Computer Science Study

# Operating System

Subject 1. Basic

Subject 2. Synchronous

Subject 3. Asynchronous

Subject 4. I/O Multiplexing

made by Seungtaek

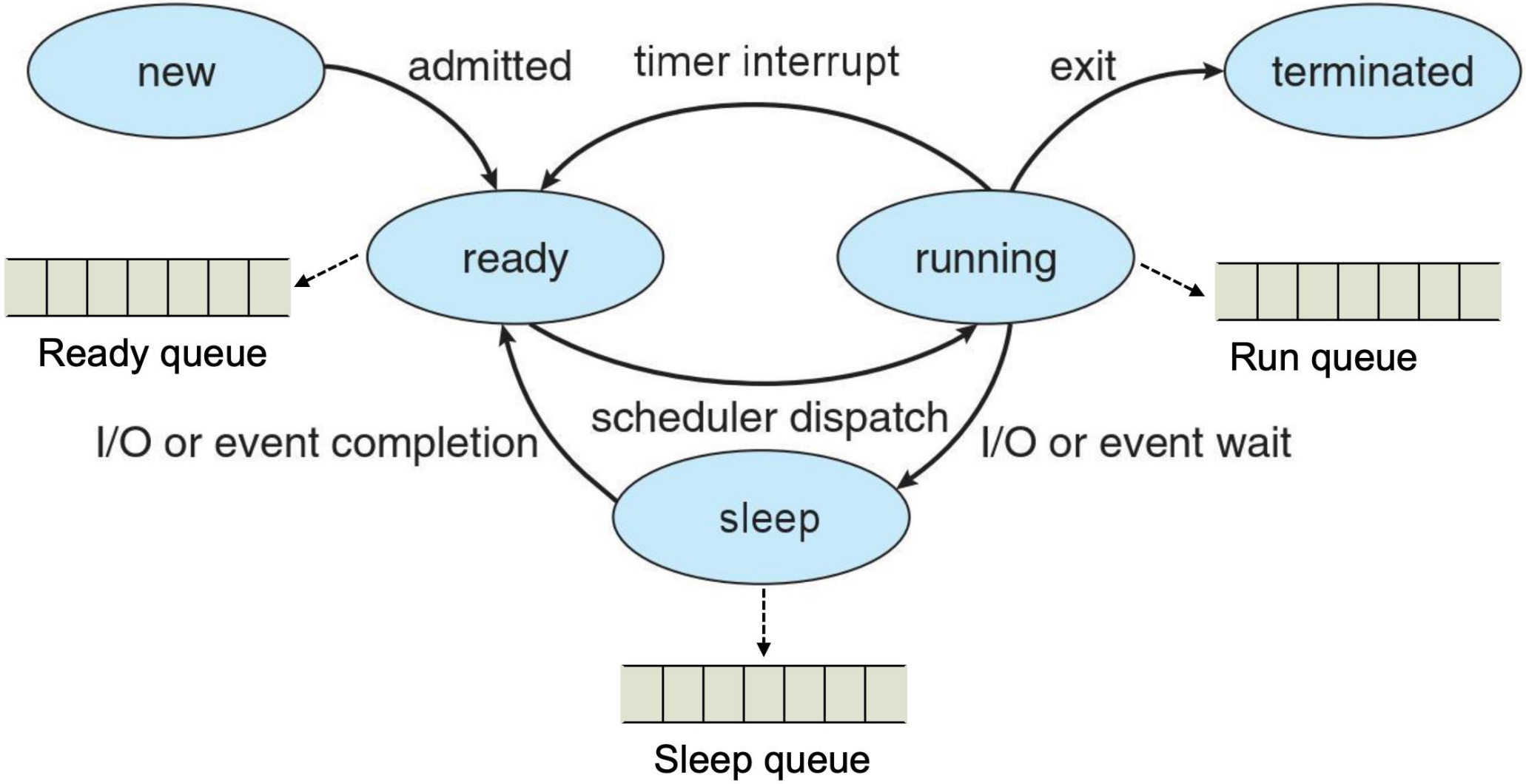
홍승택

2024. 11. 18. 19시

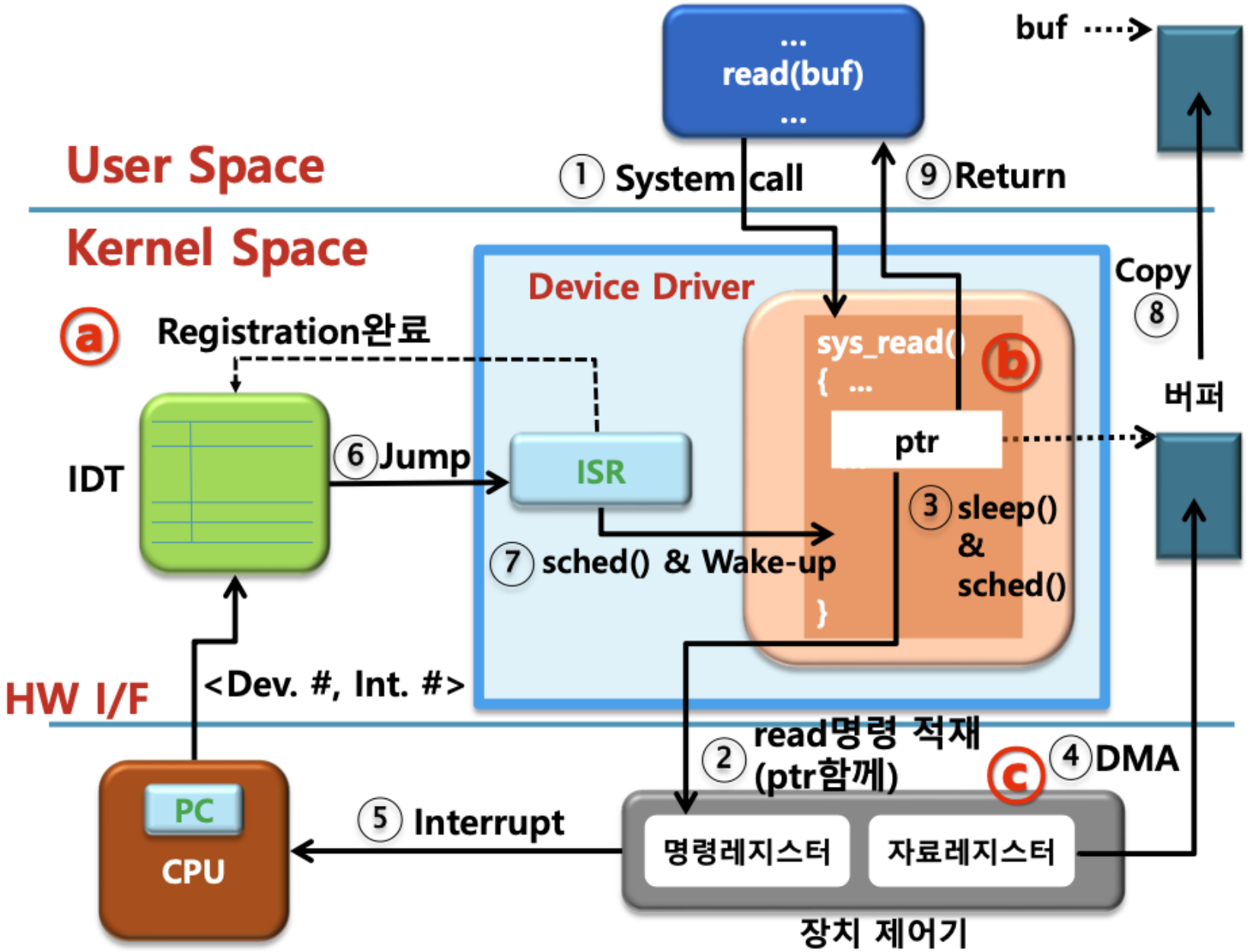
# subject 1.

## Basic

# Process State



# I/O processing



# Blocking, Non-Blocking

## Blocking

- 요청한 작업 결과를 기다린다.
- 제어권을 넘겨준다.
- 스레드나 프로세스가 일시 중지된다. (I/O)

## Non-Blocking

- 요청한 작업 결과를 기다리지 않는다.
- (거의 바로) 제어권을 돌려받는다.

