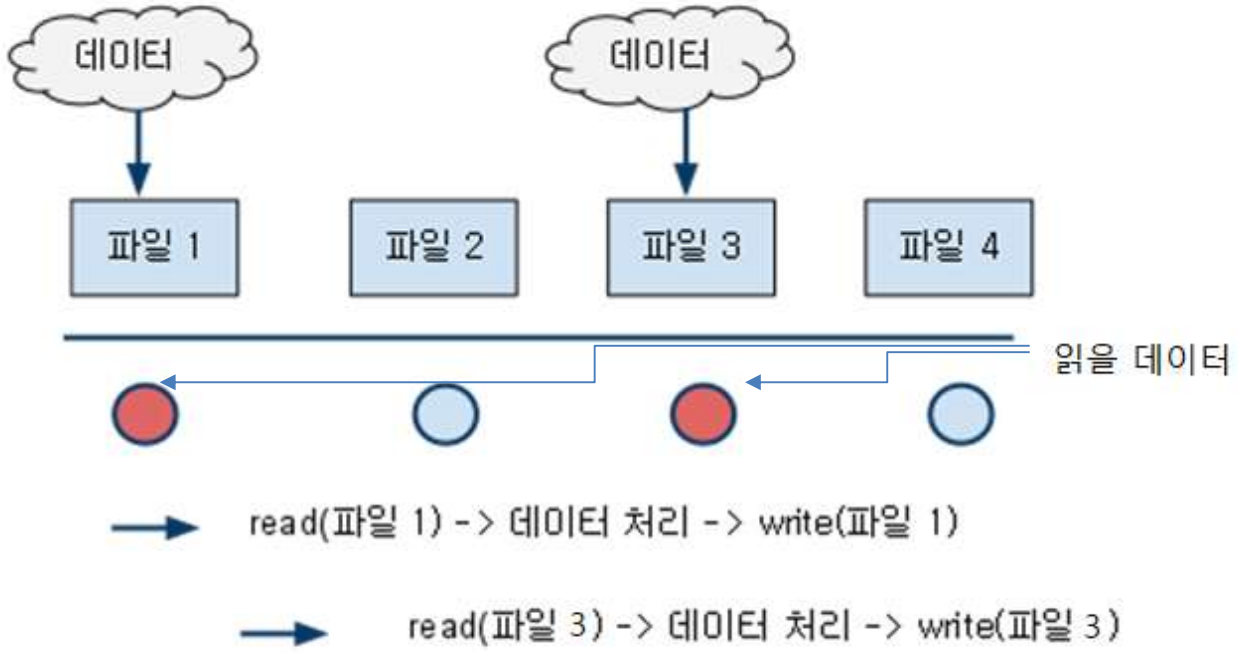
입출력 다중화(I/O Multiplexing)

