06장. I/O 인터페이스 (6)

네트워크 프로그램 설계

학습 내용

- I/O 인터페이스 개요
- 파이프(pipe)와 FIFO (생략)
- 소켓(socket) 개요
- 바이트 순서
- 소켓 통신 과정
- TCP 소켓의 기초
- TCP 서버-클라이언트 (IPv4)
- DNS 서비스 이용
- 범용 코드로 작성한 TCP 서버-클라이언트
- fork( )를 이용한 TCP 서버
- UDP 서버- 클라이언트
- UDP 브로드캐스팅
- 소켓 옵션 제어
- I/O 모델

시그널(Signal)

- 시그널이란?
  - 예상치 않은 이벤트 발생에 따른 일종의 소프트웨어 인터럽트
    - Ex) ctrl + c, ctrl + z, 자식 프로세스의 종료
  - 외부에서 프로세스에게 전달할 수 있는 유일한 통로
- 인터럽트와의 차이점
  - 인터럽트는 H/W에 의해 OS로 전달됨
  - 시그널은 OS에 의해 프로세스로 전달됨
- 시그널 값의 확인
  - /usr/include/signal.h
  - /usr/include/bits/signum.h

/usr/include/bits/signal.h

```
#define     SIGHUP      1       /* hangup */
#define     SIGINT      2       /* interrupt , ctrl + c */
#define     SIGQUIT     3       /* quit */
#define     SIGILL      4       /* illegal instruction (not reset when caught) */
#define     SIGTRAP     5       /* trace trap (not reset when caught) */
#define     SIGIOT      6       /* IOT instruction */
#define     SIGABRT     6       /* used by abort, replace SIGIOT in the future */
#define     SIGEMT      7       /* EMT instruction */
#define     SIGFPE      8       /* floating point exception */
#define     SIGKILL     9       /* kill (cannot be caught or ignored) */
#define     SIGBUS      10      /* bus error */
#define     SIGSEGV     11      /* segmentation violation */
#define     SIGSYS      12      /* bad argument to system call */
#define     SIGPIPE     13      /* write on a pipe with no one to read it */
#define     SIGALRM     14      /* alarm clock */
#define     SIGTERM     15      /* software termination signal from kill */
#if defined(__rtems__)
#define     SIGURG      16      /* urgent condition on IO channel */
#define     SIGSTOP     17      /* sendable stop signal not from tty */
#define     SIGTSTP     18      /* stop signal from tty */
#define     SIGCONT     19      /* continue a stopped process */
#define     SIGCHLD     20      /* to parent on child stop or exit */
#define     SIGCLD      20      /* System V name for SIGCHLD */
#define     SIGTTIN     21      /* to readers pgrp upon background tty read */
#define     SIGTTOU     22      /* like TTIN for output if (tp->t_local&LTOSTOP) */
#define     SIGIO       23      /* input/output possible signal */
#define     SIGPOLL     SIGIO   /* System V name for SIGIO */
#define     SIGWINCH    24      /* window changed */
#define     SIGUSR1     25      /* user defined signal 1 */
#define     SIGUSR2     26      /* user defined signal 2 */
```

커널이 시그널을 처리하는 방법

- 각 시그널은 시그널 처리기(signal handler)를 통해 기본 동작으로 수행
  - 가능한 기본 동작
    - 커널이 시그널을 무시
    - 사용자에게 통지하지 않고 프로세스를 종료
    - 프로그램이 인터럽트 되며 시그널 처리 루틴이 실행
    - 시그널이 블로킹됨

시그널 이름	기본 동작	설명
SIGINT	Quit	Interrupt
SIGILL	Dump	Illegal instruction
SIGKILL	Quit	Kill
SIGSEGV	Dump	Out of range address
SIGALRM	Quit	Alarm clock
SIGCHLD	Ignore	Child status change
SIGTERM	Quit	SW termination sent by kill

시그널 처리 과정

- 시그널 처리 과정
  - 시그널이 프로세스로 보내질 때, OS는 해당 프로세스를 중지
  - 시그널 처리기가 실행되고 내부 루틴이 실행됨
  - OS는 중지되었던 해당 프로세스를 재실행

sigaction( )을 이용한 시그널 처리

- 시그널과 시그널 핸들러를 연결시켜 주는 함수
- 특정 시그널에 대한 기본 동작을 바꿈

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact );
```

- int signo: 시그널 번호
- sigaction act: 새로운 동작이 정의된 sigaction
- sigaction old: 이전 동작이 저장된 sigaction

