

# **Network Programming 2020 Seminar**

## **6. I/O Multiplexing**

2017707063

김 문 일

---

# Network Programming

## 2020 Seminar

---

### 목차

#### 6.1 Introduction

다중 접속 서버

#### 6.2 I/O Models

#### 6.3 select Function

#### 6.4 str\_cli Function (Revisited)

#### 6.5 Batch Input and Buffering

#### 6.6 Shutdown Function

#### 6.7 str\_cli Function (Revisited Again)

#### 6.8 TCP Echo Server (Revisited)

#### 6.9 pselect Function

#### 6.10 poll Function

#### 6.11 TCP Echo Server (Revisited Again)

#### 6.12 Summary

# 6.1 Introduction

---

## I/O Multiplexing이 필요할 때 & 필요한 이유

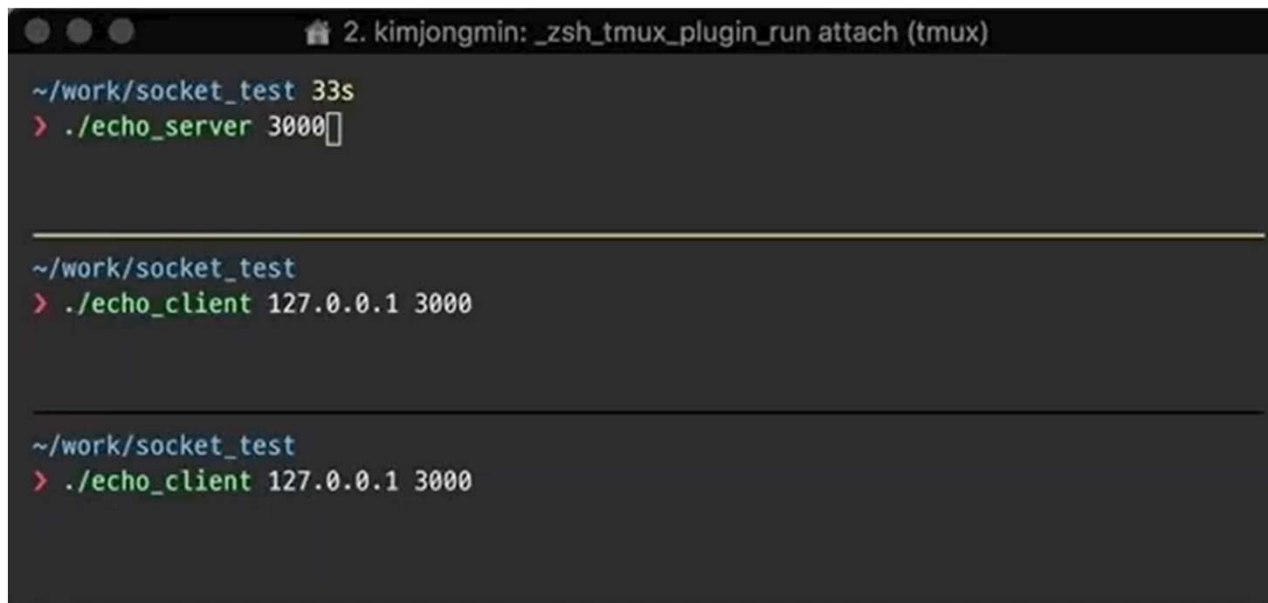
- Client가 multiple descriptors\*을 처리할 때 **반드시 필요**
- Client가 동시에 multiple socket을 처리할 때 **쓸 수도 있음**
- TCP Socket을 활용하여 통신하는 server가 listening socket과 connected sockets을 동시에 처리할 때 **보통 사용**
- Server가 TCP, UDP 기반의 Socket을 모두 처리할 때, **보통 사용**
- Server가 multiple services, multiple protocols를 처리할 때 **보통 사용**

---

descriptor: 시스템으로부터 할당 받은 파일이나 소켓을 대표하는 정수를 의미  
해당 챕터에선 Socket과 동일한 의미로 받아드려도 무방함

# 다중 접속 서버

## 다중 접속 서버 도입 이유



```
2. kimjongmin: _zsh_tmux_plugin_run attach (tmux)

~/work/socket_test 33s
> ./echo_server 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000
```

기존의 방식으로 다중 클라이언트와 동시 연결이 불가능

# 다중 접속 서버

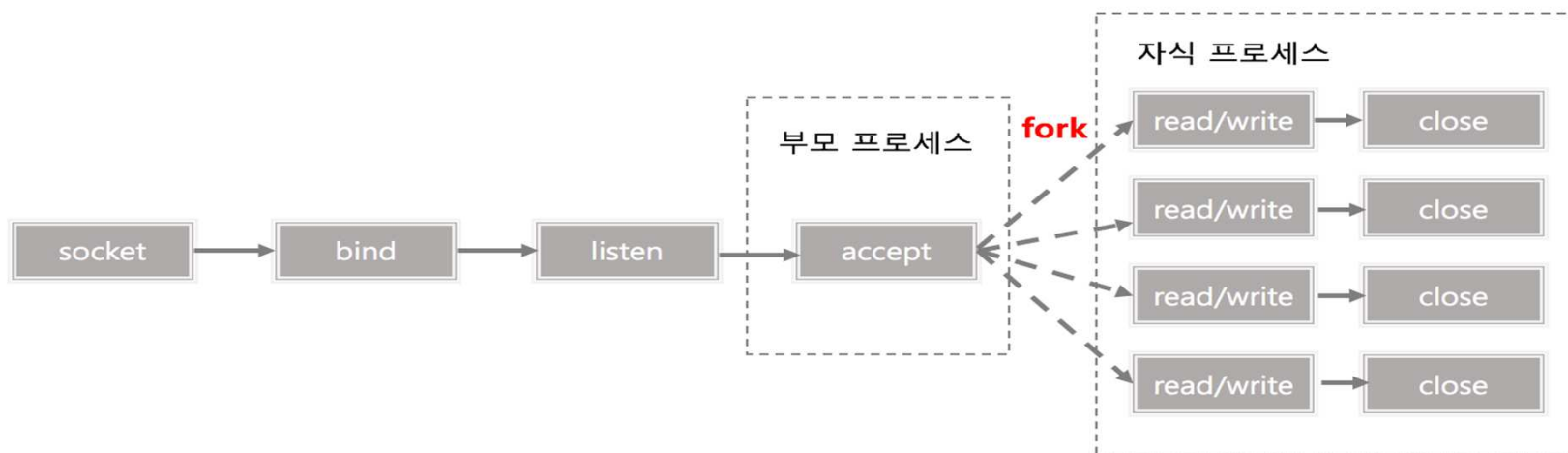
---

## 다중 접속 서버의 종류

- 멀티프로세스 기반 서버 : 다수의 프로세스를 생성하는 방식으로 서비스를 제공한다.
  - 멀티스레드 기반 서버 : 클라이언트의 수만큼 스레드를 생성하는 방식으로 서비스를 제공한다.
  - 멀티플렉싱 기반 서버 : 입출력 대상을 묶어서 관리하는 방식으로 서비스를 제공한다.
-

# 다중 접속 서버

## 멀티프로세스 기반 서버



1. 부모 프로세스는 리스닝 소켓으로 accept 함수 호출을 통해서 연결 요청을 수락함
2. 이때 얻게 되는 소켓의 파일 디스크립터(클라이언트와 연결된 연결 소켓)를 자식 프로세스를 생성해 넘겨줌
3. 자식 프로세스는 전달받은 파일 디스크립터를 바탕으로 서비스를 제공

# 다중 접속 서버

## 멀티프로세스 기반 서버

```
2. kimjongmin: _zsh_tmux_plugin_run attach (tmux)

~/work/socket_test
> ./echo_multi_process_server 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000
```

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 다중 접속 서버

## 멀티프로세스 기반 서버 장단점

### • 장점

- 프로그램 흐름이 단순하다.
- 독립적인 실행 객체로서 존재하기 때문에 안정적인 동작이 가능하다.

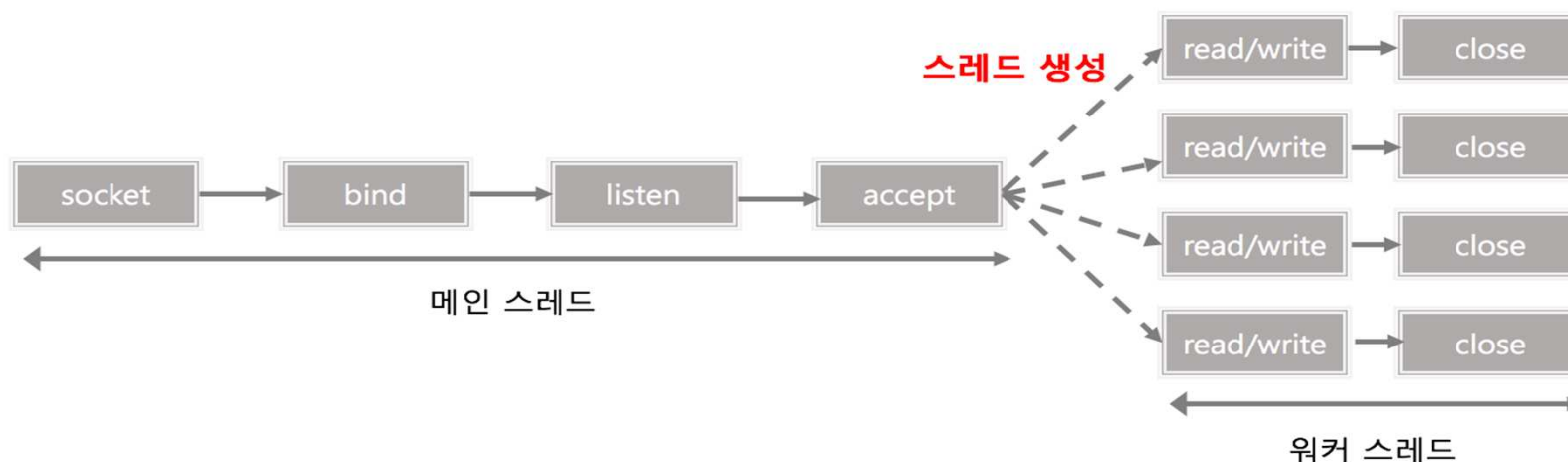
### • 단점

- 프로세스 복사에 따른 성능 저하 문제가 존재한다.
- 병렬 처리해야하는 만큼의 프로세스를 생성 해야 한다.
- 서로 다른 독립적인 메모리 공간을 갖기 때문에 프로세스간 정보 교환이 어렵다.
- IPC를 다루어야 하므로 까다로움



