입출력 모델(I/O Models)

뇌를 자극하는 TCP/IP 소켓 프로그래밍

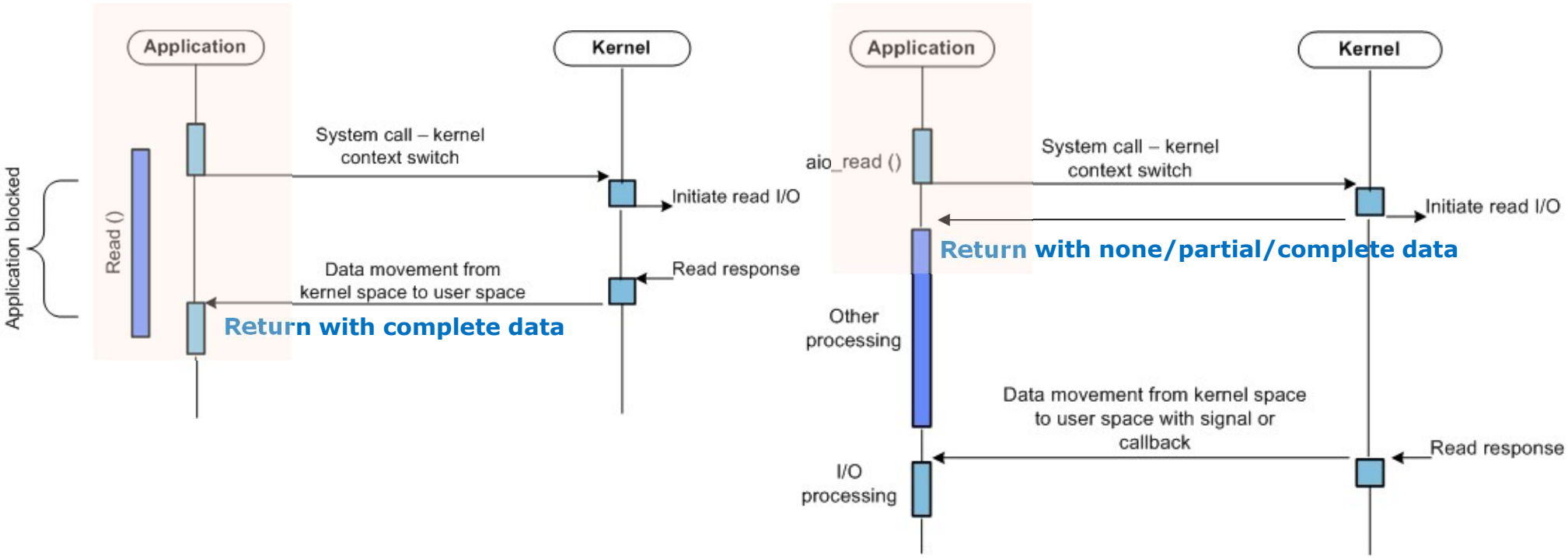
I/O Models

- Blocking I/O vs Non-blocking I/O
  - Whether the execution of application is suspended(blocked) and then resumed or not
  - Whether system call returns immediately or not
- Synchronous I/O vs Asynchronous I/O
  - Whether calling function takes care of completion of I/O or not
  - Whether calling function knows the completion time of I/O or not
  - Whether calling function synchronizes with the completion of I/O or not

Synchronous Blocking I/O	**Synchronous Non-blocking I/O**
Asynchronous Blocking I/O	**Asynchronous Non-blocking I/O**

