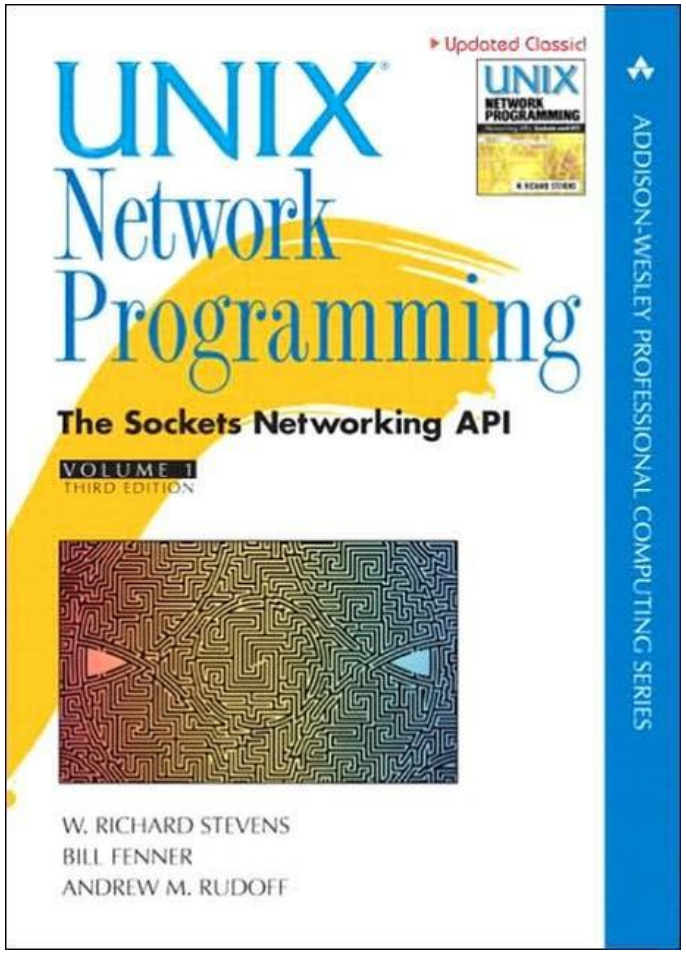
Unix Network Programming

16 : Nonblocking I/O



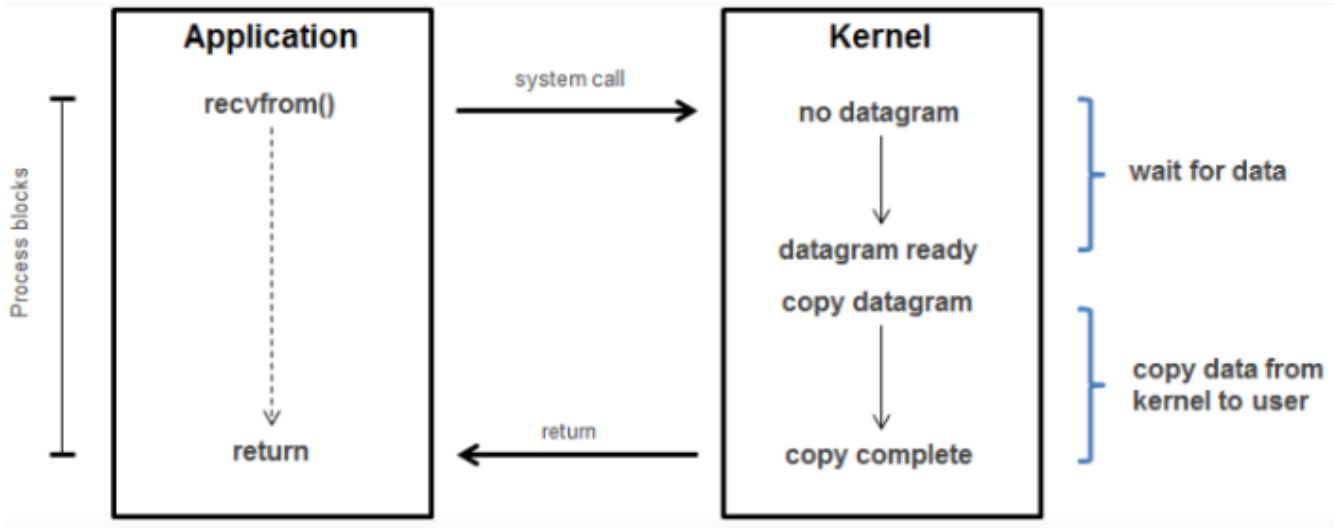
Chapter16 : Nonblocking I/O

발표자 : 김수환

Blocking vs Nonblocking

▪ Blocking Model

가장 기본적인 I/O 모델로, Linux에서 모든 소켓은 기본 blockin으로 동작한다.
I/O 작업이 진행되는 동안 유저 프로세스는 자신의 작업을 중단한 채 대기하는 방식이다.