# Synchronous, Asynchronous

## Syn + Chrono

함께 시간

**Case 1**

synchronous I/O = block I/O

asynchronous I/O = non-block I/O

**Case 2**

synchronous I/O : 요청자가 I/O 완료까지 챙겨야 할 때

asynchronous I/O : 완료를 noti 주거나 callback으로 처리

**Case 3**

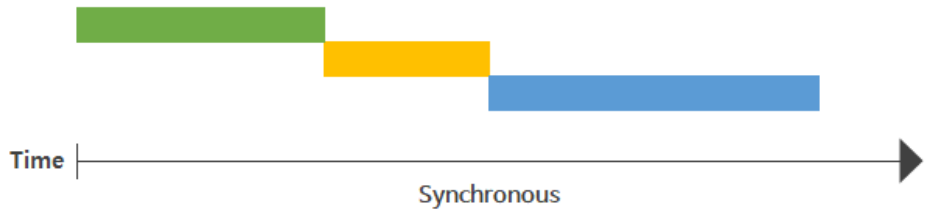
작업의 순서 보장

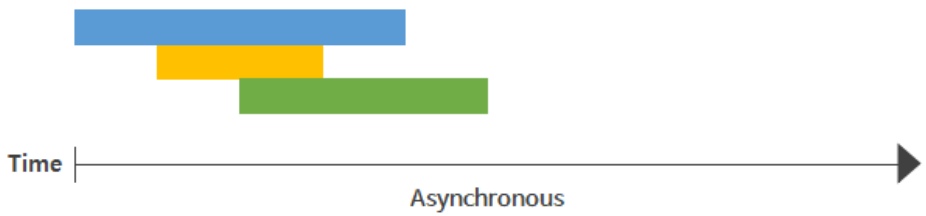
작업의 순서를 보장하지 않음

**Case 4**

asynchronous I/O : block I/O를 다른 thread에서 실행

**Case 3**

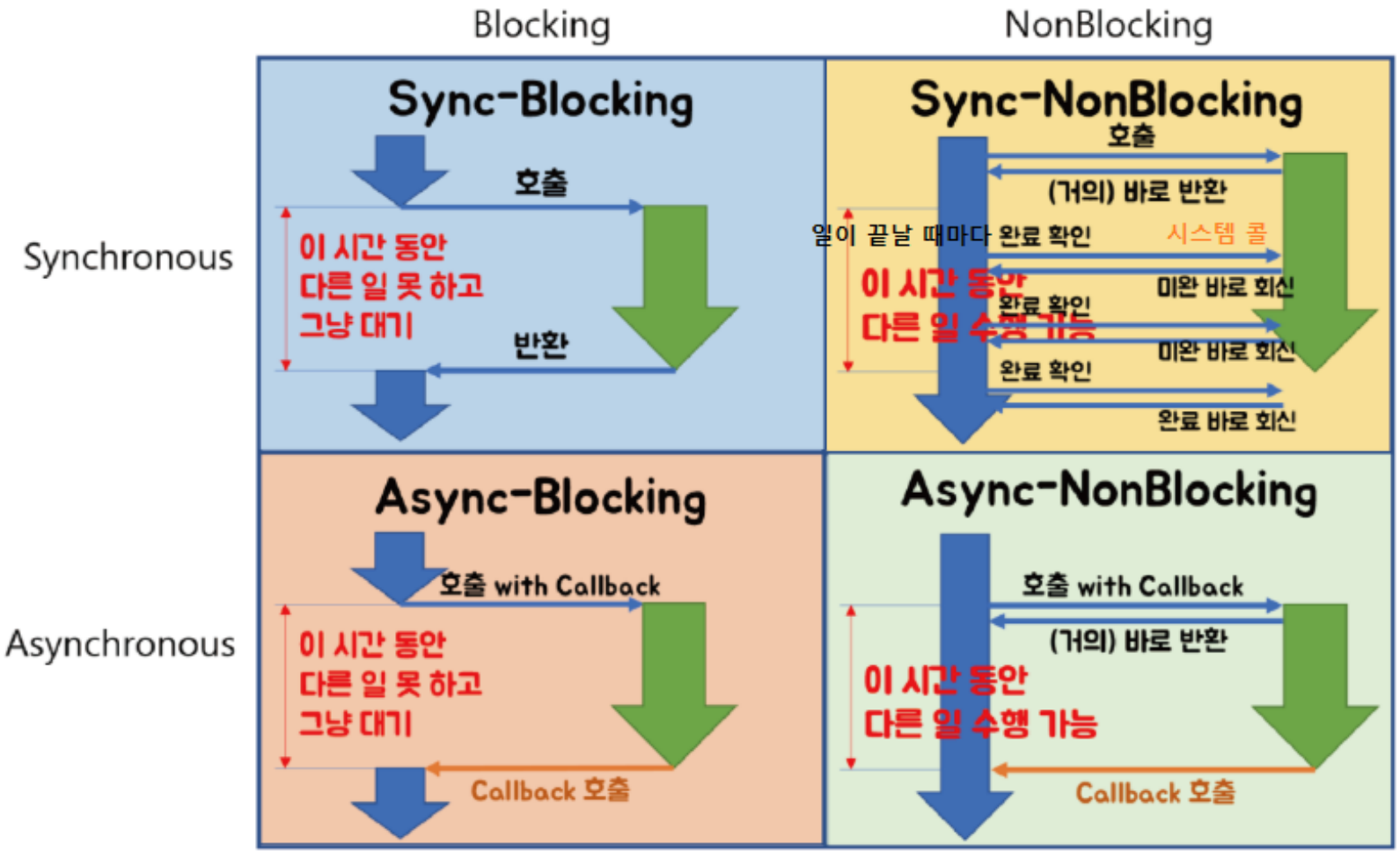



https://www.youtube.com/watch?v=EJNBLD3X2yg

# Synchronous, Asynchronous

## Case2




누가 만들었는지 모르겠음

# Synchronous, Asynchronous

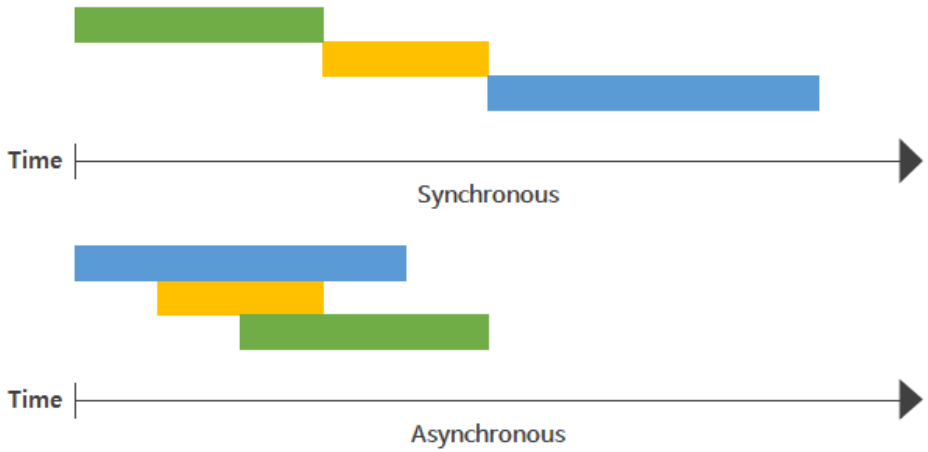
## Case 2

Synchronous I/O : 요청자가 I/O 완료까지 챙겨야 할 때

Asynchronous I/O : 완료를 noti 주거나 callback으로 처리

| | | Case 3 | |
|---|---|---|---|
| | | 순서O | 순서X |
| Case 2 | polling | while(!job_1());<br>while(!job_2()); | while(! (job1 && job2)) {<br>if(!job1) job1=job_1();<br>if(!job2) job2=job_2();<br>} |
| | callback | 콜백 함수 안에서<br>다음 작업 시작 | 마구잡이 시작 |

## Case 3

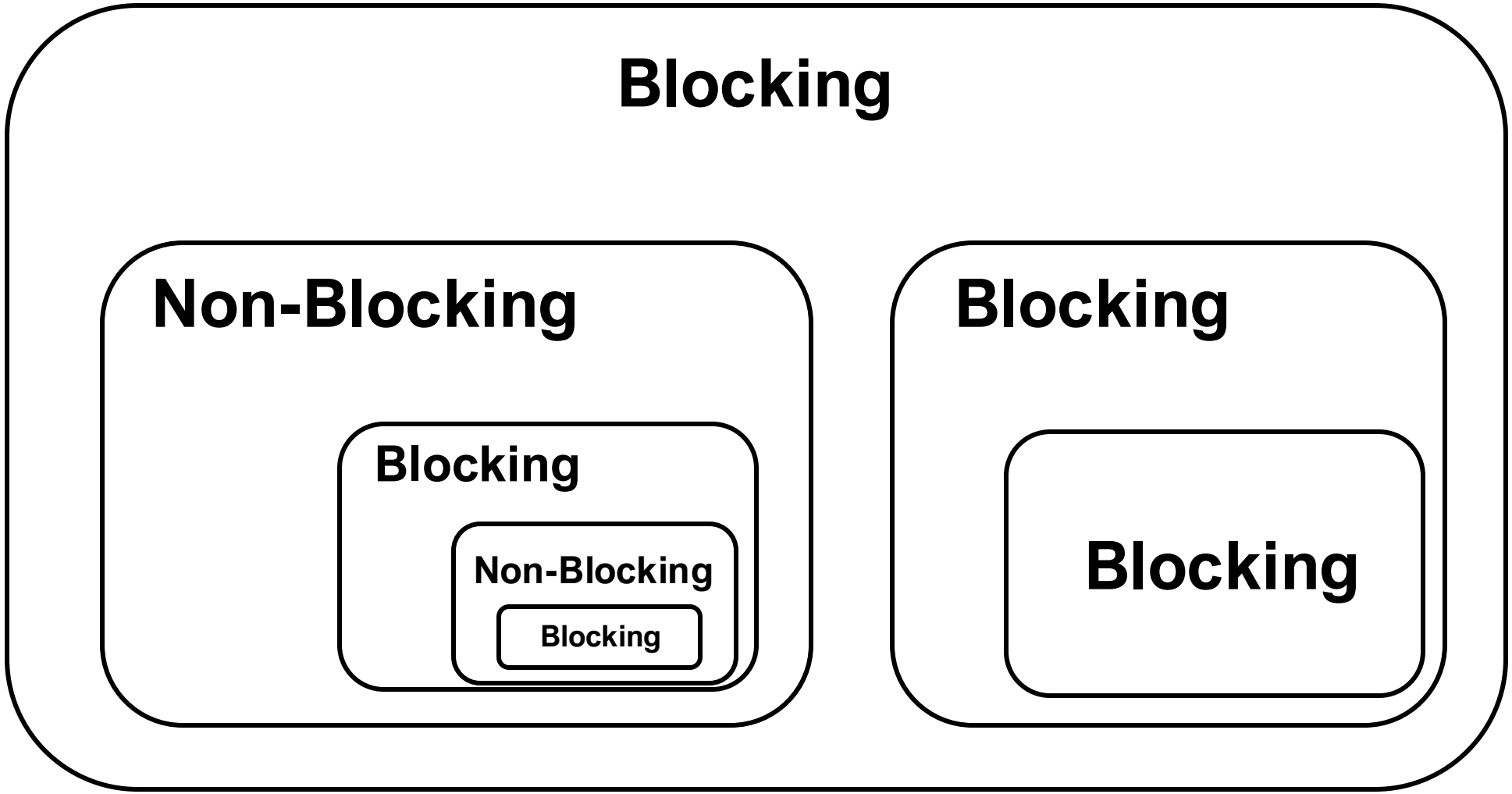



Case2 입장에선 sync여도 Case3은 async를 나타낼 수 있고,
Case2 입장에선 async여도 Case3은 sync를 나타낼 수 있다.