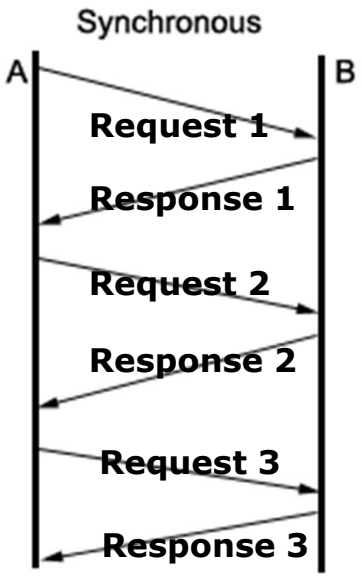
I/O Models

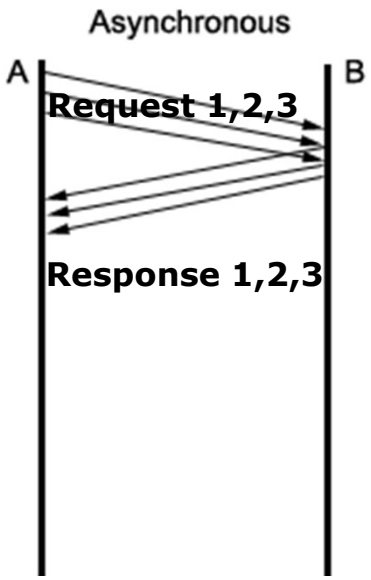
❖ Blocking I/O vs Non-blocking I/O

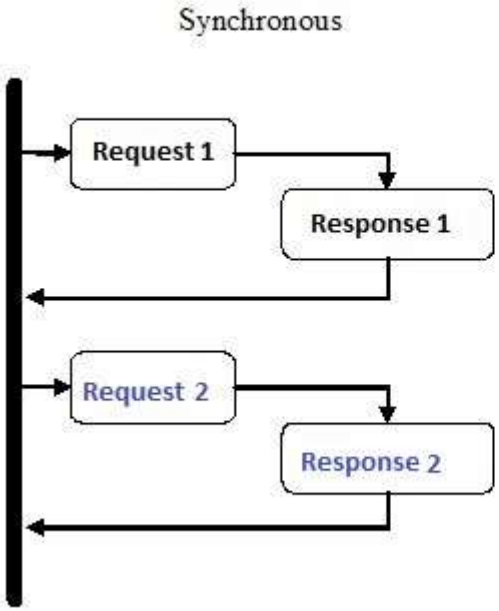


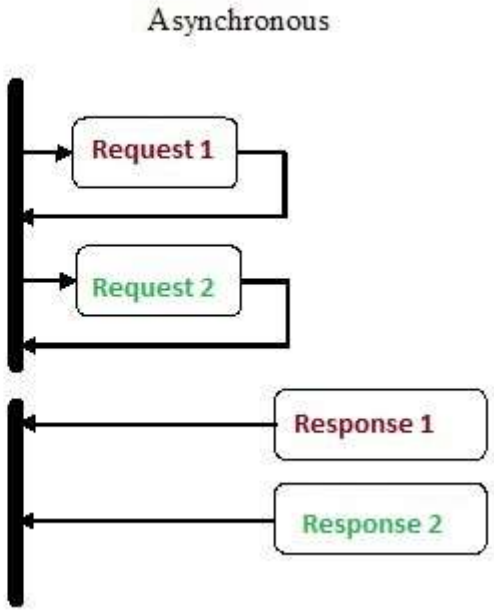
Source : https://docs.cyberoam.com/default.asp?id=44

