COMP S362F Unit 12 Asynchronous programming

Kendrew Lau

Contents

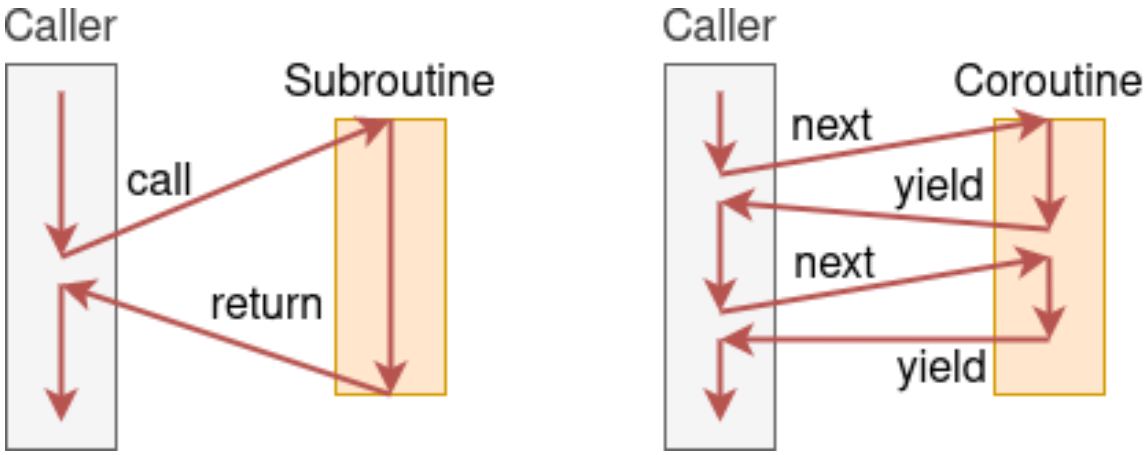
Asynchronous programming and coroutines

Asynchronous programming

- *Asynchronous programming* refers to sending code to run outside the current flow of execution and continuing without waiting for the code to complete running
- Can be supported by different mechanisms
    - Using callbacks
    - Using future or promise objects
    - Using coroutines
    - Event-driven programming
    - Reactive programming
- Python built-in module `asyncio` uses coroutines

Subroutines and coroutines

- A *subroutine*, when invoked, runs to completion when it reaches a "return" statement or the end of its code
    - Normal Python functions or methods
- A *coroutine* is a generalized subroutine that allows execution to be suspended and resumed
    - Support cooperative (i.e. non-preemptive) multitasking



Python concurrency solutions

Solution	Module	Concurrency type	Number of CPUs	Suitable tasks
Processes	`multiprocessing`	Real parallelism	Multiple CPUs	CPU-bound
Threads	`threading`	Preemptive multitasking	One CPU	I/O-bound
Asynchronous I/O	`asyncio`	Cooperative multitasking	One CPU	I/O-bound

Python concurrency example

```
# File: compare_conc.py
import asyncio, multiprocessing, threading, time

CONC_UNITS, N = 2, 3

def log(msg):
    pname = multiprocessing.current_process().name
    tname = threading.current_thread().name
    print(f"{pname} {tname}: {msg}", flush=True)

def counter():
    for i in range(N):
        time.sleep(0.1)
        log(i)

def do_processes():
    processes = [multiprocessing.Process(target=counter)
                 for i in range(CONC_UNITS)]
    for p in processes: p.start()
    for p in processes: p.join()
```

Python concurrency example (cont.)

```
# File: compare_conc.py (cont.)
def do_threads():
    threads = [threading.Thread(target=counter)
               for i in range(CONC_UNITS)]
    for t in threads: t.start()
    for t in threads: t.join()

async def co_counter():
    for i in range(N):
        await asyncio.sleep(0.1)
        log(i)

def do_coroutines():
    async def co_counters():
        coroutines = [co_counter() for i in range(CONC_UNITS)]
        await asyncio.gather(*coroutines)
    asyncio.run(co_counters())

if __name__ == "__main__":
    do_processes()
    do_threads()
    do_coroutines()
```

Python concurrency example (cont.)

```
# Output:
```

```
Process-1 MainThread: 0
Process-2 MainThread: 0
Process-1 MainThread: 1
Process-2 MainThread: 1
Process-1 MainThread: 2
Process-2 MainThread: 2
MainProcess Thread-8: 0
MainProcess Thread-9: 0
MainProcess Thread-8: 1
MainProcess Thread-9: 1
MainProcess Thread-8: 2
MainProcess Thread-9: 2
MainProcess MainThread: 0
MainProcess MainThread: 0
MainProcess MainThread: 1
MainProcess MainThread: 1
MainProcess MainThread: 2
MainProcess MainThread: 2
```