# 다중 접속 서버

## 멀티스레드 기반 서버



1. 메인 스레드는 리스닝 소켓으로 accept 함수 호출을 통해서 연결요청을 수락
2. 이때 얻게 되는 소켓의 파일 디스크립터(클라이언트와 연결된 연결 소켓)를 별도의 워커 스레드를 생성해 넘겨줌(pthread\_create 활용)
3. 워커 스레드는 전달받은 파일 디스크립터를 바탕으로 서비스를 제공

# 다중 접속 서버

## 멀티스레드 기반 서버

```
2. kimjongmin: _zsh_tmux_plugin_run attach (tmux)

~/work/socket_test
> ./echo_multi_thread_server 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000
```

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 다중 접속 서버

## 멀티스레드 기반 서버의 장단점

### • 장점

- 프로세스 복사에 따른 비용보다 스레드 생성에 대한 비용이 적다.
- 스레드간 서로 공유하는 메모리를 갖기 때문에, 스레드간 정보 교환이 쉽다.

### • 단점

- 하나의 프로세스 내의 다수의 스레드가 존재하여, 하나의 스레드에서 문제가 생긴다면 프로세스에 영향을 미쳐 나머지 다수의 스레드에도 영향을 끼칠 수 있다.

---

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 다중 접속 서버

## Multiplexing Model 서버의 도입이 필요한 이유

- Synchronous, Blocking I/O 형태의 멀티 프로세스 서버, 멀티 스레드 서버 모두 고성능 서버로는 부적합
- Client 마다 Process나 Thread를 할당하면 Context Switching 문제로 메모리 적인 문제가 발생한다
- 그렇다면 하나의 스레드에서 다수의 클라이언트에 연결된 소켓을 관리하고 소켓에 이벤트가 발생할 경우에만 별도의 스레드를 만들어 처리하는 방법은?

## -> 멀티플렉싱 서버

Q1. 멀티 프로세스 기반 서버와 멀티 스레드 기반 서버는 언제 쓰일까?

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

## 6.2 I/O Models

---

### 들어가기 앞서

1. 애플리케이션에서 I/O 작업을 하는 경우, **스레드는 데이터 준비가 완료될 때까지 대기합니다.** 예를 들어 소켓을 통해 read(reform)를 수행하는 경우 데이터가 네트워크를 통해 도착하는 것을 기다립니다. 패킷이 네트워크를 통해 도착하면 커널 내의 버퍼에 복사됩니다. (처음에 커널 공간에 생성된 소켓의 구조에서 송신 버퍼와 수신 버퍼가 있는 것을 보았습니다.)
2. 커널 내의 버퍼에 복사된 데이터를 애플리케이션에서 사용하기 위해서는 커널 버퍼 (kernel space)에서 유저 버퍼 (user space)로 복사 후 이용해야 합니다. 애플리케이션은 유버 모드에서 유저 버퍼에만 접근이 가능하기 때문입니다

---

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

## 6.2 I/O Models

---

### I/O Models 종류

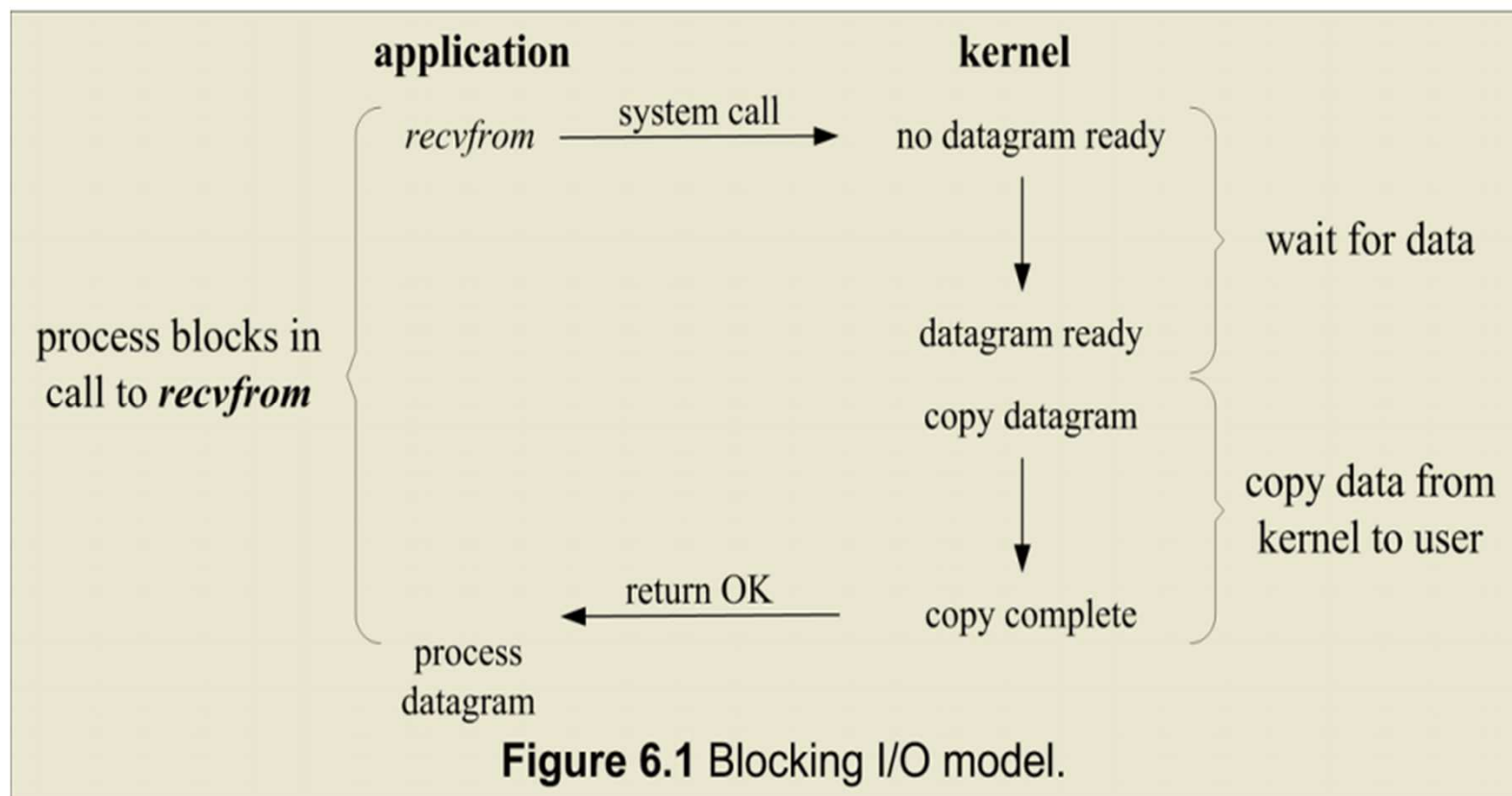
- Blocking I/O
- Nonblocking I/O
- I/O multiplexing (**select** and **poll**)
- Signal driven I/O (SIGIO)
- Asynchronous I/O (the POSIX **aio\_**functions)

---

<https://simsimjae.tistory.com/129>

# 6.2 I/O Models

## Blocking I/O



## 6.2 I/O Models

---

### Blocking I/O

- I/O 작업은 유저 영역에서 직접 수행할 수 없고, 커널 영역에서만 가능하다.

따라서 I/O 작업을 하기 위해서는 커널에 I/O 작업을 요청해야 한다.

- I/O 작업이 진행되는 동안 유저 영역 (process)의 작업은 중단한채 대기해야 한다.