→

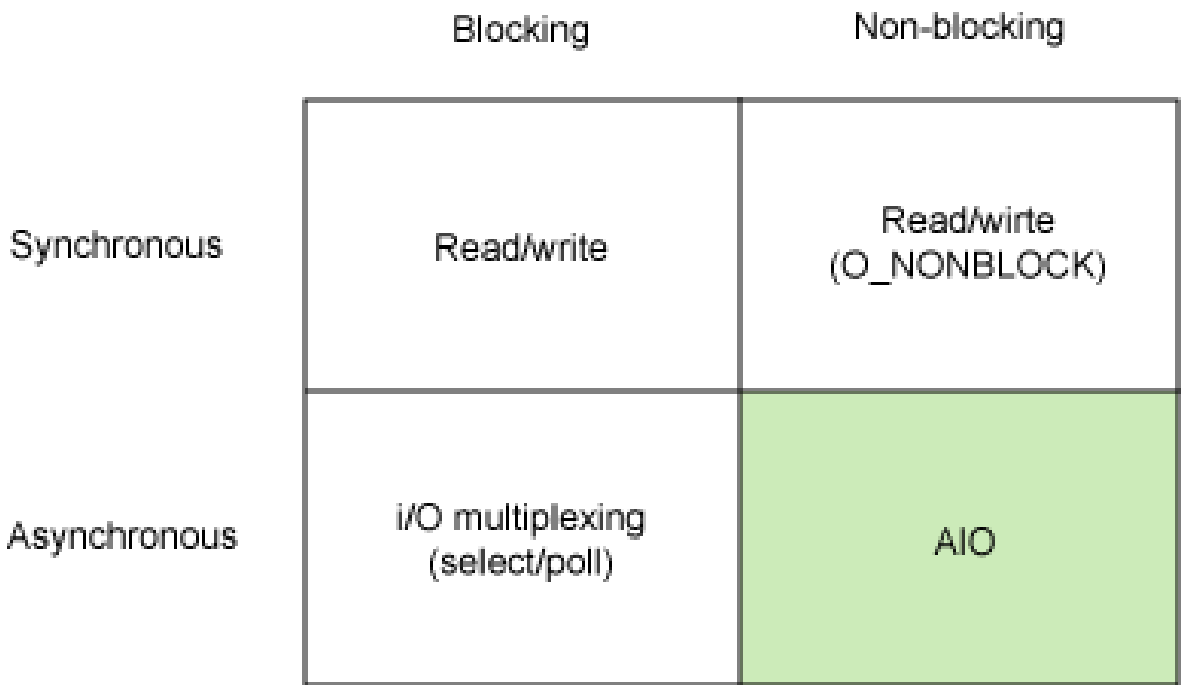
결론: Sync, Async는 문맥에 따라 다양하게 해석될 수 있다.

# Viewpoint


Blocking
Non-Blocking
Blocking
Non-Blocking
Blocking
Blocking
Blocking

# I/O model

Boost application performance using asynchronous I/O (28/August/2006)