```
struct sigaction
{
        void    (*sa_handler)( int );       /* 시그널 핸들러 지정 */
        sigset_t sa_mask;                   /* 블록될 시그널 마스킹 */
        int     sa_flags;                   /* 시그널의 설정 변경 */
}
```

sigaction()을 이용한 시그널 처리

```
/* SIGINT의 기본동작은 프로그램의 종료이다. 기본 핸들러를 바꾸어서  종료되지 않고 화면에 문자열을 출력되도록 한다. */

void handler(int sig);

int main()
{
        struct sigaction act;
        act.sa_handler = handler;                  /* 시그널 핸들러 연결 */
        sigemptyset( &act.sa_mask );               /* 블록할 시그널 없음 */
        act.sa_flags = 0;                          /* 기본 동작으로 설정 */

        sigaction( SIGINT, &act, 0 );  /* SIGINT과 handler 연결*/

         while(1) {
                printf("Hello World!\n");
                sleep(1);
        }
}

void handler(int sig) {
        printf("type of signal is %d \n", sig);
}
```

타임아웃(SIGALM 시그널) 예제

- SIGALM 시그널이란?
  - alarm(int) 함수에 의해 발생하며 int 초 이후 발생
  - 처리기의 기본동작은 프로세스 종료
  - 이를 수정하여 화면에 문자열 출력

```
void timer(int sig)  {
        puts(“alarm!! \n");   exit(0);
}

int main(int argc, char **argv)   {
        struct sigaction act;
        act.sa_handler=timer;
        sigemptyset(&act.sa_mask);
        act.sa_flags=0;

        state=sigaction(SIGALRM, &act, 0);
        alarm(5);

         while(1){
                  puts(“wait");  sleep(2);
         }
        return 0;
}
```

SIGCHLD 시그널

- SIGCHLD 시그널이란?
  - fork()로 인해 복제된 프로세스 중, 자식 프로세스가 종료되면 부모 프로세스에게 전달되는 시그널

wait() vs waitpid()

- wait()– 자식프로세스가 반환될 때까지 블럭됨

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int * status)
```

- waitpid()– WNOHANG옵션을 이용하면 자식 프로세스가 반환될 때까지 블록되지 않음
  - 시그널의 계류(pending) 특성으로 인해 SIGCHLD 시그널은 한번 도착했지만 현재 종료된 자식 프로세스는 여러 개 일 수 있음
  - 따라서 넌블럭 waitpid를 반복 호출하여 남아있는 좀비 프로세스의 제거가 가능

