slides-async (/github/nanvel/slides-async/tree/master)
/ async_talk.ipynb (/github/nanvel/slides-async/tree/master/async_talk.ipynb)

A Brief History of Async

Open in nbviewer (https://nbviewer.org/github/nanvel/slides-async/blob/master/async_talk.ipynb)

ThaiPy#85 2022-11-10

Oleksandr Polieno (https://github.com/nanvel)

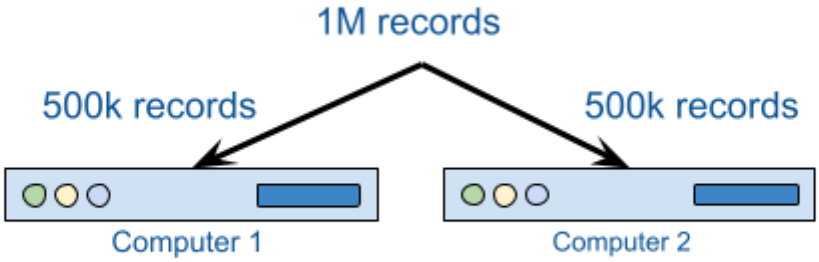
Plan:

- Parallelism and concurrency
- Why do we need async, pros/cons
- Event-driven io: from callbacks to async
- IOLoop
- Coroutines
- async/await syntax
- Overview

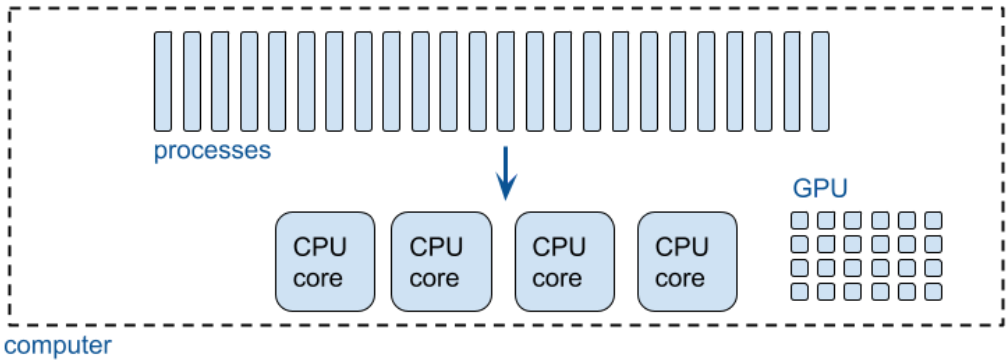
Async - a first-class citizen in Python that simplifies concurrency implementation in a single thread.

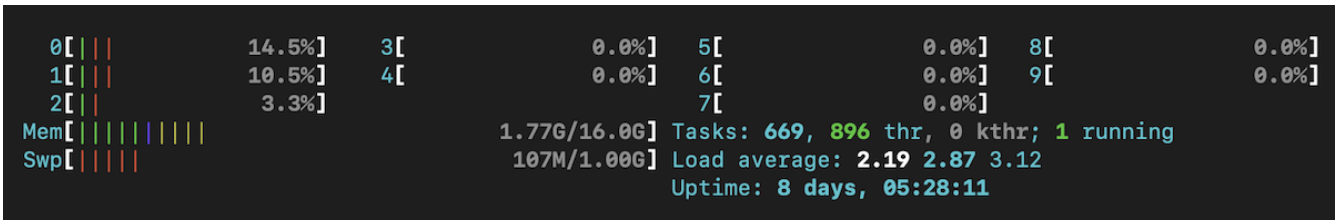
Parallelism

Distribute work across multiple computers:

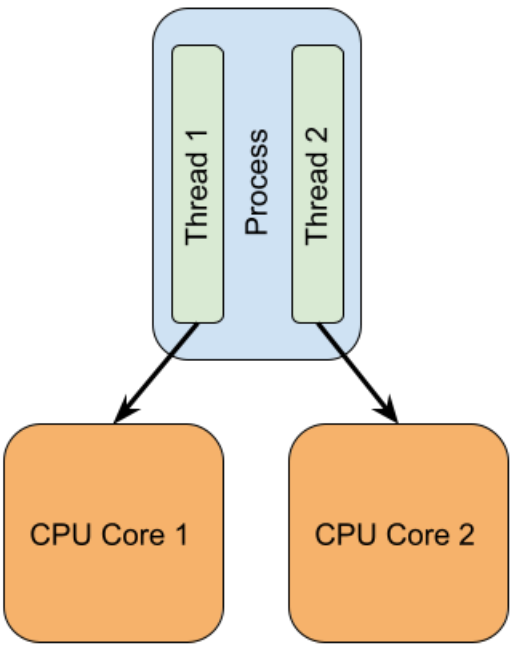


A single computer:





Using multiple CPU cores:

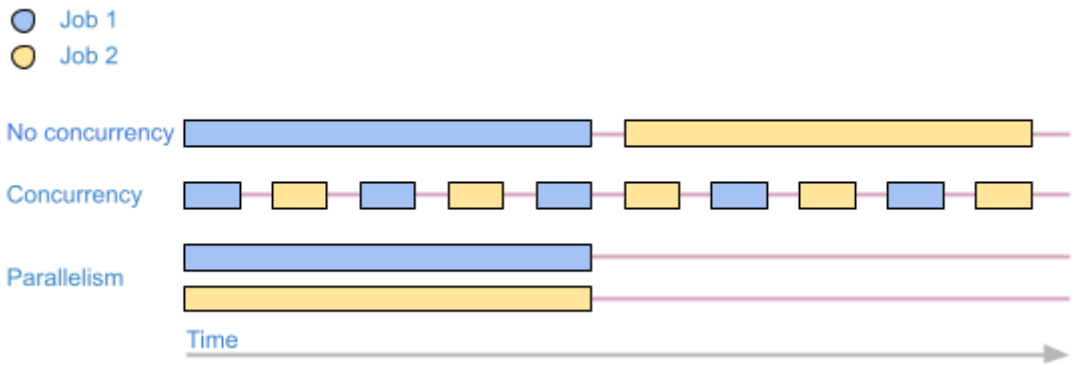


The Python Global Interpreter Lock (GIL) is a lock that allows only one thread to hold the control of the python interpreter.

Concurrency

Concurrency == simulating parallelism by switching context.

Executing multiple tasks at the same time but not necessarily simultaneously.



Sequential execution

In [40]:

```
import httpx


def job(n):
    print(f"--- request {n} sent")
    httpx.get(f"https://example.com/{n}")
    print(f"--- response {n} received")
```

In [41]:

```
%%time

job(1)
job(2)
```

```
--- request 1 sent
--- response 1 received
--- request 2 sent
--- response 2 received
CPU times: user 47.7 ms, sys: 7.25 ms, total: 54.9 ms
Wall time: 2.43 s
```

Threads

A thread is an execution context, which is all the information a CPU needs to execute a stream of instructions.

Switching context every sys.getswitchinterval() (5ms default)

In [1]:

```
import sys


print(sys.getswitchinterval())
```

```
0.005
```

In [43]:

```
%%time

from functools import partial
from threading import Thread


thread_1 = Thread(target=partial(job, n=1))
thread_2 = Thread(target=partial(job, n=2))

thread_1.start()
thread_2.start()

thread_1.join()
thread_2.join()
```

```
--- request 1 sent
--- request 2 sent
--- response 1 received
--- response 2 received
CPU times: user 37.3 ms, sys: 14.2 ms, total: 51.5 ms
Wall time: 1.09 s
```

Async (cooperative multitasking with coroutines)

Concurrency implementation that uses a single thread.

Parts of an application cooperate to switch tasks explicitly at optimal times.

