Socket Programming

유명성

0. Thread

0. Thread

0.1 Thread vs Process

Thread

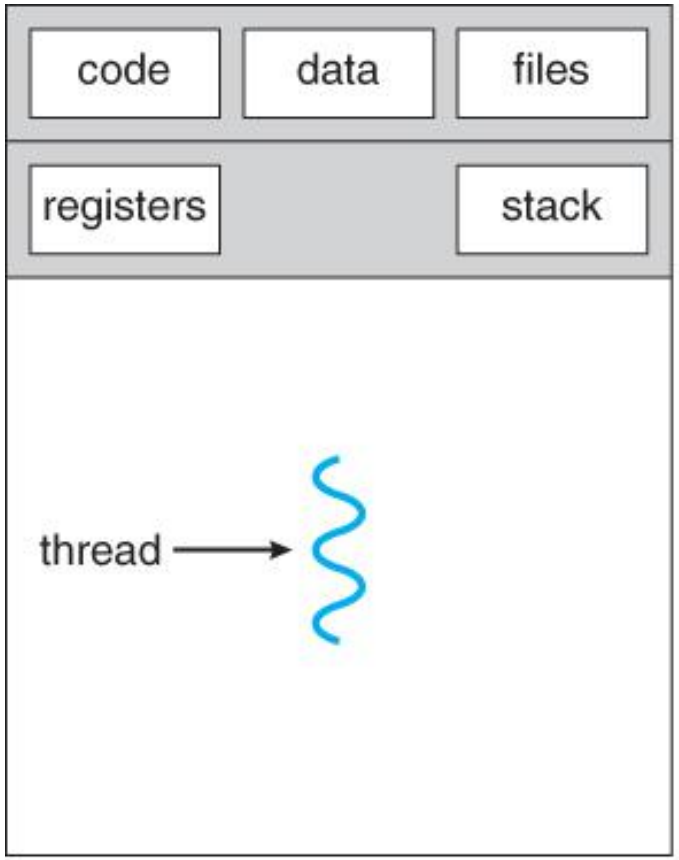
- ❖ 스레드는 경량 프로세스라고도 불리며, 프로세스 내에 존재하는 하나의 독립적인 실행 흐름이다.
- ❖ 한 프로세스 내에서 다른 스레드와 Code, Heap, Data 영역을 공유하며 Stack 영역만 독립적으로 가진다.
- ❖ 스레드는 OS 입장에서 실행의 단위이다.

Process

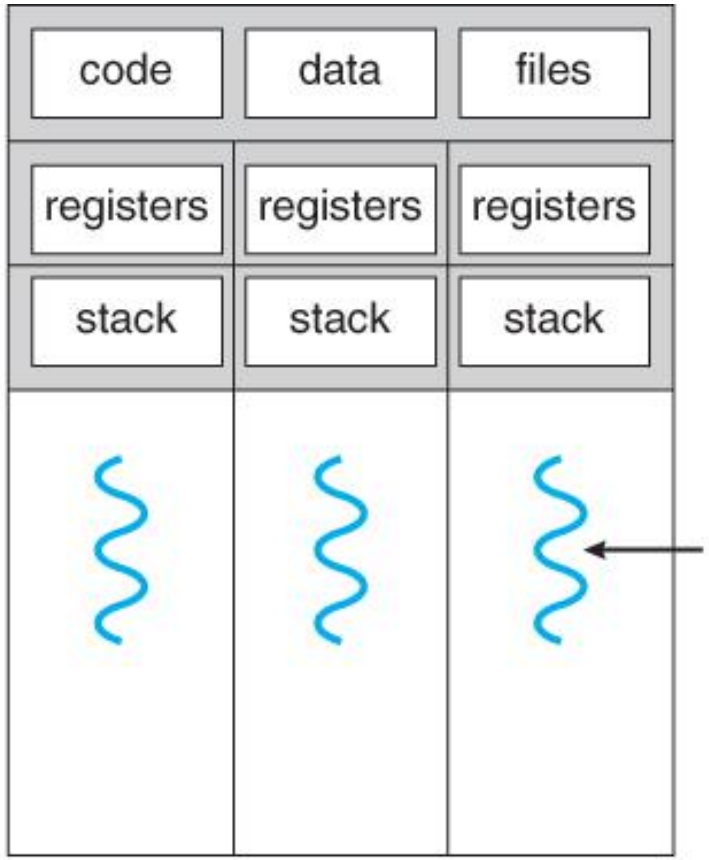
- ❖ 프로세스는 메모리에 올라가 실행중인 프로그램을 뜻한다.
- ❖ 각 프로세스의 메모리 영역은 서로 독립적이며, 가상 메모리에 매핑되기 때문에 서로 알 수 없다.
- ❖ 프로세스는 내부에 다수의 스레드를 가질 수 있다.
- ❖ 프로세스는 OS 입장에서 자원을 할당하는 단위이다.