I/O Models







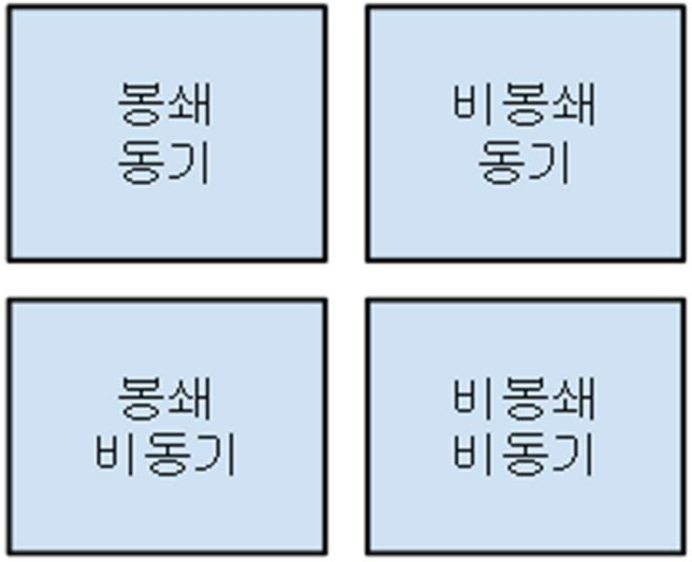


Source : http://blogs.quovantis.com/asynchronous-programming-async-and-await-in-c/

입출력 모델

- 봉쇄(Blocking) vs 비봉쇄(Non-Blocking)
- 동기(Synchronous) vs 비동기(Asynchronous)

- 이들 조합을 이용한 4개의 입출력 모델
  - 다양한 입출력 기술이 있지만 이들 4개 범주에 포함.

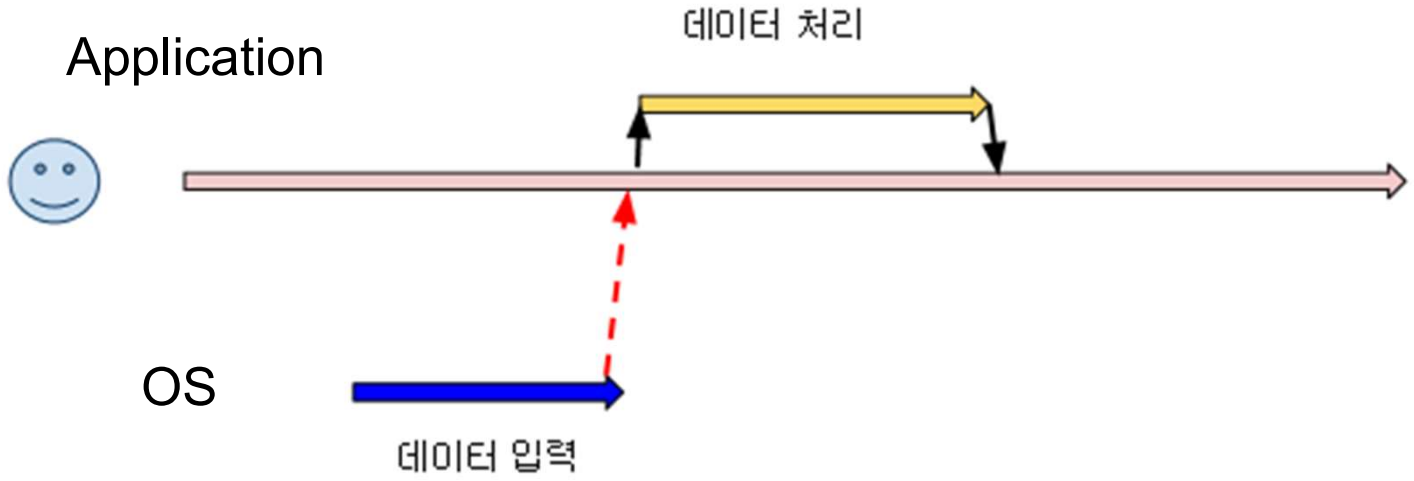


봉쇄 vs 비봉쇄

- 봉쇄 : 데이터가 준비될 때까지 봉쇄(Blocking)
  - 직관적이며, 결과를 예측하기 쉽다.
  - 프로그램이 명확하다.
  - 영원히 Blocking될 수 있으므로 주의해야 한다.
  - 봉쇄 되므로, 여러 파일을 동시에 처리하려면 다른 기술을 사용해야 한다.
- 비 봉쇄 : 즉시 반환(Non-Blocking)
  - 바로 반환 하므로, 여러 파일을 동시에 처리할 수 있다.
  - 직관적이지 않으므로 일반적으로 프로그램 개발이 까다롭다.
  - Busy wait를 주의해야 한다.
- 소켓은 생성시 기본적으로 봉쇄이며, **fcntl()** 함수로 비봉쇄로 변경 가능함

