Precisa separar os dados e depois juntá-los novamente.

# Vantagens

Velocidade na execução do script;
Consegue executar mais de uma requisição por vez;

# Multithreading




Fonte: https://www.studytonight.com/operating-system/multithreading

“Multithreading é a capacidade que o sistema operacional possui de executar várias threads simultaneamente sem que uma interfira na outra. Estas threads compartilham os recursos do processo, mas são capazes de ser executadas de forma independente” (Fonte: tecmundo)

```python
STATS = {}
def get_website_status(url, lock):
    response = requests.get(url)
    status = response.status_code
    if status != 200:
        print(url)
    lock.acquire()
    if not STATS.get(status):
        STATS[status] = 0
    STATS[status] += 1
    lock.release()
if __name__ == '__main__':
    ...
    threads = []
    lock = threading.Lock()
    for website in websites:
        t = threading.Thread(target=get_website_status, args=(website, lock))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    t1 = time.time()
    print(json.dumps(STATS))
    print("getting website statuses took {0:.1f} seconds".format(t1-
```

```
$ python3 thread-checker.py list.txt
{"200": 31}
getting website statuses took 5.7 seconds
```

Fonte: https://www.youtube.com/watch?v=qfY2cqjJMdw

# Problemas

Necessário controlar o acesso à variável e execução gerenciando locks/semáforos;

Código mais complexo.

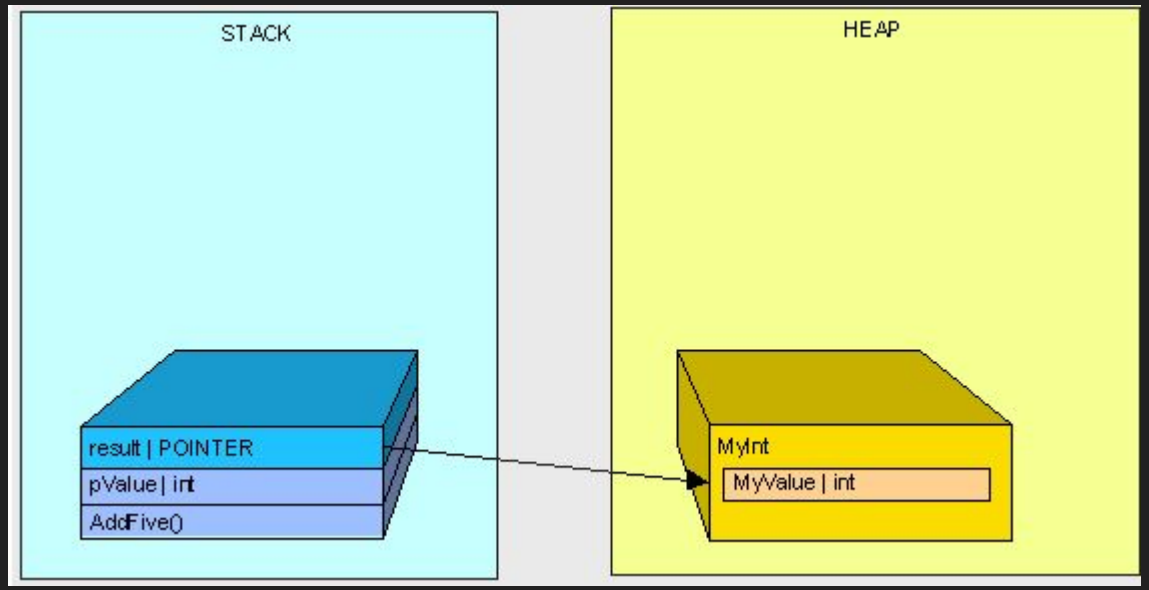
# Vantagens

Memória compartilhada;
Menos recursos consumidos;
Script com execução mais rápida.

# Perae, mas e o GIL??

“The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn’t visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.”