**이처럼 I/O 동작** (데이터를 받을 준비, 데이터를 커널 영역에서 유저 영역으로 복사 등)**으로 인해 프로세스가 block 상태가 되는 것을 blocking 된다고 한다.**

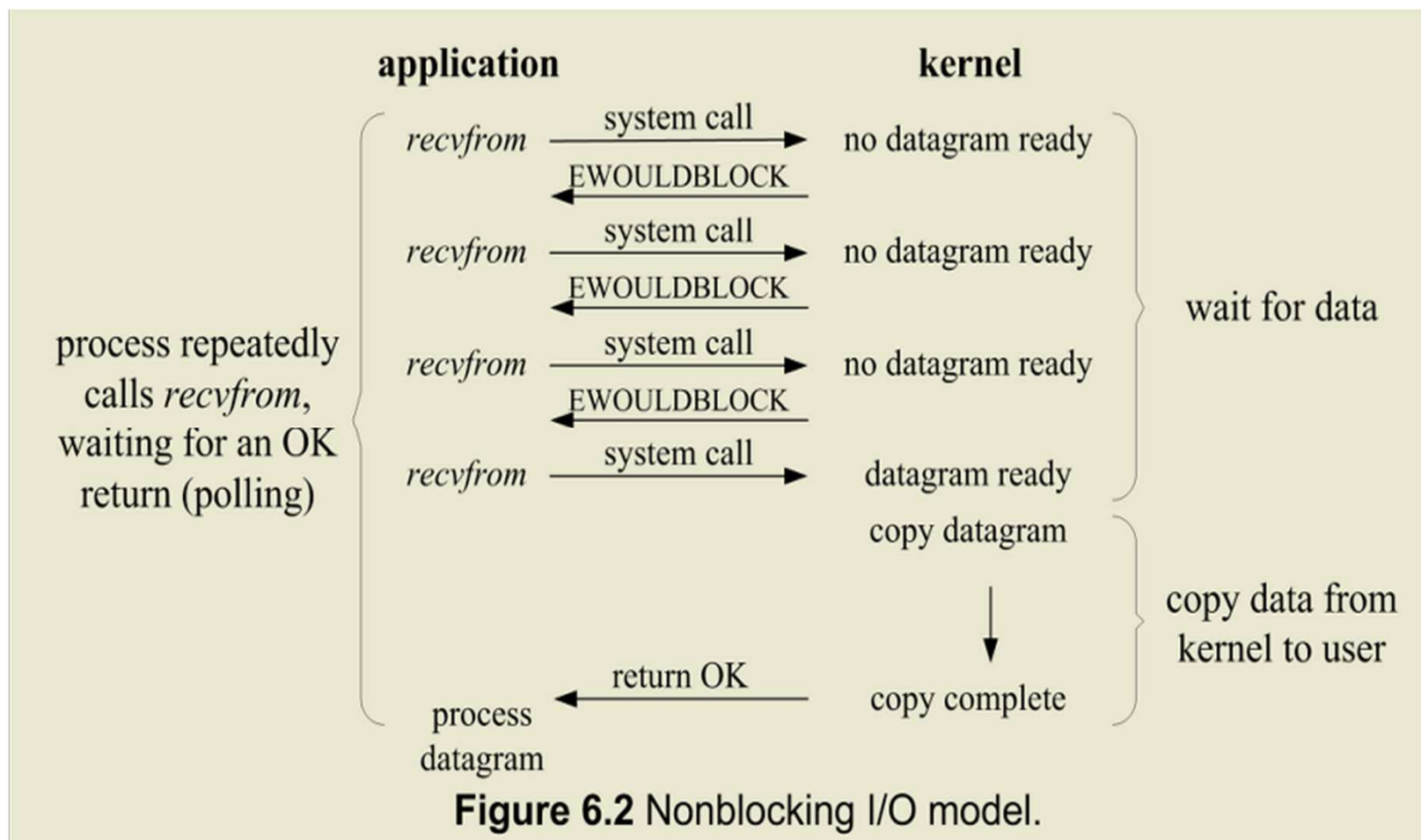
---

<https://simsimjae.tistory.com/129>



# 6.2 I/O Models

## Nonblocking I/O



## 6.2 I/O Models

---

### Nonblocking I/O

- I/O 작업을 진행하는 동안 유저 프로세스의 작업을 중단시키지 않는다.

유저 프로세스가 커널의 read를 기다리는 것이 아니라 system call을 이용하여 반복적으로 요청하고, 이에 대한 버퍼 값이 존재하면 유저 영역으로 복사해준다.