뇌를 자극하는 TCP/IP 소켓 프로그래밍

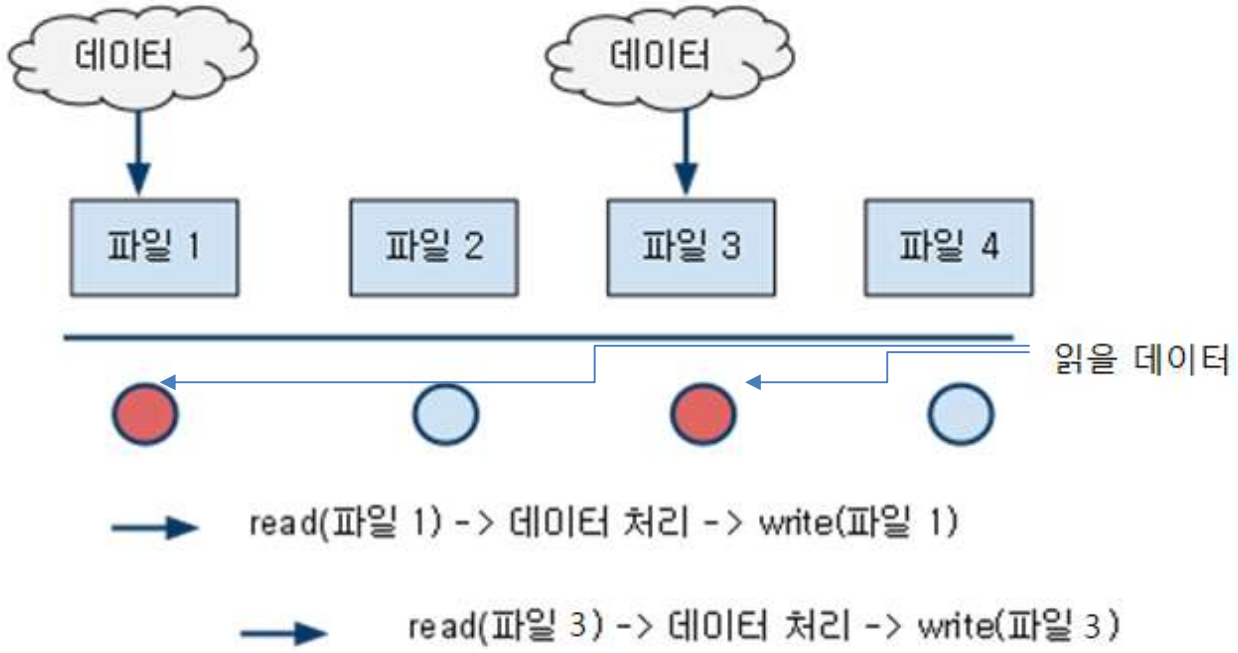
하나의 프로세스에서 여러개의 입출력을 다룰 수 없을까 ?

- 멀티 프로세스 방법은 많은 비용을 필요로 한다.
- 멀티 스레드 방법도 많은 비용이 소모된다.
- 입출력을 사건(이벤트로)다룬다면, 어떨까 ?



하나의 프로세스에서 여러개의 입출력을 다룰 수 없을까 ?

- 관리할 파일의 그룹을 만들고
- 그룹의 파일에 입출력 이벤트가 있는지를 확인.
- 입출력 이벤트가 발생한 파일의 목록을 일괄 처리



입출력 다중화 (I/O Multiplexing)

- 단일 프로세스, 단일 스레드에서 입력과 출력을 다룬다.
  - 프로세스 혹은 스레드를 만들 필요가 없다.
  - 여러 개의 입력과 출력을 관리하는 기술.