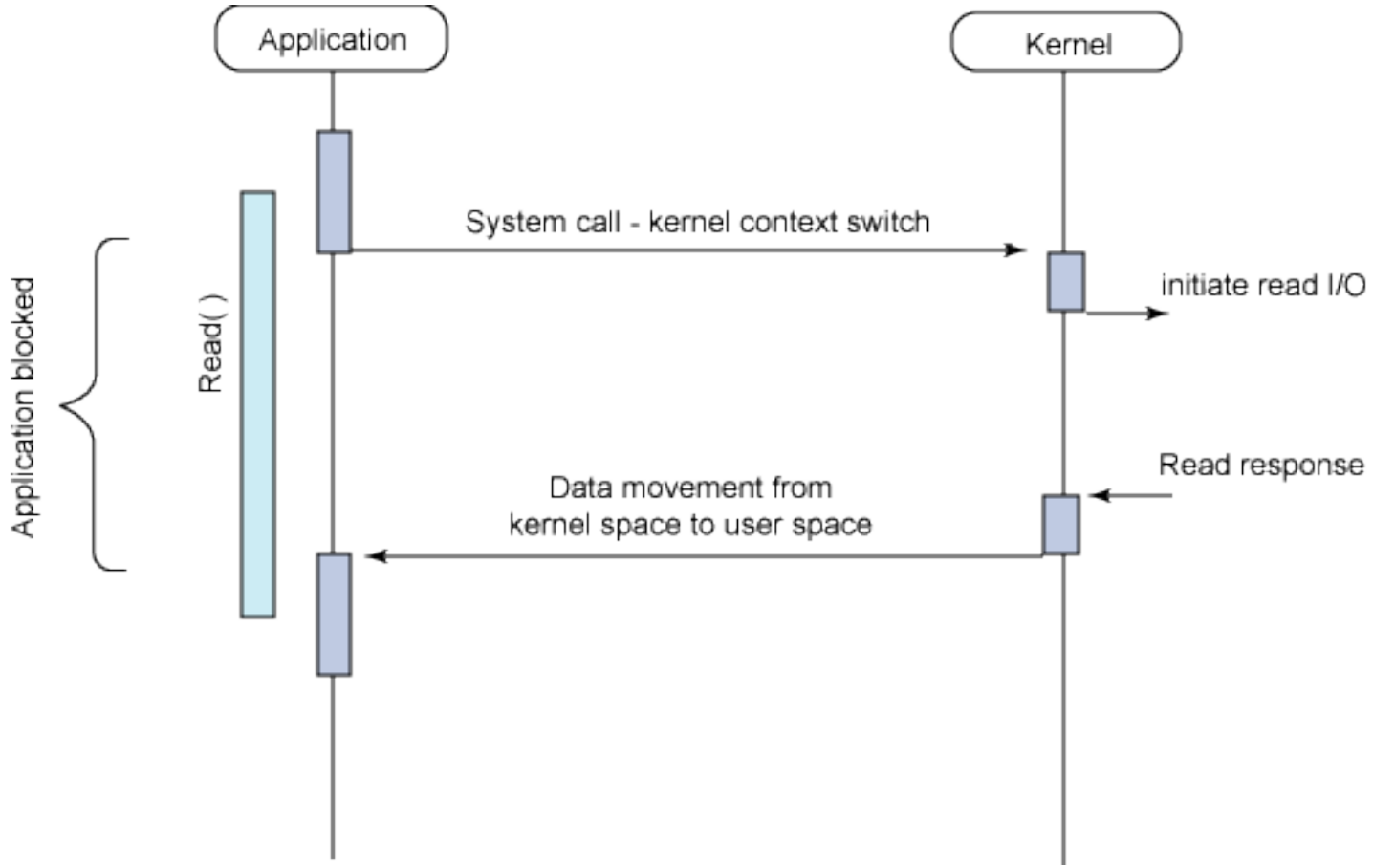



https://developer.ibm.com/articles/l-async/

# subject 2.

## Synchronous

# Synchronous Blocking I/O

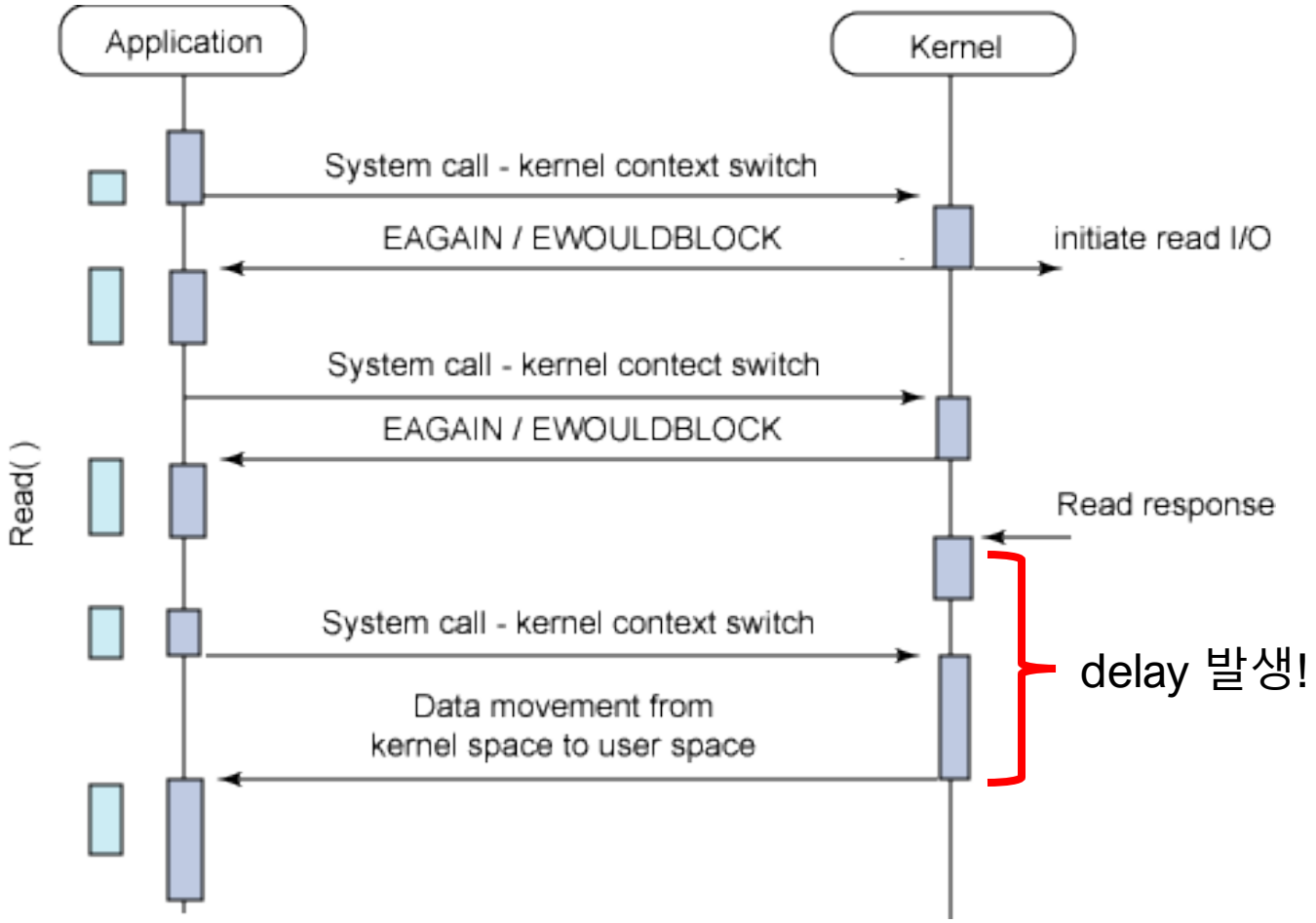



https://developer.ibm.com/articles/l-async/

# Synchronous Non-Blocking I/O

결과값 확인: busy-wait, polling

**Synchronous**
이 작업이 수행돼야 뭔가를 하는데..
관련이 없는 작업은 해도 괜찮지 않을까?




https://developer.ibm.com/articles/l-async/

# Synchronous Non-Blocking I/O

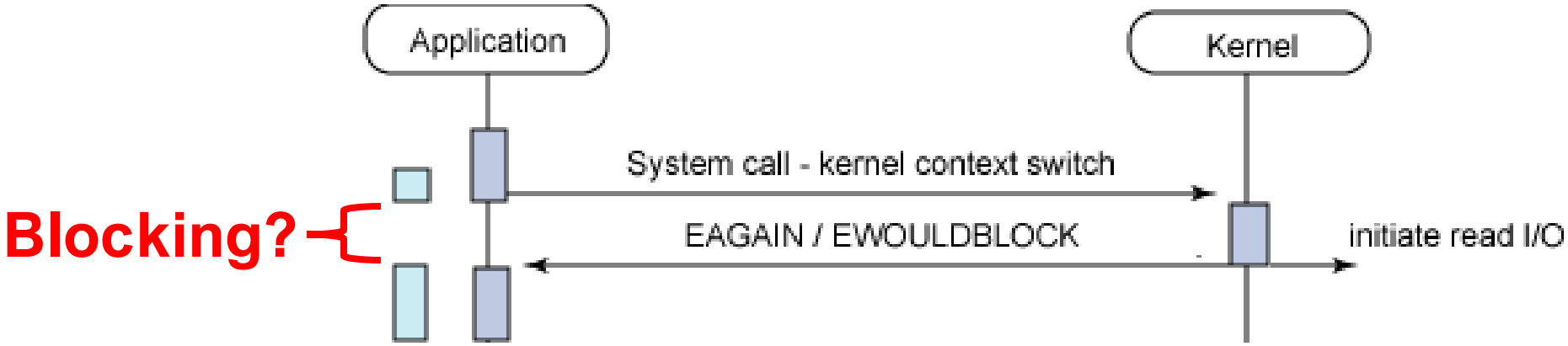
**Non-Blocking**

'즉시 반환'의 기준은 무엇인가?