0. Thread

0.2 Thread in Memory



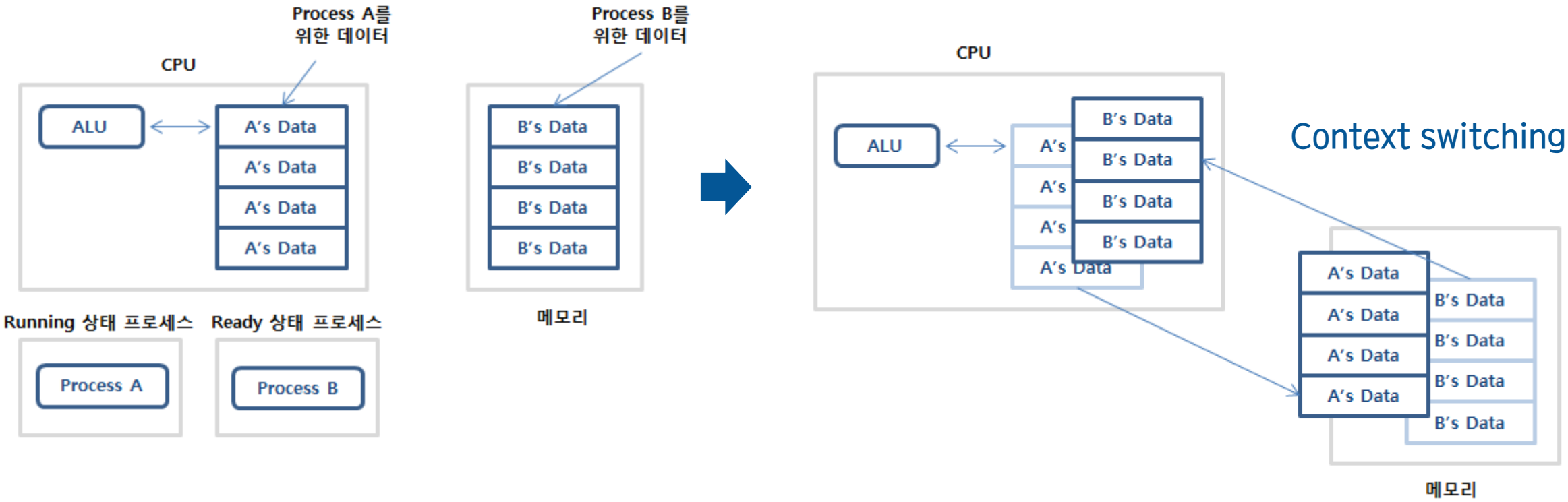
single-threaded process



multithreaded process

0. Thread

0.3 Context switching



프로그램에서 Thread를 왜 사용할까?

- 실행흐름을 나누기 위해
  - 화면을 보여주는 스레드와 내부에서 비즈니스 로직을 실행하는 스레드를 나누기 위해
  - GUI 등에서 스레드를 하나만 사용할 경우 내부 로직을 수행하면 화면이 멈추게 된다.
  - 즉 동시에 여러가지 일을 처리하기 위해 사용한다.

- 속도를 높이기 위해
  - 스레드 하나로 하면 10초 걸리는 작업을 스레드 10개로 하면 10초보다 빠르게 끝낼 수 있다.
  - 단 작업을 나누고 결과를 합치는데 오버헤드가 발생하며, 나눌 수 없는 작업도 존재한다.

Threading 모듈

- Python에서 Thread를 High-level에서 다룰 수 있게 해주는 모듈
- theading.Thread() 함수에 인자를 전달해 사용하거나, threading.Thread 객체를 상속받은 클래스의 Run() 메소드를 오버라이딩해 thread를 사용할 수 있다.

```
import threading

def print_msg(msg):
    print("{0} in sub thread : [id {1}]".format(msg, threading.get_ident()))

msg = "Hello world!!!"
print("{0} in main thread : [id {1}]".format(msg, threading.get_ident()))

thread_1 = threading.Thread(target=print_msg, args=(msg, )) # Tread() 메소드를 통해 스레드 생성
thread_1.start() # 생성된 스레드를 실행
thread_1.join() # 실행시킨 스레드가 끝날 때까지 대기
```

```
Hello world!!! in main thread : [id 14316]
Hello world!!! in sub thread : [id 12052]
```

Daemon Thread

- ❖ 데몬 스레드는 백그라운드에서 실행되는 스레드로, 파이썬 스크립트는 데몬 스레드만 남게될 경우 바로 종료된다.
- ❖ 즉 메인 스레드 등 데몬이 아닌 스레드가 모두 종료되었고, 현재 데몬 스레드들만 실행 중 이라면 파이썬 스크립트는 즉시 종료된다.

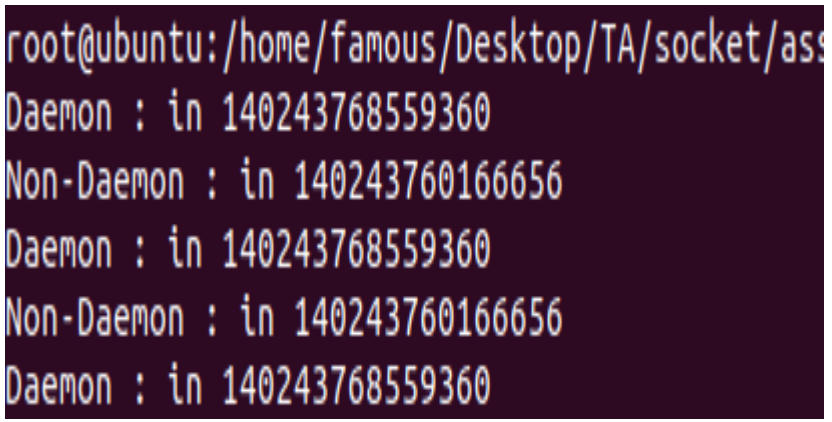
```
import threading
import time

def func(value, count):
    while count > 0:
        print("{0} : in {1}".format(value, threading.get_ident()))
        time.sleep(1)
        count -= 1

t1 = threading.Thread(target=func, args=("Daemon", 5), daemon=True)
t2 = threading.Thread(target=func, args=("Non-Daemon", 2))
t1.start()
t2.start()
```

0. Thread

0.5 Daemon Thread

```
Daemon : in 14692
Non-Daemon : in 2352
Daemon : in 14692
Non-Daemon : in 2352
Daemon : in 14692
Daemon : in 14692
Daemon : in 14692
```

Jupyter(console)



Ubuntu(script)

Jupyer, IDLE 등 ipython 콘솔에서 실행하면 메인 스레드가 콘솔을 실행하는 스레드임으로
Daemon으로 설정해도 콘솔이 종료되지 않는 한 Daemon 스레드는 작업을 계속 수행한다.

Thread-safe

❖ Thread-safe하다는 것은 다수의 thread가 특정 자원에 대해 race-condition을 일으키지 않고 각자의 일을 수행할 수 있음을 뜻한다.

```
import threading

x = 0 # A shared value

def foo(amount, value):
    global x

    for i in range(amount):
        x += value

t1 = threading.Thread(target=foo, args=(1000000, 1))
t2 = threading.Thread(target=foo, args=(1000000, -1))
t1.start()
t2.start()
t1.join()
t2.join()

print("Shared value : {0}".format(x))
```

```
Shared value : 518327
```

Thread-safe하지 않은 예, 실행할 때마다 x의 값이 달라진다.

Thread 동기화

❖ 동기화는 공유 자원에 접근하는 스레드들을 제어해, 공유 자원에 대한 작업이 순차적으로 진행되게 한다.

```
import threading

x = 0 # A shared value
lock = threading.Lock() # Mutex

def foo(amount, value):
    global x
    lock.acquire() # Mutex를 얻기위해 기다린다. running -> waitting

    for i in range(amount): # Mutex를 얻은 후에 실행된다.
        x += value

    lock.release() # Mutex를 반납한다.

t1 = threading.Thread(target=foo, args=(1000000, 1))
t2 = threading.Thread(target=foo, args=(1000000, -1))
t1.start()
t2.start()
t1.join()
t2.join()

print("Shared value : {0}".format(x))
```

```
Shared value : 0
```

CPython GIL

❖ Global Interpreter Lock은 CPython에서 여러 thread를 이용해 Python 코드를 실행할 경우 단 하나의 스레드만 Python object에 접근하도록 제한하는 Mutex이다.