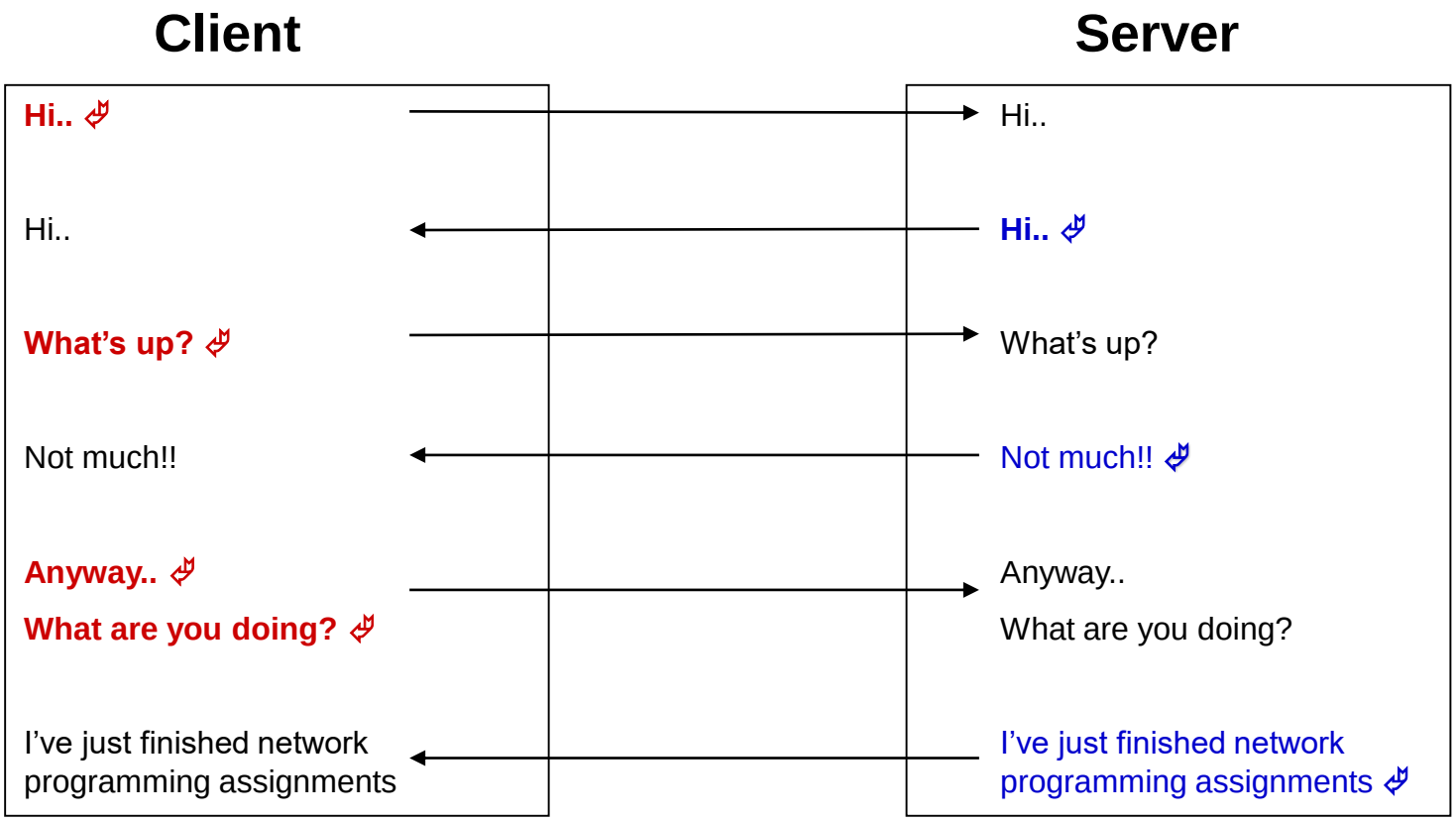
```
#include <sys/types.h>
#include <sys/wait.h>

 pid_t waitpid(pid_t pid, int * status, int options)
```

프로세스 분기와 sigaction을 이용한 자식 프로세스의 자원수거

```
void handler(int sig){
        int pid;
        int status;
        while(1) {
                pid = waitpid( WAIT_ANY, &status, WNOHANG );
                if ( pid < 0 ) {
                        perror("waitpid");
                        break;
                }
                if ( pid == 0 )
                        break;
        }
}

int main(int argc, char *argv[]) {
        struct sigaction act;
        act.sa_handler = handler;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        sigaction(SIGCHLD, &act, 0);
        pid = fork();

}
```