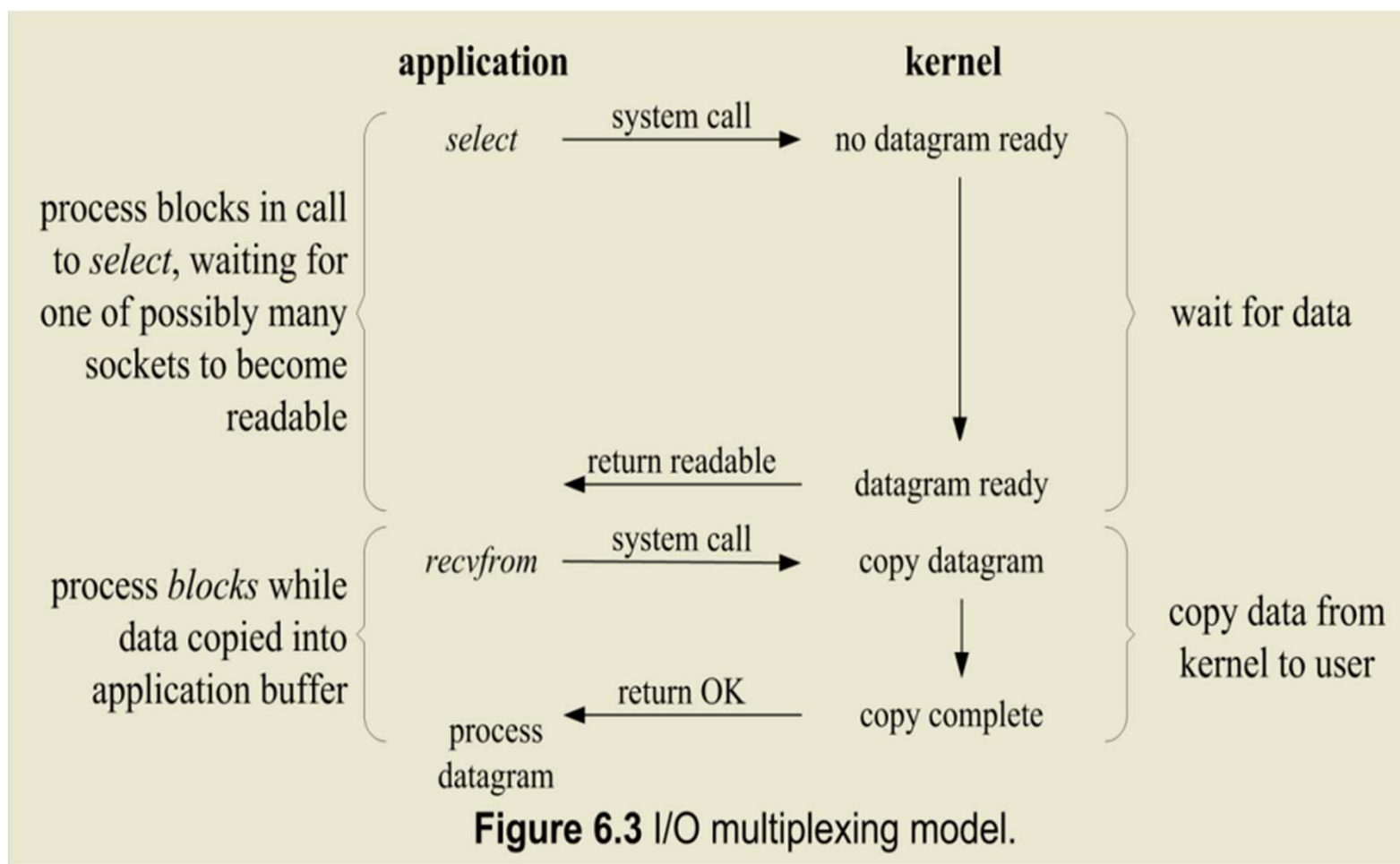
- 반복적인 system call로 리소스가 낭용

---

<https://simsimjae.tistory.com/129>

# 6.2 I/O Models

## I/O Multiplexing (select, poll)



## 6.2 I/O Models

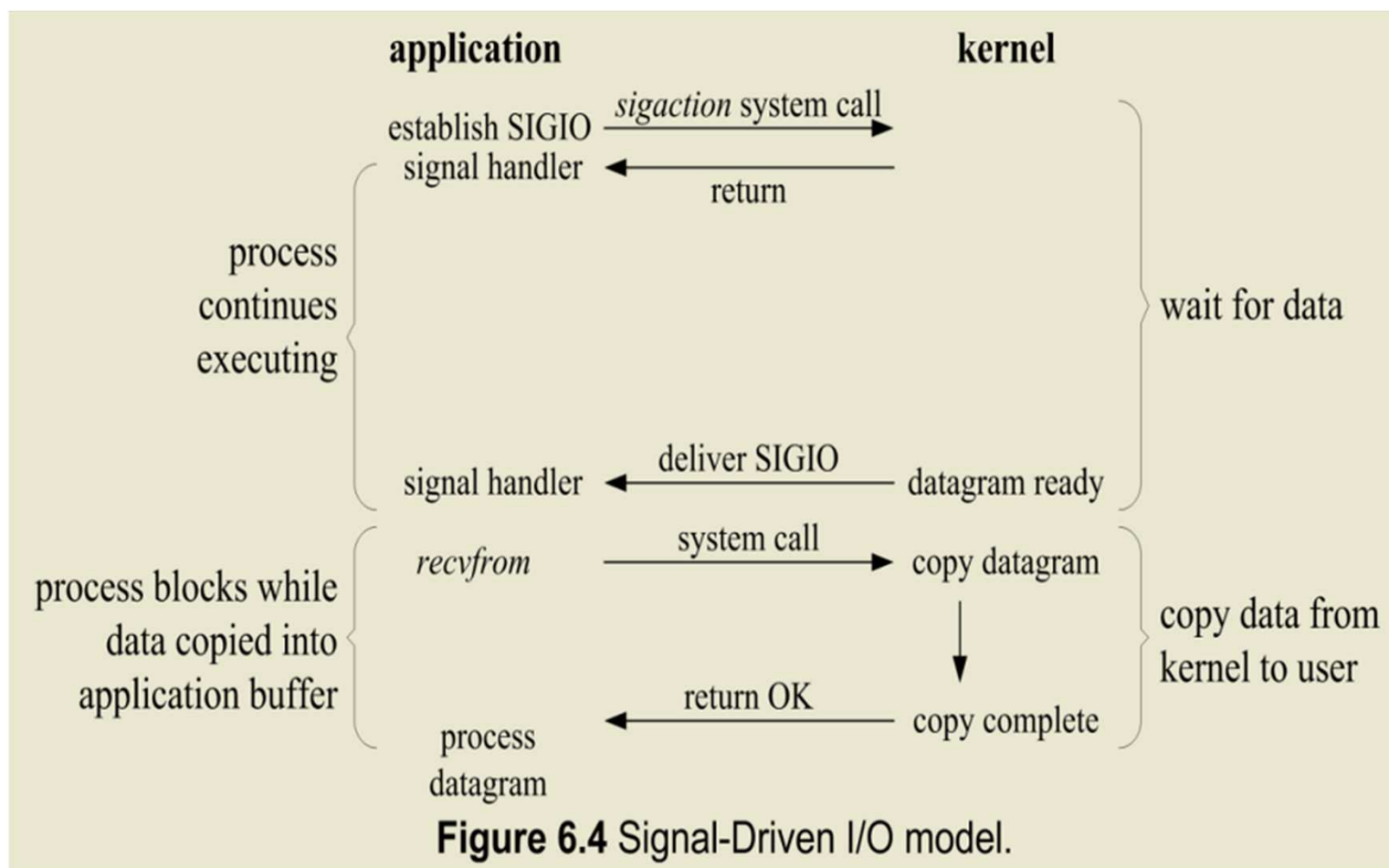
---

### I/O Multiplexing (select, poll)

- non-blocking 모델의 문제를 해결하기 위해 고안
- I/O 처리가 필요한 파일 디스크립터 등을 가려내서 알려준다.

# 6.2 I/O Models

## Signal-Driven I/O model



## 6.2 I/O Models

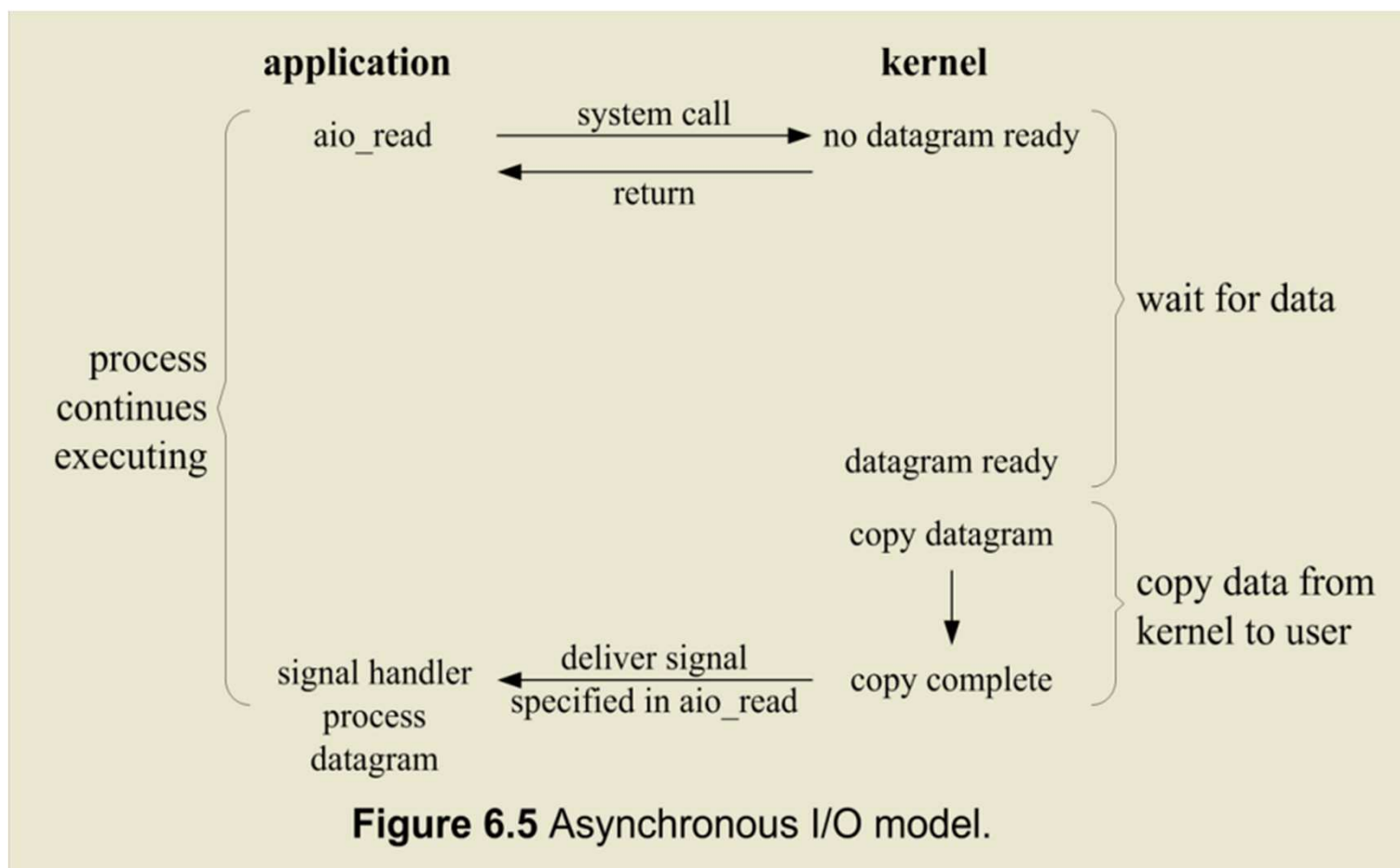
---

### Signal-Driven I/O model

- Interrupt와 유사, Application 측에서 operating system(=kernel)에게 허가요청과 비슷한 신호를 보냄, 이후 operating system이 Application 측에 준비됐을 때 알려줌. 해당 작업이 진행되기 시작할 때 Application 측은 Block됨
  - 이러한 신호가 중복되면 뒤에 온 신호는 무시 됨
  - TCP에서는 이러한 신호가 중복되는 경우가 많아서 UDP에서 주로 사용함 <- 이유를 잘모르겠습니다ㅠ
-

# 6.2 I/O Models

## Asynchronous I/O



## 6.2 I/O Models

---

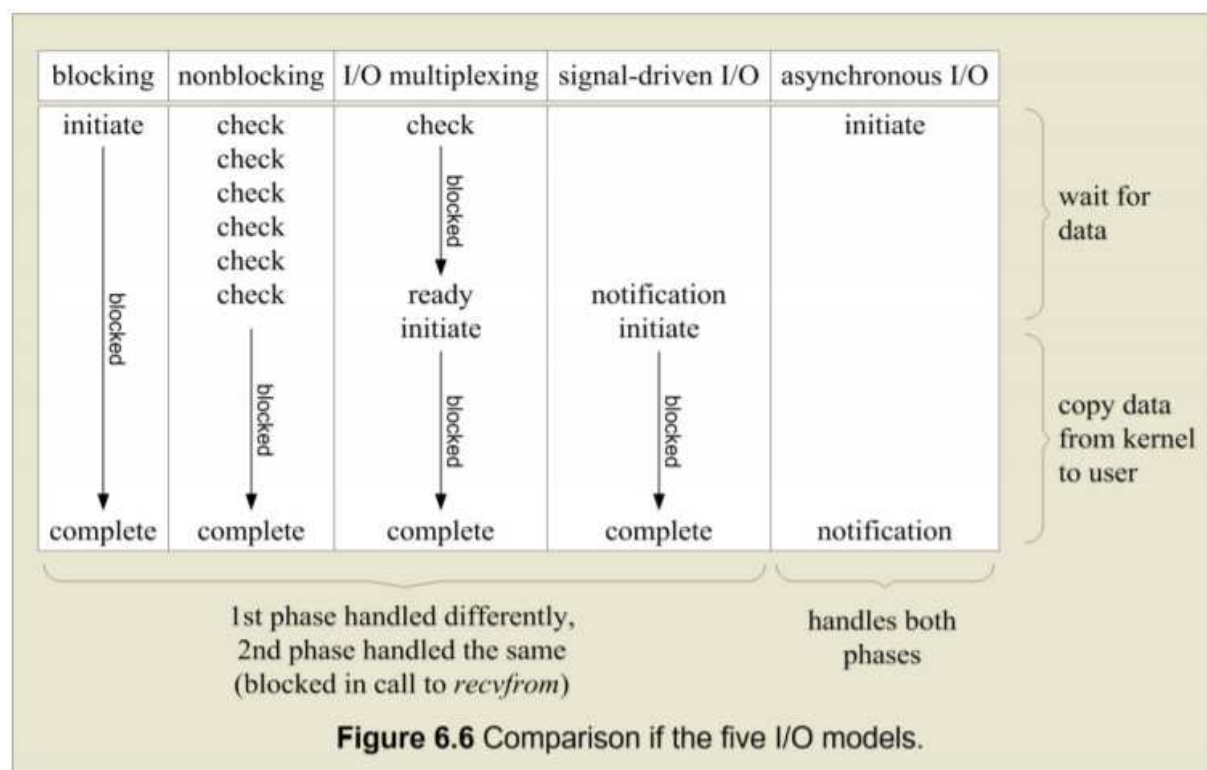
### Asynchronous I/O

- Asynchronous I/O Model은 커널에게 I/O작업을 맡기면 커널의 작업 진행사항에 대해서 프로세스가 인지할 필요가 없는 I/O Model이다.  
(즉, Application 측이 Block되지않는다.)
  - Notify의 적극적인 주체는 커널이 되며, 유저 프로세스는 수동적인 입장에서 본인 할 일을 하다가 커널의 Notify가 오면 그때 I/O 처리를 하게 된다.
-



# 6.2 I/O Models

## Comparison of I/O Models



Q2: 클라우드 서비스를 위해선 무슨 I/O Model(s)을 사용하는게 좋을까?

