

Asynchronous – Network I/O

☞ Network 통신의 상황에서 생각해보자.

Socket은 **File로 추상화** 되어있다.  File의 경우와 똑같다.

- Read할 때 나의 CPU에서 할 일이 없다.
- 다른 컴퓨터의 CPU에서 알아서 처리하고 데이터를 보내준다.

Blocking I/O

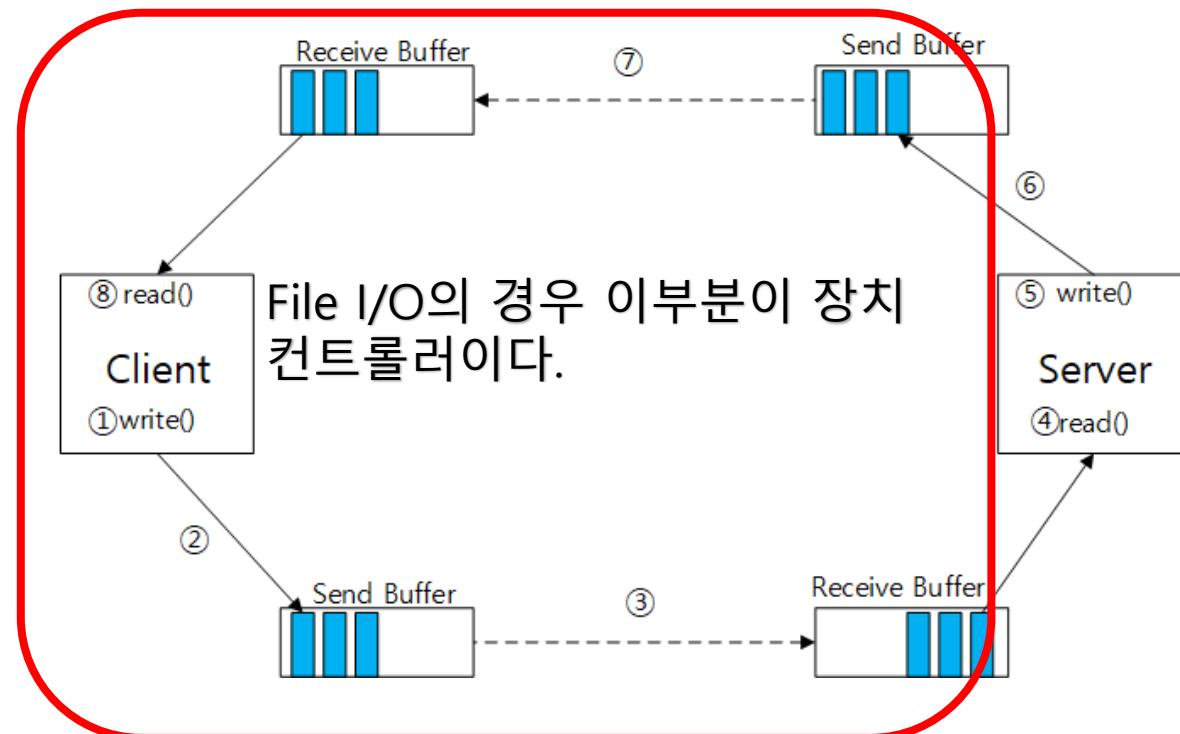
- Socket의 buffer에 Read할 Data가 비어 있다면 → Blocked (or Sleep)
- 생각해보면 File Read의 경우와 똑같다.

Non - Blocking I/O

- Read이 Blocked되지 않는다.
- Read할 Data가 있는지 Event를 확인하고 Read하면 된다.