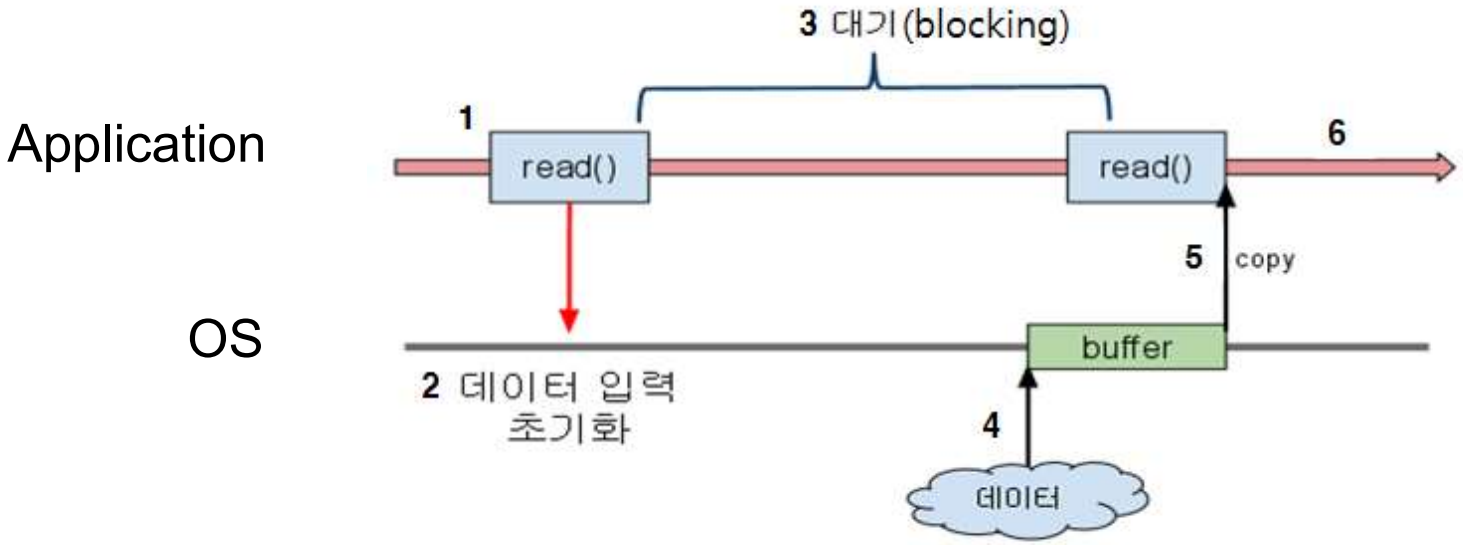
동기 vs 비동기

- 동기 : **데이터 입출력을 시점을 알고 있다.**
  - 입출력 함수를 호출 하는 시점에 데이터를 처리한다.
  - 입출력 함수 호출 시점이 동기화 시점
  - 데이터 흐름이 간결하다.
- 비동기 : **데이터의 입출력 시점을 모른다.**
  - 프로세스 진행 중에, 이벤트를 발생함으로써 데이터를 처리한다.
  - 이벤트를 받은 시점에 입출력 함수를 호출
  - 데이터를 효율적으로 처리할 수 있다.
  - 이벤트 처리로 프로그램이 복잡해질 수 있다. (이벤트 처리에는 많은 주의가 필요하다.)



