Async I/O Multithreading without threads

소개

- ch06, coroutine => 동기적
  - 망하는 경우 예시 - 외부 리소스 요청 후 hang
- ch07, 비동기적
  - 요청한 리소스가 정말 필요해질 때까지는 다른 일을 비동기적으로 처리
  - node.js, ajax와 동일한 원리
- 역사
  - 1.5부터 asynccore가 있기는 했으나... 망작
  - 그동안 외부 라이브러리 각광, gevent, eventlet
  - 3.4부터 심기일전해서 asyncio 도입
  - 3.5 부터 본격 활약. 3.5부터 쓰세요

new syntax in 3.5

- async def로 선언
- 비동기 결과 획득은 await coroutine()
- 비동기 루프는 async for ... in ...
- async with

a simple example of single-threaded parallel processing

In [ ]:
```
import asyncio

async def sleeper(delay):
    await asyncio.sleep(delay)  # 그냥 sleep는 CPU 소비, 반면 asyncio는 CPU 양보
    print('Finished sleeper with delay: %d' % delay)

loop = asyncio.get_event_loop() # default task switcher
results = loop.run_until_complete(asyncio.wait((    # scala에서 Future.join 정도 의미
    sleeper(1),
    sleeper(3),
    sleeper(2),
)))
```

In [ ]:
```
Finished sleeper with delay: 1
Finished sleeper with delay: 2
Finished sleeper with delay: 3
```

Concepts of asyncio

Future and Tasks

- 결과에 대한 일종의 약속(Promise)
- 결과가 실제 확보되면 등록된 콜백들에서 즉시 result 전달
- 보통 Future보다는 이의 wrapper 격인 Task를 쓴다.
- loop.create_task나 asyncio.ensure_future 같은 API를 통해 생성

Event loops

- CPU scheduler 처럼 등록된 여러 개의 task들에게 CPU 사용을 switch하는 역할
- default loop를 쓰거나 새로 만들거나

In [ ]:
```
loop = asyncio.get_event_loop()
result = loop.call_soon(loop.create_task, sleeper(1))
result = loop.call_later(2, loop.stop)
loop.run_forever()
```

Finished sleeper with delay: 1

In [ ]:
```
# debugging aysnc function

async def stack_printer():
    for task in asyncio.Task.all_tasks():
        task.print_stack()

loop = asyncio.get_event_loop()
result = loop.run_until_complete(stack_printer())
```

Event loop implementations

- async.SelectorEventLoop,
- async.ProactorEventLoop - 성능 우월, 단 windows's completion port
- example
  - example of binding a function to the read event (EVENT_READ) on the standard input.
- Selector
  - SelectSelector
  - KqueueSelector
  - EpollSelector,
  - DevpollSelector

In [ ]:
```
import sys
import selectors
def read(fh):
    print('Got input from stdin: %r' % fh.readline())

if __name__ == '__main__':
    # Create the default selector
    selector = selectors.DefaultSelector()

    # Register the read function for the READ event on stdin
    selector.register(sys.stdin, selectors.EVENT_READ, read)

    while True:
        for key, mask in selector.select():
            # The data attribute contains the read function here
            callback = key.data

            # Call it with the fileobj (stdin here)
            callback(key.fileobj)
```

Event loop usage

- loop.run_until_complete
- loop.run_forever
- add tasks
  - ready된 task를 위한 큐 1개 + 스케줄링된 task를 위한 큐 1개
    - ready 큐를 다 수행시에는 스케줄 due된 task룰 ready 큐로 옮겨서 수행
  - call_soon,call_soon_threadsafe => ready_queue
  - call_later : 최소 지연 시간 지정, 스케줄 큐에 추가
  - call_at : 특정 시간에 수행, 스케줄 큐에 추가
- task 추가 후 cancel을 위한 handle 리턴, 단 cancel은 thread0-safe 하지 않음

In [ ]:
```
import time
import asyncio

t = time.time()
def printer(name):
    print('Started %s at %.1f' % (name, time.time() - t))
    time.sleep(0.2)
    print('Finished %s at %.1f' % (name, time.time() - t))

loop = asyncio.get_event_loop()
result = loop.call_at(loop.time() + .2, printer, 'call_at')
result = loop.call_later(.1, printer, 'call_later')
result = loop.call_soon(printer, 'call_soon')
result = loop.call_soon_threadsafe(printer, 'call_soon_threadsafe')

# Make sure we stop after a second
result = loop.call_later(1, loop.stop)
loop.run_forever()

# 결과는 그냥 순차 수행.. time.sleep 때문
```

In [ ]:
```
Started call_soon at 0.0
Finished call_soon at 0.2
Started call_soon_threadsafe at 0.2
Finished call_soon_threadsafe at 0.4
Started call_later at 0.4
Finished call_later at 0.6
Started call_at at 0.6
Finished call_at at 0.8
```

In [ ]:
```
import time
import asyncio

t = time.time()
def printer(name):
    print('Started %s at %.1f' % (name, time.time() - t))
    await asyncio.sleep(0.2)
    print('Finished %s at %.1f' % (name, time.time() - t))

loop = asyncio.get_event_loop()
result = loop.call_at(loop.time() + .2, printer, 'call_at')
result = loop.call_later(.1, printer, 'call_later')
result = loop.call_soon(printer, 'call_soon')
result = loop.call_soon_threadsafe(printer, 'call_soon_threadsafe')

# Make sure we stop after a second
result = loop.call_later(1, loop.stop)
loop.run_forever()
```

In [ ]:
```
Started call_soon at 0.0
Started call_soon_threadsafe at 0.0
Started call_later at 0.1
Started call_at at 0.2
Finished call_soon at 0.2
Finished call_soon_threadsafe at 0.2
Finished call_later at 0.3
Finished call_at at 0.4
```