    •한 프로세스/스레드로 여러 소켓으로부터의 입출력을 처리하기 위하여

    blocking당할 수 있는 입출력(read(), accept())는

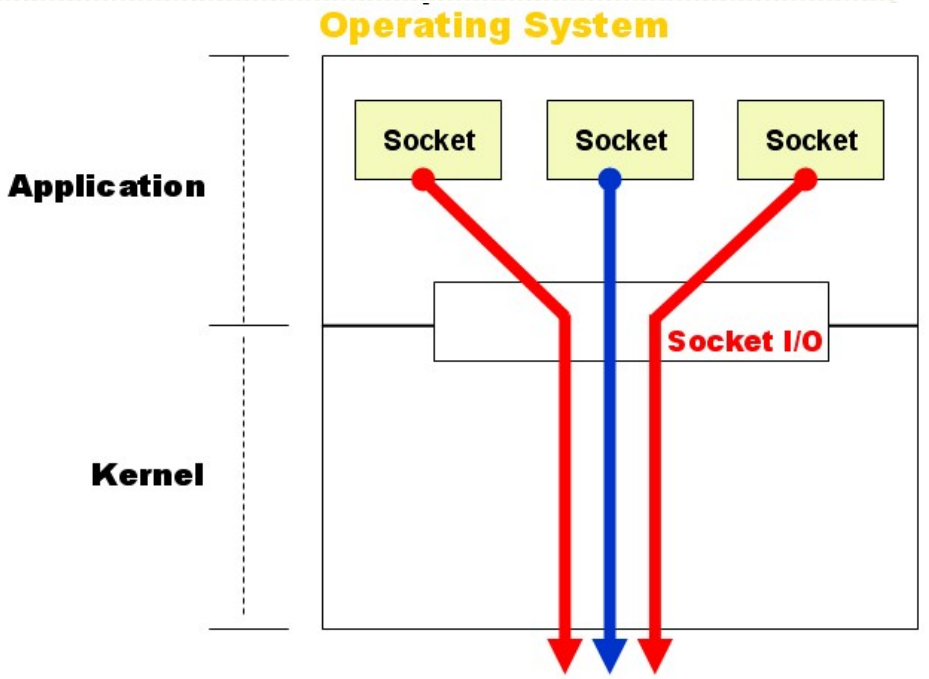
    blocking 당할 상황에서는 호출하지 않고,

    항상 입출력이 가능할 상황에서만 호출함
  - 여러 입출력의 동시 처리기술은 아님. 하나의 입력에 대한 처리가 끝날 때까지 다른 입력은 대기해야 함.
- 다른 많은 기술들이 입출력 다중화에 기반한다.

입출력 다중화 (I/O Multiplexing)

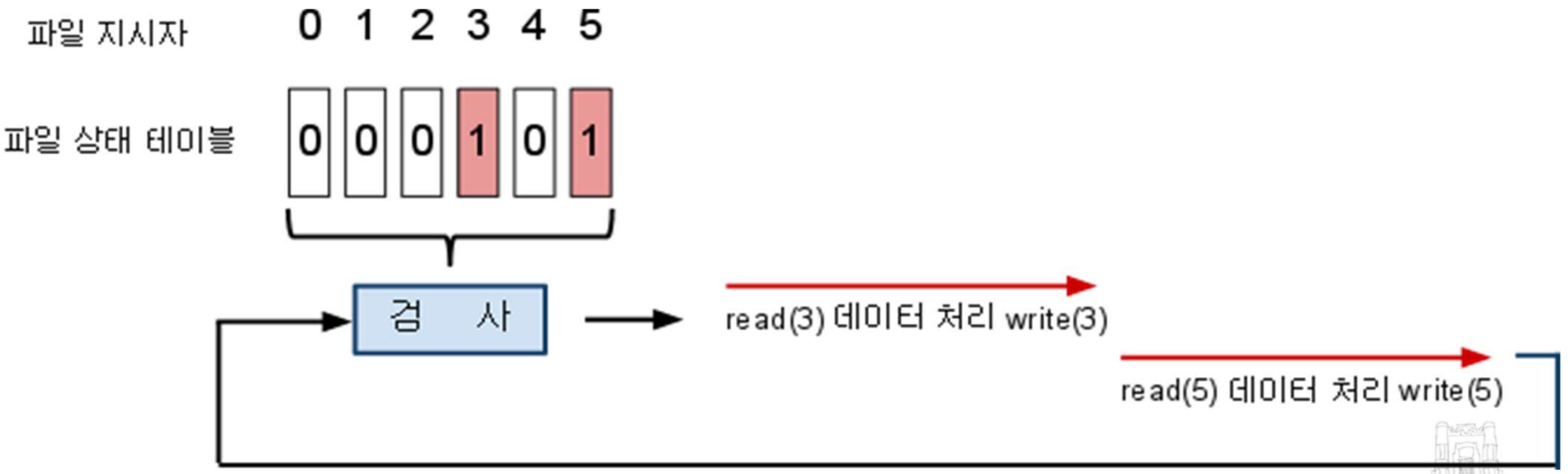
- **Multiplexing(다중화)**
  - 하나의 회선을 분할해서 각기 다른 신호를 동시에 송수신

- **I/O Multiplexing(입출력 다중화)**
  - 소켓의 I/O를 다중화해서 사용
  - 여러 소켓이 각자의 소켓 I/O를 이용해서 통신하는 것이 아니라, 하나의 소켓I/O를 통해서 통신
  - 서버 부담을 줄임