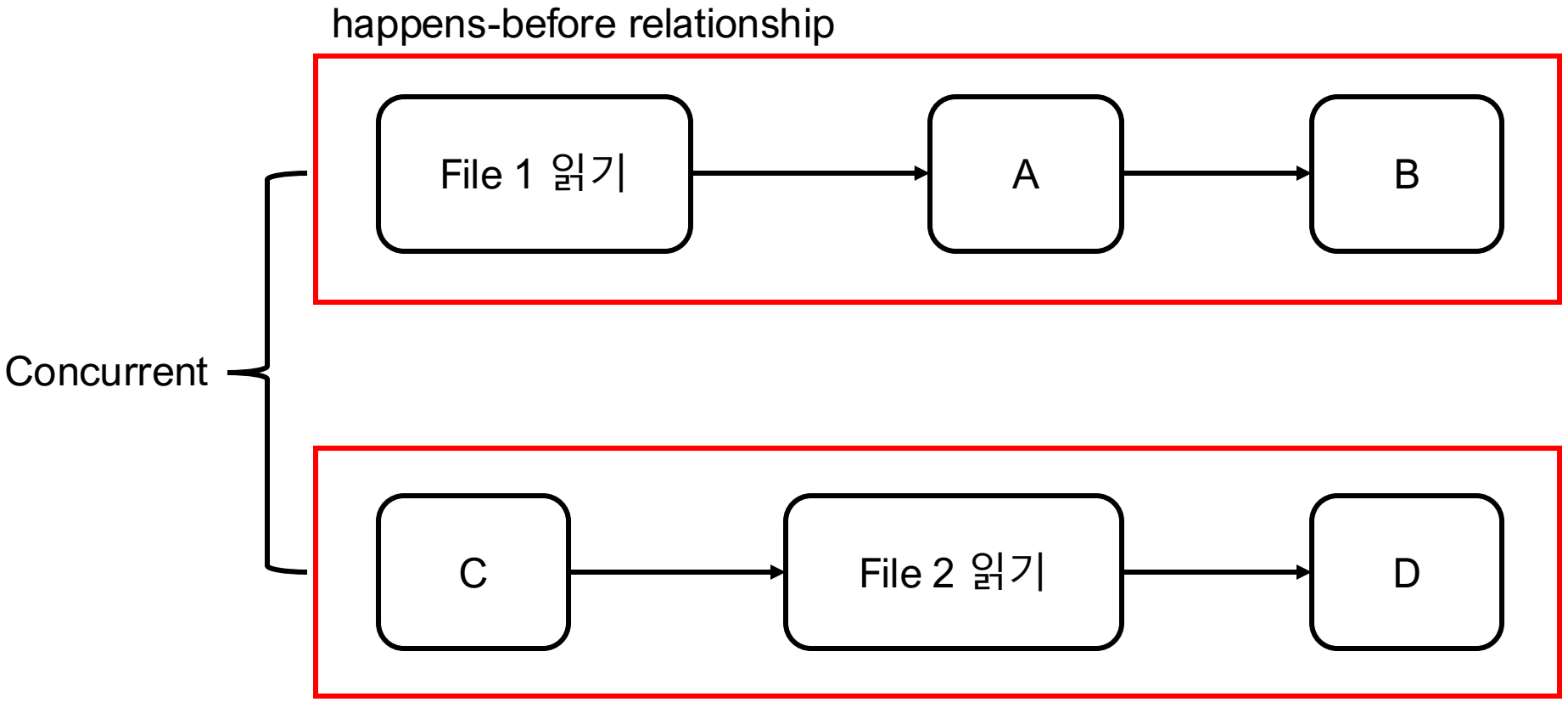
read() : fd 파일을 **읽어줘**

blocking : **읽으면** 반환해줄게

non-blocking : 바로 반환해줄게



# Synchronous Non-Blocking I/O

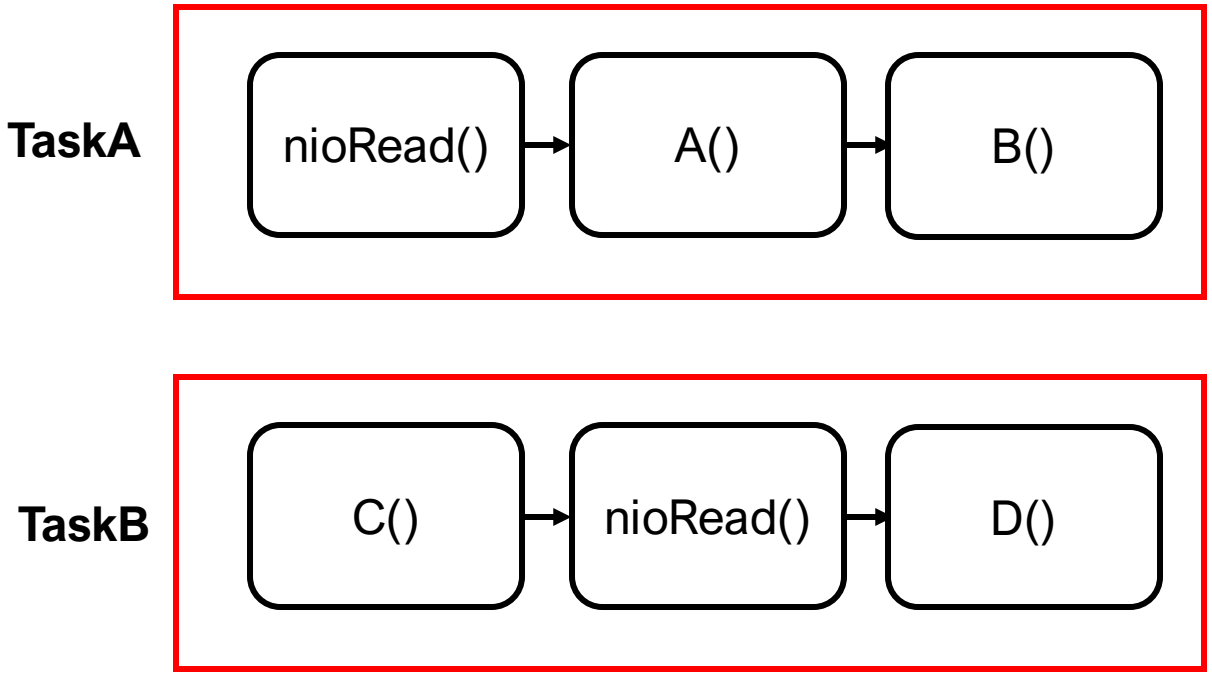


# Synchronous Non-Blocking I/O

## Practice

https://github.com/CS-Computer-Science-Study/Operating-System/tree/main/practice/Synchronous_NonBlocking

happens-before relationship




**Example**

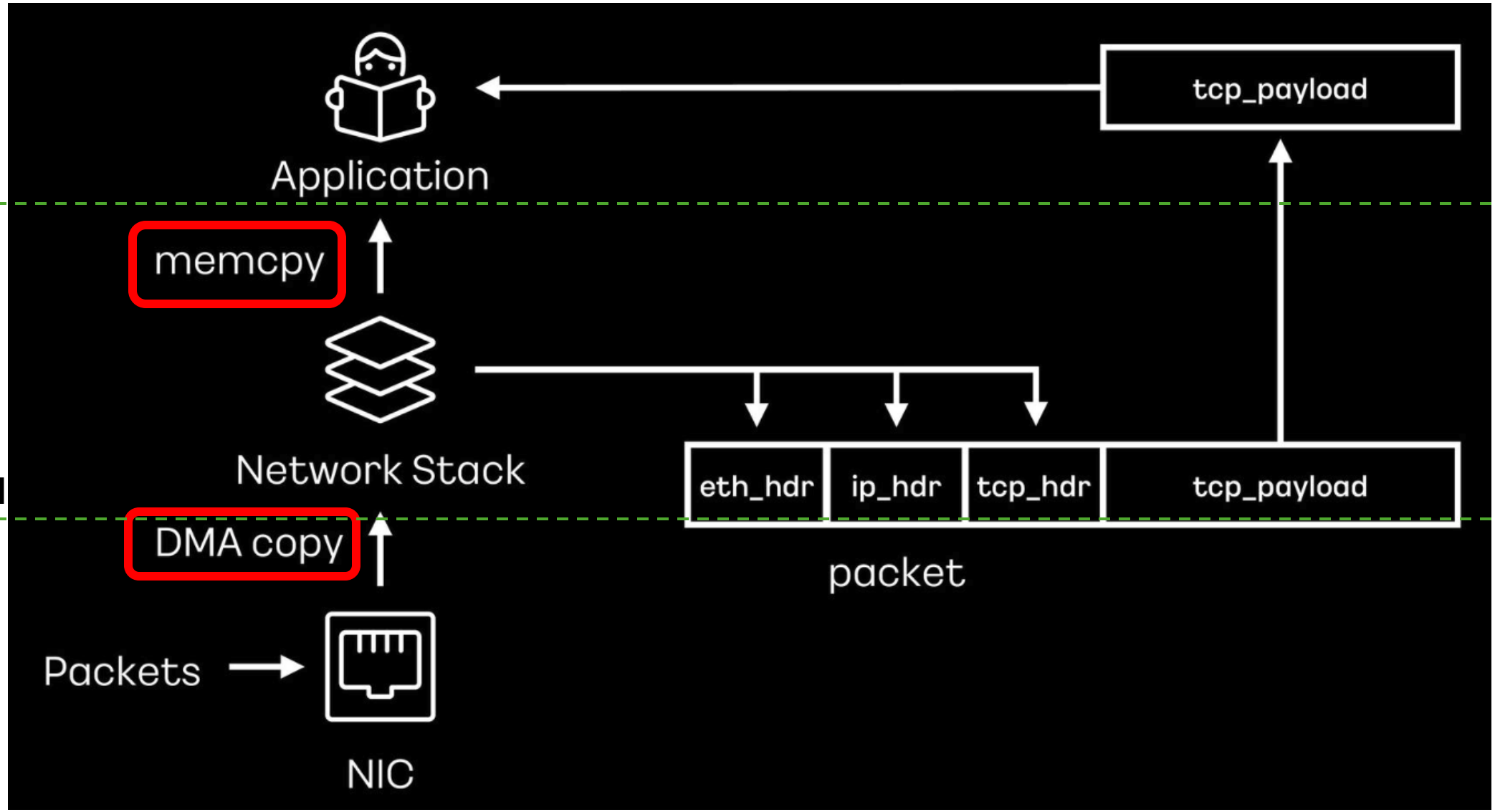
```
public void call() {
    while (!A.nioRead()) ;
    A.A();
    A.B();

    B.C();
    while (!B.nioRead()) ;
    B.D();
}
```