STACK
HEAP
result | POINTER
pValue | int
AddFive()
MyInt
MyValue | int


**Duas referências para Foo**

**Três referências para Foo**

O Python (CPython) utiliza essa contagem para fazer o gerenciamento de memória (GC);
Se houver *race condition*, pode ocorrer de incrementar/decrementar a contagem em *refcount* ao mesmo tempo;

Ou seja, nunca fazemos uso de múltiplas CPUs de forma simultânea em Python

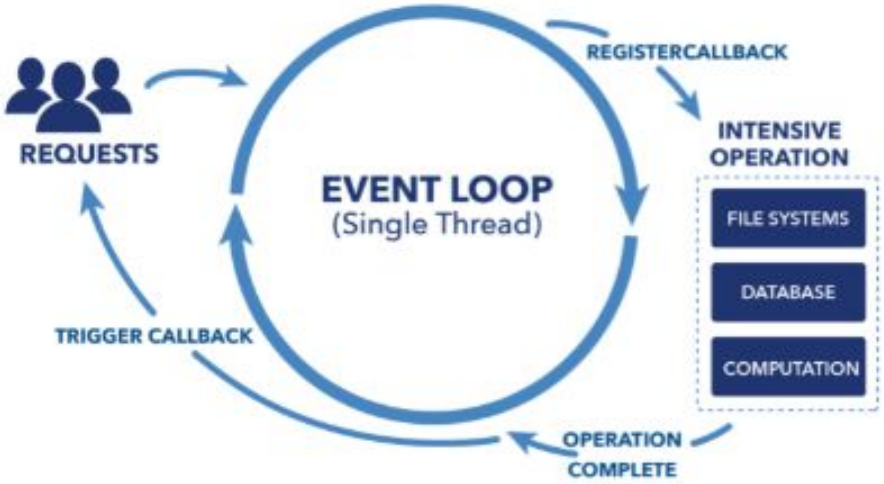
**Problema de incrementar indevidamente:** memory leak;
**Problema de decrementar indevidamente:** variável eliminada quando ainda há referências à ela.

**Espera, mas se o GIL permite que somente rode uma Thread por vez, por que tivemos um ganho de velocidade ao abrir processos/threads no script?**

“Para operações de I/O, o GIL é liberado para outro processo de forma paralela também execute até a primeira chamada retornar resultado”

https://drgarcia1986.github.io/blog/2016/02/18/threads-em-python-e-claro/

# Event Loop


REQUESTS
EVENT LOOP
(Single Thread)
REGISTERCALLBACK
INTENSIVE
OPERATION
FILE SYSTEMS
DATABASE
COMPUTATION
TRIGGER CALLBACK
OPERATION
COMPLETE

# Vantagens

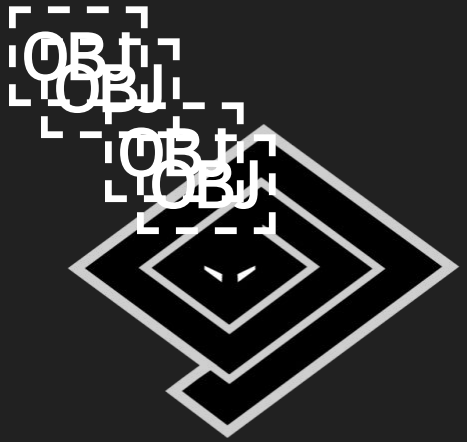
Mais rápido;
Memória compartilhada;
Mais fácil de codificar;
Boa abordagem para lidar com protocolos de rede (conexões com socket)

```python
async def get_statuses(websites):
    async with aiohttp.ClientSession() as session:
        statuses = {}
        tasks = [get_website_status(website, session) for website in websites]
        for status in await asyncio.gather(*tasks):
            if not statuses.get(status):
                statuses[status] = 0
            statuses[status] += 1
        print(json.dumps(statuses))

async def get_website_status(url, session):
    async with session.get(url) as response:
        return response.status

if __name__ == '__main__':
    with open(sys.argv[1], 'r') as f:
        websites = f.read().split('\n')
    t0 = time.time()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_statuses(websites))
    t1 = time.time()
    print("getting website statuses took {0:.1f} seconds".format(t1-t0))
```

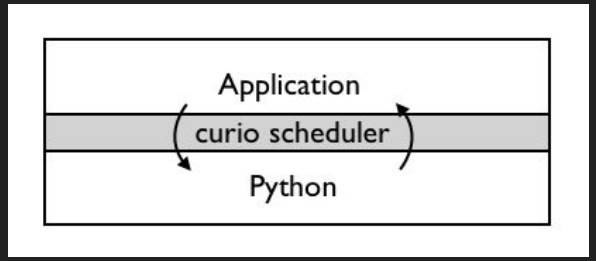
Fonte: https://www.youtube.com/watch?v=qfY2cqjJMdw

```
$ python3 asyncio-checker.py list.txt
{"200": 31}
getting website statuses took 4.8 seconds
```