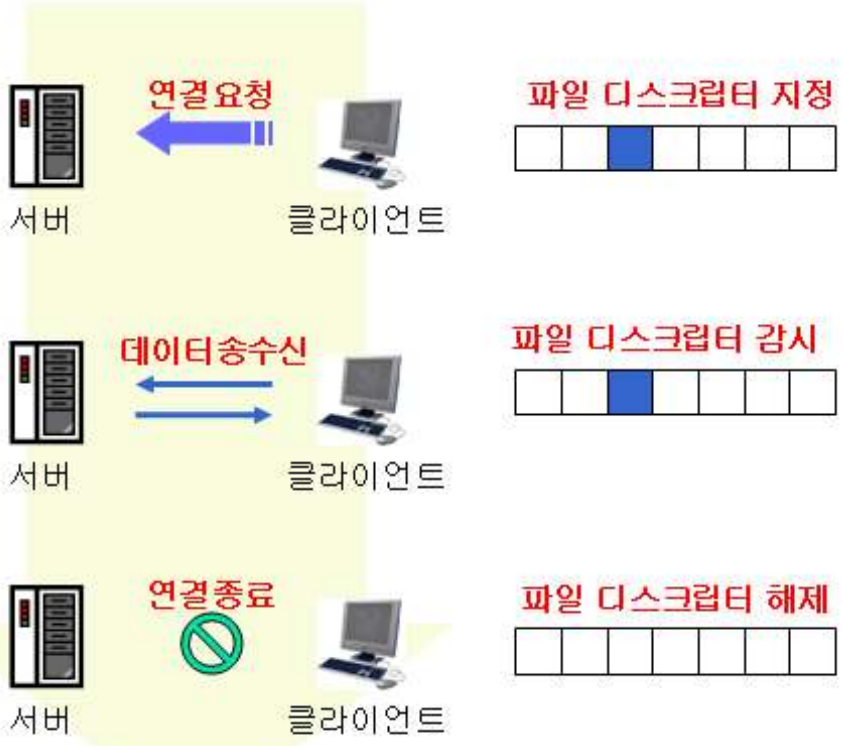
입출력 다중화 (I/O Multiplexing)

- 입출력 이벤트를 검사할 파일의 정보를 가지는 파일 디스크립터 테이블(비트 테이블, 파일지정번호 그룹)을 준비
- 비트 테이블에 검사할 파일을 체크
- 체크한 파일에 이벤트가 발생하면, **OS**는 이벤트가 발생한 비트 필드의 값을 **1**로 해서 반환
- 비트 테이블을 검사. 비트 필드가 **1**이면, 이에 대응하는 파일에 대해서 입출력 함수를 호출



입출력 다중화 (I/O Multiplexing)

- select() 는 파일 디스크립터 전체에서 I/O가 발생했다는 것을 알려주지만
- 정확하게 몇 번째 파일 디스크립터에서 I/O가 발생했다는 것을 알려주지 않음
  - 루프를 돌면서 찾아야 함
  - 현재 발생한 I/O 종류를 구분해야 함
    - 듣기 소켓(서버 소켓)에 입력이벤트 - 새로운 연결 요청 – accept() 호출
    - 연결 소켓(클라이언트 소켓)에 입력이벤트 - 데이터 수신 – read() 호출



select() system call

- **select()** function allows
  - the process to instruct the kernel
  - to **wait** for any one of multiple events to occur and
  - to **wake up** the process only
    - when one or more of these events occurs or
    - when a specified amount of time has passed

- We tell the kernel
  - what descriptors we are interested in (for reading, writing, or exception condition) and
  - how long to wait

select() system call

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

- nfds : 파일 테이블의 최대 크기
    - •tests **file descriptors** in the range of **0 to nfds-1**
- readfds : 읽기 이벤트를 검사할 파일정보를 포함한 비트 테이블
- writefds : 쓰기 이벤트를 검사할 파일정보를 포함한 비트 테이블
- exceptfds : 예외 이벤트를 검사할 파일정보를 포함한 비트 테이블
- timeout : 이벤트를 기다릴 시간 제한

select() system call

- On successful completion, the objects pointed to by the readfds, writefds, and exceptfds arguments are modified to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively. For each file descriptor less than nfds, the corresponding bit is set on successful completion if it was set on input and the associated condition is true for that file descriptor.

- If the time limit expires before any event occurs that would cause one of the masks to be set to a non-zero value, select() completes successfully and returns 0.