1. 유저는 커널에게 read 작업 요청
2. Input이 들어올 때까지 대기
3. Input이 들어오면 커널에서 유저에게 결과가 전달되면 본래 작업으로 복귀

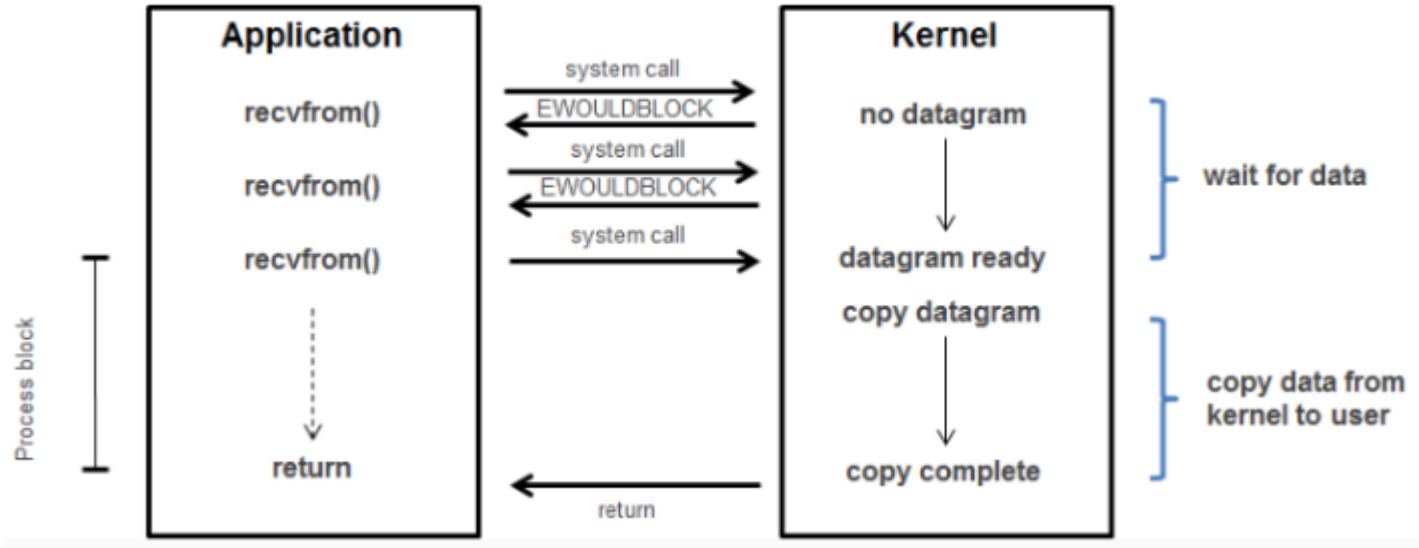
말 그대로 block이 되고, 어플리케이션에서 다른 작업을 수행하지 못하고 wait하게 되므로 자원이 낭비된다.

ㄴex) 카톡이 메시지를 보낼때까지 다른 일을 못하고 무한정 대기한다고 생각!

Blocking vs Nonblocking

▪ Non-Blocking Model

앞의 blocking 방식의 비효율성을 극복하고자 도입된 방식.
I/O 작업이 진행되는 동안 유저 프로세스의 작업을 중단시키지 않는 방식



1. 유저가 커널에게 read 작업 요청
2. Input이 있든 없든, 바로 결과 반환
   (Input이 없으면 EWOULDBLOCK 반환)
3. 입력 데이터가 있을 때까지 1-2 반복
4. Input이 있으면 유저에게 결과 반환

I/O 진행시간과 관계가 없기 때문에 작업을 오랜 시간 중지하지 않고도 I/O 작업을 진행 가능

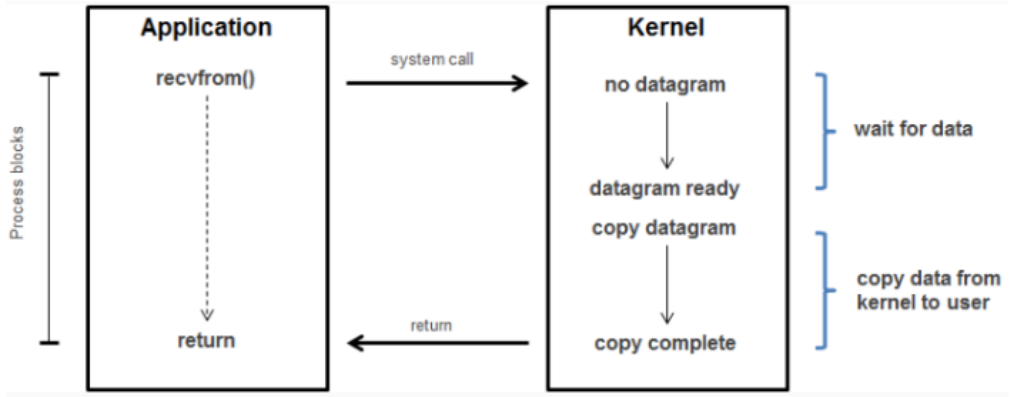
ㄴBut! 반복적으로 시스템 호출이 발생하여 이 경우 역시 자원이 낭비

Blocking vs Nonblocking

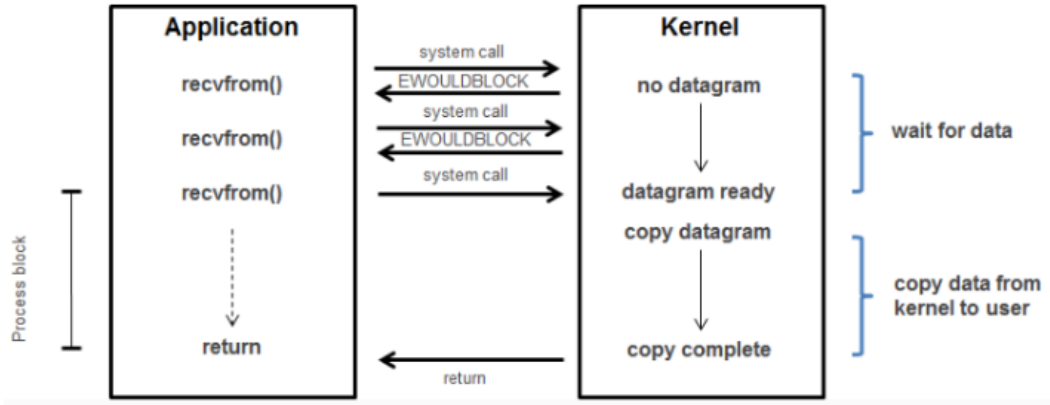
▪ Non-Blocking Socket의 장점 / 단점

- 장점 : Multi-Thread를 사용하지 않고도 다른 작업을 할 수 있다.
- 단점 : 프로그램이 복잡해지며, **CPU 사용량이 증가**한다.