Python generators

Python generators

- A *generator* is like a "simplified" coroutine for generating values
  - No real support of concurrency
  - May maintain some internal states
- A *generator function* is a function that uses the `yield` keyword in the function body
  - `yield x` passes the value `x` to the caller, and pauses execution
- Calling a generator function returns a generator object
  - But the code inside the function is not (yet) executed!
- `next(gen_obj)` obtains the next value from `gen_obj`
  - Result in `StopIteration` if there is no more values

Generator next() example

```
# File: generator_next.py
N = 3
def gen_numbers():
    for i in range(N):
        print("Generator generates", i)
        yield i

if __name__ == "__main__":
    gen_obj = gen_numbers()
    print("Caller creates", gen_obj)
    for i in range(N):
        n = next(gen_obj)
        print("Caller gets", n)
    #n = next(gen_obj)  # raises StopIteration
```

```
Caller creates <generator object gen_numbers at 0x7f95881a19e0>
Generator generates 0
Caller gets 0
Generator generates 1
Caller gets 1
Generator generates 2
Caller gets 2
```

Using generators

- `next()` is rarely used
- Typical usage of a generator
  - Use it in a for-loop, obtaining the generated values one by one
  - Convert it to a list, obtaining the generated values all at once
- A generator can generate infinite values
  - The caller should implement logic to stop, i.e. obtain finite values only
- A generator can be created using *generator comprehension*
  - E.g. `(2*n for n in range(10))`

Generator loop/list example

```python
# File: generator_loop_list.py
N = 3
def gen_numbers():
    for i in range(N):
        print("Generator generates", i, end="; ")
        yield i

if __name__ == "__main__":
    for n in gen_numbers():
        print("Caller gets", n)
    ns = list(gen_numbers())
    print("Caller gets a list", ns)
```

```
Generator generates 0; Caller gets 0
Generator generates 1; Caller gets 1
Generator generates 2; Caller gets 2
Generator generates 0; Generator generates 1; Generator generates 2;
Caller gets a list [0, 1, 2]
```

Generator infinite example

```python
# File: generator_infinite.py
def fibonacci():
    f0, f1 = 0, 1
    while True:
        yield f0
        f0, f1 = f1, f0+f1

if __name__ == "__main__":
    for i, fib in enumerate(fibonacci()):
        print(i, fib, end=", ")
        if i == 10:
            break
    print()
    print(fibonacci())  # OK
    #print(list(fibonacci()))  # Don't do this!
```

```
0 0, 1 1, 2 1, 3 2, 4 3, 5 5, 6 8, 7 13, 8 21, 9 34, 10 55,
<generator object fibonacci at 0x7fd4285b0740>
```

Generator comprehension example

```
# File: generator_comp.py
if __name__ == "__main__":
    gen = (2*n for n in range(10))
    print(gen)
    print(list(gen))
    total = sum(n for n in range(11))
    print(total)
```

```
<generator object <genexpr> at 0x7fd4285b0120>
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
55
```

Python asyncio basics

The asyncio module

- Python doc: `asyncio` is a library to write concurrent code using the `async`/`await` syntax
  - Good for I/O-bound code
- Support *event loops* for running coroutines concurrently
  - An event loop is the “caller” of all coroutines, executing their code in one thread
  - To perform blocking/long operation, a coroutine suspends itself and lets the event loop run another coroutine, i.e. cooperative multitasking

Coroutines and the async keyword

- A coroutine is defined using `async def` in a *coroutine function* definition
  - Syntax: `async def co_fn(...): ...`
- Calling a coroutine function returns a coroutine object
  - But the code inside the function is not (yet) executed!
- Main ways of executing a coroutine
  - Schedule it to run and wait for its completion, with `asyncio` setup: `asyncio.run(co_fn())`
  - Schedule it to run: `asyncio.gather(co_fn(), co_fn())`, `asyncio.create_task(co_fn())`
  - Schedule it to run and await for its completion: `await co_fn()`

Things a coroutine may do

- Things that a coroutine function may do but a normal function may not do
  - `await x`, where `x` is called an “awaitable”, suspends the current coroutine and waits for `x` to complete execution
  - `async with x`, where `x` is an *asynchronous context manager*
  - `async for i in x`, where `x` is an *asynchronous iterator*
- A coroutine may but should not invoke blocking/long operations directly
  - E.g. `time.sleep()`, `socket.recv()`