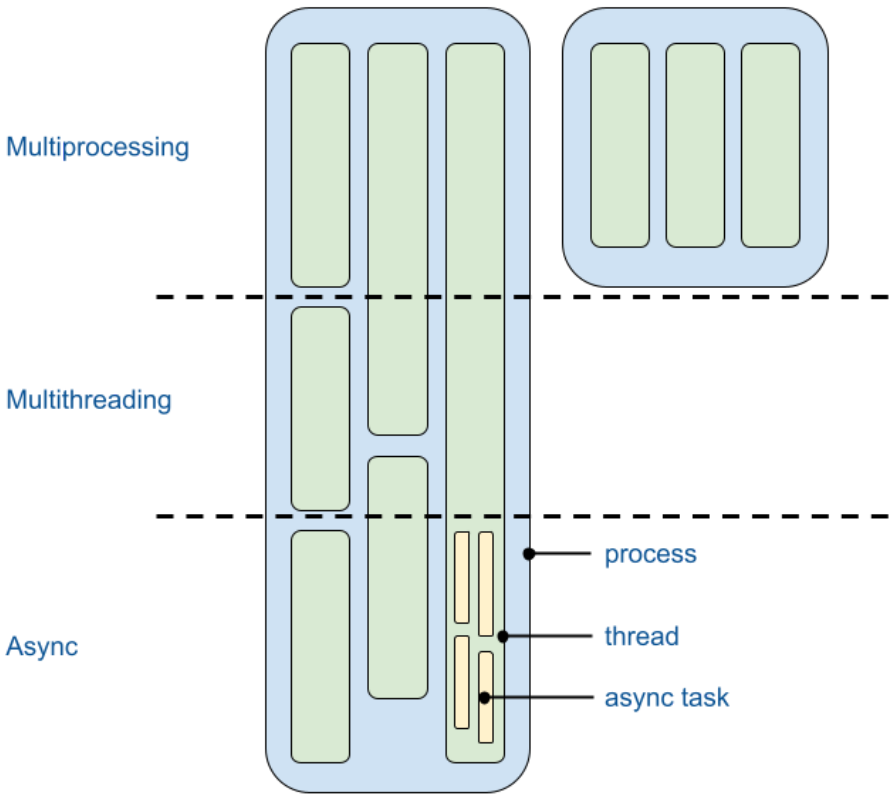
In [44]:

```
async def ajob(client, n):
    print(f"--- request {n} sent")
    await client.get(f"https://example.com/{n}")
    print(f"--- response {n} received")
```

In [47]:

```
import asyncio
import time


async def amain():
    async with httpx.AsyncClient() as client:
        await asyncio.gather(
            ajob(client=client, n=1),
            ajob(client=client, n=2)
        )


start = time.time()
await amain()
print(time.time() - start)
```

```
--- request 1 sent
--- request 2 sent
--- response 1 received
--- response 2 received
0.8680598735809326
```



Only one thread can be active inside a process at a time. Only one async task can be active inside a thread at a time.

Why do we need async?

Cons of threads:

- threads are heavier
- the code has to be thread-safe
- switching is not under our control (*)

Cons of async:

- special syntax (steeper learning curve)
- need ioloop to execute
- no blocking code allowed (*)

Timeline

- 1991 - The first python release
- 2001 - Simple generators (PEP 255 (https://peps.python.org/pep-0255/))
- 2002 - Twisted - event driven networking engine
- 2005
  - Python 2.5
  - Coroutines via enhanced generators (PEP 342 (https://peps.python.org/pep-0342/))
- 2008
  - Python 2.6
  - Python 3.0
- 2009 - Tornado opensourced by Facebook (developed by FriendFeed)
- 2010 - Python 2.7
- 2012
  - Python 3.3
  - proposed to make Tulip/asyncio a part of stdlib (the Tulip project is the asyncio module for Python)
  - `yield from` : Syntax for Delegating to a Subgenerator (PEP 380 (https://peps.python.org/pep-0380/) - 2009)
  - `return value = raise StopIteration(value)` in generators
- 2014
  - Python 3.4
  - asyncio is a part of stdlib
- 2015
  - Python 3.5
  - async/await syntax (native coroutines) (PEP 492 (https://peps.python.org/pep-0492/))
- 2016
  - Python 3.6
  - Asynchronous generators (PEP 525 (https://peps.python.org/pep-0525/))
  - Asynchronous comprehensions (PEP 530 (https://peps.python.org/pep-0530/))
- 2018
  - Python 3.7
  - support for generator-based coroutines is deprecated (https://docs.python.org/3.7/library/asyncio-task.html#generator-based-coroutines) and is scheduled for removal in Python 3.10
  - FastAPI first release
  - Tornado integration with asyncio by default (Tornado IOLoop is a wrapper around asyncio.ioloop)
- 2021 - Python 3.10