핵심

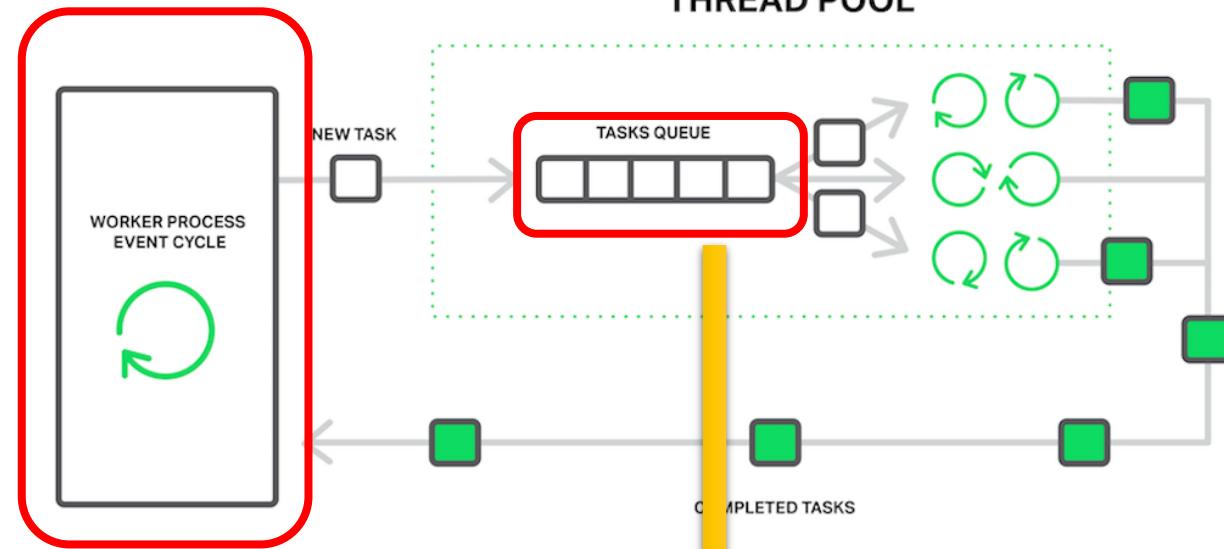
즉, Network I/O 또한 Async와 Event 또는 Callback 함수로 실행할 수 있다.



Thread Per Request Model

간단한 동작 원리 ※ 내 생각이다.

1. 서버 소켓에서 연결 수락 후
2. 클라이언트 소켓을 Thread Pool의 Thread에 전달



```
// 서버 소켓 생성 및 리스닝  
ServerSocket serverSocket = new ServerSocket(port);  
System.out.println("서버가 " + port + " 포트에서 실행 중입니다.");  
  
while (true) {  
    // 클라이언트 연결 요청을 수락하고 Acceptor Thread를 사용  
    Socket clientSocket = serverSocket.accept();  
    System.out.println("클라이언트 연결 수락됨.");  
  
    // 클라이언트와의 통신을 처리하기 위해 Worker Thread를 풀에서 가져와 실행  
    executorService.execute(new WorkerThread(clientSocket));
```

참고

Thread Pool 내부에 구현 되어있는 것 같다.. (정확하지는 않다.)

Thread Per Request Model

문제점

만약 Thread가 **Blocking I/O를 호출**한다면? → Thread는 Sleep 상태가 된다.

다음과 같은 상황을 생각해보자.

1. Thread Pool에 Idle Thread가 존재하지 않고
2. 요청이 들어와 대기를 하고 있다면

이러한 Thread의 Sleep는 늦은 응답의 원인이 된다.

INSITE

만약 I/O가 많은 요청이 아닌 CPU 연산이 많은 요청이면

- Webflux나 Thread Per Request나 똑같이 응답이 느리다.

INSITE

만약 Request가 Thread Pool Size 보다 작다면

- Webflux나 Thread Per Request나 똑같이 응답 속도가 빠르다.

장점

- 내가 작성한 코드가 어느 Thread에서 동작하는지 신경 쓸 필요가 하나도 없다.
→ 따라서 코드 작성 및 디버깅이 쉽다.
- 서버 구성 시 Reactive Model보다 Instance의 수를 많이 사용하게 된다.
→ 장애 발생시 타격이 크지 않다.