1. 동기 & 봉쇄(Synchronous Blocking) 모델

- 입출력 함수를 호출 데이터가 있을 때까지 기다린다.
- 단순 하고 직관적



1. Application에서 **read** 함수 호출
2. 커널은 데이터 입력 초기화
3. Application은 데이터가 들어올 때까지 대기(blocking)
4. OS는 데이터를 수신하면 **커널 버퍼**에 저장
5. OS는 read 함수에서 지정한 **유저 버퍼에** 복사한 후 **read함수 반환**
6. Application에서 봉쇄가 풀리며, 다음작업(유저 버퍼에서 데이터 읽음) 수행

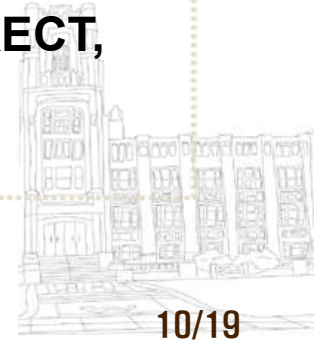
1. 동기 & 봉쇄(Synchronous Blocking) 모델

- 기본적으로 동기/봉쇄 모델임
- 단점 : **한번에 하나의 입력만 처리**할 수 있다, 즉 둘 이상의 소켓을 동시에 다룰 수 없다.
- 단점을 보완해주는 기술(입출력 다중화, 멀티 스레드, 멀티 프로세스)와 함께 사용하면 유용함

fcntl() - manipulate file descriptor

- #include <unistd.h>
  #include <fcntl.h>
  int fcntl(int fd, int cmd, ... /* arg */);

- Description
  - fcntl() performs one of the operations described below on the open file descriptor fd. The operation is determined by cmd.
  - fcntl() can take an optional third argument. Whether or not this argument is required is determined by cmd.
- **File status flags**
  - Each open file description has certain **associated status flags**, **initialized by open()** and **possibly modified by fcntl()**.
  - **F_GETFL** (void)
    - **Get the file access mode and the file status flags**; arg is ignored.
  - **F_SETFL (int)**
    - **Set the file status flags to the value specified by arg**. On Linux this command can change only the **O_APPEND, O_ASYNC, O_DIRECT, O_NOATIME**, and **O_NONBLOCK** flags.
- RETURN VALUE
  - F_GETFD, F_GETFL : **Value of flags**



Blocking Socket vs. Non-blocking Socket

- socket() 으로 생성되는 소켓은 기본값으로 봉쇄(Blocking) 소켓. fcntl() 함수를 이용하여 봉쇄 소켓을 비 봉쇄 소켓으로 만듬

```
int opts, cpyopts;
opts = fcntl(sockfd, F_GETFL);
cpyopts = opts;
opts = (opts | O_NONBLOCK);
fcntl(sockfd, F_SETFL, opts);

fcntl(sockfd, F_SETFL, cpyotps);          // return to blocking socket
```

Blocking Socket vs. Non-blocking Socket

- **accept()**
  - BS : backlog(connection 요청 큐)가 비어 있을때 blocking
  - NS : backlog(connection 요청 큐)가 비어있을때 -1 return, errno==EWOULDBLOCK/EAGAIN // if(errno==EAGAIN)
- **connect()**
  - BS : 서버가 connection 요청을 받을때까지 blocking
  - NS : connection이 이루어지 않더라도 곧바로 return. 나중에 getsockopt로 connection이 완전히 이루어졌는지 확인가능.

- **read()**
  - BS : read 버퍼가 비어 있을때 blocking
    - read 버퍼에 존재하는 데이터의 크기가 read시 요청한 데이터의 크기보다 작은 경우, read 버퍼에 존재하는 데이터만큼 리턴되며 block 되지 않음.
  - NS : read 버퍼가 비어있을때 -1 return, errno==EWOULDBLOCK/EAGAIN
- **write()**
  - BS : write 버퍼가 꽉 차 있을때 blocking
  - NS : write 버퍼가 꽉 차있을때 -1 return, errno==EWOULDBLOCK/EAGAIN