  => 멀티스레드 환경에서는 사용하지 않는 것이 좋다!



Blocking I/O



Non-Blocking I/O

Introduction

▪ Blocking Categories

소켓은 default로 blocking으로 동작한다. 이것은 우리가 작업을 요청했을 때, Input이 없으면 프로세스가 sleep에 빠지게 됨을 의미한다. 이러한 block 상태가 될 수도 있는 함수들 카테고리가 4가지가 있다.

① Input Operations
- read, readv, recv, recvfrom, recvmsg

② Output Operations
- write, writev, send, sendto, sendmsg

③ Accepting incoming connections
- accept

④ Initiating outgoing connections
- connect

Introduction

① Input Operations - read. readv. recv. recvfrom. recvmsg

Blocking TCP 소켓인 경우, 소켓수신버퍼에 수신된 데이터가 없으면, 프로세스는 sleep한다.
데이터가 도착하면 (그것이 충분한 크기가 아닐지라도) 프로세스는 깨어난다. 원하는 크기만큼 도착할때까지 기다리려면, while로 계속 받아 붙이던가, 아니면 MSG_WAITALL 플래그를 이용한다.

UDP 소켓인 경우, 소켓수신버퍼가 비어 있으면, 프로세스는 sleep한다. UDP 패킷이 도착하면, 프로세스는 깨어난다.

Nonblocking 소켓의 경우, 수신버퍼가 비어 있는 경우, 에러 EWOULDBLOCK으로 바로 리턴한다.

② Output Operations - write. writev. send. sendto. sendmsg

Blocking TCP 소켓인 경우, 출력 함수는 어플리케이션의 데이터를 커널의 소켓전송버퍼에 복사한다.
만약 소켓전송버퍼에 공간이 없으면, 프로세스는 sleep한다.

Nonblocking TCP 소켓의 경우, 소켓전송버퍼에 공간이 없는 경우, 에러 EWOULDBLOCK으로 바로 리턴한다.
만약 소켓전송버퍼에 약간의 공간이 있는 경우, 복사가능한 공간을 바이트로 리턴한다.

UDP 소켓은 실제로 소켓전송버퍼가 없다. 즉, 호출 즉시 UDP/IP 스택으로 전달하므로, block 되지 않는다.

Introduction

③ Accept Operations - accept

Blocking 소켓인 경우, 새로운 연결이 없으면, 프로세스는 sleep한다.

Nonblocking TCP 소켓의 경우, 에러 EWOULDBLOCK으로 리턴한다.

④ Connect Operations - connect

Blocking 소켓인 경우, 실제 연결이 될 때까지 (SYN에 대한 ACK을 받을 때까지) block 되어 있는다.

Nonblocking TCP 소켓의 경우, 에러 einprogress로 리턴한다.
(같은 호스트의 경우에는 바로 정상 연결이 이루어 질 수 있다.)

UDP 소켓은 connect는 가상의 연결을 만드는 것이라서 바로 리턴한다.

Nonblocking Reads & Writes: str_cli Function

- **앞의 Section 6.4에서 봤던 select() 를 이용한 Blocking I/O 코드를 조금 수정** (to & fr buffer 활용)