# Implementações Event Loop



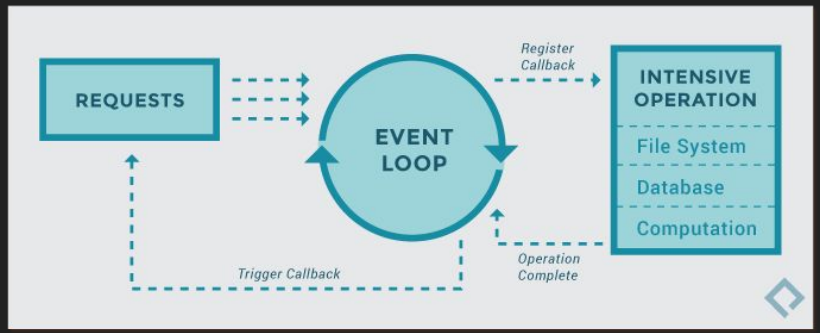

Twisted




Curio

gevent

gevent




asyncio

# Asyncio

- Biblioteca nativa Python (3.5>) para escrever código concorrente usando ***async/await***;
- Base para frameworks assíncronos em Python que fornecem alta performance rede, servidores *web*, banco de dados, filas de tarefas distribuídas etc;
    - Evita *callback hell* utilizando geradores;

# O que temos no asyncio?

- Event Loop (plugável);
- Interfaces para protocolos e transporte;
- Fábricas para servers e conexões;
- Futures/Tasks: callbacks, corrotinas, timeouts, cancellation
- Subprocessos, filas, mecanismos de sincronização

# Uvloop

99,9% compatível com event loop;
Escrito em Cython (Escrever extensões em C - sintaxe similar ao Python, porém estática);
Usa libuv por baixo dos panos: não usa o socket nativo Python (tudo roda em cima da libuv);
I/O é mais rápido.

# Libuv

# Libuv

Async I/O multiplataforma;
Desenvolvido em C;
Uso nativo em Node;

# O que temos na libuv?

Event Loop (Single Thread);
Processos;
Timers;
Sockets TCP/UDP;
Named pipes;
Operações em sistema de arquivos (Thread Pool);
Signal handling;
Processos filhos;
Utilitários para Threads

“We take care of the sheet. You don’t have to.” IBARRA, Saul

# Arquitetura do Libuv

Event Loop (onde as coisas ocorrem);
Handles;
Requests;
Outros utilitários;

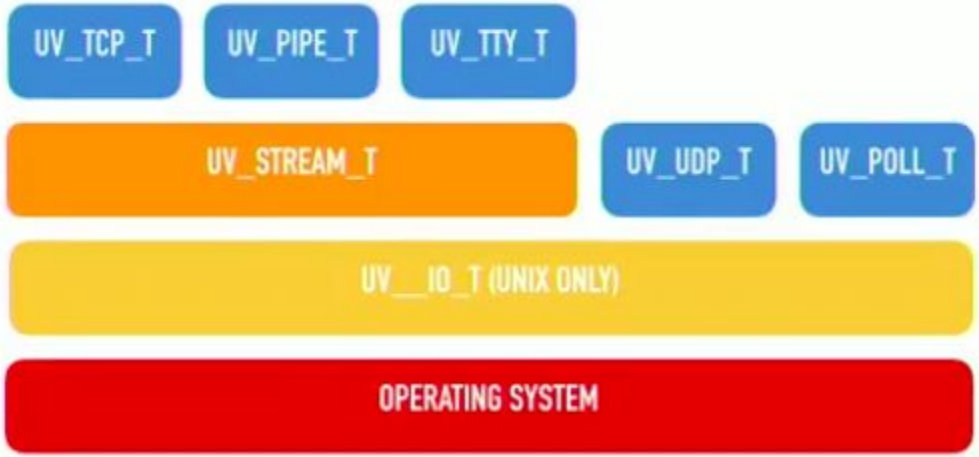
# Handles

Específicos para fazer/lidar com algum tipo de trabalho;
Exemplos: Timer (tipo de Handle): chama um callback após um determinado tempo;
TCPHandles: lidam com a conexão na escrita/leitura