Blocking Socket vs. Non-blocking Socket

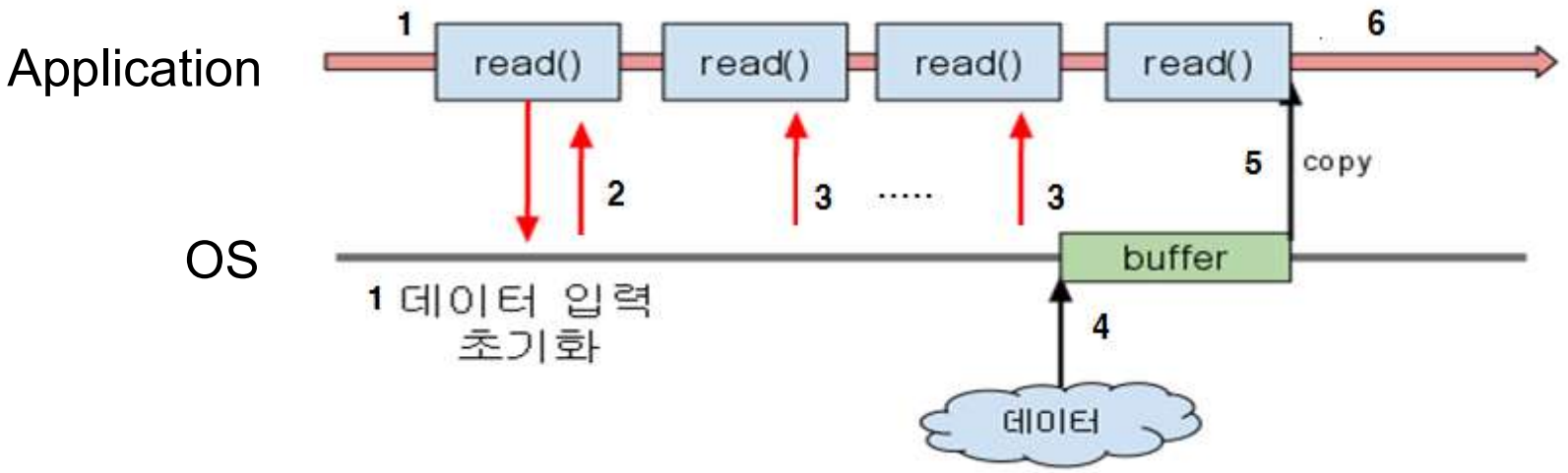
- Non-blocking Socket의 경우, 입출력 함수(read,write,accept)의 반환 값만으로는 실제 에러인지, 비봉쇄에 의한 반환인지 확인할 수 없음
  - errno를 확인 : 가장 최근 발생한 함수의 에러 코드를 저장하는 외부변수
  - errno값이 **EAGAIN** 혹은 **EWOULDBLOCK**이면 비봉쇄에 의한 반환
- busy wait로 인하여 CPU 효율이 나쁨
  - 동기&비봉쇄 모델은 거의 사용하지 않는 모델
- Nonblocking 소켓의 장점/단점
  - 장점 : 멀티스레드를 사용하지 않고도 다른 작업을 할 수 있다.
  - 단점 : 프로그램이 복잡해지며, CPU 사용량이 증가한다.

2. 동기 & 비봉쇄 (Synchronous Non-blocking) 모델

- 소켓을 비 봉쇄로해서 처리한다.
- **busy wait** 문제.
  - 데이터 입력을 계속 검사하기 때문에.
- 입출력 다중화로 해결 할 수 있음

- busy wait로 인하여 CPU 효율이 나쁨
  - 동기&비봉쇄 모델은 거의 사용하지 않는 모델

2. 동기 & 비봉쇄 (Synchronous Non-blocking) 모델



1. Application에서 read 함수 호출 : 커널은 데이터 입력 초기화
2. **read함수 즉시 반환** : 데이터가 없다면 **errno를 EAGAIN으로 설정하고 -1반환**
3. Application은 데이터가 입력되었는가를 확인하기 위하여 계속 물어보면서 반복
4. OS는 데이터를 수신하면 커널 버퍼에 저장
5. 이제 read함수 수행하면 커널은 유저 버퍼로 복사함
6. Application에서 이제 유저 버퍼에서 데이터를 읽으면서 처리함

3. 비동기 & 봉쇄 (Asynchronous Blocking) 모델

- Blocking 소켓을 사용함.