Section 6.4 strcliselect..c

```c
#include    "unp.h"

void str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1;
    fd_set      rset;
    char        sendline[MAXLINE], recvline[MAXLINE];

    FD_ZERO(&rset);
    for ( ; ; ) {
        FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd) + 1;
        Select(maxfdp1, &rset, NULL, NULL, NULL);

        if (FD_ISSET(sockfd, &rset)) {  /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0)
                err_quit("str_cli: server terminated prematurely");
            Fputs(recvline, stdout);
        }

        if (FD_ISSET(fileno(fp), &rset)) {  /* input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL)
                return;     /* all done */
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

Nonblocking Reads & Writes: str_cli Function

- **기존** str_cli **Func**
  - Blocking I/O 사용
  - 소켓 송신 버퍼가 가득 차 있으면, Written 호출은 block 된다.
  - Writen 호출에서 block 되어 있는 동안, data는 소켓 수신 버퍼로부터 읽기 가능

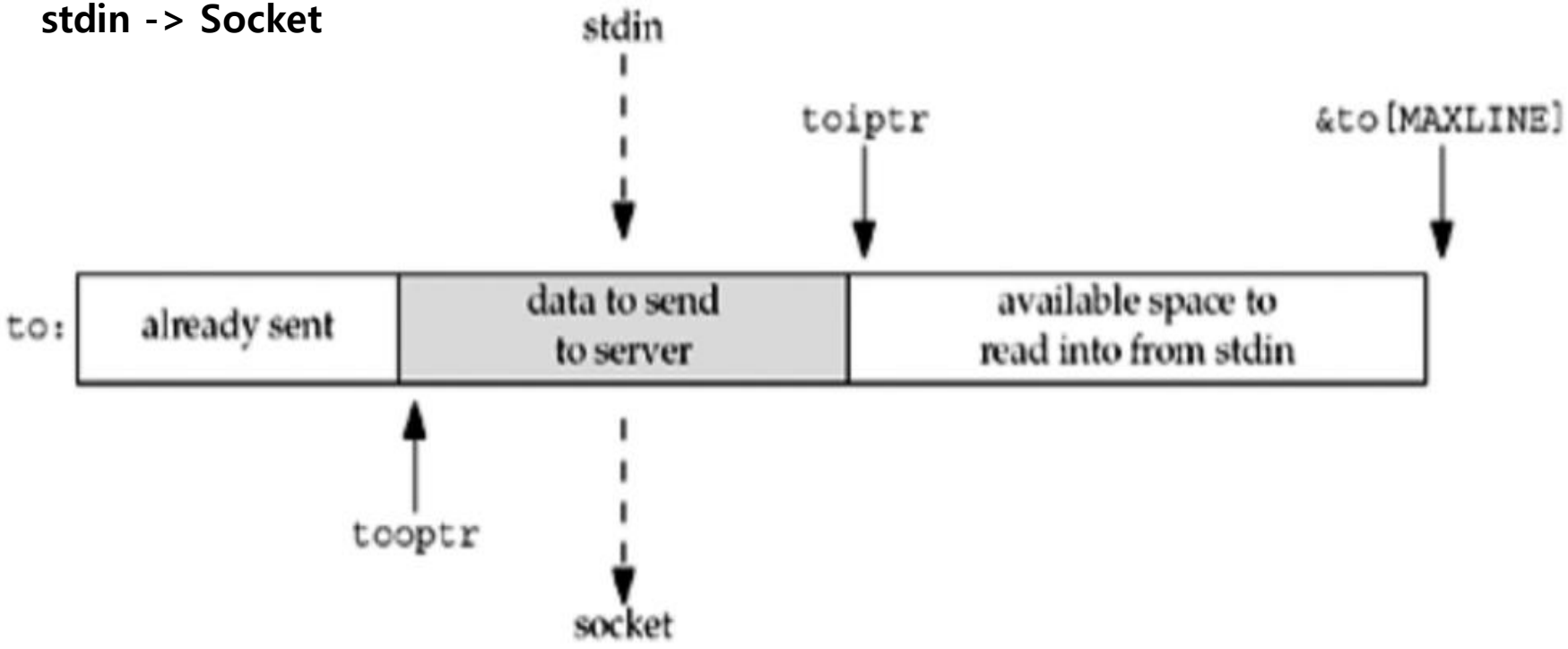
- **Non-blocking** str_cli**의 목표**
  - 기존 str_cli Func를 non-blocking으로 변형하는 것
  - Block 되어 다른 일을 못하게 되는 것을 방지

Nonblocking Reads & Writes: str_cli Function

- **to Buffer**

  소켓에 전달될 표준 입력(stdin)의 데이터가 들어있는 버퍼

stdin -> Socket

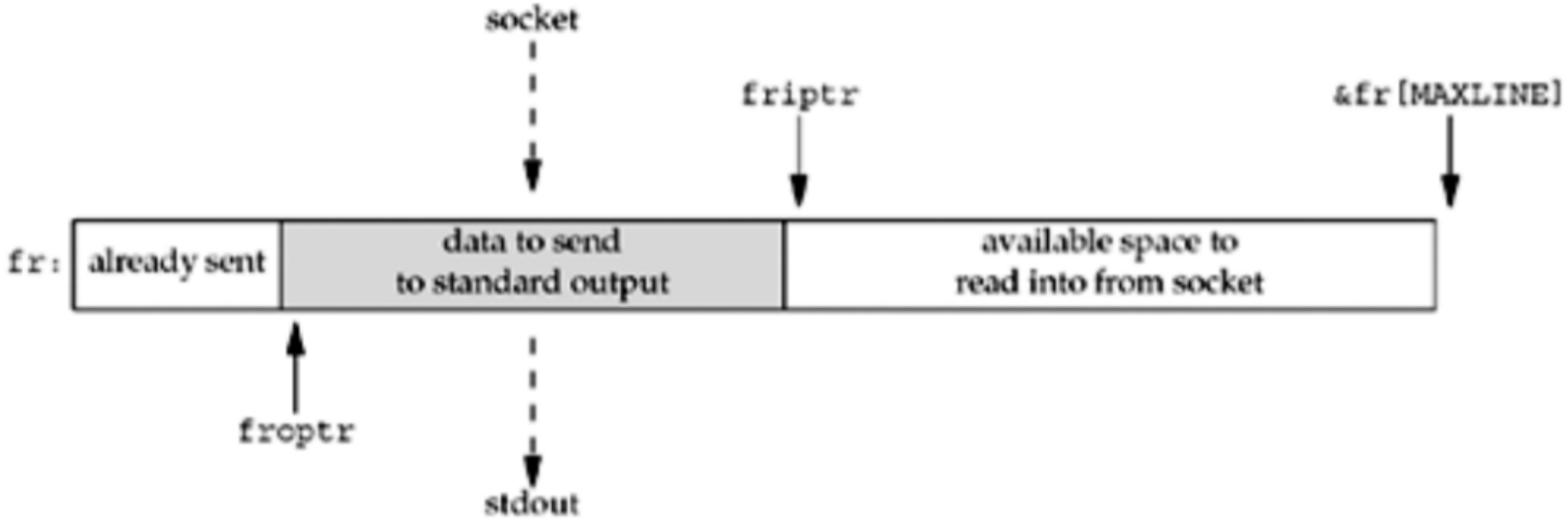


Nonblocking Reads & Writes: str_cli Function

- **fr Buffer**

  표준 출력(stdout)에 전달될 소켓의 데이터가 들어있는 버퍼

Socket -> stdout



Nonblocking Reads & Writes: str_cli Function

- **to Buffer & fr Buffer 정리**

① **optr == iptr**

Read or Write 할 데이터가 없음을 의미

② **optr < iptr**

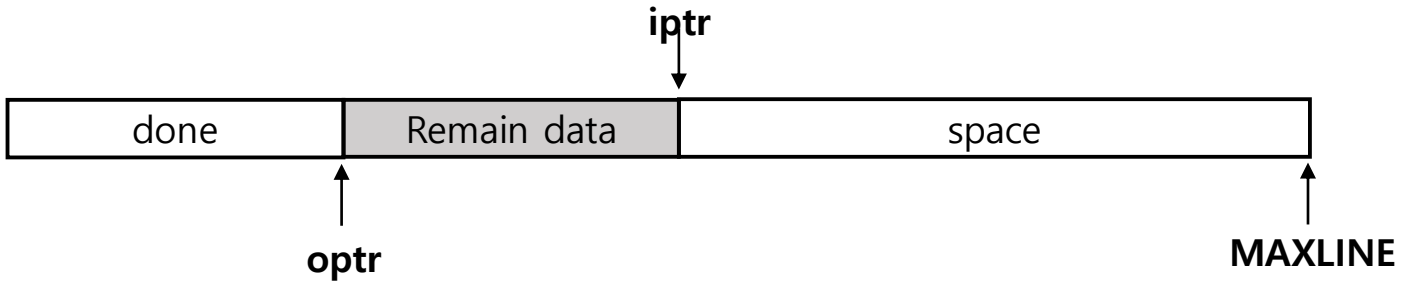
Read or Write 할 데이터가 남아 있음을 의미

③ **optr > iptr**

Error!! (일어나서는 안 되는 경우)

④ **lptr == MAXLINE**

버퍼가 꽉 찬 것을 의미



Nonblocking Reads & Writes: str_cli Function

- str_cli **function** - ①

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, val, stdineof;
    ssize_t     n, nwritten;
    fd_set      rset, wset;
    char        to[MAXLINE], fr[MAXLINE];
    char        *toiptr, *tooptr, *friptr, *froptr;

    val = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

    toiptr = tooptr = to;   /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;

    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
    for ( ; ; ) {
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        if (stdineof == 0 && toiptr < &to[MAXLINE])
            FD_SET(STDIN_FILENO, &rset);    /* read from stdin */
        if (friptr < &fr[MAXLINE])
            FD_SET(sockfd, &rset);          /* read from socket */
        if (tooptr != toiptr)
            FD_SET(sockfd, &wset);          /* data to write to socket */
        if (froptr != friptr)
            FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */

        Select(maxfdp1, &rset, &wset, NULL, NULL);
```