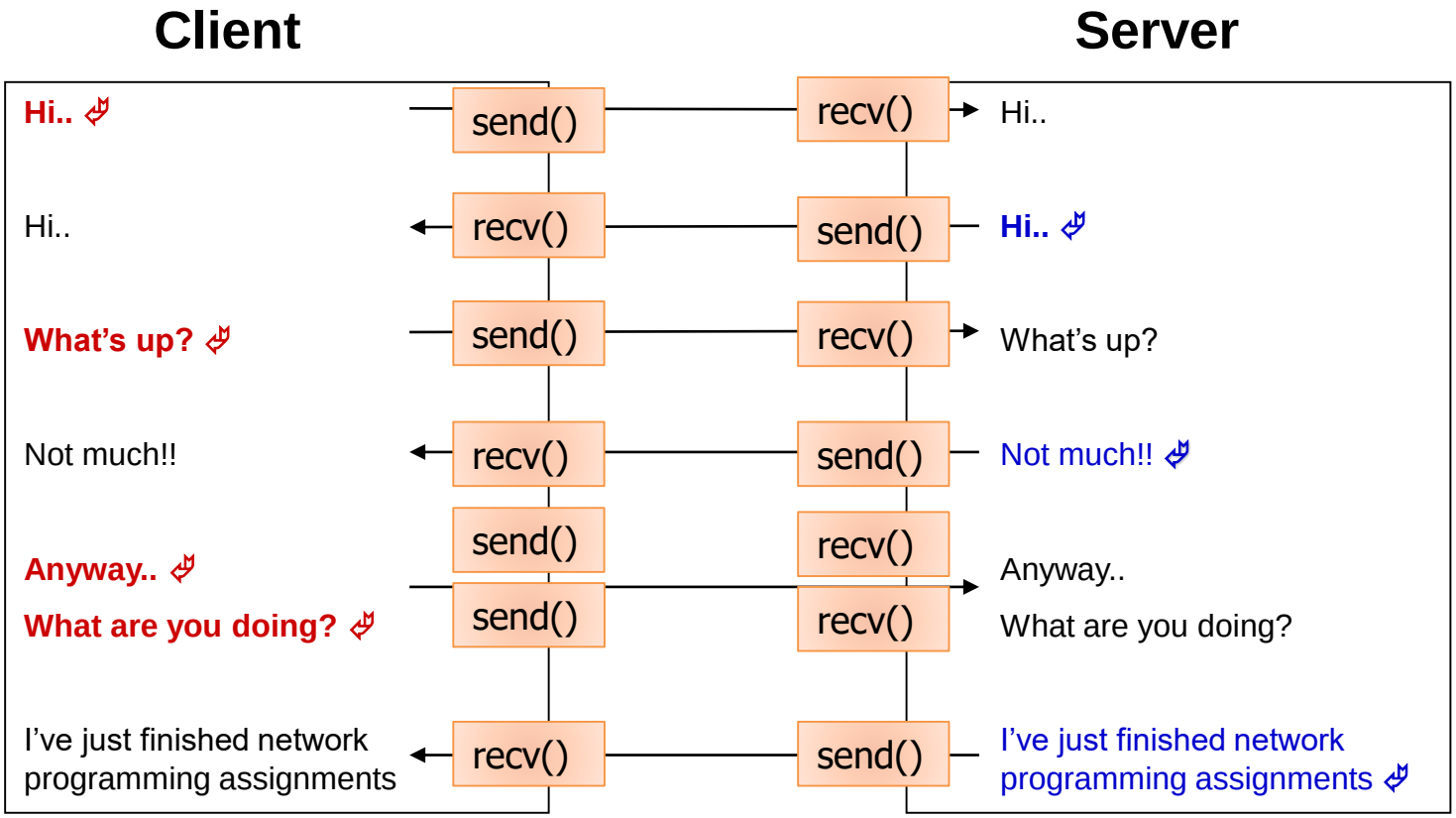
넌 블록 I/O 모델

- 다음과 같은 채팅 프로그램의 구현이 가능한가?



넌 블록 I/O 모델 (계속)

- send()를 연이어 두 번 하거나, 클라이언트 혹은 서버 중 어느 한쪽이 먼저 채팅을 시작하게 할 수 있는가?



앞 예제의 문제점

- 사용자의 입출력 패턴과 recv(), send()의 동기화가 필요함
  - send()의 경우, 사용자 입력이 있을 때까지 기다림
  - recv()와 같은 함수의 경우, 수신할 데이터가 있을 때까지 기다림
    - 블록 함수
  - 동기화 되지 않을 경우, block되어 진행 불가
- 사용자의 입력패턴을 정확히 예상하고 send(), recv()를 코딩
  - 현실적으로 불가능
  - 한 턴씩 진행되는 간단한 경우에 사용 가능

해결 방법

- 문제점
  - 블록 함수의 사용으로 인한 고착 상태
- 해결 방안
  - Non-Blocking 함수의 사용
    - 블록 되지 않고 바로 리턴
    - 필요에 따라 폴링(polling) 루틴 작성 필요
  - 비동기(Asynchronous I/O)사용
    - 소켓(파일)에서 어떤 I/O 변화가 발생하면 그 사실을 응용 프로그램이 알 수 있도록 하여 그 때 원하는 동작을 할 수 있게 하는 모드
    - I/O가 발생시 전달되는 SIGIO처리를 통해 폴링이 아닌 인터럽트 방식으로 처리하는 방식