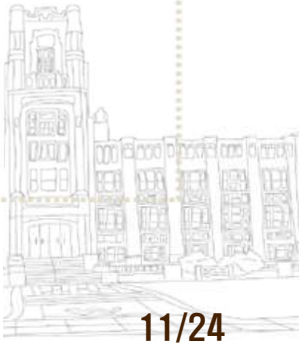
- 반환 값 : 이벤트가 발생한 파일 수

select() system call

- A *timeval* structure specifies the number of seconds and microseconds

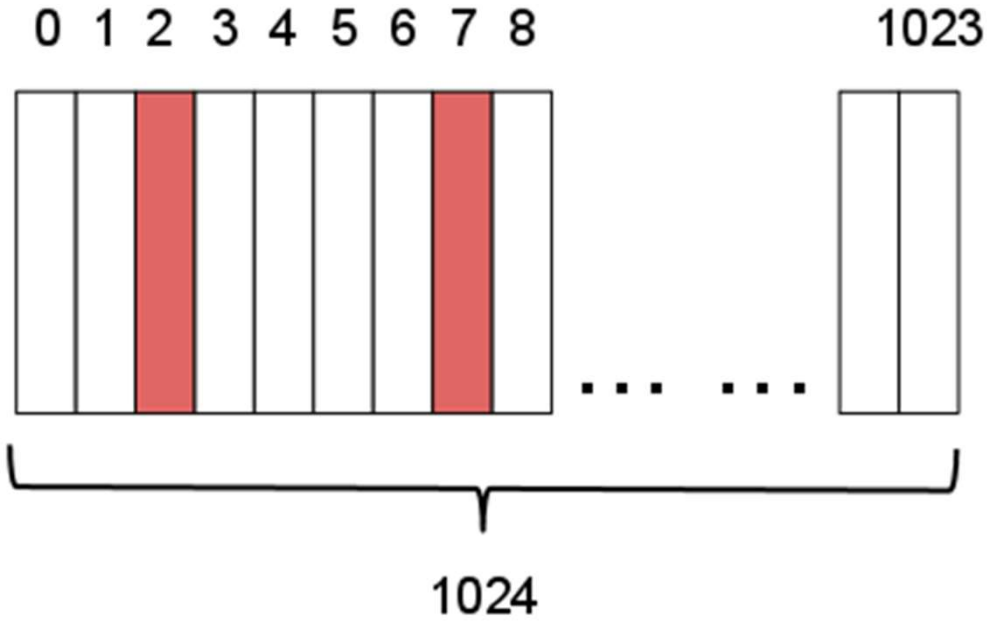
- struct timeval {
  long tv_sec; /* seconds */
  long tv_usec; /* microseconds */
  }

- There are three possibilities.
  1. Wait forever: - null pointer
  2. Wait up to a fixed amount of time:
  3. Do not wait at all:



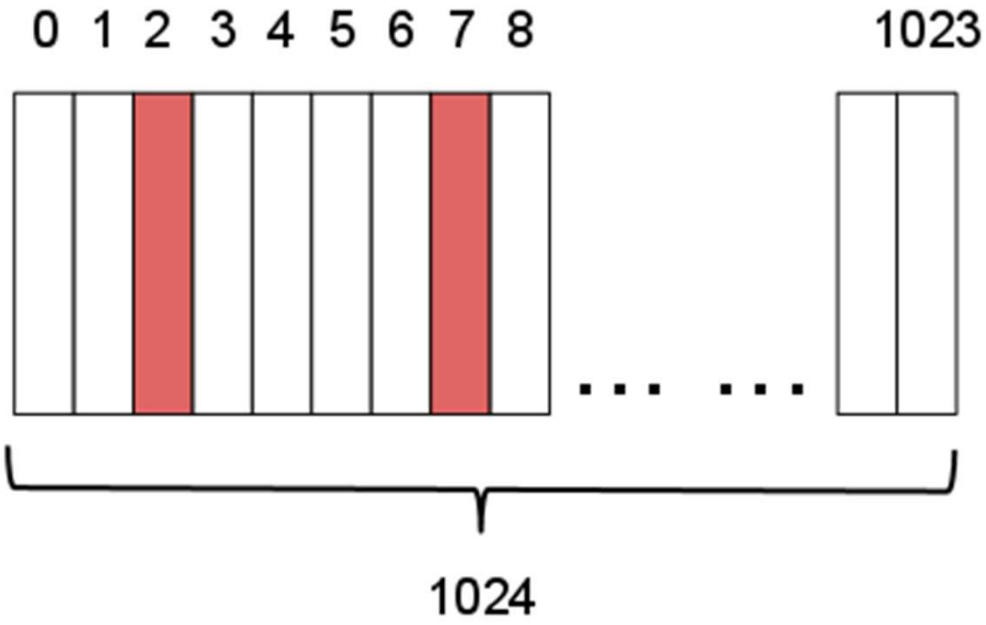
select() system call

- nfds = 최대 파일 지정 번호 + 1
  - 비트 테이블은 비트 array로 0번째 부터 시작.
  - ex) 7번 소켓이 만들어졌다면, 7+1 = 8



select 의 문제점

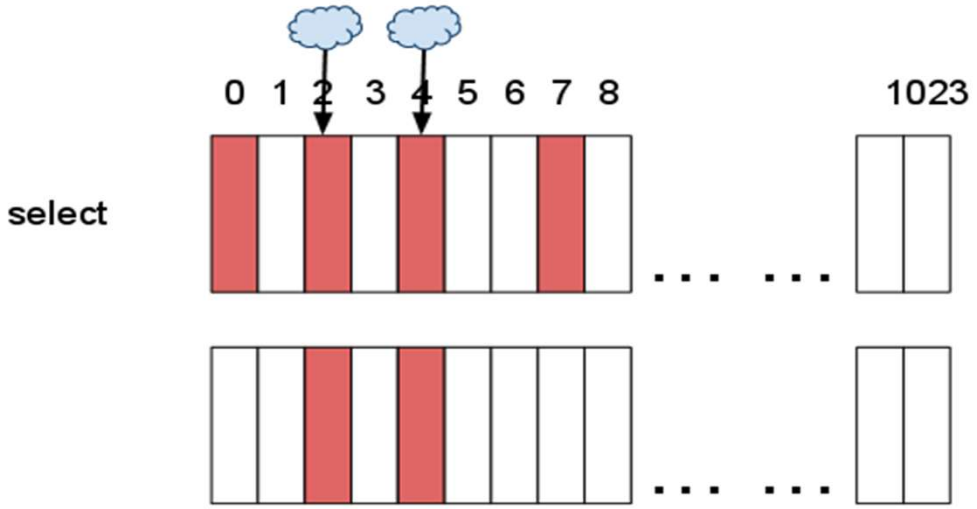
- 최대 파일지정번호 + 1만큼 검사
- 2와 7 두개의 파일만 검사할 경우에도 **8개의 필드를 모두 검사**해야 함



select 의 문제점

- **fd_set**은 이전 상태를 기억하지 못함
- 매번 파일 테이블을 복사해야 함

```
fd_set readfds;
readfds(0, 2, 4, 7);
select(int nfds, &readfds, ....);
```



- fd_set에 0,2,4,7을 지정
- 2, 4에 입출력 이벤트 발생
- 이전 상태인 0,2,4,7을 잃어버림

select 함수 매크로 사용

- select는 fd_set의 제어가 핵심
- fd_set 비트 연산을 돕기 위한 매크로 함수 제공
  - •비트 테이블 초기화
  - •비트 테이블 값 설정
  - •비트 테이블 값 검사