# Requests

Operações em um handler ou por si só
Exemplo: UVWrite é uma requisição para escrever dados em uma conexão stream (usa o StreamHandler)

# LIBUV ARCHITECTURE: NETWORK I/O



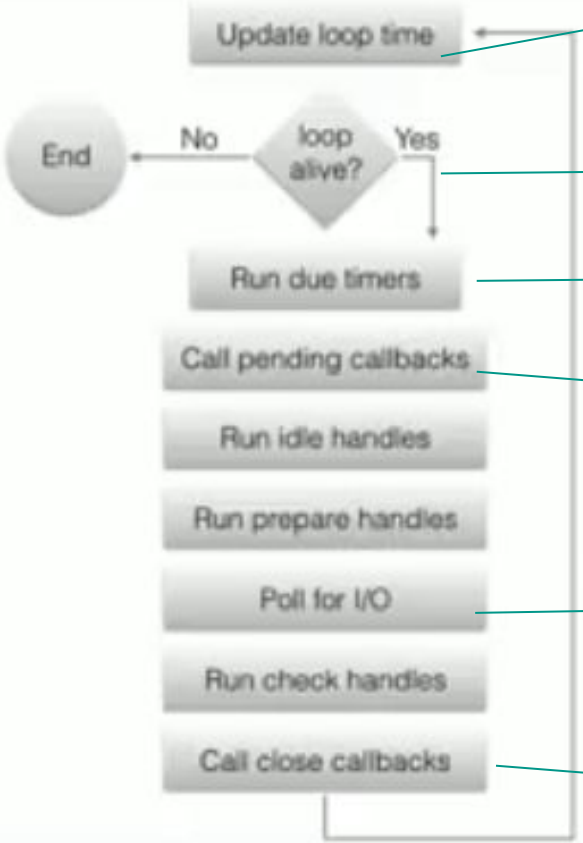
# LIBUV ARCHITECTURE: FILE I/O

# Libuv Event Loop

Onde as “coisas acontecem”;
Single Threaded.

# Como Event Loop do Libuv trabalha?

# LIBUV: THE EVENT LOOP




Atualiza o tempo atual sempre que volta ao início

Verifica se o loop deve continuar rodando

Verifica os timers que já estão no passado Comparando com o tempo atual. Se sim, executa.

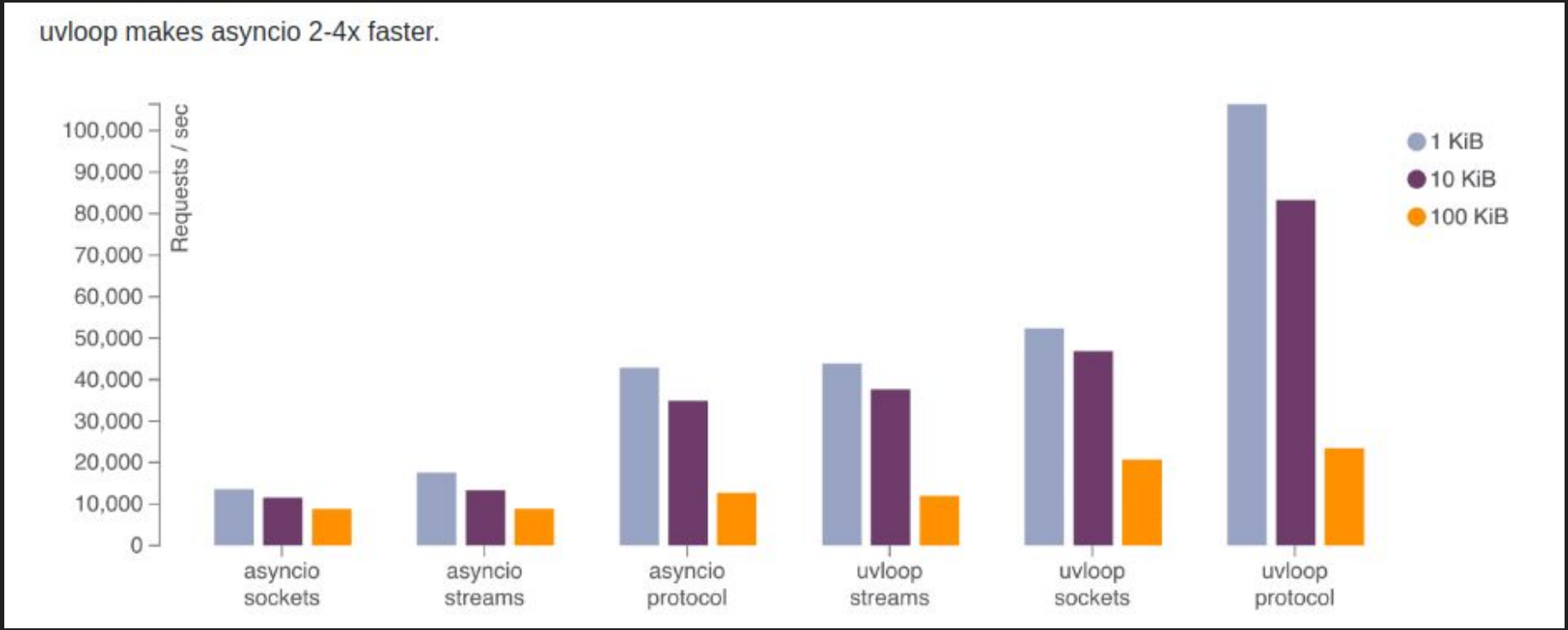
Callbacks de operações anteriores que foram completados. Exemplo: Escrever em um TCP Handle e a mesma termina e o Callback com o resultado é devolvido.