Q3: signal-driven I/O, asynchronous I/O 기반의 Server는 없나?

## 6.3 Select Function

---

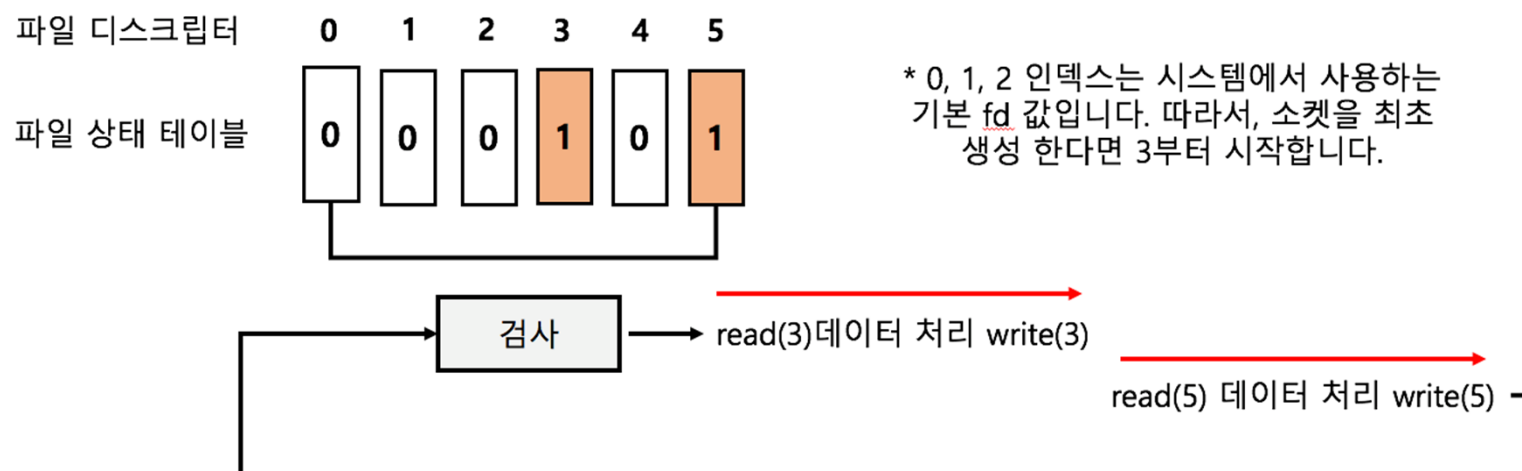
### Select 원리

- Select는 싱글 스레드로 다중 I/O를 처리하는 Multiplexing I/O Model 구현을 위한 대표적인 방법
  - 해당 파일 디스크립터가 I/O를 할 준비가 되었는지 알 수 있다면  
그 파일 디스크립터가 할당 받은 커널 Buffer에 데이터를 복사해주기만 하면 된다.
  - Select는 많은 파일 디스크립터들을 한꺼번에 관찰하는 **FD\_SET** 구조체를 사용하여 빠르고 간편하게 유저에게 파일 디스크립터의 상황을 알려준다.
-

# 6.3 Select Function

## FD\_SET

- FD\_SET은 하나의 FD 상태를 하나의 비트로 표현한 구조체
- 파일 디스크립터(소켓)의 번호를 Index로 사용



## 6.3 Select Function

---

### FD\_SET 관리 매크로

```
FD_ZERO(fd_set* set);           //fdset을 초기화
FD_SET(int fd, fd_set* set);    //fd를 set에 등록
FD_CLR(int fd, fd_set* set);    //fd를 set에서 삭제
FD_ISSET(int fd, fd_set* set); //fd가 준비되었는지 확인
```

## 6.3 Select Function

### Select ()

```
int select(int nfds, fd_set readfds, fd_set writefds, fd_set exceptfds, struct timeval timeout)
```

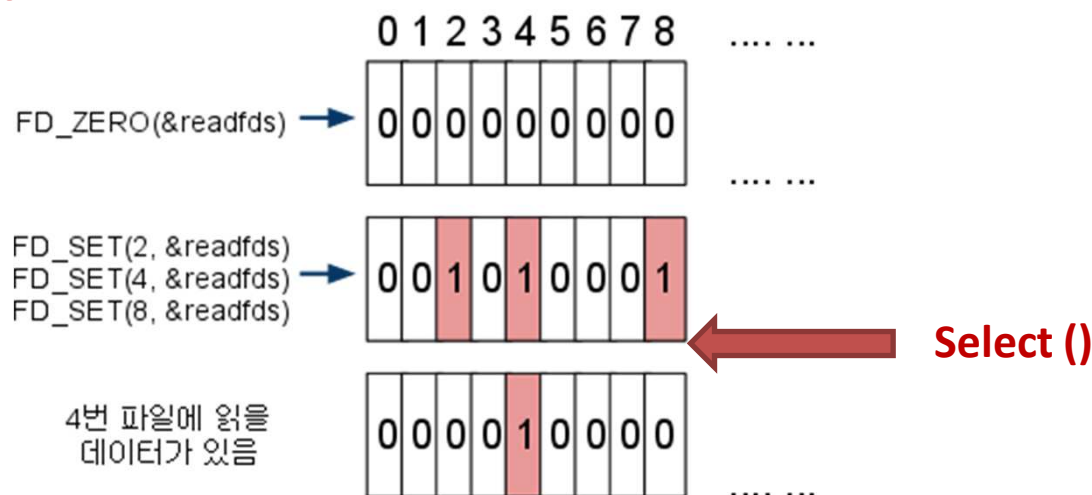
- `nfds`: 검사 대상이 되는 파일 디스크립터의 수
- `readfds`: 읽기 이벤트를 검사할 파일 디스크립터의 목록
- `writefds`: 쓰기 이벤트를 검사할 파일 디스크립터의 목록
- `exceptfds`: 예외 이벤트를 검사할 파일 디스크립터의 목록
- `timeout`: 이벤트를 기다릴 시간 제한
- 반환 값: 이벤트가 발생한 파일의 갯수 ← `fd_set` 배열 중 1의 개수

NULL이면 하나라도 1될 때까지 Block

NULL이 아니면 시간 제한만큼 기다리고, 그 시간 동안 발생한 이벤트의 개수 반환

# 6.3 Select Function

## Select (): 주의할 점



**Select 함수를 호출 시 fd\_set 배열의 바뀌지않은 비트들이 모두 초기화된다!**

-> Socket 통신에 활용하려면 호출 전 반드시 저장해야 한다.

-> Select 함수를 사용한 I/O Multiplexing 방식의 단점

Q4: Select 함수 호출 시, 비트가 1인 소켓들만 검사하는가?

## 6.3 Select Function

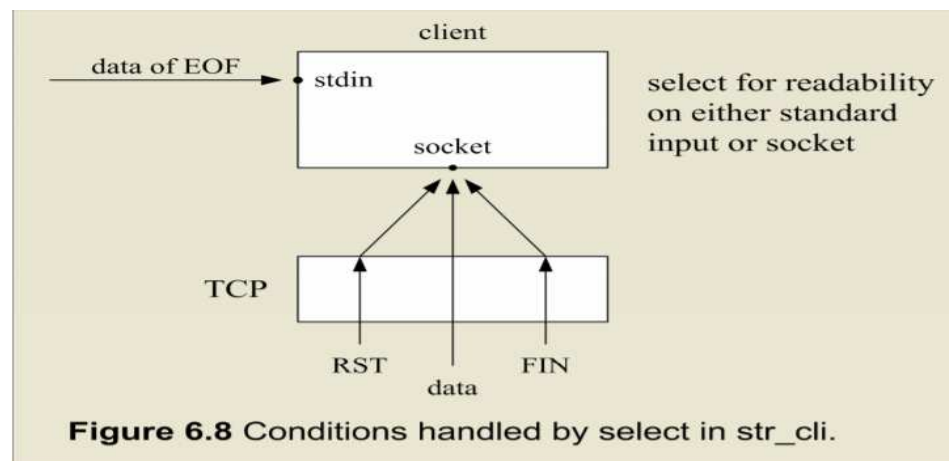
**Select (): 호출 후 1을 반환하는 Condition**

Condition	Readable?	Writable?	Exception?
Data to read	•		
Read half of the connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

**Figure 6.7** Summary of conditions that cause a socket to be ready for *select*.

Q5: Half of the connection closed 란?