Nonblocking Reads & Writes: str_cli Function

- str_cli **function** - ①

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, val, stdineof;
    ssize_t     n, nwritten;
    fd_set      rset, wset;
    char        to[MAXLINE], fr[MAXLINE];
    char        *toiptr, *tooptr, *friptr, *froptr;

    val = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

    toiptr = tooptr = to;   /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;

    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
    for ( ; ; ) {
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        if (stdineof == 0 && toiptr < &to[MAXLINE])
            FD_SET(STDIN_FILENO, &rset);    /* read from stdin */
        if (friptr < &fr[MAXLINE])
            FD_SET(sockfd, &rset);          /* read from socket */
        if (tooptr != toiptr)
            FD_SET(sockfd, &wset);          /* data to write to socket */
        if (froptr != friptr)
            FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */

        Select(maxfdp1, &rset, &wset, NULL, NULL);
```

Fcntl을 이용하여 모든 descriptor를 nonblocking 방식화

Nonblocking Reads & Writes: str_cli Function

- str_cli **function** - ①

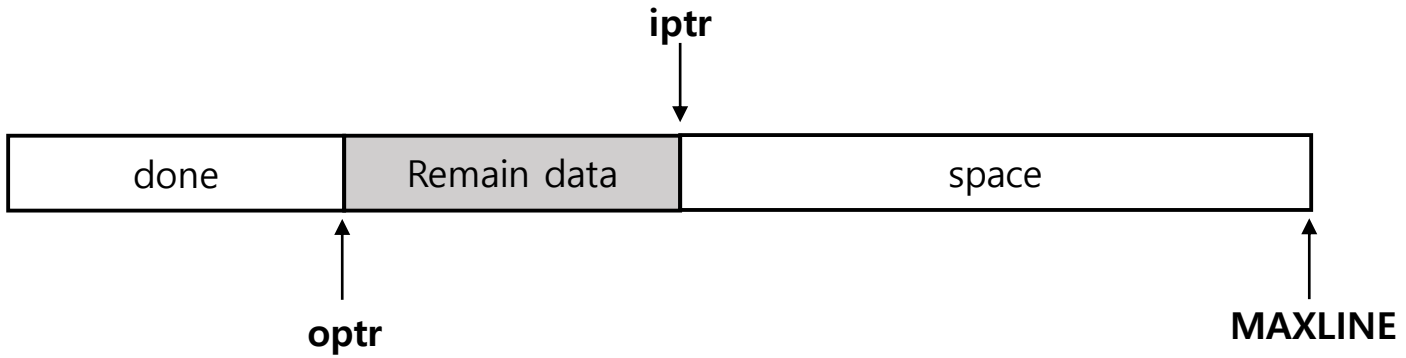
```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, val, stdineof;
    ssize_t     n, nwritten;
    fd_set      rset, wset;
    char        to[MAXLINE], fr[MAXLINE];
    char        *toiptr, *tooptr, *friptr, *froptr;

    val = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

    toiptr = tooptr = to;   /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;