블록 모드 vs 넌블럭 모드

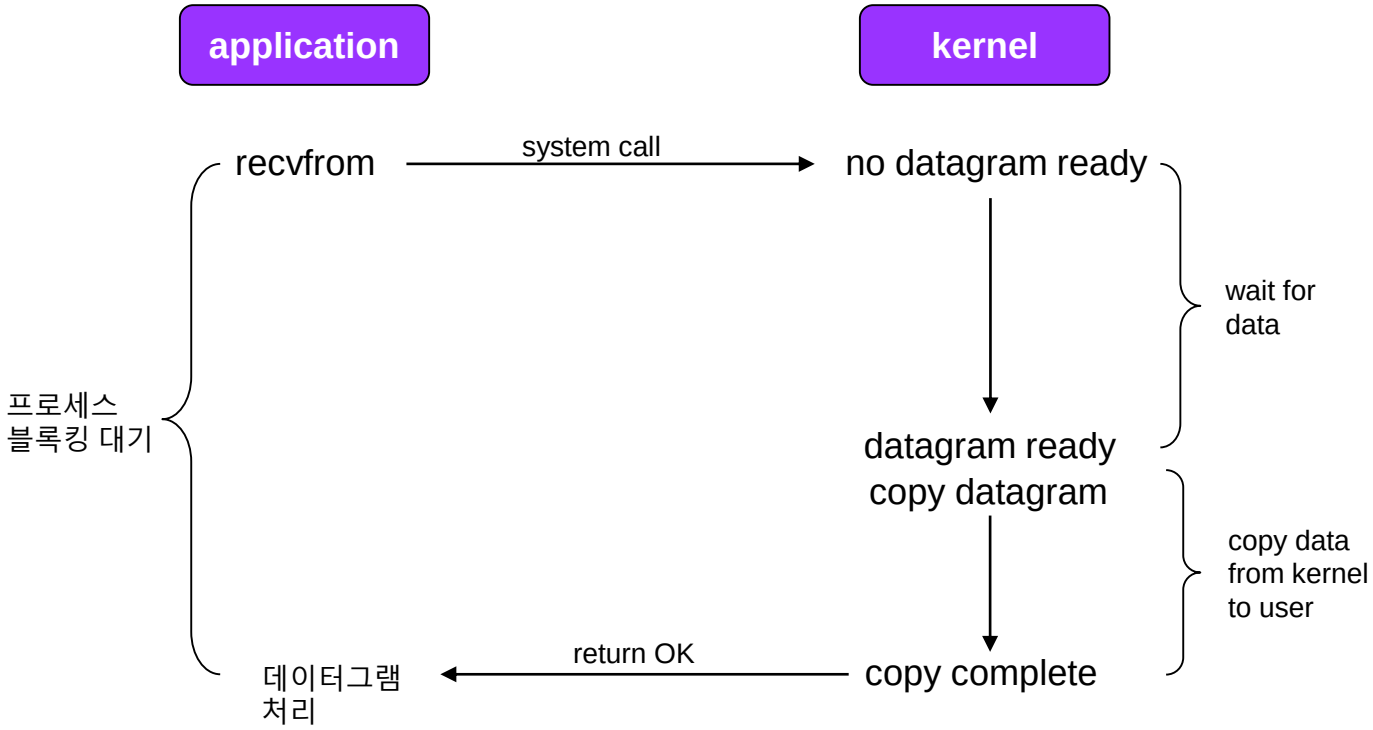
- blocking 모드
  - 어떤 시스템 콜을 호출하였을 때 네트워크 시스템이 동작을 완료할 때까지 그 시스템 콜에서 프로세스가 멈춤
    - 소켓 생성시 디폴트 blocking 모드
  - block 될 수 있는 소켓 시스템 콜
    - listen(),connect(), accept(), recv(), send(), read(), write(), recvfrom(), sendto(), close()
  - I/O시 처리가 될 때까지 기다려야 함
  - 비동기적인 작업 수행 불가능
  - 일 대 일 통신을 하거나 프로그램이 한가지 작업만 하면 되는 경우는 blocking 모드로 프로그램 작성 가능