Block de I/O: novos eventos e novas conexões. Callbacks de leitura de dados.

```c
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  int timeout, r, ran_pending;

  r = uv__loop_alive(loop);
  if (!r)
    uv__update_time(loop);

  while (r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
    ran_pending = uv__run_pending(loop);
    uv__run_idle(loop);
    uv__run_prepare(loop);

    timeout = 0;
    if ((mode == UV_RUN_ONCE && !ran_pending) || mode == UV_RUN_DEFAULT)
      timeout = uv_backend_timeout(loop);

    uv__io_poll(loop, timeout);
    uv__run_check(loop);
    uv__run_closing_handles(loop);

    if (mode == UV_RUN_ONCE) {
      uv__update_time(loop);
      uv__run_timers(loop);
    }

    r = uv__loop_alive(loop);
    if (mode == UV_RUN_ONCE || mode == UV_RUN_NOWAIT)
      break;
  }

  loop->stop_flag = 0;
```

# Voltando ao Uvloop...

# Uvloop Performance



# Uvloop Performance

Se otimizado corretamente, consegue atingir nível de performance em Go para redes

```
class AbstractEventLoopPolicy:
    """Abstract policy for accessing the event loop."""

    def get_event_loop(self):
        """Get the event loop for the current context.

        Returns an event loop object implementing the BaseEventLoop interface,
        or raises an exception in case no event loop has been set for the
        current context and the current policy does not specify to create one.

        It should never return None."""
        raise NotImplementedError

    def set_event_loop(self, loop):
        """Set the event loop for the current context to loop."""
        raise NotImplementedError

    def new_event_loop(self):
        """Create and return a new event loop object according to this
        policy's rules. If there's need to set this loop as the event loop for
        the current context, set_event_loop must be called explicitly."""
```

```
def set_event_loop_policy(policy):
    """Set the current event loop policy.

    If policy is None, the default policy is restored."""
    global _event_loop_policy
    assert policy is None or isinstance(policy, AbstractEventLoopPolicy)
    _event_loop_policy = policy
```

```
if __name__ == '__main__':
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.get_event_loop()
```

```
(Pdb) type(loop)
<class 'uvloop.Loop'>
```

# Então só usar uvloop já faz minha aplicação ficar super rápida?

# Não!

Depende de **COMO** libs que utilizam asyncio foram implementadas

# Formas de implementar libs com asyncio

- Sockets (*sock_sendall, sock_recv, sock_connect*);
- Streams (*StreamReader / StreamWriter*);
- Protocolos de baixo nível e Transportes