Process

- 외부의 long-running task와 연동해야 할 경우

In [ ]:
```
# 아래는 프로세스 순차 수행

import time
import subprocess

t = time.time()

def process_sleeper():
    print('Started sleep at %.1f' % (time.time() - t))
    process = subprocess.Popen(['sleep', '0.1'])
    process.wait()
    print('Finished sleep at %.1f' % (time.time() - t))
for i in range(3):
    process_sleeper()
```

In [ ]:
```
# 아래는 병렬 프로세스 수행이지만 messy code

import time
import subprocess

t = time.time()

def process_sleeper():
    print('Started sleep at %.1f' % (time.time() - t))
    return subprocess.Popen(['sleep', '0.1'])

processes = []
for i in range(5):
    processes.append(process_sleeper())

for process in processes:
    returncode = process.wait()
    print('Finished sleep at %.1f' % (time.time() - t))
```

In [ ]:
```
# 아래는 asyncio 기반 병렬 프로세스 수행

import time
import subprocess

t = time.time()

async def async_process_sleeper():
    print('Started sleep at %.1f' % (time.time() - t))
    process = await asyncio.create_subprocess_exec('sleep', '0.1')
    await process.wait()
    print('Finished sleep at %.1f' % (time.time() - t))

loop = asyncio.get_event_loop()
for i in range(5):
    task = loop.create_task(async_process_sleeper())

future = loop.call_later(.5, loop.stop)
loop.run_forever()
```

In [ ]:
```
# 이제는 프로세스의 수행의 리턴 결과가 있는 경우

import asyncio

async def run_script():
    process = await asyncio.create_subprocess_shell(
        'python3',
        stdout=asyncio.subprocess.PIPE,
        stdin=asyncio.subprocess.PIPE,
    )

    # Write a simple Python script to the interpreter
    process.stdin.write(b'\n'.join((
        b'import math',
        b'x = 2 ** 8',
        b'y = math.sqrt(x)',
        b'z = math.sqrt(y)',
        b'print("x: %d" % x)',
        b'print("y: %d" % y)',
        b'print("z: %d" % z)',
        b'for i in range(int(z)):',
        b' print("i: %d" % i)',
    )))

    # Make sure the stdin is flushed asynchronously
    await process.stdin.drain()

    # And send the end of file so the Python interpreter will
    # start processing the input. Without this the process will
    # stall forever.
    process.stdin.write_eof()

    # Fetch the lines from the stdout asynchronously
    async for out in process.stdout:
        # Decode the output from bytes and strip the whitespace
        # (newline) at the right
        print(out.decode('utf-8').rstrip())

    # Wait for the process to exit
    await process.wait()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run_script())
    loop.close()
```

Asynchronous servers and clients

In [ ]:
```
import time
import sys
```

```
import asyncio

HOST = '127.0.0.1'
PORT = 1234
start_time = time.time()

def printer(start_time, *args, **kwargs):
    '''Simple function to print a message prefixed with the
    time relative to the given start_time'''
    print('%.1f' % (time.time() - start_time), *args, **kwargs)

async def handle_connection(reader, writer):
    client_address = writer.get_extra_info('peername')
    printer(start_time, 'Client connected', client_address)
    # Send over the server start time to get consistent timestamps
    writer.write(b'%.2f\n' % start_time)
    await writer.drain()

    repetitions = int((await reader.readline()))
    printer(start_time, 'Started sending to', client_address)
    for i in range(repetitions):
    message = 'client: %r, %d\n' % (client_address, i)
    printer(start_time, message, end='')
    writer.write(message.encode())
    await writer.drain()

    printer(start_time, 'Finished sending to', client_address)
    writer.close()

async def create_connection(repetitions):
    reader, writer = await asyncio.open_connection(host=HOST, port=
PORT)
    start_time = float((await reader.readline()))
    writer.write(repetitions.encode() + b'\n')
    await writer.drain()

    async for line in reader:
        # Sleeping a little to emulate processing time and make
        # it easier to add more simultaneous clients
        await asyncio.sleep(1)
        printer(start_time, 'Got line: ', line.decode(), end='')
        writer.close()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    if sys.argv[1] == 'server':
        server = asyncio.start_server(
            handle_connection,
            host=HOST,
            port=PORT,
        )
        running_server = loop.run_until_complete(server)
        try:
```

```
            result = loop.call_later(5, loop.stop)
            loop.run_forever()
        except KeyboardInterrupt:
            pass
        running_server.close()
        loop.run_until_complete(running_server.wait_closed())
    elif sys.argv[1] == 'client':
        loop.run_until_complete(create_connection(sys.argv[2]))
        loop.close()
```

In [ ]:
```
# python3 simple_connections.py server
0.4 Client connected ('127.0.0.1', 59990)
0.4 Started sending to ('127.0.0.1', 59990)
0.4 client: ('127.0.0.1', 59990), 0
0.4 client: ('127.0.0.1', 59990), 1
0.4 client: ('127.0.0.1', 59990), 2
0.4 Finished sending to ('127.0.0.1', 59990)
2.0 Client connected ('127.0.0.1', 59991)
2.0 Started sending to ('127.0.0.1', 59991)
2.0 client: ('127.0.0.1', 59991), 0
2.0 client: ('127.0.0.1', 59991), 1
2.0 Finished sending to ('127.0.0.1', 59991)

The first client:
# python3 simple_connections.py client 3
1.4 Got line: client: ('127.0.0.1', 59990), 0
2.4 Got line: client: ('127.0.0.1', 59990), 1
3.4 Got line: client: ('127.0.0.1', 59990), 2
```