* Mutex : 다수의 thread가 하나의 자원을 사용할 때 충돌이 발생하지 않도록 제어하는 장치

❖ GIL이 필요한 이유는 CPython 구현체가 전역변수 등에 많이 의존하는 등 메모리 관리를 thread-safe하게 하지 않기 때문이다.

❖ GIL은 CPython 인터프리터 내부의 상태를 보호하는 것이지, 사용자가 작성한 코드에 있는 자료구조에 대한 접근을 보호하지 않는다.

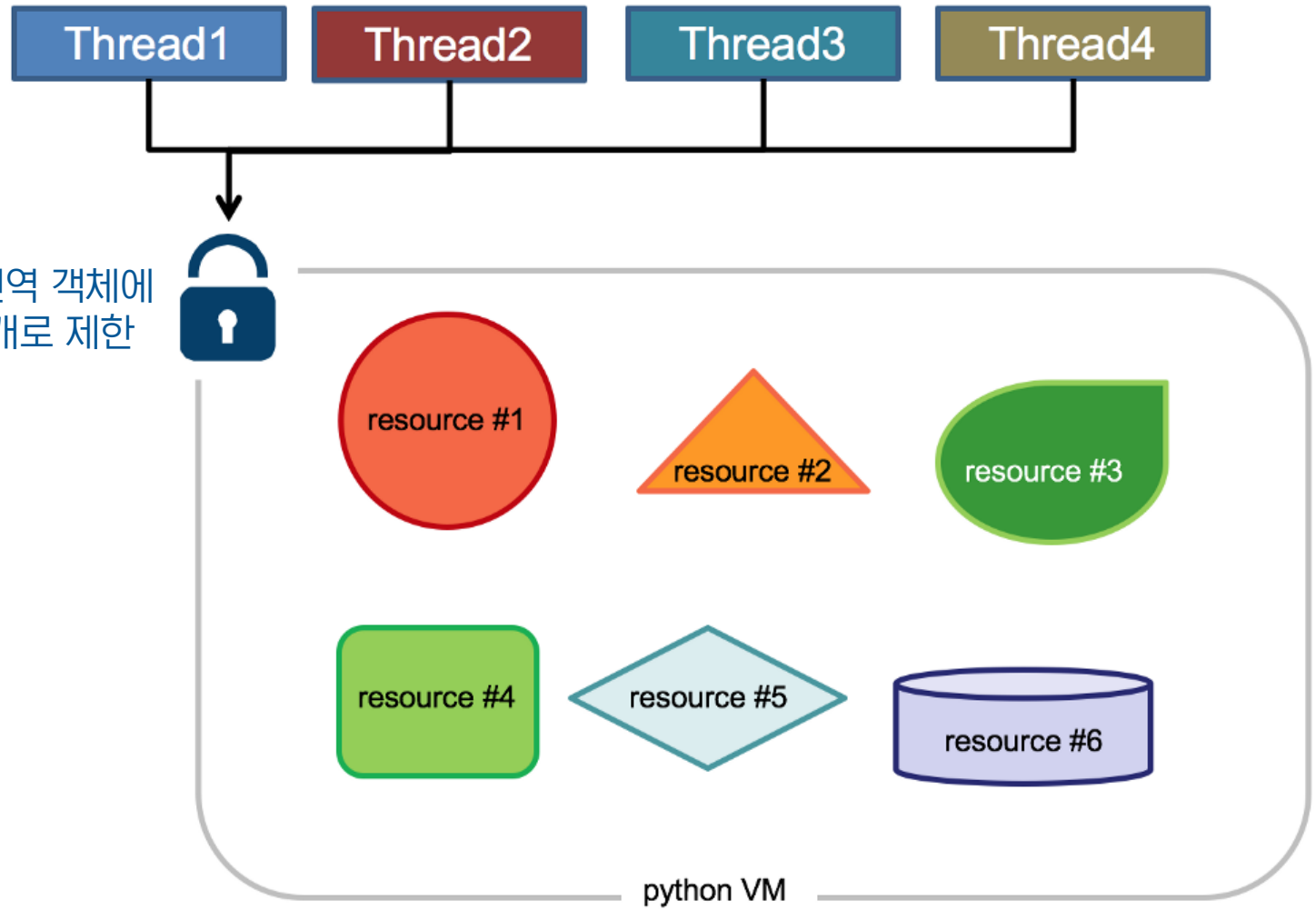
❖ 다양한 파이썬 구현체

- CPython : python 구현 표준으로, C를 이용해 구현한 python 인터프리터
- Jython : Java로 구현한 JVM위에서 실행되는 python 인터프리터
- PyPy : 정적 python으로 구현한 python 인터프리터
- IronPython : .NET 프레임워크에서 구현된 python 인터프리터