Awaitables and the await keyword

- An *awaitable* object is one that can be used in an `await` expression
  - The `x` in the expression `await x`
- Three main types of awaitable objects
  - Coroutines, which can be awaited from other coroutines
  - Tasks, which wrap coroutines for scheduling execution, e.g. `task = asyncio.create_task(co_fn())`
  - Futures, which represent eventual results of asynchronous operations (low-level)
- Calling `await asyncio.sleep(0)` suspends the current coroutine
  - Also known as to “yield execution” and “yield the CPU”

Async hello example

```python
# File: async_hello.py
import asyncio

async def hello():
    print("hello")
    await asyncio.sleep(1)
    print("world")

asyncio.run(hello())
```

```
hello
world
```

Async counter example

```python
# File: async_counter.py
import asyncio

COROUTINES, N = 5, 10000
counter = 0

async def inc():
    global counter
    for i in range(N):
        counter += 1
        await asyncio.sleep(0)

async def main():
    coroutines = [inc() for i in range(COROUTINES)]
    await asyncio.gather(*coroutines)
    print(counter)

asyncio.run(main())
```

```
50000
```

Coroutines and blocking operations

- A coroutine may but should not invoke blocking/long operations directly
  - E.g. `time.sleep()`, `socket.recv()`
- Arrange the blocking code to run in a thread/pool executor
- `loop = asyncio.get_running_loop()`: get the event loop
- `loop.run_in_executor(executor, func, *args)`: arrange for `func` to be called in the specified executor
  - Specify `None` for `executor` to use the default built-in executor

Async executor example

```
# File: async_executor.py
import asyncio, datetime, socket, timeit

def get_page(server):
    print(f"{datetime.datetime.now()}: {server} starts")
    with socket.socket() as s:
        s.connect((server, 80))
        s.send(b"GET / HTTP/1.0\r\n\r\n")
        file = s.makefile("rb")
        data = file.read()
    print(f"{datetime.datetime.now()}: {server} ends")
    return data

SERVERS = ("www.hkmu.edu.hk", "ole.hkmu.edu.hk", "www.google.com")
```

Async executor example (cont.)

```
# File: async_executor.py (cont.)
async def main1():
    start_time = timeit.default_timer()
    loop = asyncio.get_event_loop()
    for server in SERVERS:
        data = await loop.run_in_executor(None, get_page, server)
        print(f"{len(data)} bytes: {data[:40]} ...")
    print(f"main1(): {timeit.default_timer() - start_time:.4f}s")

async def main2():
    start_time = timeit.default_timer()
    loop = asyncio.get_event_loop()
    futures = [loop.run_in_executor(None, get_page, server)
               for server in SERVERS]
    results = await asyncio.gather(*futures)
    for data in results:
        print(f"{len(data)} bytes: {data[:40]} ...")
    print(f"main2(): {timeit.default_timer() - start_time:.4f}s")

asyncio.run(main1())
asyncio.run(main2())
```

Async executor example (cont.)

```
# Output:
```

```
2021-12-10 15:54:41.785748: www.hkmu.edu.hk starts
2021-12-10 15:54:41.836901: www.hkmu.edu.hk ends
96 bytes: b'HTTP/1.0 302 Found\r\nLocation: https://' ...
2021-12-10 15:54:41.837739: ole.hkmu.edu.hk starts
2021-12-10 15:54:41.880460: ole.hkmu.edu.hk ends
241 bytes: b'HTTP/1.1 302 Found\r\nServer: Lotus-Domino' ...
2021-12-10 15:54:41.881284: www.google.com starts
2021-12-10 15:54:42.012177: www.google.com ends
51504 bytes: b'HTTP/1.0 200 OK\r\nDate: Fri, 10 Dec 2021 ' ...
main1(): 0.2277s
2021-12-10 15:54:42.015348: www.hkmu.edu.hk starts
2021-12-10 15:54:42.015813: ole.hkmu.edu.hk starts
2021-12-10 15:54:42.016565: www.google.com starts
2021-12-10 15:54:42.056568: www.hkmu.edu.hk ends
2021-12-10 15:54:42.059583: ole.hkmu.edu.hk ends
2021-12-10 15:54:42.138279: www.google.com ends
96 bytes: b'HTTP/1.0 302 Found\r\nLocation: https://' ...
241 bytes: b'HTTP/1.1 302 Found\r\nServer: Lotus-Domino' ...
51448 bytes: b'HTTP/1.0 200 OK\r\nDate: Fri, 10 Dec 2021 ' ...
main2(): 0.1241s
```

Python asyncio networking

Python asyncio networking

- `asyncio` supplies asynchronous functions and streams to work with network connections
- `open_connection(host, port, ...)`: establish a client network connection and return `(reader, writer)` (coroutine)
- `start_server(client_connected_cb, host, port, ...)`: start a socket server (coroutine)
    - `async def client_connected_cb(reader, writer)` is called whenever a new client connection is established