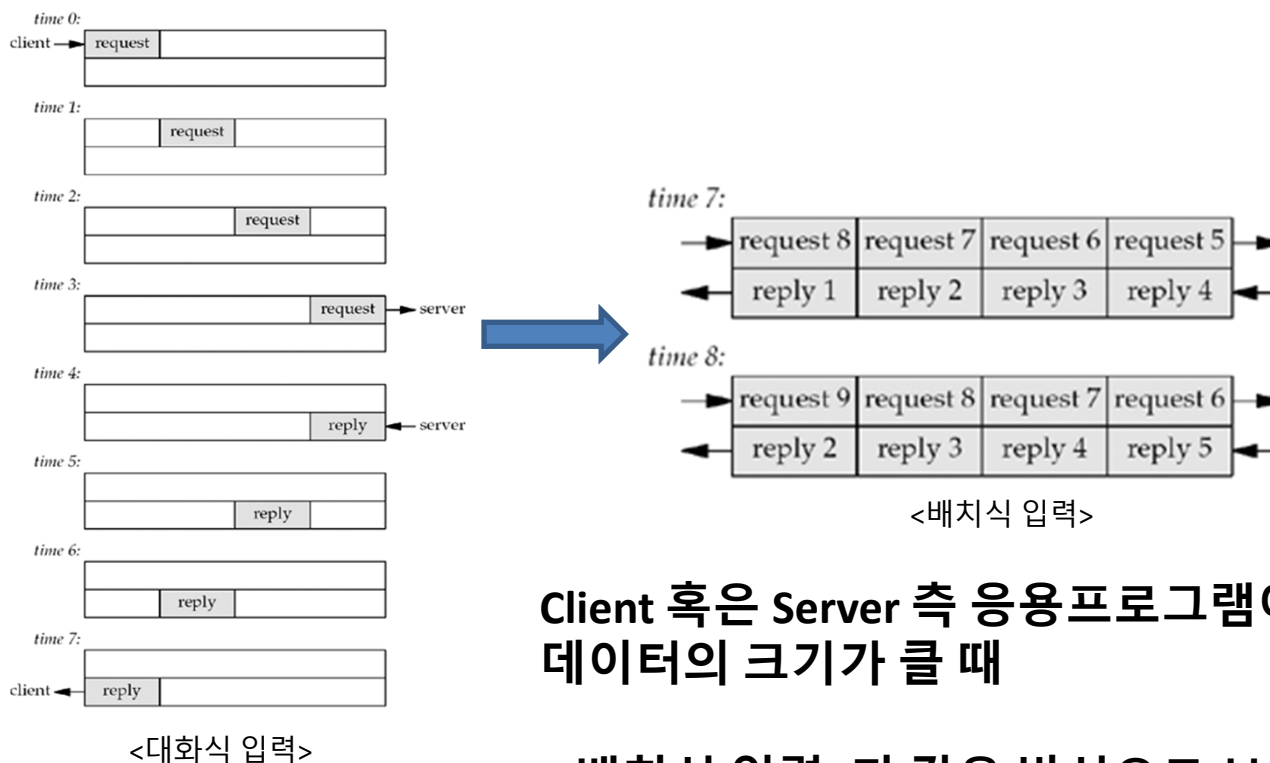
# 6.4 str\_cli Function



1. 만약 peer TCP가 data를 보낸다면, socket이 읽을 수 있어야 하고 read는 0보다 큰 값을 return한다(예를 들면, data의 byte 수).
2. 만약 peer TCP가 FIN을 보낸다면(peer process는 종료), socket이 읽을 수 있어야 하고 read는 0을 return한다(EOF).
3. 만약 peer TCP가 RST를 보낸다면(peer host는 충돌되어 재부팅), socket이 읽을 수 있어야 하고 read는 -1을 return하고, `errno`는 명확한 error 코드를 포함한다.



# 6.5 Batch Input and Buffering



Client 혹은 Server 측 응용프로그램이 원하는 단위 데이터의 크기가 클 때

<배치식 입력>과 같은 방식으로 보낼 필요가 있음

Q6: 어떤 경우 대화식 입력 방식을 배치식 입력 방식으로 바꾸어야 할까?

# 6.5 Batch Input and Buffering

## Socket option

```
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname, void
*optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname, const void
*optval, socklen_t optlen);
Both return: 0 if OK, -1 on error
```

**getsockopt: Socket의 옵션을 가져와 확인하는 함수**

**setsockopt: Socket의 옵션을 설정하는 함수**

sock - 옵션 확인을 위한 소켓의 파일 디스크립터 전달.

level - 확인(getsockopt) 또는 변경(setsockopt)할 옵션의 프로토콜 레벨 전달

optname - 확인 또는 변경할 옵션의 이름 전달

optval - 확인결과와 저장을 위한 버퍼의 주소 값 전달

optlen - 주의: getsockopt의 경우 포인터 값을 매개변수로 받고, setsockopt의 경우 타입이 socklen\_t\*가 아닌 socklen\_t이다.

(getsockopt) 네 번째 매개변수 optval로 전달된 주소 값의 버퍼 크기를 담고 있는 변수의 주소 값 전달, 함수 호출이 완료되면 이 변수에는 네 번째 인자를 통해 반환된 옵션 정보의 크기가 바이트 단위로 계산되어 저장된다.

Protocol Level	옵션명	get	set	설명
SOL_SOCKET (일반)	SO_SNDBUF	OK	OK	소켓 송신 버퍼의 크기
	SO_RCVBUF	OK	OK	소켓 입력 버퍼의 크기
	SO_REUSEADDR	OK	OK	이미 사용중인 주소나 포트에 대해서도 바인드 허용
	SO_LINGER	OK	OK	소켓을 닫을 때 남은 데이터의 처리 규칙 지정
	SO_KEEPALIVE	OK	OK	keepalive 메시지 활성화 (프로토콜에 구현되어 있을 경우)
	SO_BROADCAST	OK	OK	브로드캐스트 허용
	SO_DONTROUTE	OK	OK	라우팅하지 않고 직접 인터페이스로 전송
	SO_OOBINLINE	OK	OK	OOB 데이터를 설정할 때 일반 입력 큐에서 데이터를 읽을 수 있게 합니다.
	SO_SNDLOWAT	OK	OK	전송할 최소 바이트 수
	SO_RCVLOWAT	OK	OK	recv()가 반환될 수 있는 최소 바이트 수
IPPROTO_IP (IP레벨-IPv4)	SO_ERROR	OK	X	에러상태를 반환하고 클리어됩니다.
	SO_TYPE	OK	X	소켓의 타입. (ex: SOCK_STREAM)
	IP_TOS	OK	OK	패킷의 우선순위 설정 (Type Of Service)
	IP_TTL	OK	OK	유니캐스트 시 TTL 값 설정
	IP_MULTICAST_TTL	OK	OK	멀티캐스트 시 TTL 값 설정
	IP_MULTICAST_LOOP	OK	OK	멀티캐스트 소켓이 자신이 보낸 패킷을 수신도록 합니다.
IPPROTO_TCP (TCP레벨)	IP_MULTICAST_IF	OK	OK	멀티캐스트 패킷을 보낼 인터페이스 설정
	TCP_KEEPAIVE	OK	OK	keep alive 시간 간격 지정
	TCP_NODELAY	OK	OK	데이터를 합치기 위해 Nagle's 알고리즘 사용시 지연을 허용 안함
	TCP_MAXSEG	OK	OK	TCP 최대 세그먼트 지정

# 6.6 Shutdown function

## Shutdown function(for half close)

```
#include <sys/socket.h>
int shutdown(int sock, int howto);
성공시 0, 실패시 -1 반환
- sock : 종료할 소켓의 파일 디스크립터 전달
- howto : 종료방법에 대한 정보 전달
```

SHUT_RD	입력 스트림 종료
SHUT_WR	출력 스트림 종료
SHUT_RDWR	입출력 스트림 종료

A와 B가 서로 TCP 통신 시, A 쪽에서 일방적으로 close()로 연결을 종료하면 B는 A에게 데이터를 보낼 것이 남아있음에도 보낼 수 없다.

ex) FIN이 손실될 경우

이를 해결하기 위한 함수가 Shutdown()

데이터전송이 끝났음을 알리는 EOF를 날릴 때, 출력 스트림만 닫아서 EOF를 보내는 방식

Q7: TIME-WAIT timer를 사용하여 close하는 것과 차이점이 무엇인가?

# 6.6 Shutdown function

## Shutdown function(for half close)

```
#include <sys/socket.h>
int shutdown(int sock, int howto);
성공시 0, 실패시 -1 반환
- sock : 종료할 소켓의 파일 디스크립터 전달
- howto : 종료방법에 대한 정보 전달
```

SHUT_RD	입력 스트림 종료
SHUT_WR	출력 스트림 종료
SHUT_RDWR	입출력 스트림 종료

A와 B가 서로 TCP 통신 시, A 쪽에서 일방적으로 close()로 연결을 종료하면 B는 A에게 데이터를 보낼 것이 남아있음에도 보낼 수 없다.

ex) FIN이 손실될 경우

이를 해결하기 위한 함수가 Shutdown()

데이터전송이 끝났음을 알리는 EOF를 날릴 때, 출력 스트림만 닫아서 EOF를 보내는 방식

Q7: TIME-WAIT timer를 사용하여 close하는 것과 차이점이 무엇인가?