0. Thread

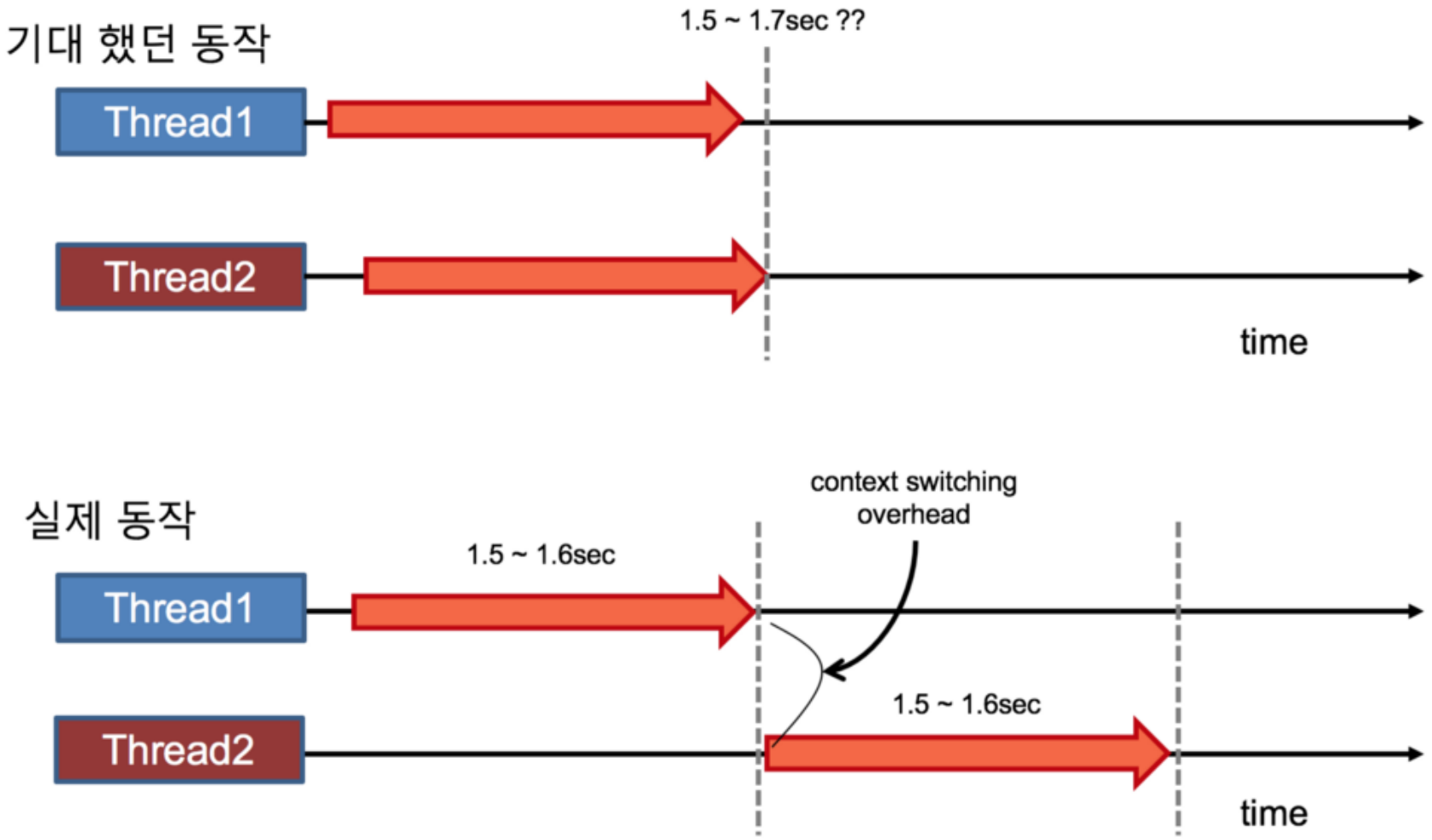
0.7 Cpython GIL

GIL이 인터프리터의 전역 객체에
접근하는 thread를 1개로 제한



0. Thread

0.7 Cpython GIL



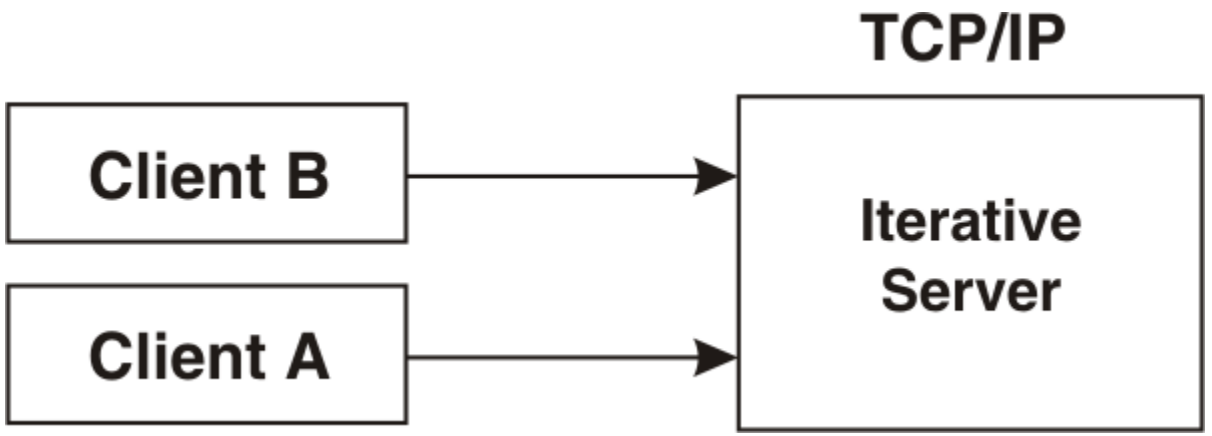
1. Server Model

1. Server Model

1.1 Iterative Server

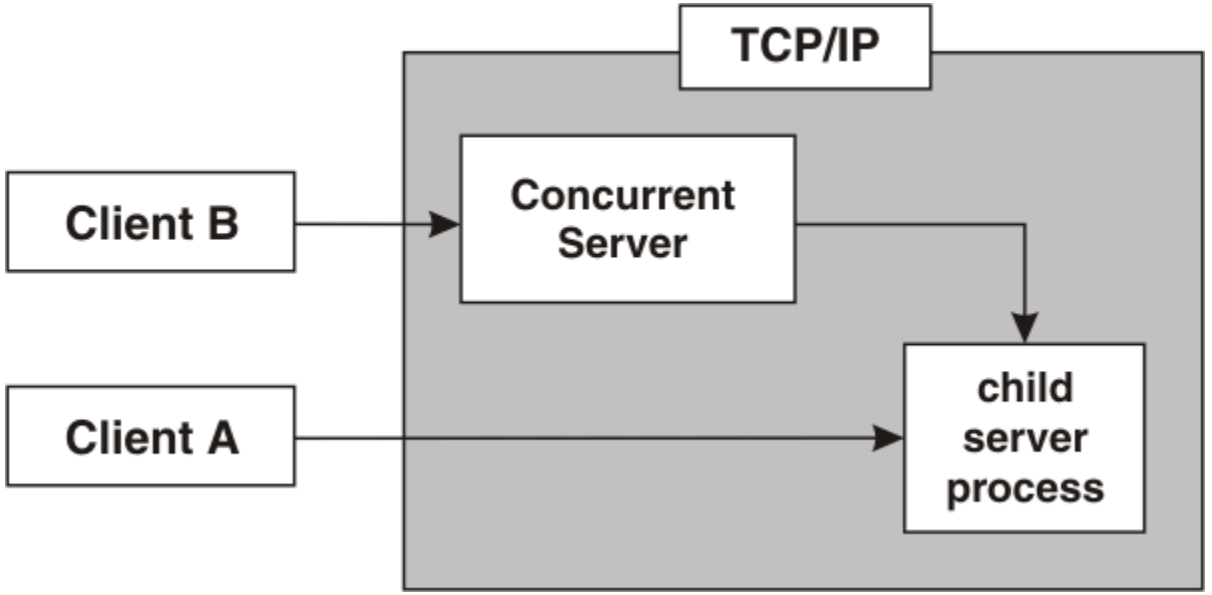
반복서버

- ❖ 접속한 클라이언트를 하나씩 차례대로 처리한다.
- ❖ 하나의 스레드로 모든 요청을 처리하기 때문에 시스템 자원 소모가 적다.
- ❖ 서버와 클라이언트의 통신이 길어지면 그 시간만큼 다른 클라이언트의 요청을 처리할 수 없다.
- ❖ 주로 이벤트 기반 비동기처리를 통해 구현한다. 대표적으로 node.js 서버가 있다.