From callbacks to async

Synchronous

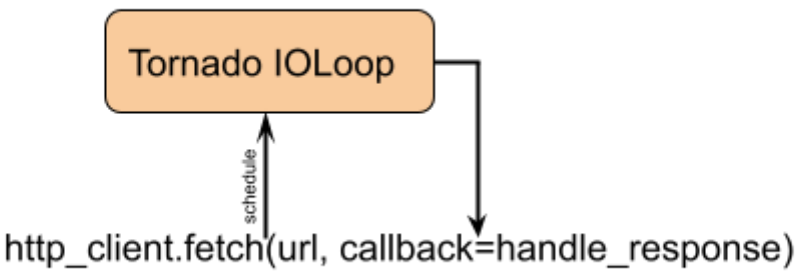
In [ ]:

```
from tornado.httpclient import HTTPClient


def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

With callbacks

In [ ]:

```
from tornado.httpclient import AsyncHTTPClient


def asynchronous_fetch_callbacks(url, callback):
    http_client = AsyncHTTPClient()

    def handle_response(response):
        callback(response.body)

    http_client.fetch(url, callback=handle_response)
```



`AsyncHTTPClient().fetch()` is not blocking the function.

With `Future`

In [ ]:

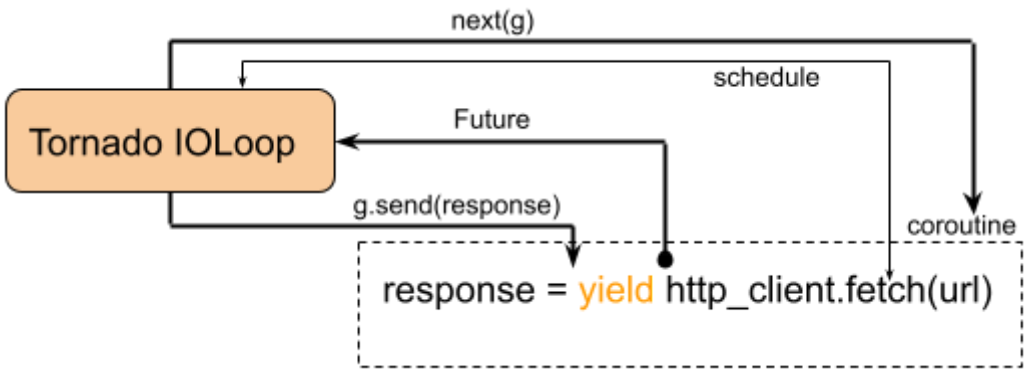
```
from tornado.concurrent import Future
from tornado.httpclient import AsyncHTTPClient


def asynchronous_fetch_future(url):
    http_client = AsyncHTTPClient()
    my_future = Future()
    fetch_future = http_client.fetch(url)

    def on_fetch(f):
        my_future.set_result(f.result().body)

    fetch_future.add_done_callback(on_fetch)
    return my_future
```

With Tornado `gen` (generator based coroutine)

Can be used in Python 2.5+.

In [ ]:

```
from tornado import gen
from tornado.httpclient import AsyncHTTPClient


@gen.coroutine
def asynchronous_fetch_gen(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    raise gen.Return(response.body)
```



`yield from` and `return` for generator based coroutines

In [ ]:

```
@asyncio.coroutine
def asynchronous_fetch_gen(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    return response.body