# Sockets

Maior facilidade de implementação se você já utiliza sockets síncronos;
Porém não consegue fazer buffer dos dados
Não há controle sobre o fluxo;

# loop.sock_* methods

```
async def echo_server(loop, address, unix):
    with sock:
        while True:
            client, addr = await loop.sock_accept(sock)
            loop.create_task(echo_client(loop, client))


async def worker(loop, client):
    with client:
        while True:
            data = await loop.sock_recv(client, 1000000)
            if not data:
                break
            await loop.sock_sendall(client, data)
```

# Streams

Nível maior de abstração;
Tem uma performance melhor que a implementação com sockets, pois event loop sabe mais sobre a aplicação;
Facilidade para utilização, mas é muito genérico

# aiohttp

Abstração de streaming customizada sob medida para o protocolo concreto para utilizar **async/await**

# Streams

```
async def streams_worker(reader, writer):
    while True:
        data = await reader.read(1000000)
        if not data:
            break
        writer.write(data)
    writer.close()

loop.run_until_complete(
    asyncio.start_server(streams_worker, *addr))
loop.run_forever()
```

# Protocolos/Transporte

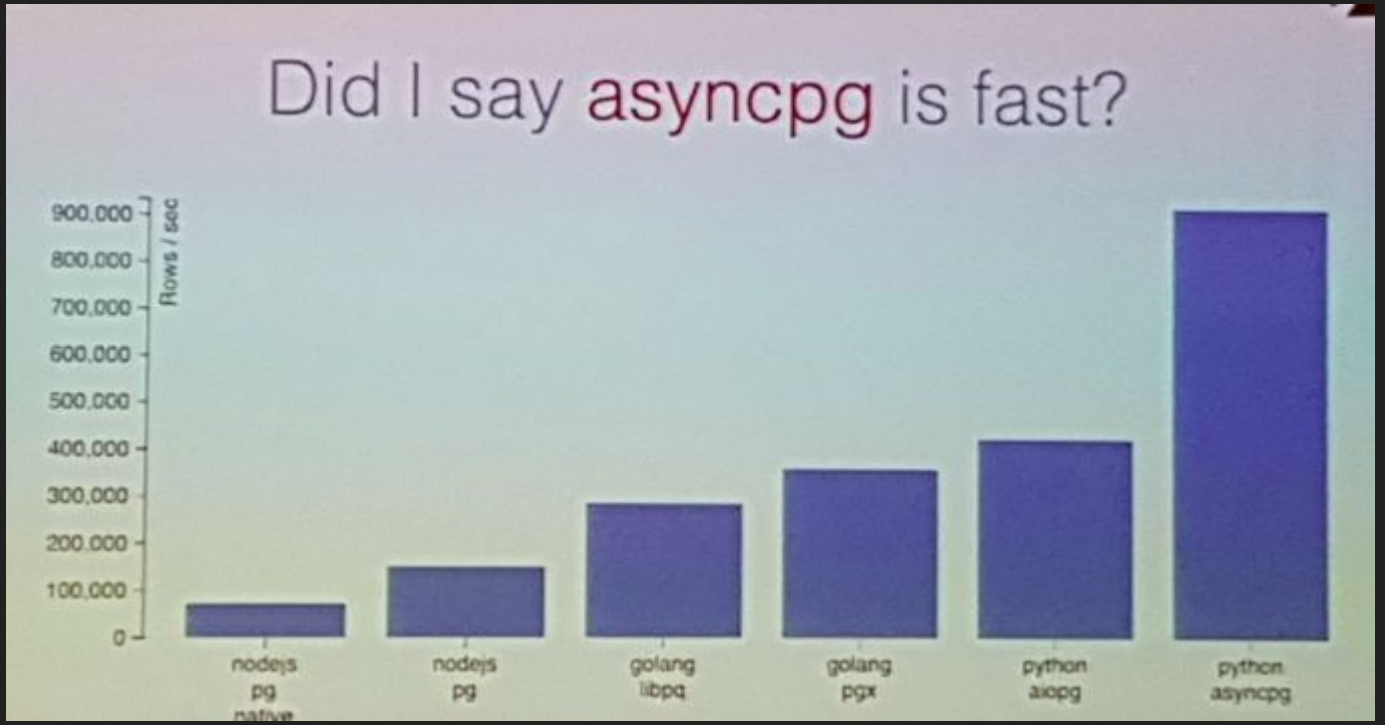
Permite você escrever em Cython,C,Rust, whatever;
Melhor opção para performance;
Controle total do fluxo I/O (“Event Loop pare de me mandar dados”);
Implementação de buffers de leitura/escrita personalizados
Callback hell (use Facade);