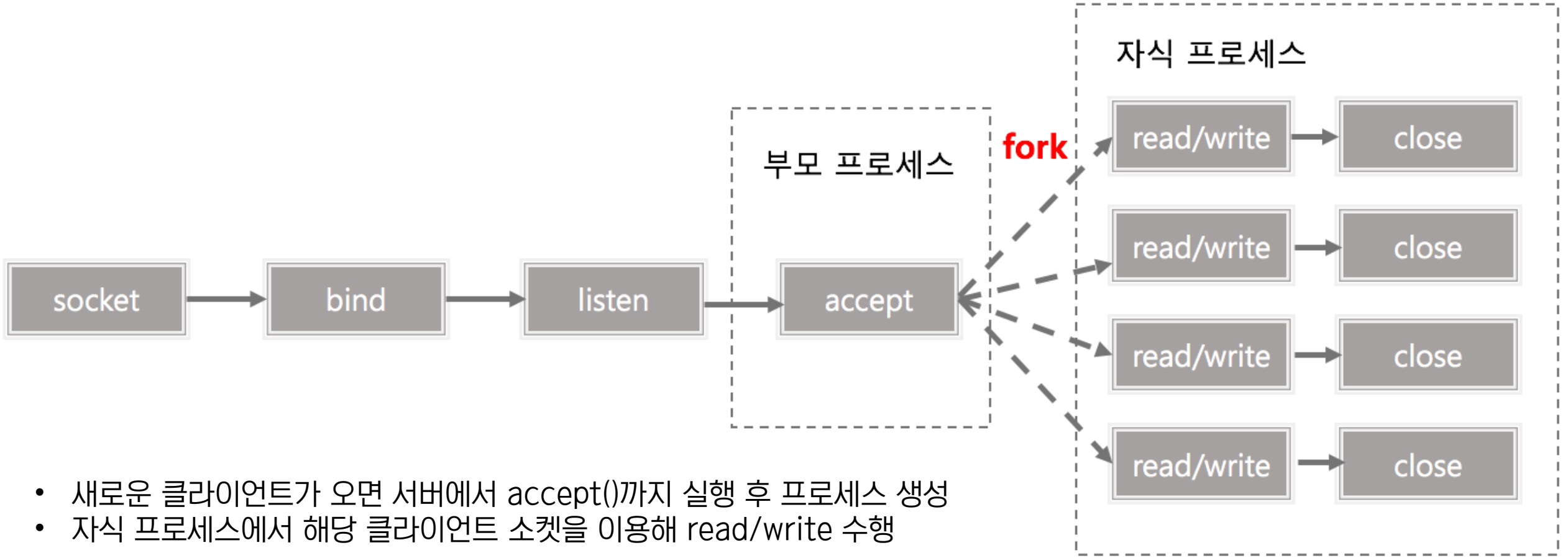
병행서버

- ❖ 다수의 스레드 혹은 프로세스를 이용해 많은 클라이언트의 요청을 병렬로 처리
- ❖ 서버와 클라이언트간 통신이 길어져도 다른 클라이언트 요청을 즉각 처리할 수 있다.
- ❖ 멀티 스레드 혹은 멀티 프로세스를 이용해 구현하기 때문에 시스템 자원 소모가 크다.
- ❖ 대표적인 예로 Apache 서버가 있으며 pre-fork 방식을 사용한다.



1. Server Model

1.2 Concurrent Server



- 새로운 클라이언트가 오면 서버에서 accept()까지 실행 후 프로세스 생성
- 자식 프로세스에서 해당 클라이언트 소켓을 이용해 read/write 수행

1. Server Model

1.2 Concurrent Server

- 새로운 클라이언트가 오면 서버에서 accept()까지 실행 후 스레드 생성
- 새로운 스레드에서 해당 클라이언트 소켓을 이용해 read/write 수행