@asyncio.coroutine
def amain():
    result = yield from asynchronous_fetch_gen('https://example.com')
    return result
```

`yield from` and `return` support was added in Python 3.3 (PEP 380 – Syntax for Delegating to a Subgenerator (https://peps.python.org/pep-0380/)).

With `async` / `await` (native coroutine)

In [ ]:

```
from tornado.httpclient import AsyncHTTPClient


async def asynchronous_fetch(url):
    http_client = AsyncHTTPClient()
    response = await http_client.fetch(url)
    return response.body
```

async/await was added in Python 3.5 (PEP 492 – Coroutines with async and await syntax (https://peps.python.org/pep-0492/)).

Native coroutine (`async def`):

- doesn't require `await`
- runtime warning when garbage collected and not awaited
- can not use `next()` on the coroutine (native coroutine is not a generator)
- can not use `yield from` inside native coroutines
- await validates that the right argument is awaitable

Awaitable:

- Coroutine
- Future
- Task (a subclass of Future)

Future: low-level representation of a future result.

Task: a subclass of Future that knows how to wrap and manage the execution of a coroutine; it is possible to cancel a task by using the task object. When a coroutine is wrapped in a task - automatically scheduled to run soon.

`await` can be used only inside a coroutine.

`async for` : supports async iterator, `__next__` -> `__anext__` .

`async with` : supports async context managers, `__enter__` and `__exit__` -> `__aenter__` and `__aexit__` .

`asyncio.Queue` : not thread safe, put/get are coroutines.

IOLoop

ioloop:

- run asynchronous tasks and callbacks
- perform network IO operations (efficiently handling io events, system events)
- run blocking code in a thread or process pool

In [ ]:

```
# IOLoop Hello World!

import asyncio


def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()


loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

In [ ]:

```
import asyncio
import concurrent.futures


def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)


if __name__ == '__main__':
    ioloop = asyncio.get_event_loop()

    with concurrent.futures.ThreadPoolExecutor() as pool:
        ioloop.run_until_complete(
            ioloop.run_in_executor(
                pool,
                blocking_io
            )
        )

    ioloop.close()
```

In long-running tasks, we can release IOLoop by calling `await asyncio.sleep(0)`.

Selectors (https://docs.python.org/3/library/selectors.html)

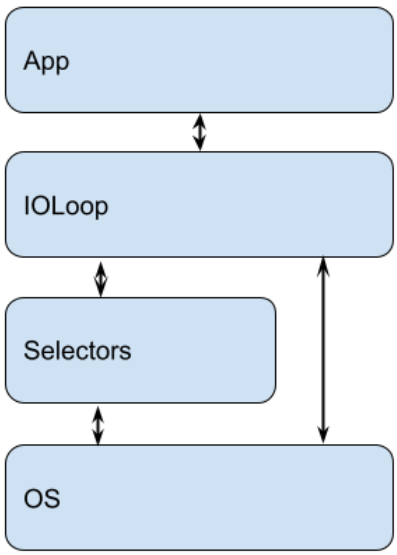
Allows high-level and efficient I/O multiplexing.

Can be used to wait for I/O readiness notification on multiple file objects.

Based on select (https://docs.python.org/3/library/select.html) that provides access to the select() and poll() functions available in most operating systems.

In [ ]:

```
# https://docs.python.org/3/library/selectors.html

import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```



Generators and coroutines

Generators are able to give up control to the caller without losing their state, with an option to resume the execution.

In [7]:

```
import time


def my_generator():
    print('start')
    yield '1'
    yield '2'
    time.sleep(1)
    yield '3'


g = my_generator()
i = next(g)
print(i)

for i in g:
    print(i)


print(type(g))

generator_exp = (i for i in range(3))

print(type(generator_exp))
```

```
start
1
2
3
<class 'generator'>
<class 'generator'>
```

A coroutine is a generator function that can both yield values and accept values from the outside.

In [14]:

```
# Generator based coroutine

def my_generator():
    rec = yield "return 1"
    print(f"Received {rec}")
    rec = yield "return 2"
    print(f"Received {rec}")


g = my_generator()

print(next(g))
print(next(g))
print(g.send('data'))
```

```
return 1
Received None
return 2
Received data

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent cal
Cell In [14], line 14
     12 print(next(g))
     13 print(next(g))
---> 14 print(g.send('data'))