- **blocking**될 수 있는 **accept(), read()** 함수는 입력이 있는가를 확인하여 데이터가 있을 경우만 **호출**함, 즉 blocking 당하지 않도록 한다.

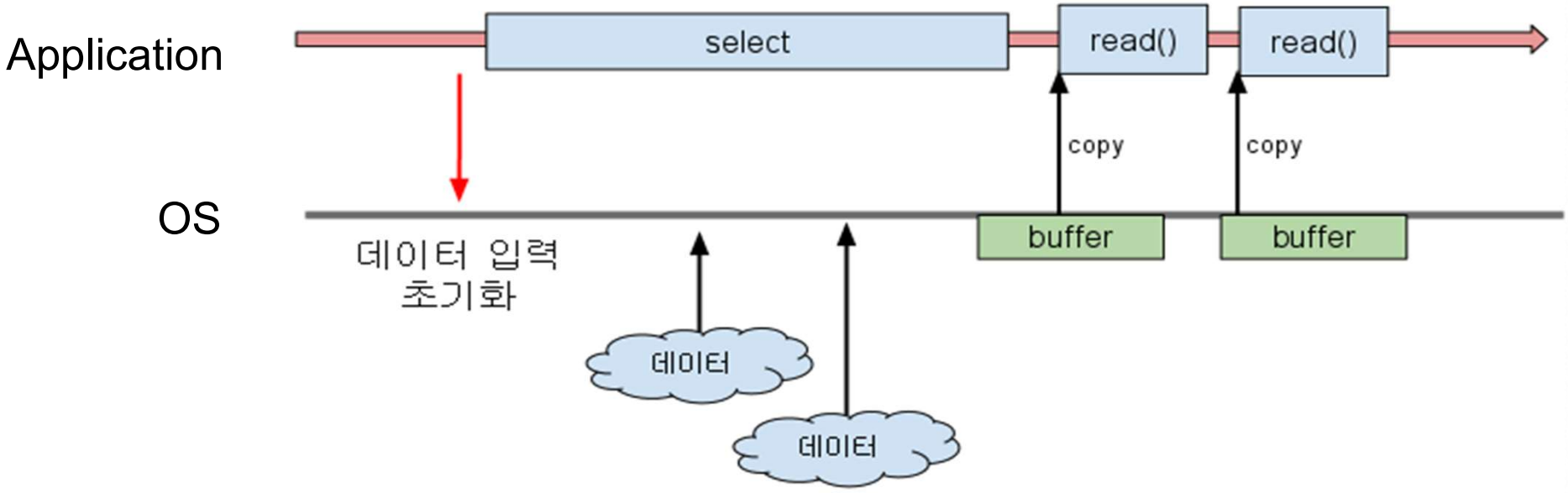
- accept(), read() 입출력 함수 호출 앞 단에 **소켓에 어떤 데이터의 변화가 있는지 입력 이벤트를 확인**하도록 함

- 데이터가 들어오면 이벤트를 발생하고, 이때 입출력 함수를 호출한다.
- 단일 프로세스로 여러 입출력을 처리할 수 있다.
- 매우 효율적이며, 프로그램이 직관적이다.