# 6.8 TCP Echo Server

## TCP Echo Server Example(1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/select.h>

#define BUF_SIZE 100
void error_handling(char *buf);

int main(int argc, char *argv[]) {
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_addr, clnt_addr;
    struct timeval timeout;
    // 파일 상태 테이블 선언
    fd_set reads, cpy_reads; // 파일 디스크립터들도 띄워진 비트 배열 구조체

    socklen_t addr_sz;
    int fd_max, str_len, fd_num, i;
    char buf[BUF_SIZE];
    if (argc != 2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(atoi(argv[1]));

    if (bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) ==
        -1)
        error_handling("bind() error");
    if (listen(serv_sock, 5) == -1)
        error_handling("listen() error");
```

```
FD_ZERO(&reads); // fd_set 테이블을 초기화한다.
FD_SET(serv_sock, &reads); // 서버 소켓 (리스팅 소켓)의 이벤트 검사를 위해
// fd_set 테이블에 추가한다.
fd_max = serv_sock;

while(1) {
    cpy_reads = reads;
    timeout.tv_sec = 5;
    timeout.tv_usec = 5000;

    // result
    // -1: 오류 발생
    // 0: 타임 아웃
    // 1 이상 : 등록된 파일 디스크립터에 해당 이벤트가 발생하면 이벤트가 발생한 파일
    // 디스크립터의 수를 반환한다.
    if ((fd_num = select(fd_max+1, &cpy_reads, 0, 0, &timeout)) == -1)
        break; // 오류 발생

    if (fd_num == 0)
        continue; // 이벤트 발생 X 시 fd_set 테이블 검사 안함
```

Q8: sockaddr\_in 구조체 타입으로 serv\_addr를 선언했는데  
왜 bind 함수의 2번째 인자로 사용할 때 sockaddr 구조체 타입으로 형 변환 하는가?

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 6.8 TCP Echo Server

## TCP Echo Server Example

```
for (i=0; i<fd_max+1; i++) {
    if (FD_ISSET(i, &cpy_reads)) { // fd_set 테이블을 검사한다.
        // 서버 소켓(리스닝 소켓)에 이벤트(연결 요청) 발생
        if (i == serv_sock) { // connection request!
            adr_sz = sizeof(clnt_adr);
            clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_adr,
                                &adr_sz);
            FD_SET(clnt_sock, &reads); // fd_set 테이블에 클라이언트 소켓
            // 디스크립터를 추가한다.
            if (fd_max < clnt_sock)
                fd_max = clnt_sock;
            printf("connected client: %d \n", clnt_sock);
        }
        // 클라이언트와 연결된 소켓에 이벤트 발생
        else { // read message!
            str_len = read(i, buf, BUF_SIZE);
            if (str_len == 0) { // close request!
                FD_CLR(i, &reads); // fd_set 테이블에서 파일 디스크립터를
                // 삭제한다.
                close(i);
                printf("closed client: %d \n", i);
            } else {
                write(i, buf, str_len); // echo!
            }
        }
    }
}
} // ← while(1)
close(serv_sock);
return 0;
} // ← main

void error_handling(char *buf) {
    fputs(buf, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 6.8 TCP Echo Server

## TCP Echo Server Example

```
2. kimjongmin: _zsh_tmux_plugin_run attach (tmux)

~/work/socket_test
> ./echo_select_server 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000
```

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

## 6.8 TCP Echo Server

### Select()를 사용한 Multiplexing Server의 장단점

- 장점

- 단일 프로세스(스레드)에서 여러 파일의 입출력 처리가 가능하다.
- 지원하는 OS가 많아 이식성이 좋다. (POSIX 표준)

- 단점

- 커널에 의해서 완성되는 기능이 아닌, 순수하게 함수에 의해 완성되는 기능이다.
- Select 함수의 return 값이 이벤트가 발생한 파일 디스크립터의 개수이기 때문에 fd\_set 항상 테이블 전체를 검사해야 한다.
- Select 호출 때마다 데이터를 복사해야 한다.

Q9: 순수하게 함수에 의해 완성되는 기능인 점이 어째서 단점인가?



## 6.9 pselect Function

---

### pselect()

```
#include <sys/select.h>
```

원형:

```
int pselect(int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  
const struct timespec *timeout,  
const sigset_t *sigmask);
```

- timeval 구조체를 활용하는 select()에 비해  
timespec 구조체를 활용함으로써 시간 정밀도가 높다. (마이크로초vs나노초)
- 만료시간 인자가 const로 선언되어 있기 때문에 pselect() 호출에 의해 값이 변경되지 않는다.
- sigmask를 통해서 신호 마스크를 지정할 수 있다. ( Ctrl+C 등의 시그널을 통하지 않게 할 수 있다.

---

Q10: select()는 호출할 때 마다 값이 어떻게 바뀌는가?

# 6.10 poll Function

## poll()

**int poll(struct pollfd \*ufds, unsigned int nfds, int timeout);**

```
struct pollfd
{
    int fd;          // 관심있어하는 파일지시자
    short events;    // 발생한 이벤트
    short revents;   // 돌려받은 이벤트
};
```

```
#define POLLIN      0x0001 // 읽을 데이터가 있다.
#define POLLPRI     0x0002 // 긴급한 읽을 데이터가 있다.
#define POLLOUT     0x0004 // 쓰기가 봉쇄(block)가 아니다.
#define POLLERR     0x0008 // 에러발생
#define POLLHUP     0x0010 // 연결이 끊겼음
#define POLLNVAL    0x0020 // 파일지시자가 열리지 않은것같은
                        // Invalid request (잘못된 요청)
```

**nfd**s: pollfd의 배열의 크기(=조사하고자 하는 최대 파일 디스크립터 개수)

**timeout**: select의 time과 같은 역할

[https://www.joinc.co.kr/w/Site/Network\\_Programing/Documents/Poll](https://www.joinc.co.kr/w/Site/Network_Programing/Documents/Poll)

# 6.10 poll Function

---

## poll()

1. pollfd 구조체에 모든 fd를 -1로 초기화 //fd 가 -1이면 할당된 file descriptor 존재X
2. listen 이후 pollfd 구조체에 listen socket을 할당
3. Listen 소켓이 되돌려준 이벤트가 POLLIN이라면 Accept
4. 현재 파일 디스크립터의 총 개수 만큼 루프를 돌면서 POLLIN(읽을 데이터 존재)과 POLLERR(에러)의 발생을 찾고, 데이터를 받아들임
5. 2~4 반복

---

[https://www.joinc.co.kr/w/Site/Network\\_Programing/Documents/Poll](https://www.joinc.co.kr/w/Site/Network_Programing/Documents/Poll)

# 6.10 poll Function

## poll()의 Select 대비 장단점

### • 장점

- select와 같이 단일 프로세스(스레드)에서 여러 파일의 입출력 처리가 가능하다
- select와 다르게 pollfd를 계속 복사할 필요가 없다. (본인 생각에)
- select 방식처럼 표준 입출력 에러를 따로 감시할 필요가 없다.
- Select는 timeval이라는 구조체를 사용해 타임아웃 값을 세팅하지만, poll은 별다른 구조체 없이 타임아웃 기능을 지원한다.

### • 단점

- 일부 unix 시스템에서는 poll을 지원하지 않습니다.

[https://www.joinc.co.kr/w/Site/Network\\_Programing/Documents/Poll](https://www.joinc.co.kr/w/Site/Network_Programing/Documents/Poll)

# 6.10 poll Function

## select()와 poll()의 문제점

- select 함수의 return 값이 이벤트가 발생한 파일 디스크립터의 개수이기 때문에 fd\_set 항상 테이블 전체를 검사해야한다.
- 위 사항은 poll 함수도 마찬가지이다.
- select 함수는 호출 때마다 데이터를 복사해야 한다.

시간  
자원  
낭비

Solution?



epoll()

[https://www.joinc.co.kr/w/Site/Network\\_Programing/Documents/Poll](https://www.joinc.co.kr/w/Site/Network_Programing/Documents/Poll)

# 6.10 poll Function

---

## epoll()

- 파일 디스크립트(fd) 수가 무제한
- 파일 디스크립트를 커널에서 관리하므로 상태가 바뀐 것 만  
을 직접 통지 -> fd\_set 복사가 필요 없다
- 성능 또한 select < poll < epoll
- 커널 레벨 멀티플렉싱 지원. 커널에 관찰대상에 대한 정보를  
한 번만 전달하고, 관찰대상의 범위, 또는 내용에 변경이 있을  
때만 변경 사항을 알려줌.

---

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 6.10 poll Function

---

## epoll()

- **epoll\_create**  
epoll 파일 디스크립터 저장소 생성
- **epoll\_ctl**  
저장소에 파일 디스크립터 등록 및 삭제
- **epoll\_wait**  
select 함수와 마찬가지로 파일 디스크립터의 변화를 대기한다.