    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
    for ( ; ; ) {
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        if (stdineof == 0 && toiptr < &to[MAXLINE])
            FD_SET(STDIN_FILENO, &rset);    /* read from stdin */
        if (friptr < &fr[MAXLINE])
            FD_SET(sockfd, &rset);          /* read from socket */
        if (tooptr != toiptr)
            FD_SET(sockfd, &wset);          /* data to write to socket */
        if (froptr != friptr)
            FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */

        Select(maxfdp1, &rset, &wset, NULL, NULL);
```

to & fr 버퍼 포인터 초기화

Nonblocking Reads & Writes: str_cli Function

- str_cli **function** - ①

```
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    int         maxfdp1, val, stdineof;
    ssize_t     n, nwritten;
    fd_set      rset, wset;
    char        to[MAXLINE], fr[MAXLINE];
    char        *toiptr, *tooptr, *friptr, *froptr;

    val = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDIN_FILENO, F_GETFL, 0);
    Fcntl(STDIN_FILENO, F_SETFL, val | O_NONBLOCK);

    val = Fcntl(STDOUT_FILENO, F_GETFL, 0);
    Fcntl(STDOUT_FILENO, F_SETFL, val | O_NONBLOCK);

    toiptr = tooptr = to;   /* initialize buffer pointers */
    friptr = froptr = fr;
    stdineof = 0;

    maxfdp1 = max(max(STDIN_FILENO, STDOUT_FILENO), sockfd) + 1;
for ( ; ; ) {
    FD_ZERO(&rset);
    FD_ZERO(&wset);
    if (stdineof == 0 && toiptr < &to[MAXLINE])
        FD_SET(STDIN_FILENO, &rset);    /* read from stdin */
    if (friptr < &fr[MAXLINE])
        FD_SET(sockfd, &rset);          /* read from socket */
    if (tooptr != toiptr)
        FD_SET(sockfd, &wset);          /* data to write to socket */
    if (froptr != friptr)
        FD_SET(STDOUT_FILENO, &wset);   /* data to write to stdout */

    Select(maxfdp1, &rset, &wset, NULL, NULL);
```



to & fr 버퍼 포인터를 이용한 Select
(발표자료 11p 참고)

(참고) select() Function

- fd_set 구조체

  file Descriptor (fd) 를 저장하는 구조체이다. (배열이라 생각하면 편하다)

- FD_SET()

  fd_set 구조체에 특정 FD를 저장

  ex)

```
fd_set testFds;
FD_SET(2, &testFds);
FD_SET(, &testFds);
select(), &testFds, NULL, NULL, NULL);
```

  => 위 구조체의 배열 값을 출력해보면

  0 **1** 0 0 **1** 0 0 0 0 ~ ...

위의 select()의 뜻은 FD 이벤트는 6 미만 값만 체크하며,
6미만 중 2와 5 FD 이벤트가 발생했을 때만 깨어난다는 뜻이다.

(참고) select() Function

- **Event 발생 시**

  만약 2번 FD의 이벤트가 발생하여 select() 함수가 깨어나고,
  select 함수 이후에 testFds를 찍어보면,

  0 **1** 0 0 0 0 0 0 0 ~ ...

  식으로 해당 비트만 1로 값이 변경된다.

  => select()는 위처럼 FD를 변경시키기 때문에, 매번 **testFds를 다시 세팅**해주어야 한다.

  ==> 그래서 **반복문 안에서 매번 FD_ZERO(), FD_SET()으로 다시 세팅**을 해주는 것!

Nonblocking Reads & Writes: str_cli Function

- str_cli **function - ②** (③도 같은 구조이므로 생략)

```
if (FD_ISSET(STDIN_FILENO, &rset)) {
    if ( (n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
        if (errno != EWOULDBLOCK)
            err_sys("read error on stdin");

    } else if (n == 0) {
        fprintf(stderr, "%s: EOF on stdin\n", gf_time());
        stdineof = 1;            /* all done with stdin */
        if (tooptr == toiptr)
            Shutdown(sockfd, SHUT_WR);/* send FIN */

    } else {
        fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(), n);
        toiptr += n;             /* # just read */
        FD_SET(sockfd, &wset);  /* try and write to socket below */
    }
}

if (FD_ISSET(sockfd, &rset)) {
    if ( (n = read(sockfd, friptr, &fr[MAXLINE] - friptr)) < 0) {
        if (errno != EWOULDBLOCK)
            err_sys("read error on socket");

    } else if (n == 0) {
        fprintf(stderr, "%s: EOF on socket\n", gf_time());
        if (stdineof)
            return;      /* normal termination */
        else
            err_quit("str_cli: server terminated prematurely");