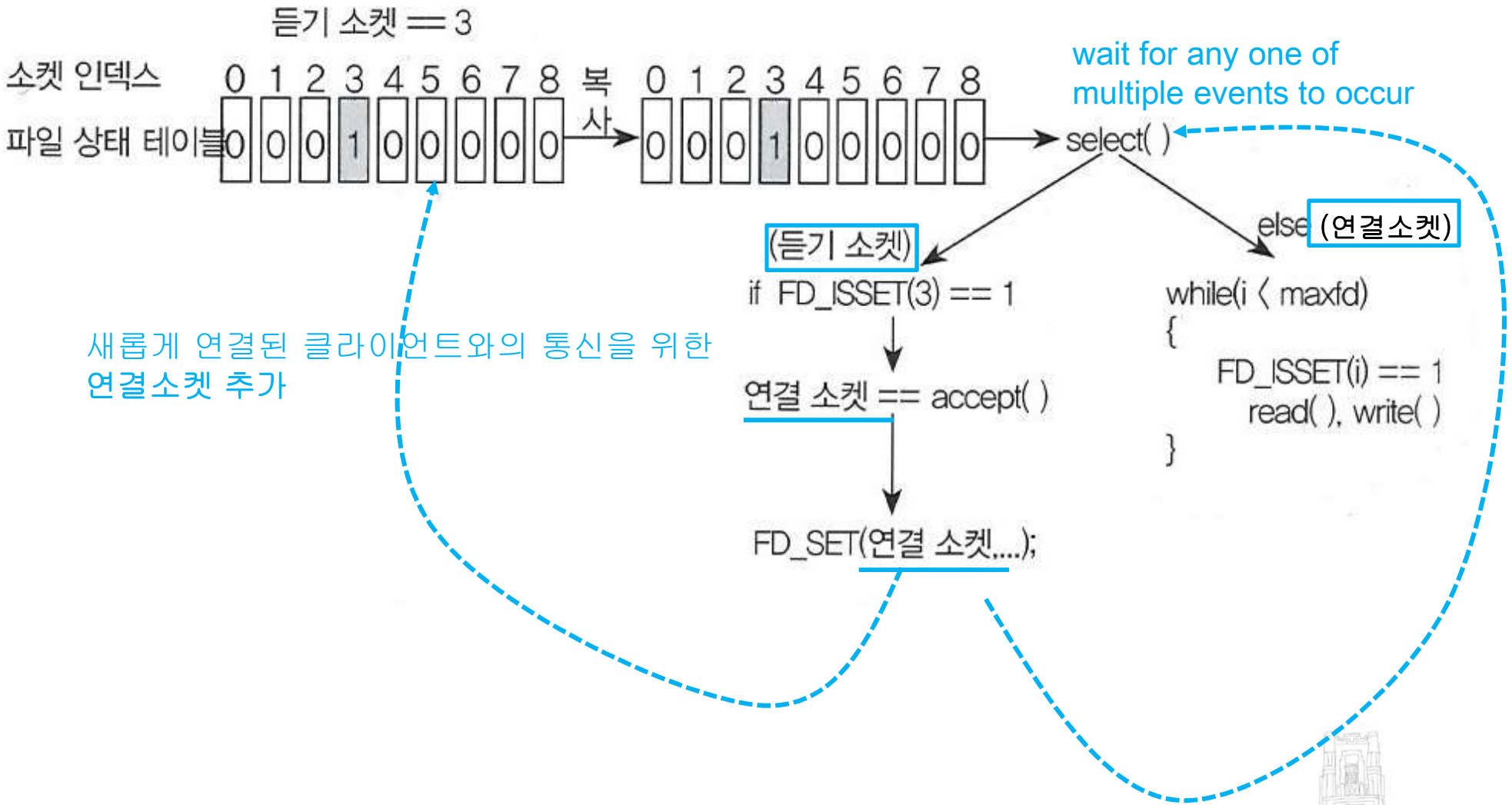
- fd_set 테이블을 0으로 초기화
  - **FD_ZERO**(fd_set *fds);

- fd_set 테이블에 검사할 파일 목록 추가
  - **FD_SET**(int fd, fd_set *fds);

- fd_set 테이블에서 파일 삭제
  - **FD_CLR**(int fd, fd_set *fds);