이상적인 서버모델

1. Socket 함수 호출 시 Blocking 최소화
2. I/O를 다른 작업과 병행
3. Thread 및 프로세스의 개수를 최소화
4. 시스템 자원(CPU, memory) 사용량 최소화

1. Server Model

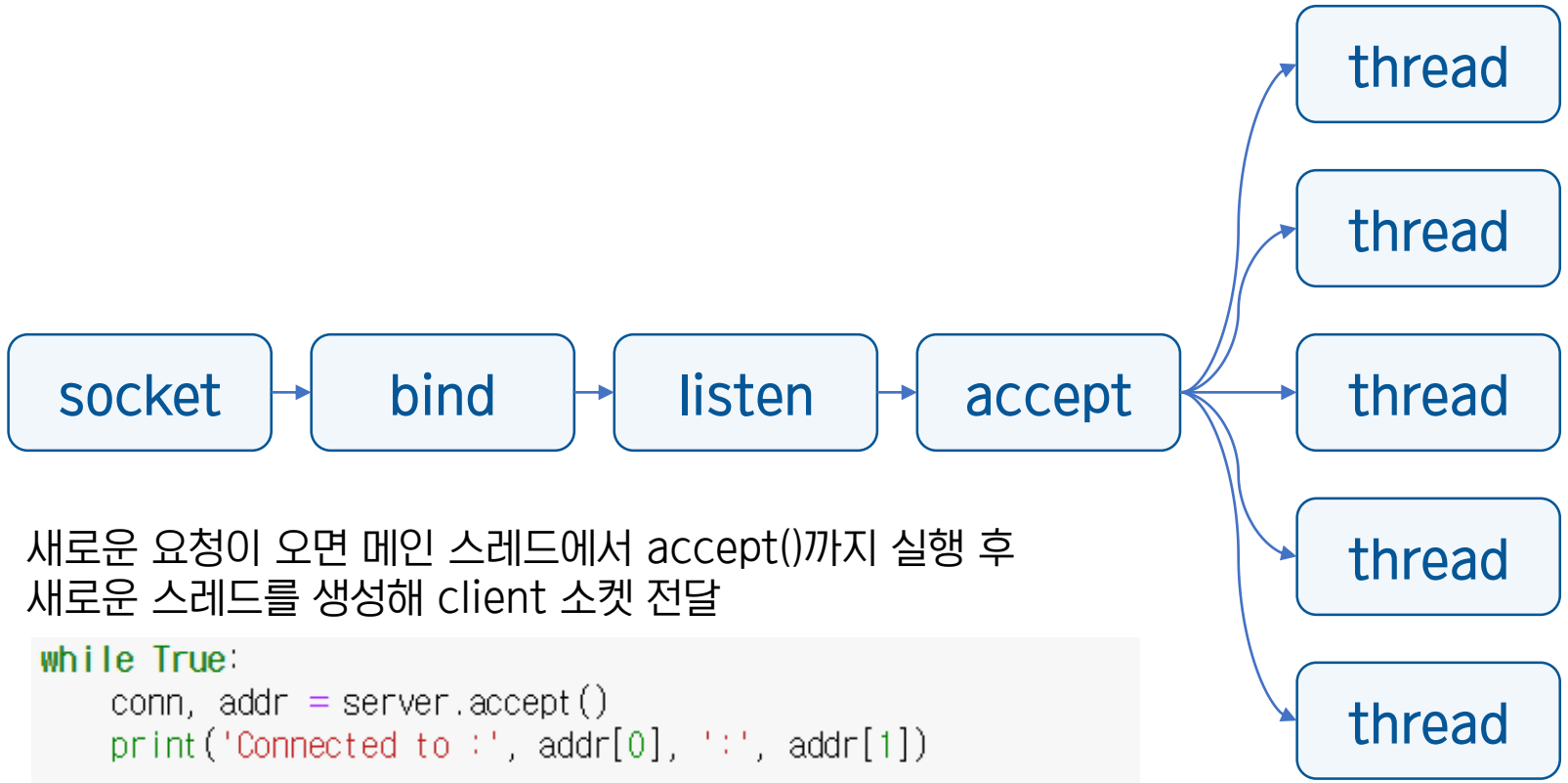
F.Y.I

- -Select model : 반복서버 + Non-Blocking + sync I/O
- -Overapped I/O model : 병행서버 + Non-Blocking + Async I/O
- -IOCP(I/O Completion Port) model : 병행서버 + Non-Blocking + Async I/O, 스레드 개수 최소화

2. Concurrent Server

2. Concurrent Server

2.1 Concurrent Server



새로운 요청이 오면 메인 스레드에서 accept()까지 실행 후
새로운 스레드를 생성해 client 소켓 전달

```
while True:
    conn, addr = server.accept()
    print('Connected to :', addr[0], ':', addr[1])

    t = threading.Thread(target=socket_handler, args=(conn,))
    t.start()
```

각 스레드에서는 전달받은 클라이언트 소켓을 이용해 recv/send를 수행

```
def socket_handler(conn):
    data = conn.recv(1024)
    data = data.decode()[::-1]
    conn.sendall(data.encode())
    conn.close()
```

2. Concurrent Server

2.1 Concurrent Server

```
def socket_handler(conn):
    data = conn.recv(1024)
    data = data.decode()[::-1]
    conn.sendall(data.encode())
    conn.close()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Thread server -p port")
    parser.add_argument('-p', help = "port_number", required = True)

    args = parser.parse_args()

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('', int(args.p)))
    server.listen(5)

    while True:
        conn, addr = server.accept()
        print('Connected to :', addr[0], ':', addr[1])

        t = threading.Thread(target=socket_handler, args=(conn,)) # 메인 스레드에서 accept까지 처리 후 새로운 스레드 생성
        t.start()

    server.close()
```

2. Concurrent Server

2.1 Concurrent Server

서버에서 단순히 클라이언트가 보낸 문자열을 뒤집어 전송하는게 아니라,
지속적으로 서버와 클라이언트가 메시지를 주고 받으려면?

새로운 요청처리 스레드

사용자 입력 스레드

클라이언트 입력 수신 스레드

Assignment #6

- threading 모듈을 사용해 서버와 클라이언트가 대화를 주고 받을 수 있는 프로그램 작성
  - 서버는 클라이언트가 전송한 문자열 출력, input()으로 사용자 입력을 받아서 클라이언트에 전달
  - 클라이언트는 서버가 전송한 문자열 출력, input()으로 사용자 입력을 받아서 서버에 전달
  - 난이도 조절을 위해 서버는 1개의 클라이언트만 처리
  - python thread_talk_server.py –p 8888
  - python thread_talk_client.py –p 8888 –i 127.0.0.1
- 팀 대표가 barcel@naver.com으로 제출 (4.15일까지)
  - Title : [컴퓨터네트워크][학번][이름][과제_N]
  - Content : github repo url

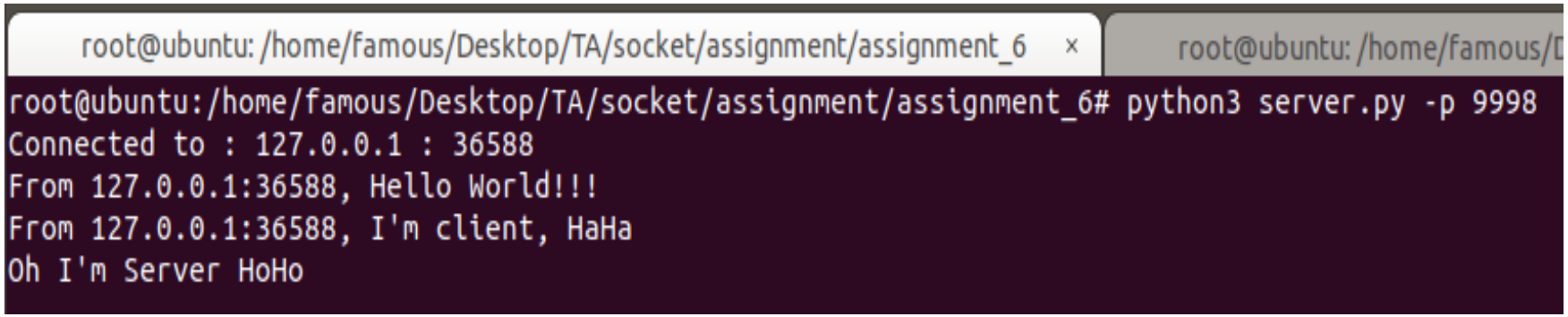
    팀명 : 길동이네

    팀원 : 홍길동(학번), 고길동(학번)