# CS

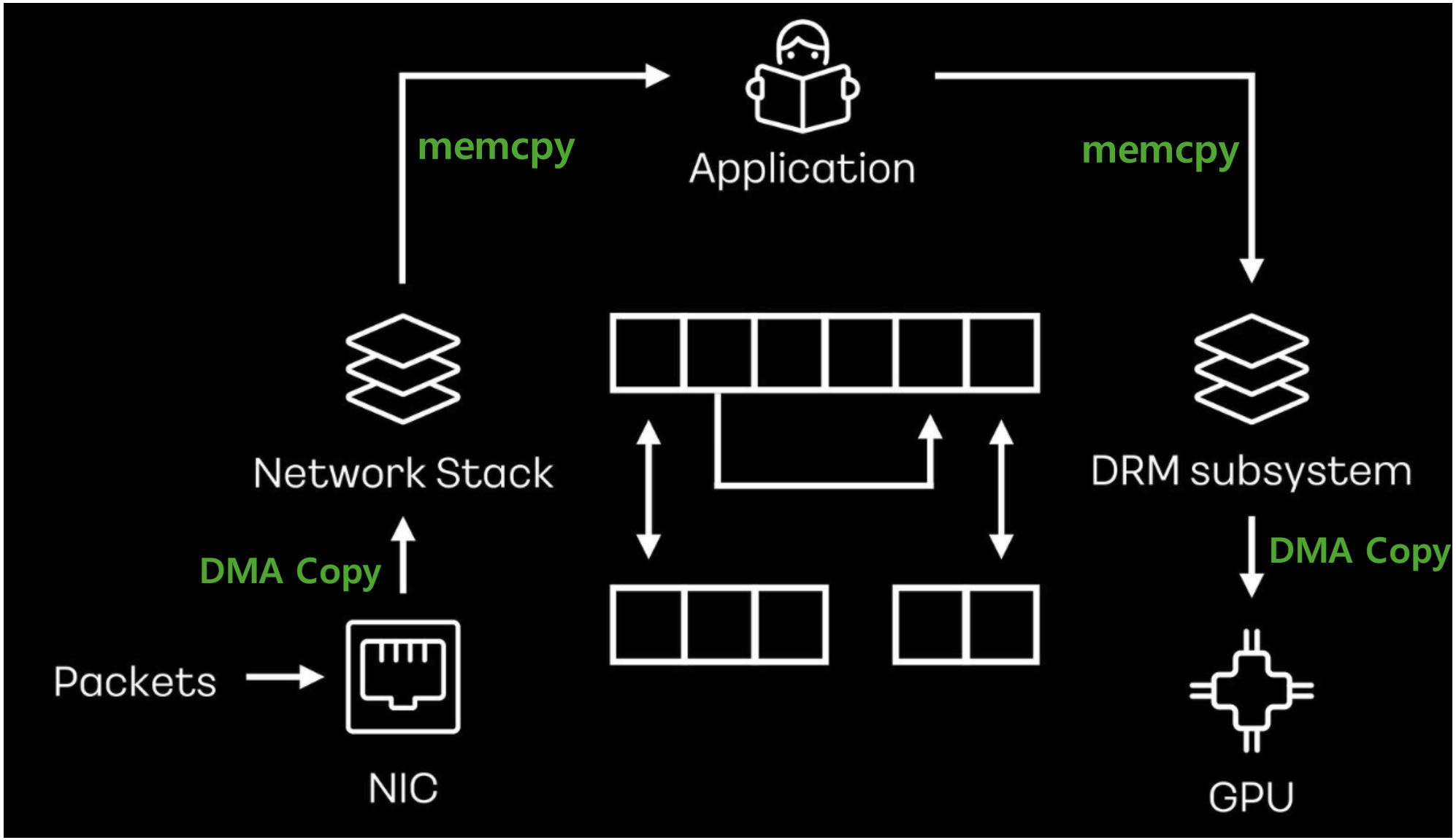
## Common Sense

# Ethernet NIC in Linux kernel / if(kakaoAI) 2024




https://www.youtube.com/watch?v=UTvGJwXbNzs

# Ethernet NIC in Linux kernel / if(kakaoAI) 2024

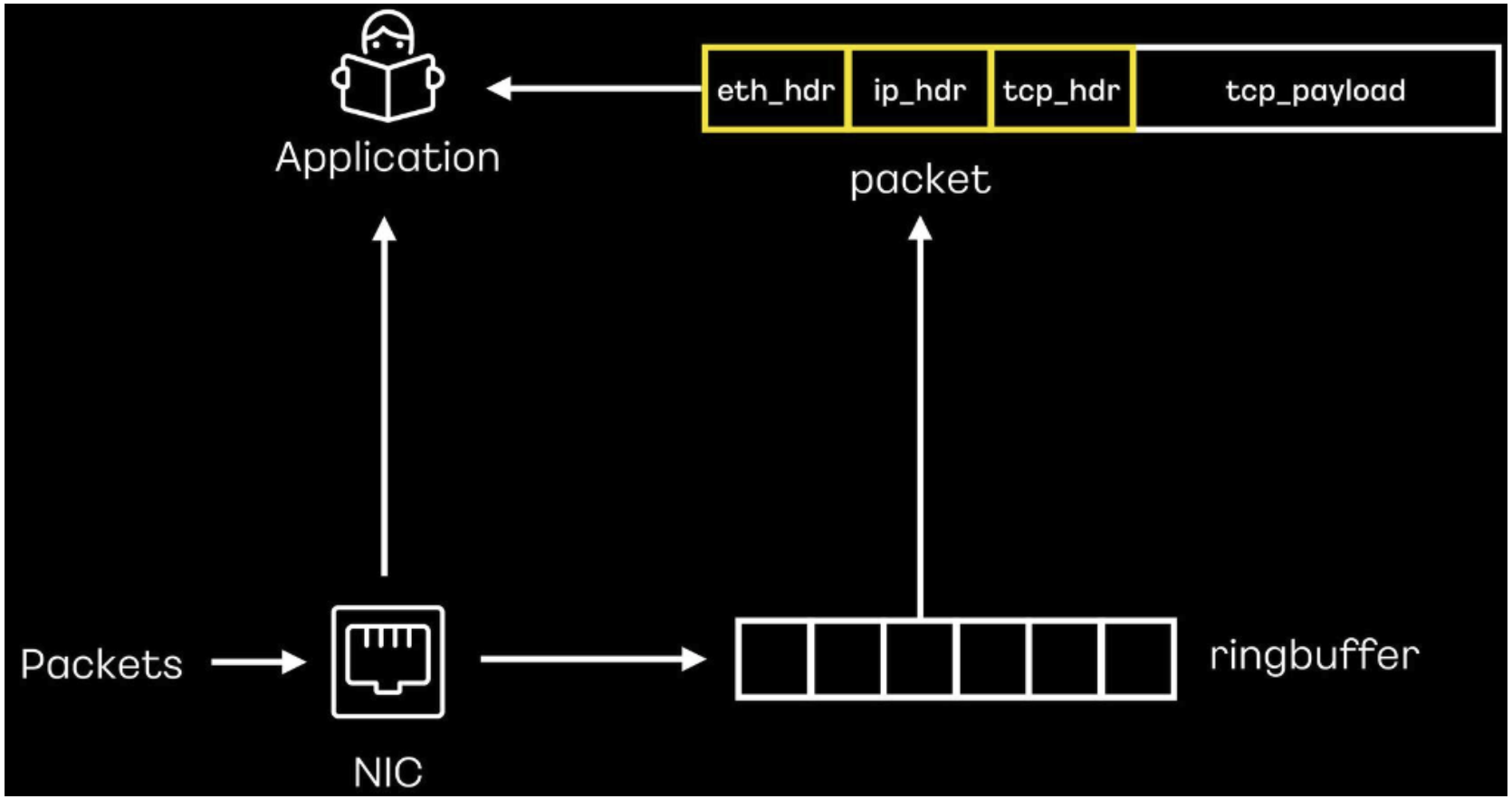



https://www.youtube.com/watch?v=UTvGJwXbNzs

# RX zerocopy

**Bypass network stack**

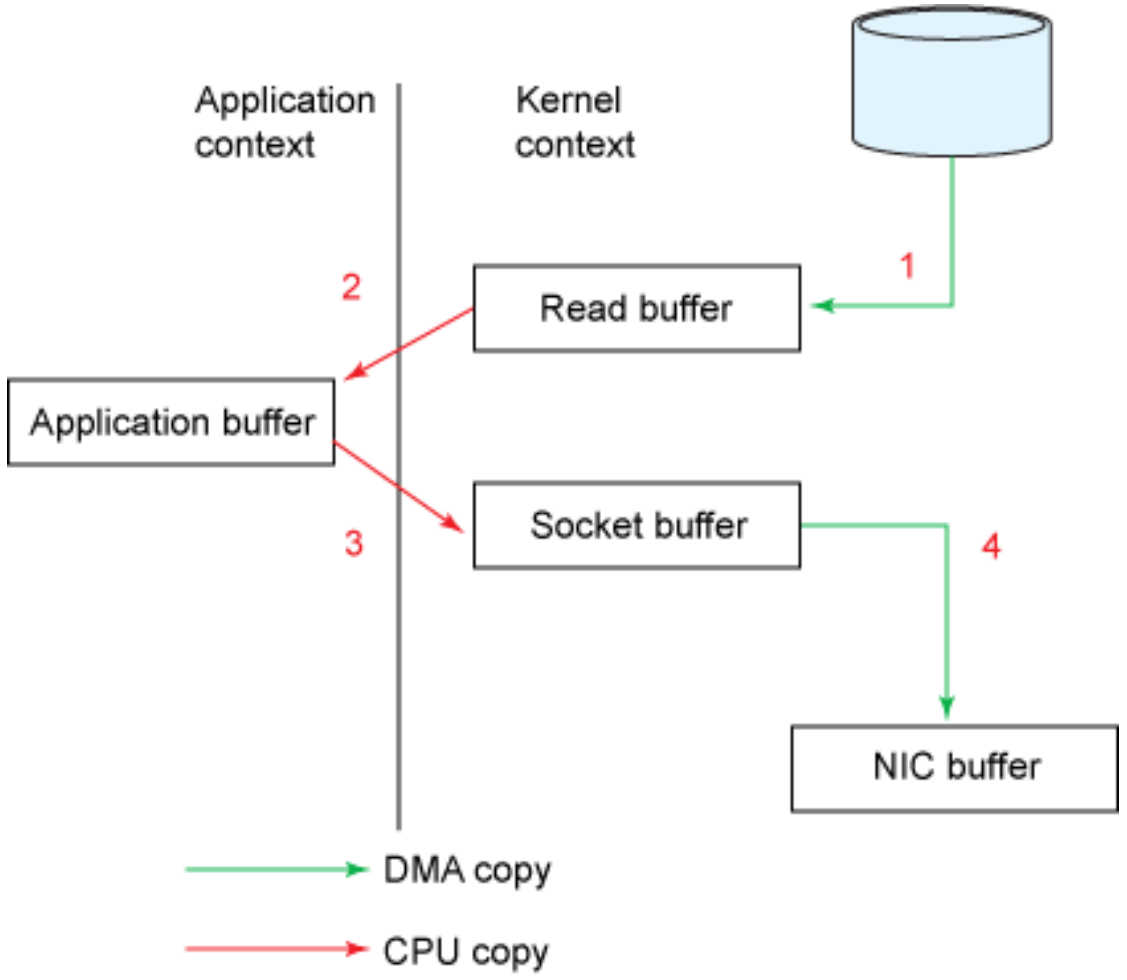
- 헤더 정보를 열어볼 수 있는 장점
- 내가 구현한 패킷 프로세싱 성능이 떨어진다면?
- Linux Network Stack의 도움을 받을 수 없음(Protocol handling, TC, Netfilter, …)