    } else {
        fprintf(stderr, "%s: read %d bytes from socket\n",
                        gf_time(), n);
        friptr += n;         /* # just read */
        FD_SET(STDOUT_FILENO, &wset);   /* try and write below */
    }
}
```

Error Handling

End

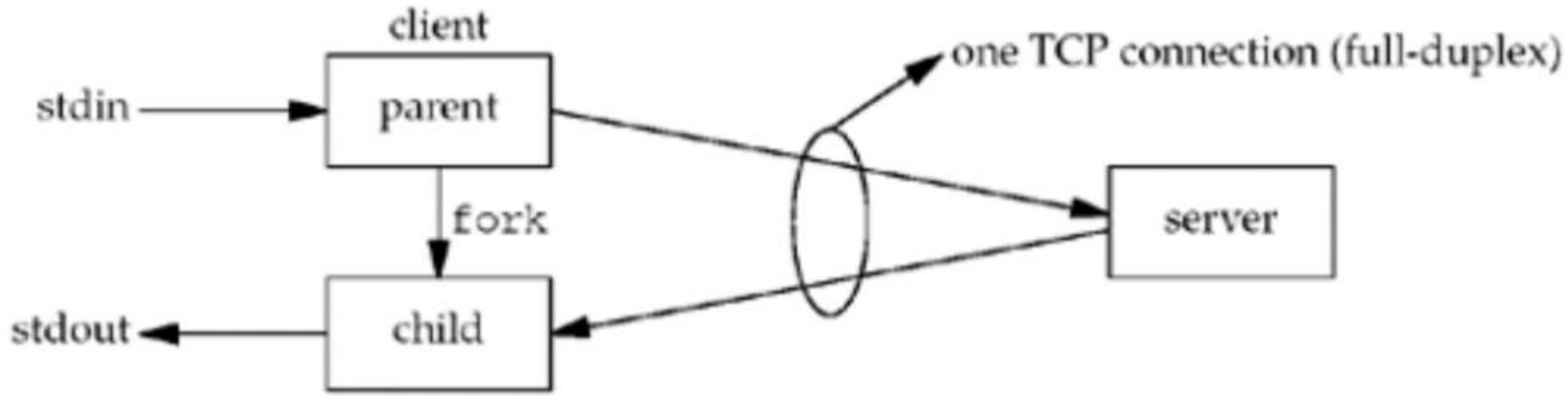
Read

Nonblocking Reads & Writes: str_cli Function

- **a Simpler Version of** str_cli

앞에까지의 코드에서 봤듯이, blocking I/O를 Non-blocking I/O로 바꾸게 되면 프로그램이 복잡해진다. (40line -> 135line)
하지만, 이렇게까지 프로그램을 복잡하게 하면서 non-blocking I/O를 써야 하느냐? 에 대한 대답은 **No**이다.

우리는 fork() 를 이용하여 더 간단한 방식의 Non-blocking I/O를 사용할 수 있다.



Nonblocking Reads & Writes: str_cli Function

- **a Simpler Version of** str_cli

```c
#include    "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
    pid_t   pid;
    char    sendline[MAXLINE], recvline[MAXLINE];

    if ( (pid = Fork()) == 0) {     /* child: server -> stdout */
        while (Readline(sockfd, recvline, MAXLINE) > 0)
            Fputs(recvline, stdout);

        kill(getppid(), SIGTERM);   /* in case parent still running */
        exit(0);
    }

        /* parent: stdin -> server */
    while (Fgets(sendline, MAXLINE, fp) != NULL)
        Writen(sockfd, sendline, strlen(sendline));

    Shutdown(sockfd, SHUT_WR);  /* EOF on stdin, send FIN */
    pause();
    return;
}
```

Child : stdout

Parent : stdin

Nonblocking Reads & Writes: str_cli Function

- **Timing of** str_cli

  2000 line을 Solaris 클라이언트로부터 서버에 175ms RTT로 복사할 때에 각 버젼들에 필요한 시간

  - 354.0 sec, stop-and-wait (Figure 5.5)
  - 12.3 sec, select and blocking I/O (Figure 6.13)
  - 6.9 sec, nonblocking I/O (Figure 16.3)
  - 8.7 sec, fork (Figure 16.9)
  - 8.5 sec, threaded version (Figure 23.2)

  Nonblocking I/O 버전이 select와 blocking I/O를 사용한 버전보다 2배 빠르다.

  Fork를 사용한 간단한 버전이 nonblocking I/O보다 느리지만 코드의 복잡성이 덜하다.

Nonblocking connect

- **동작 과정**
  - TCP 소켓이 nonblocking으로 설정되고 connec를 호출하면,
    connec는 EINPROGRESS Error를 리턴하지만, TCP three-way handshake는 계속된다.
  - 연결의 성공 여부는 select()를 이용해 점검
- **Nonblocking** connect**의 3가지 용도**
  - Three-way handshake를 다른 프로세스와 overlap 할 수 있다.
    - 연결이 완료되는 동안 다른 일을 수행할 수 있다.
  - 위의 기법을 이용하여 multiple-connection을 설정 가능하다.
    - Web Browser에서 널리 사용된다.
  - Select에서 시간 한계를 지정함으로써 connect의 시간 만료를 줄일 수 있다.
- **Berkeley-derived implmentations (and POSIX) 2 Rules of Nonblocking Connections**
  1. 연결이 성공하면, descriptor는 writable 해진다.
  2. 연결이 실패하면, descriptor는 read와 write가 동시에 가능해진다. (readable & writable)

Nonblocking connect: Daytime Client

- **Connect_nonb() Func**

```
int connect_nonb(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
    int             flags, n, error;
    socklen_t       len;
    fd_set          rset, wset;
    struct timeval  tval;

    flags = Fcntl(sockfd, F_GETFL, 0);
    Fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);

    error = 0;
    if ( (n = connect(sockfd, saptr, salen)) < 0)
        if (errno != EINPROGRESS)
            return(-1);