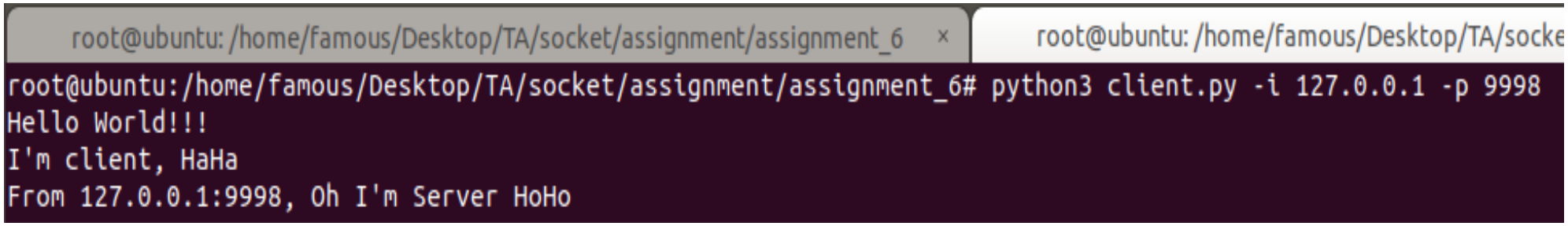
2. Concurrent Server

2.2 Assignment 6

Server



Client



3. I/O Processing Type

3. I/O Processing Type

3.1 Overview

Blocking / Non-Blocking : 호출되는 함수가 바로 return하는지에 따라 구분.

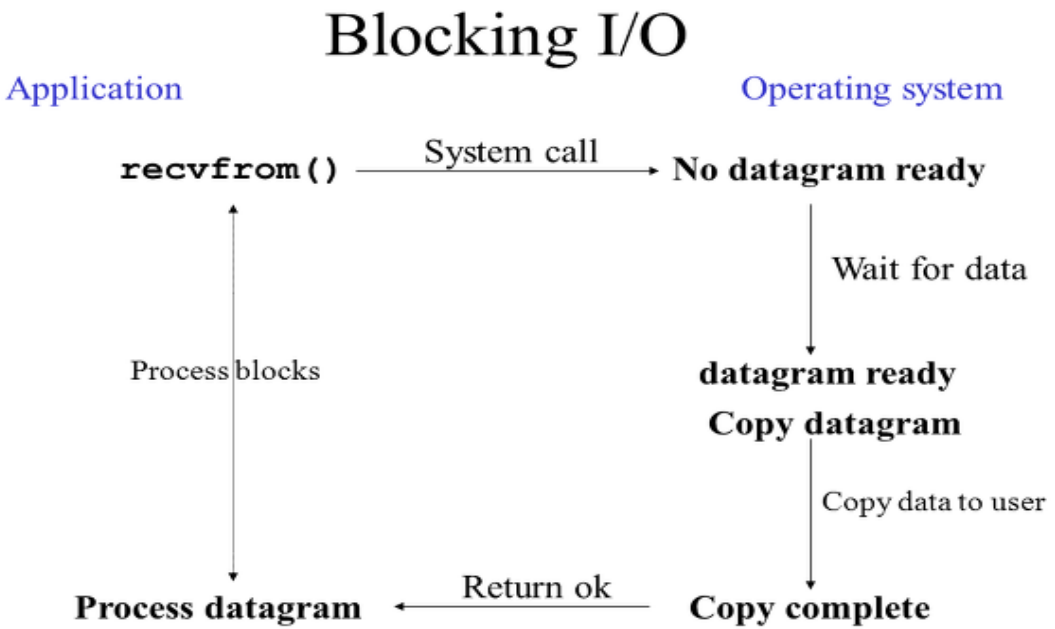
Sync / Async : 호출되는 함수의 작업 완료를 누가 처리하는지에 따라 구분.

3. I/O Processing Type

3.2 Blocking I/O

Blocking Socket

- ❖ Socket API 호출 시 조건이 만족되지 않으면 API를 호출한 스레드가 Block 된다.(run -> wait)
- ❖ Socket API는 기본적으로 Blocking mode이다.
- ❖ Blocking된 Socket API가 리턴될 때까지 다른 작업을 할 수 없다. -> Multi thread를 이용
- ❖ 1대1 통신이거나, 한가지 작업만 할 때 사용

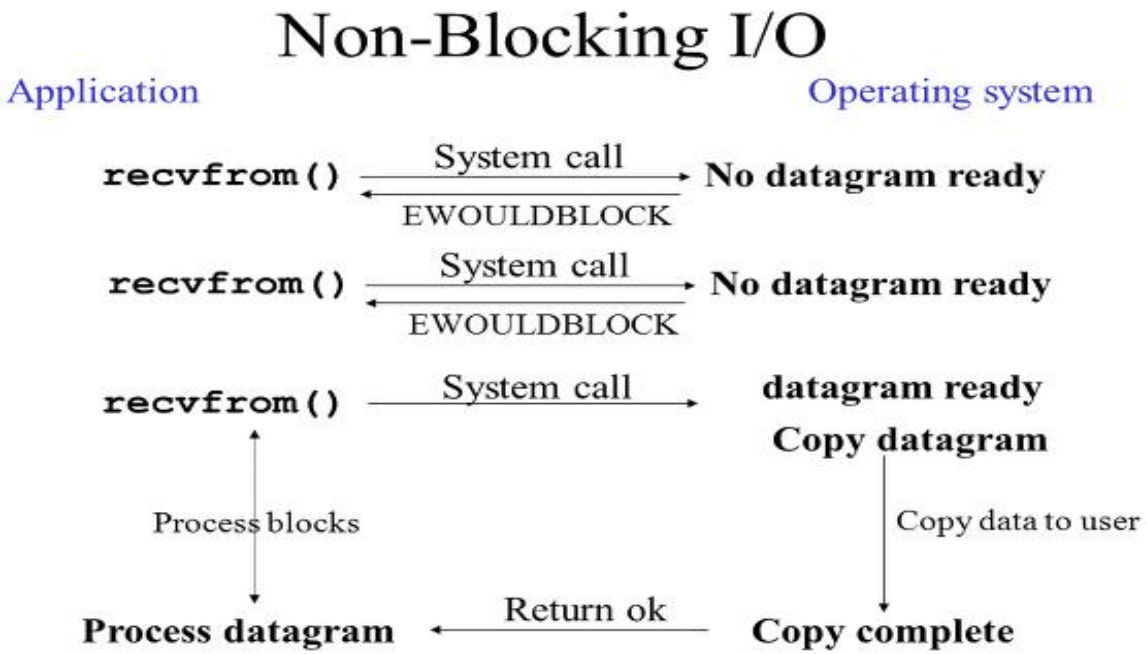


3. I/O Processing Type

3.3 Non-Blocking I/O

Non-Blocking Socket

- ❖ Socket API 호출 후 조건에 관계없이 바로 return하는 모드.
- ❖ 일반적으로 어떤 시스템 콜이 완료되었는지 보기 위해 루프를 돌며 확인하는 Polling을 사용해야 한다.
- ❖ 통신 대상이 여럿이거나, 다른 작업을 병행해야 하는 경우 Non-Blocking 또는 Async 모드를 사용해야 한다.