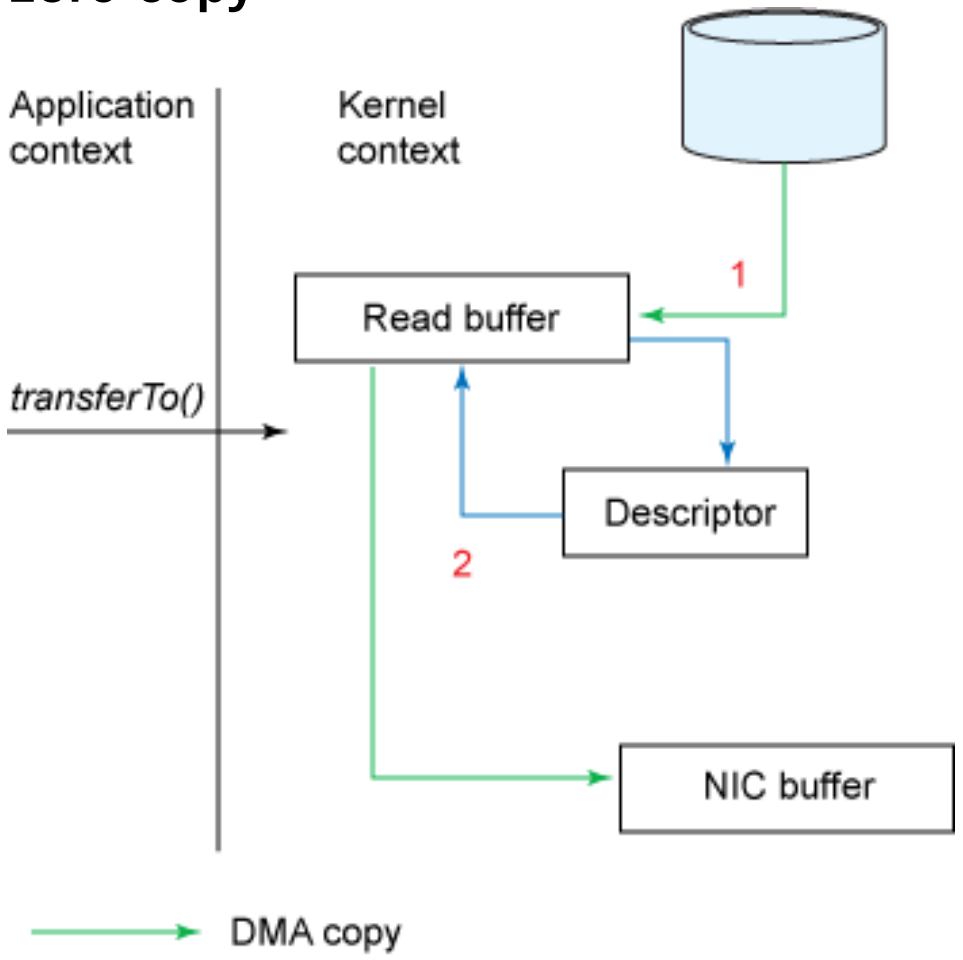

https://www.youtube.com/watch?v=UTvGJwXbNzs

# zero copy

## Traditional




## zero copy




https://developer.ibm.com/articles/j-zerocopy/

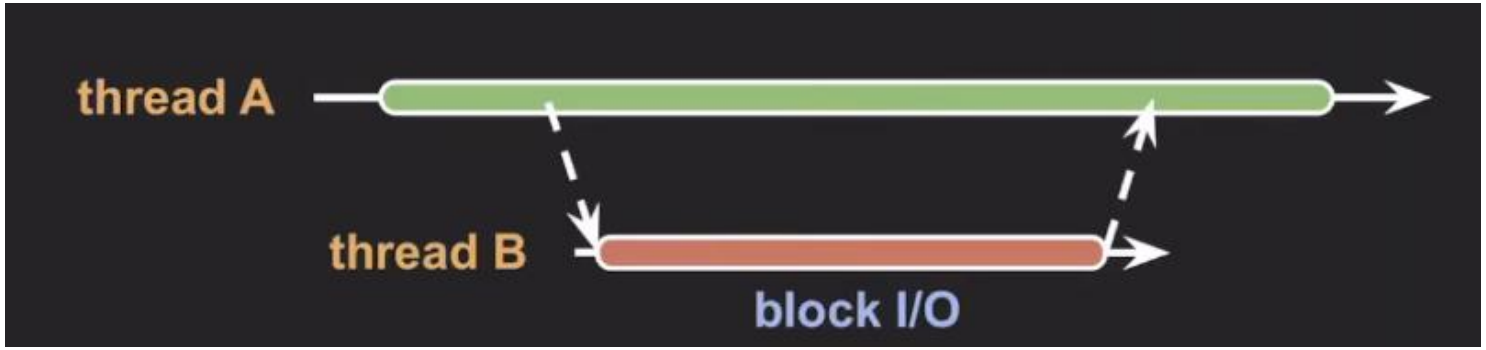
# subject 3.

## Asynchronous

# Asynchronous programming

**작업간의 선후관계가 없다.**
multi-threads + non-block I/O ?

**Blocking 함수를 사용한다면**
다른 스레드한테 던지기?



# Asynchronous + Multi-thread

## Practice

https://github.com/CS-Computer-Science-Study/Operating-System/tree/main/practice/asynchronous

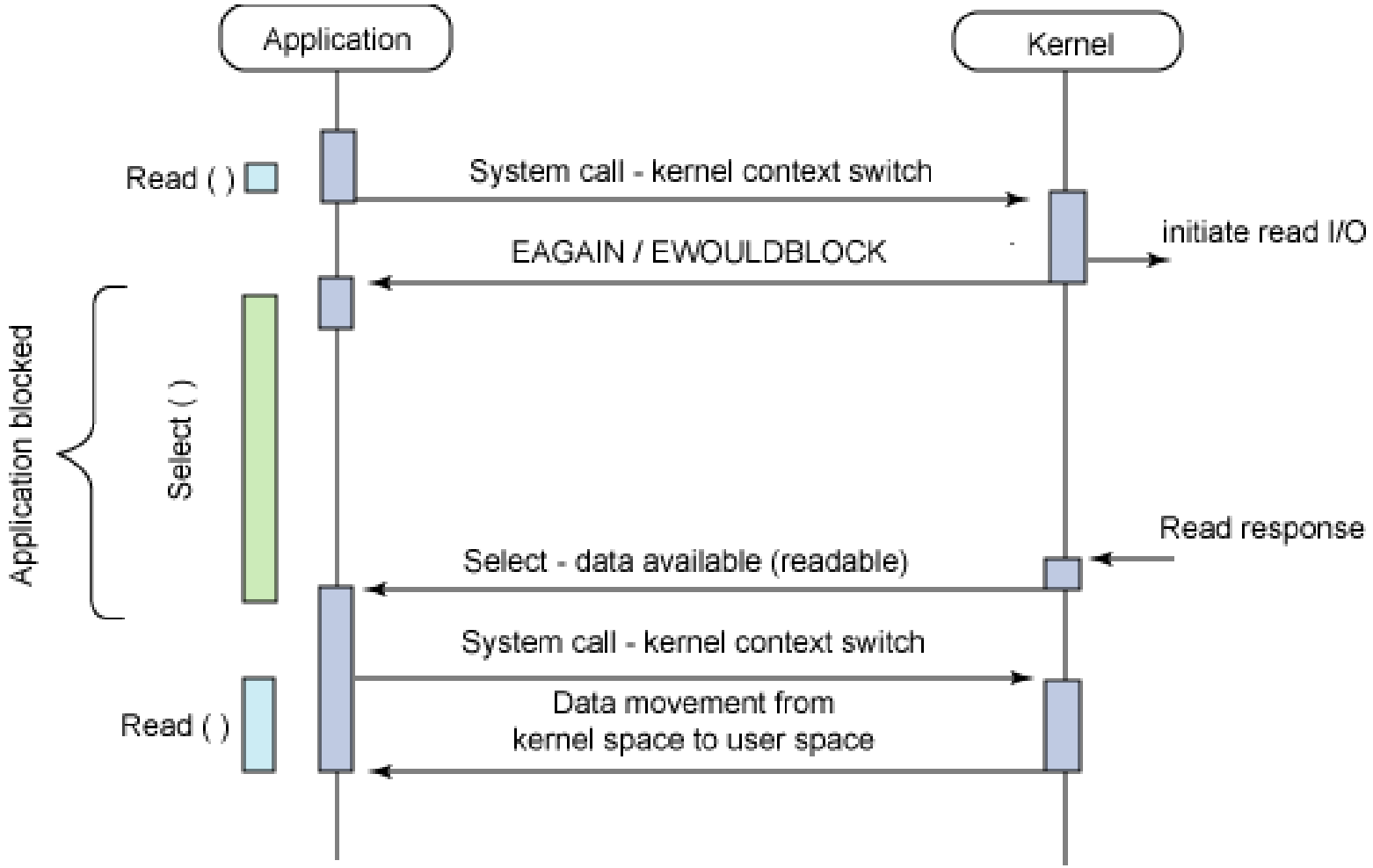
단위: nano

| | CPU intensive | I/O |
|---|---|---|
| **1 thread** | 4553775667 | 19643030375 |
| **HW threads number** | 1695922208 | 12940036708 |
| **20 threads** | 2505987291 | 8298103583 |