- fd_set 테이블을 검사
  - **FD_ISSET**(int fd, fd_set *fds);

select 함수 사용 예 (echo_server_multi.c)



select 함수 사용 예 (echo_server_multi.c)

```
int main()
{
  int listen_fd, client_fd;
  int fd_num;
  int maxfd = 0;
  fd_set readfds, allfds;

  listen_fd = socket(..);
  bind(listen_fd, ..);
  listen(listen_fd, ..);

  FD_ZERO(&readfds);                  // fd_set을 초기화 함 (0 으로 채움)
  FD_SET(listen_fd, &readfds);        // fd_set에 듣기소켓 추가
  maxfd = listen_fd;                  // 검사할 비트 테이블 크기 설정
```

select 함수 사용 예

```
while(1)
{
   allfds = readfds;
   printf("Select Wait %d\n", maxfd);

  // select 함수로 입출력 이벤트 대기 - wait forever
  fd_num = select(maxfd + 1, &allfds, (fd_set *) 0, (fd_set *) 0, NULL);
  // fd_num : 이벤트가 발생한 파일 수

  // 만약 듣기 소켓으로부터 데이터가 있다면(즉, 클라이언트로부터 연결요청)
  if( FD_ISSET(listen_fd, &allfds) )
  {
    client_fd = accept(listen_fd,..);                    // accept 함수를 호출하고
    FD_SET(client_fd, &readfds);       // 연결소켓을 fd_set에 추가
    if (client_fd > maxfd) maxfd = client_fd;
    printf("Accept OK\n");
    continue;                                  // wait for events to occur - skip the next slide
  }

   // 만일 연결소켓으로부터 읽을 데이터가 있다면
    see the next slide
}
```

```
// 어떤 연결소켓으로부터 읽을 데이터가 있는가 확인 - maxfd 만큼 반복처리
// 이벤트가 발생한 소켓인지를 확인 후 read 함수를 호출하여 데이터 처리
for (i = 0; i <= maxfd ; i++)
{ sockfd = i;
  if( FD_ISSET(sockfd, &allfds) )
  {
     if (read(sockfd, buf, MAXLINE) <= 0)
     {
        close(sockfd);          FD_CLR(sockfd, &readfds);
     }
     else
     {
         if (strncmp(buf, "quit\n", 5) ==0)
         {
            close(sockfd);       FD_CLR(sockfd, &readfds);
         }
         else // echo
         {
            printf("Read : %s", buf);
            write(sockfd, buf, strlen(buf));
         }
         if (--fd_num <= 0)       break; // fd_num : 이벤트가 발생한 파일 수
     } // end of else
  } // end of if
} // end of for
```

select 함수 사용 예

Lab

osnw00000000@osnw00000000-osnw: ~/lab09

```
osnw00000000@osnw00000000-osnw:~/lab09$ ./echo_server_multi
Select Wait 3
Accept OK
Select Wait 4
Accept OK
Select Wait 5
Read : osnw2023
Select Wait 5
Read : from 1st client
Select Wait 5
Read : final examination is comming
Select Wait 5
Read : from 2nd client
Select Wait 5
```

osnw00000000@osnw00000000-osnw: ~/lab09

```
osnw00000000@osnw00000000-osnw:~/lab09$ ./echo_client_loop
osnw2023
read : osnw2023
from 1st client
read : from 1st client
```

osnw00000000@osnw00000000-osnw: ~/lab09

```
osnw00000000@osnw00000000-osnw:~/lab09$ ./echo_client_loop
final examination is comming
read : final examination is comming
from 2nd client
read : from 2nd client
```

입출력 다중화의 특징과 적용처

- 단일 프로세스, 단일 스레드에서 여러 소켓 처리 가능
  - 일반적으로 멀티 프로세스나 멀티 스레드 다중접속서버 보다 효율적

- 동시 실행이 아닌 순차적 처리임.
  - 데이터를 읽어서 처리하고 응답하는데 많은 시간이 걸리는 서비스에는 적당하지 않음. 그 시간 동안 다른 입출력은 대기해야 함
  - 앞의 서비스 처리가 늦어지면, 그 시간만큼 지연이 생김
  - 요청처리에 걸리는 시간이 짧은 서비스에 적합하며, 대용량 파일전송 서비스, DB를 열람해야 하는 정보서비스에는 부적절함

- 최대 **1024** 만큼의 동시 접속만을 지원함
  - 멀티 프로세스(or 멀티 스레드) 기반의 서버에 비해서는 많은 수의 클라이언트 처리

Thank You !

뇌를 자극하는 TCP/IP 소켓 프로그래밍