StreamReader and StreamWriter

- Class `StreamReader`: for reading data from an I/O stream
  - `read(max_bytes=-1)` *coroutine*: read up to a maximum number of bytes
  - `readline()` *coroutine*: read a line, which ends with `\n`
- Class `StreamWriter`: for writing data to an I/O stream
  - `write(data)`: attempt to write data to the underlying socket immediately
  - `writelines(data)`: attempt to write a list of bytes to the underlying socket immediately
  - `close()`: close the stream and the underlying socket
  - `drain()` *coroutine*: wait until it is appropriate to resume writing to the stream (called after `write()` and `writelines()`)
  - `wait_closed()` *coroutine*: wait until the stream is closed (called after `close()`)

HTTP client example

```python
# File: http_client.py
import asyncio, datetime, timeit

async def get_page(server):
    print(f"{datetime.datetime.now()}: {server} starts")
    reader, writer = await asyncio.open_connection(server, 80)
    writer.write(b"GET / HTTP/1.0\r\n\r\n")
    await writer.drain()
    data = await reader.read()
    writer.close()
    await writer.wait_closed()
    print(f"{datetime.datetime.now()}: {server} ends")
    return data
```

HTTP client example (cont.)

```python
# File: http_client.py
SERVERS = ("www.hkmu.edu.hk", "ole.hkmu.edu.hk", "www.google.com")

async def main():
    start_time = timeit.default_timer()
    coroutines = [get_page(server) for server in SERVERS]
    results = await asyncio.gather(*coroutines)
    for data in results:
        print(f"{len(data)} bytes: {data[:40]} ...")
    print(f"main(): {timeit.default_timer() - start_time:.4f}s")

asyncio.run(main())
```

HTTP client example (cont.)

```
# Output:
```

```
2021-12-10 16:00:37.901993: www.hkmu.edu.hk starts
2021-12-10 16:00:37.902843: ole.hkmu.edu.hk starts
2021-12-10 16:00:37.903171: www.google.com starts
2021-12-10 16:00:37.948363: www.hkmu.edu.hk ends
2021-12-10 16:00:37.949681: ole.hkmu.edu.hk ends
2021-12-10 16:00:38.030133: www.google.com ends
96 bytes: b'HTTP/1.0 302 Found\r\nLocation: https://' ...
241 bytes: b'HTTP/1.1 302 Found\r\nServer: Lotus-Domino' ...
51502 bytes: b'HTTP/1.0 200 OK\r\nDate: Fri, 10 Dec 2021 ' ...
main(): 0.1285s
```

Echo server example

```python
# File: echo_server.py
import asyncio, logging, sys
HOST, PORT = "localhost", 7000

async def handle_client(reader, writer):
    addr = writer.get_extra_info("peername")
    logging.info(f"Client connected from {addr}")
    while True:
        data = await reader.read(1024)
        if len(data) <= 0:
            break
        writer.write(data)
        await writer.drain()
    logging.info(f"Client disconnected from {addr}")
    writer.close()
    await writer.wait_closed()
```

Echo server example (cont.)

```python
# File: echo_server.py (cont.)
async def echo_server(host, port):
    server = await asyncio.start_server(handle_client, host, port)
    logging.info(f"Server started, listening on {(host, port)}")
    async with server:
        await server.serve_forever()

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    host = sys.argv[1] if len(sys.argv) > 1 else HOST
    port = int(sys.argv[2]) if len(sys.argv) > 2 else PORT
    asyncio.run(echo_server(host, port))
```

```
INFO:root:Server started, listening on ('localhost', 7000)
INFO:root:Client connected from ('::1', 45148, 0, 0)
INFO:root:Client disconnected from ('::1', 45148, 0, 0)
```

Echo client example

```python
# File: echo_client.py
import asyncio, random, sys
HOST, PORT = "localhost", 7000

async def echo_client(host, port):
    reader, writer = await asyncio.open_connection(host, port)
    n = random.randint(1, 100)
    text = f"Hello, {n} is my lucky number!"
    writer.write(text.encode("utf-8"))
    await writer.drain()
    print("Sent to server:       ", text)
    data = await reader.read(1024)
    text = data.decode("utf-8")
    print("Received from server: ", text)
    writer.close()
    await writer.wait_closed()
```

Echo client example (cont.)

```python
# File: echo_client.py (cont.)
if __name__ == "__main__":
    host = sys.argv[1] if len(sys.argv) > 1 else HOST
    port = int(sys.argv[2]) if len(sys.argv) > 2 else PORT
    asyncio.run(echo_client(host, port))
```

```
Sent to server:         Hello, 3 is my lucky number!
Received from server:   Hello, 3 is my lucky number!
```

The end