- read함수 호출 : 커널은 데이터 입력을 초기화 한다.
- 데이터가 들어오면 커널 버퍼에 데이터가 쌓이고, 이벤트를 발생
- 이벤트를 catch해서, 이벤트가 발생한 파일에서 데이터를 읽어서 처리

3. 비동기 & 봉쇄 (Asynchronous Blocking) 모델

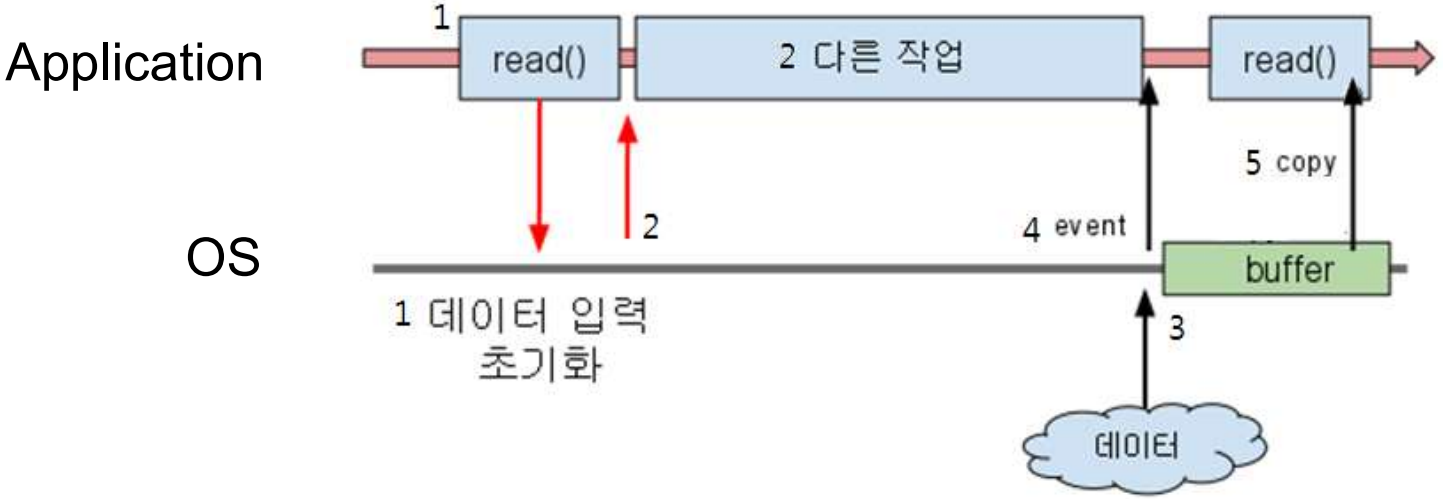
- 데이터가 들어오면 이벤트를 발생하고, 이때 입출력 함수를 호출한다.
- 단일 프로세스로 여러 입출력을 처리할 수 있다.
- 매우 효율적이며, 프로그램이 직관적이다.



- read함수 호출 : 커널은 데이터 입력을 초기화 한다.
- 데이터가 들어오면 커널 버퍼에 데이터가 쌓이고, 이벤트를 발생
- 이벤트를 catch해서, 이벤트가 발생한 파일에서 데이터를 읽어서 처리

4. 비동기 & 비봉쇄 (Asynchronous Non-blocking) 모델

- 데이터 입출력과 유저모드 프로세스가 분리
- 데이터가 준비되면 이벤트로 통지
- 매우 효율적이다.
- 다루기가 까다롭다.
  - 비동기 기술과 비봉쇄 기술은 다루기 까다로운 기술이다.



1. Application에서 read 함수 호출 : 커널은 데이터 입력 초기화
2. **read함수 반환,** Application은 **read한 데이터를 사용하지 않는 다른 작업** 수행
3. OS는 데이터를 수신하면 커널 버퍼에 저장하고
4. Application에 이벤트로 통지
5. Application에서 유저 버퍼에서 데이터를 읽음

Thank You !

뇌를 자극하는 TCP/IP 소켓 프로그래밍