# Protocolos x Transporte

Transporte: “Como os dados serão transmitidos?”
Protocolo: “Quais dados serão transmitidos”

Transporte: Abstração para um socket (I/O endpoint);
Protocolo: Abstração para uma aplicação;

Transporte: Chama o protocolo para enviar dados;
Protocolo: Chama o transporte para passar dados que foram recebidos


Did I say asyncpg is fast?
Rows / sec
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
nodejs pg
nodejs pg
golang libpq
golang pgx
python aiopg
python asyncpg

asyncpg

# Protocols and Transports

```
class EchoProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        self.transport = transport

    def connection_lost(self, exc):
        self.transport = None

    def data_received(self, data):
        self.transport.write(data)

loop.run_until_complete(
    loop.create_server(EchoProtocol, *addr))
loop.run_forever()
```

# Implementação

Formato de dados binário ao invés de texto;

Nem todos os tipos podem ser utilizados e parseados como texto no Postgres;

Escrito em Cython/Python;

Cython é utilizado para trabalhar com char para criar buffers em Python, assim evita que o Python aloque e desaloque memória muitas vezes (parse do PostgreSQL protocol).

# Gabaritando performance asyncio + uvloop

# Preferência por usar/criar libs que...

- Implemente protocolos/transporte sempre que possível para melhor performance;
- Utilize dados em formato binário (menos dados / melhor parse) sempre que possível;
- Torne público apenas async/await e esconda quaisquer complexidades envolvendo callbacks;
- Use uma linguagem mais “low level”: Cython/C/Rust;

# Script para obter a variação de bitcoins do ano de 2016

API retorna por dia;
Logo: 365 requisições

```
}
{"date": "2016-02-01", "opening": 1705.99999, "closing": 1674.65, "lowest": 1622.52684, "highest": 1
714.99899, "volume": 130035.001122, "quantity": 77.93155503, "amount": 318, "avg_price": 1668.579576
94}
{"date": "2016-01-28", "opening": 1762.97999, "closing": 1707.99999, "lowest": 1673.0, "highest": 17
62.98, "volume": 221460.00305749, "quantity": 128.51576824, "amount": 411, "avg_price": 1723.2126928
1}
{"date": "2016-01-20", "opening": 1715.00001, "closing": 1789.99999, "lowest": 1690.02, "highest": 1
798.8, "volume": 303158.29709106, "quantity": 173.80483499, "amount": 571, "avg_price": 1744.2454757
3}
{"date": "2016-01-19", "opening": 1757.0, "closing": 1715.00001, "lowest": 1715.00001, "highest": 17
82.999, "volume": 275638.62773794, "quantity": 157.78104553, "amount": 382, "avg_price": 1746.969205
41}
{"date": "2016-01-18", "opening": 1754.0, "closing": 1774.999, "lowest": 1700.00001, "highest": 1799
.9, "volume": 515680.28371721, "quantity": 295.33519499, "amount": 513, "avg_price": 1746.08476221}
```

# Sem Uvloop

```python
if __name__ == '__main__':
    #asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.get_event_loop()
    start_time = time()
    print('Starting executing script')
    loop.run_until_complete(
        get_year_balance(from_year=2016)
    )
    print('\n\nDone! It took {} seconds'.format(time() - start_time))
    loop.close()
```

```
{"date": "2016-01-18", "opening": 1754.0
.9, "volume": 515680.28371721, "quantity


Done! It took 7.859303951263428 seconds
```

# Com uvloop

```python
if __name__ == '__main__':
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.get_event_loop()
    start_time = time()
    print('Starting executing script')
    loop.run_until_complete(
        get_year_balance(from_year=2016)
    )
    print('\n\nDone! It took {} seconds'.format(time() - start_time))
    loop.close()
```

```
79.97967, "volume": 132664.20415029, "qu
79}


Done! It took 4.439315319061279 seconds
```

# Dúvidas/Sugestões?