Synchronous I/O

❖ 동기식 I/O는 요청과 요청에 대한 결과를 같은 곳에서 처리한다는 뜻이다.
❖ 함수를 호출할 때 호출한 함수의 처리 결과를 호출한 쪽에서 처리하면 동기식이다.
❖ 아래의 input() 메소드는 호출한 쪽에서 input()이 반환한 문자열을 처리함으로 동기식 I/O이다.

```
msg = input("Syncronous I/O >>> ")
print(msg)
```

```
Syncronous I/O >>> Hello World!!!
Hello World!!!
```

❖ input()의 처리결과는 msg라는 변수에 담기고, 이를 자신이 직접 처리했다.

Asynchronous I/O

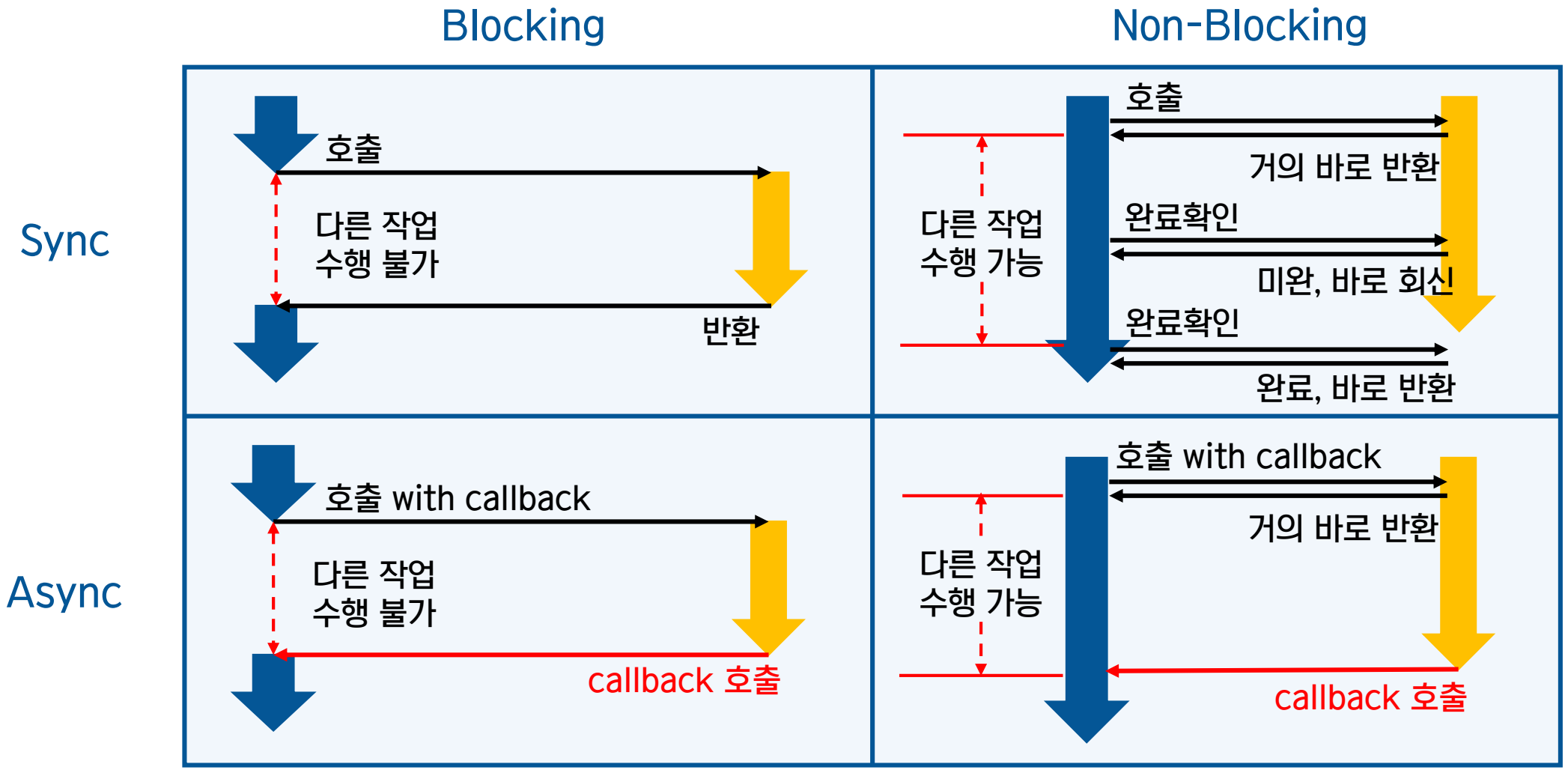
- ❖ 비동기식 I/O는 요청과 이에 대한 결과를 같은 곳에서 처리한다는 뜻이다.
- ❖ 함수를 호출하면 함수의 처리결과를 호출한 쪽에서 처리하지 않으면 비동기이다.
- ❖ 일반적으로 비동기 호출 시 요청에 대한 결과를 처리하기 위한 함수(callback)를 같이 전달한다.

```
> setTimeout( foo, 3000);

  function foo(){
      console.log("2");
  }
  console.log("1");
  1                                                    VM76:6
< undefined
  2                                                    VM76:4
```

- ❖ setTimeout()에 시간과 callback함수를 같이 전달했다. 호출 후 3초가 지나면 호출당한 쪽(VM76:4)에서 callback 함수를 실행해 수행결과를 처리한다.

3. I/O Processing Type

3.6 Put it in together



3. I/O Processing Type

3.6 Put it in together

	Blocking	Non-Blocking
Sync	Read/Send	Read/Send O_NONBLOCK
Async	I/O Multiplexing 구현체에 따라 상이	AIO

Q & A