StopIteration:
```

Code examples

In [ ]:

```
# running a coroutine

import asyncio

async def coro():
    await asyncio.sleep(0.5)

asyncio.run(coro())
```

In [ ]:

```
# schedules the coroutine in the current loop
task = asyncio.ensure_future(coro_or_future)
```

In [2]:

```
# async generator

import asyncio


async def gen():
    await asyncio.sleep(1)
    yield 1
    await asyncio.sleep(1)
    yield 2


async for res in gen():
    print(res)
```

```
1
2
```

Async generators support was added in Python 3.6 (PEP 525 – Asynchronous Generators (https://peps.python.org/pep-0525/))

In [ ]:

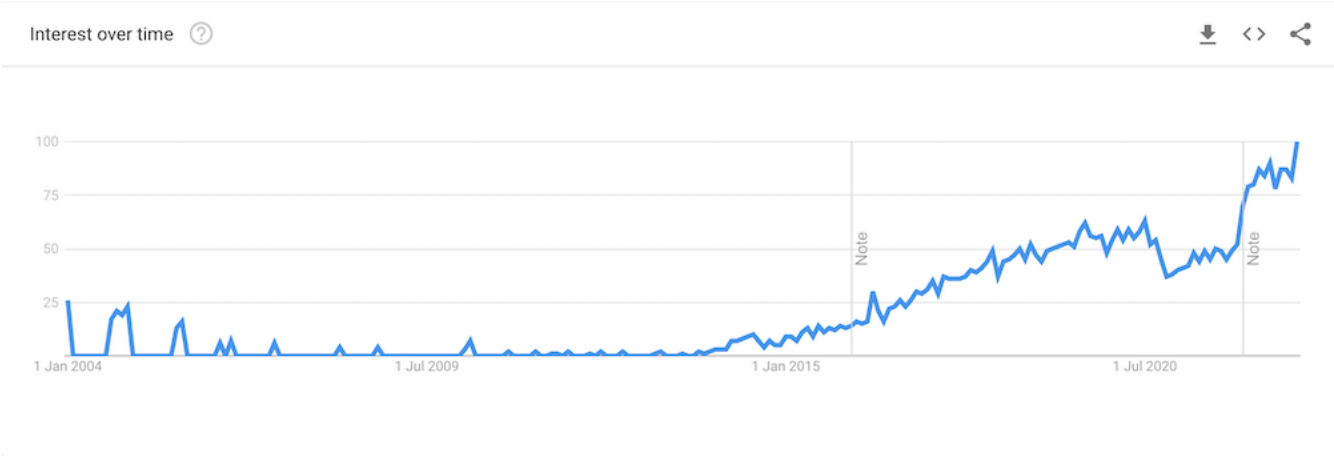
```
# async comprehensions

result = [await fun() for fun in funcs]
result = {await fun() for fun in funcs}
result = {fun: await fun() for fun in funcs}

result = [await fun() for fun in funcs if await smth]
result = {await fun() for fun in funcs if await smth}
result = {fun: await fun() for fun in funcs if await smth}

result = [await fun() async for fun in funcs]
result = {await fun() async for fun in funcs}
result = {fun: await fun() async for fun in funcs}

result = [await fun() async for fun in funcs if await smth]
result = {await fun() async for fun in funcs if await smth}
result = {fun: await fun() async for fun in funcs if await smth}
```

Async comprehensions support was added in Python 3.6 (PEP 530 – Asynchronous Comprehensions (https://peps.python.org/pep-0530/))

Overview

`asyncio` success:

- Growing world connectivity and a request for event-driven networking
- A part of stdlib
- Unified ioloop interface
- Native coroutines and async/await syntax
- Wide support


Interest over time
100
75
50
25
Note
Note
1 Jan 2004
1 Jul 2009
1 Jan 2015
1 Jul 2020