# Asynchronous Blocking I/O

**I/O Multiplexing**

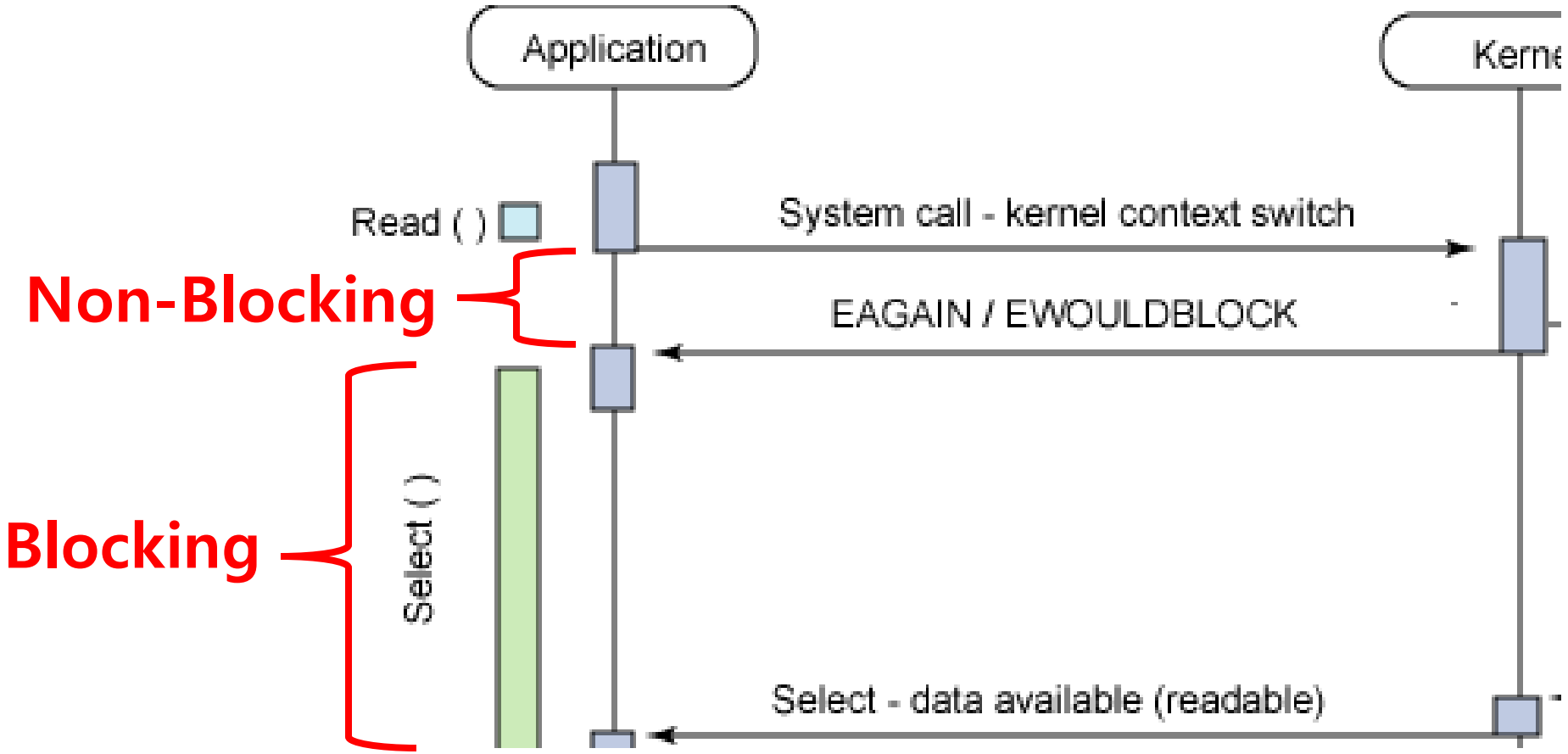
select, poll, epoll(linux), kqueue(bsd), iocp(windows)




https://developer.ibm.com/articles/l-async/

# Asynchronous Blocking I/O

논란 1.



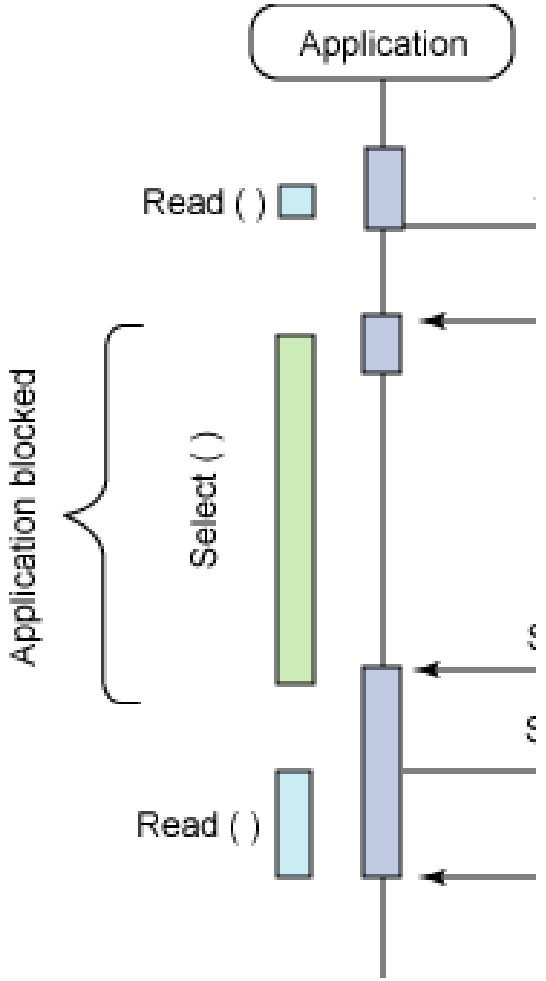
# Asynchronous Blocking I/O

**논란 2.**

실제 select()를 사용하는 코드 예시

```
fd_num = select( … );
for (int i = 0; i <= fd_max; i++) {
    if(FD_ISSET(i, ..)) { // do something }
}
```

내가 직접 챙겨야하는데?



# subject 4.

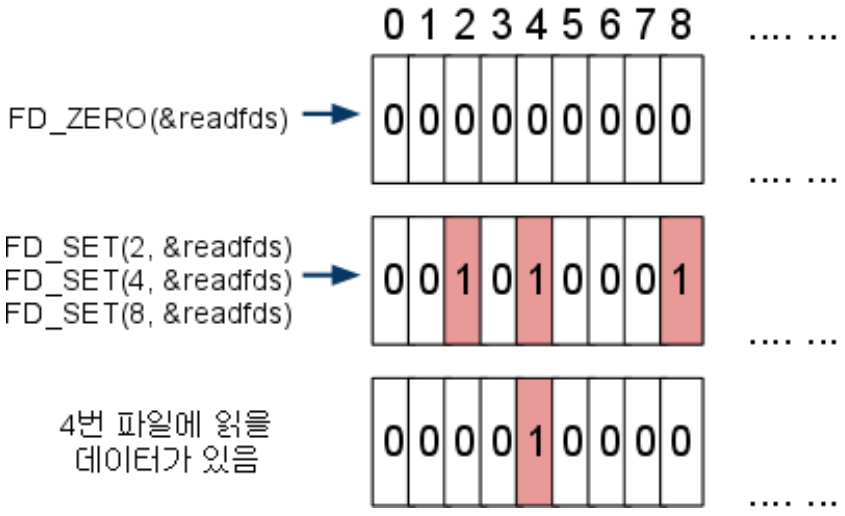
## I/O Multiplexing

# select()

bit table로 상태를 관리 – 최대 FD 수는 1024

return value : 데이터가 변경된 파일의 개수

어떤 FD가 변경됐는지는 안알려줌 -> for문 돌면서 확인해야 함




```c
fd_set set, cpy_reads;
FD_ZERO(&set);
FD_SET(target_fd, &set); // target fd에 bit 세팅

while(1) {
    // 다시 select 함수에 들어가기 전에 cpy_reads 세팅을 다시 해야 한다.
    cpy_reads = reads;

    if((fd_num = select(fd_max + 1, &cpy_reads, 0, 0, &timeout)) == -1) {
        break; // error
    }
    if(fd_num == 0) continue; // timeout

    for(int i = 0; i < fd_max + 1; i++) {
        if(FD_ISSET(i, &cpy_reads)) {
            // do something
            FD_CLR(i, &set); // 이제 i번은 검사하지 않을거야
        }
    }
}
```

# epoll()

FD 수 무제한
return : 감지된 FD의 목록 -> 사용자 레벨에서 루프를 돌 필요가 없다.

level-triggered : 입력 buffer에 데이터가 남아있는 동안 계속 이벤트 등록 (default in socket)
edge-triggered : 입력 buffer로 데이터가 수신된 상황에 딱 한번 이벤트 등록

```
int epfd = epoll_create(EPOLL_SIZE); // epoll FD

struct epoll_event event;
event.events = EPOLLIN; // 수신 이벤트
event.data.fd = target_fd;

// EPOLL_CTL_ADD: 관심있는 FD를 추가하겠다.
epoll_ctl(epfd, EPOLL_CTL_ADD, target_fd, &event); // 관찰대상이 되는 FD 등록하기

struct epoll_event *ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);
while(1) {
    int event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if (event_cnt == -1) break; // error

    for (int i = 0; i < evnet_cnt; i++) {
        int trigger_fd = ep_events[i].data.fd;
        // do something
        epoll_ctl(epfd, EPOLL_CTL_DEL, trigger_fd, NULL); // 이제 관심 없어
    }
}
```

# Refernece

operating system(korea university, 유혁)

https://blog.naver.com/n_cloudplatform/221896690 84

https://www.youtube.com/watch?v=EJNBLD3X2yg

Operating System Concepts (10/E, Silberschatz)

Computer Systems A Programmer’s Perspective (3/E, Randal E. Bryan)

Computer Organization And Design (6/E, David A. Patterson)