    /* Do whatever we want while the connect is taking place. */

    if (n == 0)
        goto done;  /* connect completed immediately */

    FD_ZERO(&rset);
    FD_SET(sockfd, &rset);
    wset = rset;
    tval.tv_sec = nsec;
    tval.tv_usec = 0;

    if ( (n = Select(sockfd+1, &rset, &wset, NULL,
                     nsec ? &tval : NULL)) == 0) {
        close(sockfd);      /* timeout */
        errno = ETIMEDOUT;
        return(-1);
    }

    if (FD_ISSET(sockfd, &rset) || FD_ISSET(sockfd, &wset)) {
        len = sizeof(error);
        if (getsockopt(sockfd, SOL_SOCKET, SO_ERROR, &error, &len) < 0)
            return(-1);         /* Solaris pending error */
    } else
        err_quit("select error: sockfd not set");

done:
    Fcntl(sockfd, F_SETFL, flags);  /* restore file status flags */

    if (error) {
        close(sockfd);      /* just in case */
        errno = error;
        return(-1);
    }
    return(0);
}
```

Nonblocking connect: Daytime Client

- **connect_nonb() Func**

  기존의 connect Func를 다음과 같이 바꿈

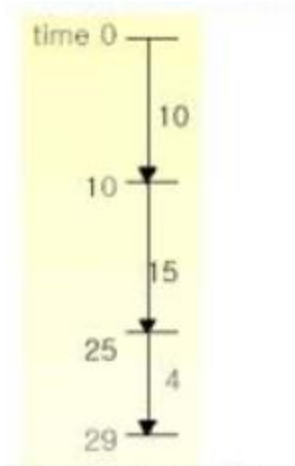
```
if ( connect_nonb(sockfd, (SA) &servaddr, sizeof(servaddr), 0) < 0 )
    err_sys("connect error");
```

  3번째 인수까지는 기존의 connect Func와 동일하지만, **4번째 parameter**가 새로 추가되었다.

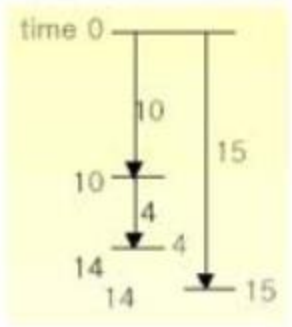
  4번째 params : 연결이 완료될 때까지 기다리는 sec
  (값이 0이면 시간 만료가 없음을 의미)

Nonblocking connect: **Web Client**

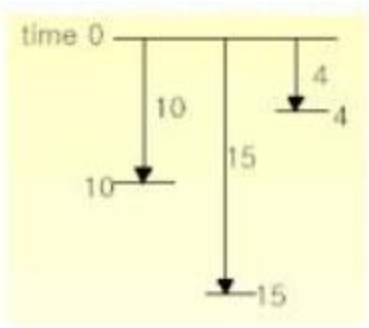
- **Nonblocking** connect**의 실질적인 예**
  - 클라이언트는 웹 서버와 HTTP 연결을 맺고 홈페이지를 가져온다.
    (해당 페이지는 다른 웹페이지에 대한 많은 참조를 가지고 있다고 가정)
  - 하나씩 가져오지만, Non-blocking connect를 사용하여 동시에 여러 개를 가져온다.



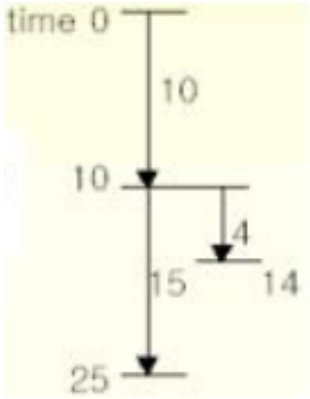
ıree connections
done serially



three connections
done in parallel:
maximum of two
connections at a time



three connections
done in parallel:
maximum of three
connections at a time



Complete first connection,
then multiple connections
in parallel.

Nonblocking connect: Web Client

- **web.h**

```
#include    "unp.h"

#define MAXFILES    20
#define SERV        "80"    /* port number or service name */

struct file {
  char  *f_name;            /* filename */
  char  *f_host;            /* hostname or IPv4/IPv6 address */
  int    f_fd;              /* descriptor */
  int    f_flags;           /* F_xxx below */
} file[MAXFILES];

#define F_CONNECTING    1   /* connect() in progress */
#define F_READING       2   /* connect() complete; now reading */
#define F_DONE          4   /* all done */

#define GET_CMD     "GET %s HTTP/1.0\r\n\r\n"

            /* globals */
int     nconn, nfiles, nlefttoconn, nlefttoread, maxfd;
fd_set  rset, wset;

            /* function prototypes */
void    home_page(const char *, const char *);
void    start_connect(struct file *);
void    write_get_cmd(struct file *);
```

자세한 코드는 생략…

Nonblocking accept

- **accept에서 발생 가능한 문제점**

  1. select가 서버 프로세스에게 readable (연결가능) 이라고 return하지만 서버가 accept를 호출하는데 매우 짧은 시간이 걸린다.
  2. 서버가 select로부터의 return과 accept를 호출하는 것 사이에 RST를 클라이언트로부터 받는다.
  3. 완료된 연결이 큐에서 제거되고 다른 완료된 연결은 존재하지 않는다고 가정한다.
  4. 서버가 accept를 호출하지만, 완료된 연결이 없기 때문에 block 상태에 빠지게 된다.

- **해결책**

  - select로 connection이 언제 accept 될 준비가 되었는지 확인하려면 listening socket을 항상 nonblocking으로 둔다.
  - accept 호출에서 다음의 오류는 무시한다.
    - EWOULDBLOCK (Berkeley-derived 구현에서 클라이언트가 연결을 파기할 때)
    - ECONNABORTED (Posix. 1g 구현에서 클라이언트가 연결을 파기할 때)
    - EPROTO (SVR4 구현에서 클라이언트가 연결을 파기할 때)
    - EINTR (신호를 포착할 때)