---

<https://jacking75.github.io/choiheungbae>

# 6.10 poll Function

## epoll()

```
int epoll_create(int size); //size는 epoll_fd의 크기정보를 전달한다.  
// 반환 값 : 실패 시 -1, 일반적으로 epoll_fd의 값을 리턴  
  
int epoll_ctl(int epoll_fd,          // epoll_fd  
              int operate_enum,      // 어떤 변경을 할지 결정하는 enum값  
              int enroll_fd,         // 등록할 fd  
              struct epoll_event* event // 관찰 대상의 관찰 이벤트 유형  
              );  
// 반환 값 : 실패 시 -1, 성공시 0  
  
int epoll_wait(int epoll_fd,          // epoll_fd  
               struct epoll_event* event, // event 버퍼의 주소  
               int maxevents,          // 버퍼에 들어갈 수 있는 구조체 최대 개수  
               int timeout              // select의 timeout과 동일 단위는 1/1000  
               );  
// 성공시 이벤트 발생한 파일 디스크립터 개수 반환, 실패시 -1 반환
```

<https://jacking75.github.io/choiheungbae>



# 6.10 poll Function

---

## epoll\_create

```
#include <sys/epoll.h>

int epoll_create(int size);
성공시 epoll 파일 디스크립터, 실패시 -1 반환

size : epoll 인스턴스의 크기 정보
```

---

<https://jacking75.github.io/choiheungbae/%EB%AC%B8%EC%84%9C/epoll%EC%9D%84%20%EC%82%AC%EC%9A%A9%ED%95%9C%20%EB%B9%84%EB%8F%99%EA%B8%B0%20%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D.pdf>

2020-07-20

Network Programming 2020 seminar

# 6.10 poll Function

---

## epoll\_ctl

```
#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event* event);
성공시 0, 실패시 -1 반환

epfd : 관찰대상을 등록할 epoll 인스턴스의 파일 디스크립터
op : 관찰대상의 추가, 삭제 또는 변경여부 지정
fd : 등록할 관찰대상의 파일 디스크립터
event : 관찰대상의 관찰 이벤트 유형
```

---

<https://jacking75.github.io/choiheungbae>

# 6.10 poll Function

---

## epoll\_wait

```
#include <sys/epoll.h>
```

```
int epoll_wait(int epfd, struct epoll_event* events, int maxevents, int timeout);
```

성공시 이벤트가 발생한 파일 디스크립터의 수, 실패시 -1 반환

epfd : 이벤트 발생의 관찰영역인 epoll 인스턴스의 파일 디스크립터

events : 이벤트가 발생한 파일 디스크립터가 채워질 버퍼의 주소 값

maxevents : 두 번째 인자로 전달된 주소 값의 버퍼에 등록 가능한 최대 이벤트 수

timeout : 1/1000초 단위의 대기시간, -1 전달 시, 이벤트가 발생 할 때까지 무한 대기

---

<https://jacking75.github.io/choiheungbae>

## 6.10 poll Function

---

- **EPOLLIN** : 수신할 데이터가 존재하는 상황 (EPOLL\_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- **EPOLLOUT** : 출력데이터가 버퍼에서 밀려서 밀려 데이터 전송할 수 없는 상황 (EPOLL\_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- **EPOLLPRI** : OOB 데이터가 수신된 상황 (EPOLL\_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- **EPOLLRDHUP** : 연결이 종료되거나 half-close 가 진행된 상황. 이는 엡지 트리거 방식에서 유용하게 사용할 수 있다. 상대편 소켓 닫으면 (EPOLL\_CTL의 입력, WAIT 의 출력에 모두 사용됨)
- **EPO LLERR** : 에러가 발생한 상황 (EPOLL\_WAIT의 출력으로만 사용됨)
- **EPOLLHUP** : 장애발생 (hangup) (EPOLL\_WAIT의 출력으로만 사용됨)
- **EPOLLET** : 이벤트의 감지를 엡지 트리거 방식으로 동작 (EPOLL\_CTL의 입력에만 사용됨)
- **EPOLLONESHOT** : 이벤트가 한번 감지되면, 해당 파일 디스크립터에서는 더 이상 이벤트를 발생시키지 않는다. 따라서 **epoll\_ctl** 함수의 두번째 인자로 **EPOLL\_CTL\_MOD**을 전달해서 이벤트를 재설정해야 한다. (EPOLL\_CTL의 입력에만 사용됨)

---

<https://jacking75.github.io/choiheungbae>

# 6.11 TCP Echo Server

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
// 리눅스에서만 사용 가능
#include <sys/epoll.h>

#define BUF_SIZE 100
#define EPOLL_SIZE 50
void error_handling(char *buf);

int main(int argc, char *argv[]) {
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_addr, clnt_addr;
    socklen_t adr_sz;
    int str_len, i;
    char buf[BUF_SIZE];

    struct epoll_event *ep_events;
    struct epoll_event event;
    int epfd, event_cnt;

    if (argc != 2) {
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }

    serv_sock = socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(atoi(argv[1]));
    if (bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) ==
-1)
        error_handling("bind() error");
    if (listen(serv_sock, 5) == -1)
        error_handling("listen() error");

    // 커널이 관리하는 epoll 인스턴스와 불리는 파일 디스크립터의 자장소 생성
    // 성공 시 epoll 파일 디스크립터, 실패 시 -1 반환
    epfd = epoll_create(EPOLL_SIZE);
    ep_events = malloc(sizeof(struct epoll_event)*EPOLL_SIZE);

    event.events = EPOLLIN;
    event.data.fd = serv_sock;
    // 파일 디스크립터(serv_sock)를 epoll 인스턴스에 등록한다. (관할대상의 관찰 이벤트
    // 유형은 EPOLLIN)
    epoll_ctl(epfd, EPOLL_CTL_ADD, serv_sock, &event);
```

```
while(1) {
    // 성공 시 이벤트가 발생한 파일 디스크립터의 수, 실패 시 -1 반환
    // 두 번째 인자로 전달된 주소의 메모리 공간에 이벤트 발생한 파일 디스크립터에
    // 대한 정보가 들어있다.
    event_cnt = epoll_wait(epfd, ep_events, EPOLL_SIZE, -1);
    if (event_cnt == -1) {
        puts("epoll_wait() error");
        break;
    }

    for (i=0; i<event_cnt; i++) {
        if (ep_events[i].data.fd == serv_sock) {
            // 서버에서 이벤트 발생?
            adr_sz = sizeof(clnt_addr);
            clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr,
&adr_sz);

            event.events = EPOLLIN;
            event.data.fd = clnt_sock;
            // 파일 디스크립터(clnt_sock)를 epoll 인스턴스에 등록한다. (관할대상의
            // 관찰 이벤트 유형은 EPOLLIN)
            epoll_ctl(epfd, EPOLL_CTL_ADD, clnt_sock, &event);
            printf("connected client: %d\n", clnt_sock);
        } else {
            str_len = read(ep_events[i].data.fd, buf, BUF_SIZE);
            if (str_len == 0) { // close request!
                epoll_ctl(epfd, EPOLL_CTL_DEL, ep_events[i].data.fd,
NULL);

                close(ep_events[i].data.fd);
                printf("closed client: %d\n", ep_events[i].data.fd);
            } else {
                write(ep_events[i].data.fd, buf, str_len); // echo!
            }
        }
    }

    close(serv_sock);
    close(epfd);
    return 0;
}

void error_handling(char *buf) {
    fputs(buf, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

## 6.11 TCP Echo Server

---

```
2. kimjongmin: _zsh_tmux_plugin_run attach (tmux)

~/work/socket_test
> ./echo_epoll_server 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000

~/work/socket_test
> ./echo_client 127.0.0.1 3000
```

---

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

# 6.11 TCP Echo Server

---

## epoll()의 장단점

- 장점
  - 상태변화의 확인을 위한, 전체 파일 디스크립터를 대상으로 하는 반복문이 필요 없다.
  - select 함수에 대응하는 epoll\_wait 함수호출 시, 커널에서 상태정보를 유지하기 때문에 관찰대상의 정보를 매번 전달할 필요가 없다.
- 단점
  - 리눅스의 select 기반 서버를 윈도우의 select 기반 서버로 변경하는 것은 간단하나, 리눅스의 epoll 기반의 서버를 윈도우의 IOCP 기반으로 변경하는 것은 select를 이용하는 것보다 번거롭다.

---

<https://jongmin92.github.io/2019/02/28/Java/java-with-non-blocking-io/>

## 6.12 Summary

---

- 다중 접속 서버의 종류에는 멀티 프로세스 방식, 멀티 스레드 방식, I/O 멀티플렉싱 방식이 있다.
  - 멀티 프로세스, 멀티 스레드 방식은 성능 저하 혹은 안정성 문제를 동반한다.
  - 이를 해결하기 위해 나온 것이 멀티 플렉싱 방식의 서버를 도입한다.
  - 멀티 플렉싱 방식의 서버에는 `select()`를 활용한 서버, `poll()`을 활용한 서버, `epoll()`을 활용한 서버가 있다.
  - `select()` 방식은 비교적 사용이 간편하고, window 운영체제로의 이식이 간편하나 매 번 `fd_set`을 복사 해야 함과 `fd_set`에 등록된 모든 소켓들을 매 번 검사해야 함에 따른 성능 저하가 야기된다.
-



## 6.12 Summary

---

- 이러한 성능 저하를 줄이기 위해 나온 것이 poll() 방식이다. 이는 select() 방식의 매 번 fd\_set을 복사해야 한다는 점을 개선하여, 성능을 조금 더 개선하였다.
  - 하지만 poll() 또한 pollfd를 모두 검사해야 한다는 점은 변함없다.
  - 이 점을 개선하기 위해 나온 것이 epoll()이다.
  - epoll()은 변화가 발생한 것들만 검사함을 통해 성능 개선을 이뤄냈다.
  - 리눅스 운영체제에서 이러한 서버를 구축할 계획이라면, epoll()이 최고인 것 같다. 성능이 그다지 중요하지않고, 다른 운영체제와 호환성이 중요하다면 select()를 사용하는 것도 좋아보인다.
-