블록 모드 vs 넌블럭 모드 (계속)

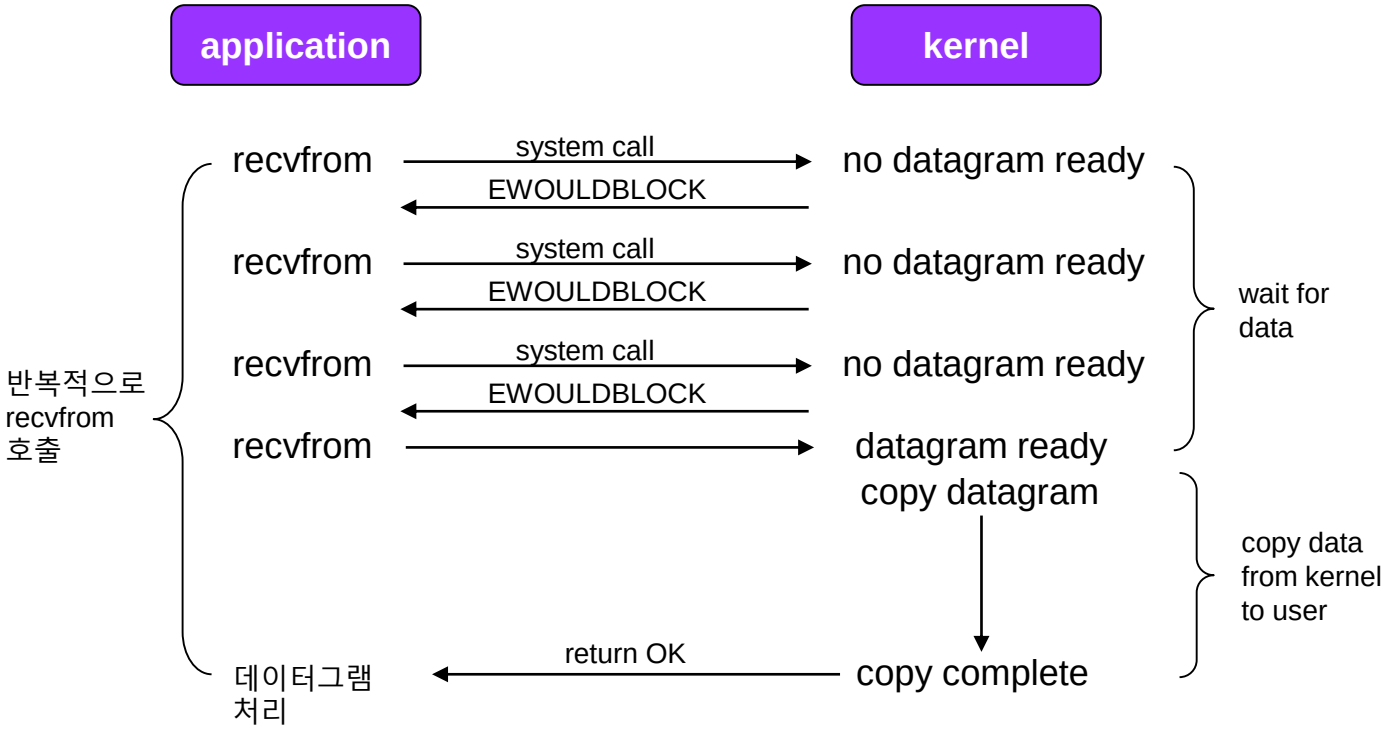
- Non-blocking 모드
  - 소켓 관련 시스템 콜에 대하여 네트워크 시스템이 즉시 처리할 수 없는 경우라도 시스템 콜이 바로 리턴 되어 응용 프로그램이 block되지 않게 하는 소켓 모드
  - 통신 상대가 여럿이거나 여러 가지 작업을 병행하려면 nonblocking 또는 비동기 모드를 사용하여야 함

- non-blocking 모드를 사용 시 동작 방식
  - 넌블럭 모드를 사용하는 경우에는 일반적으로 어떤 시스템 콜이 성공적으로 실행될 때까지 계속 루프를 돌면서 주기적으로 확인하는 방법(폴링)을 사용

Blocking I/O Model



Nonblocking I/O Model



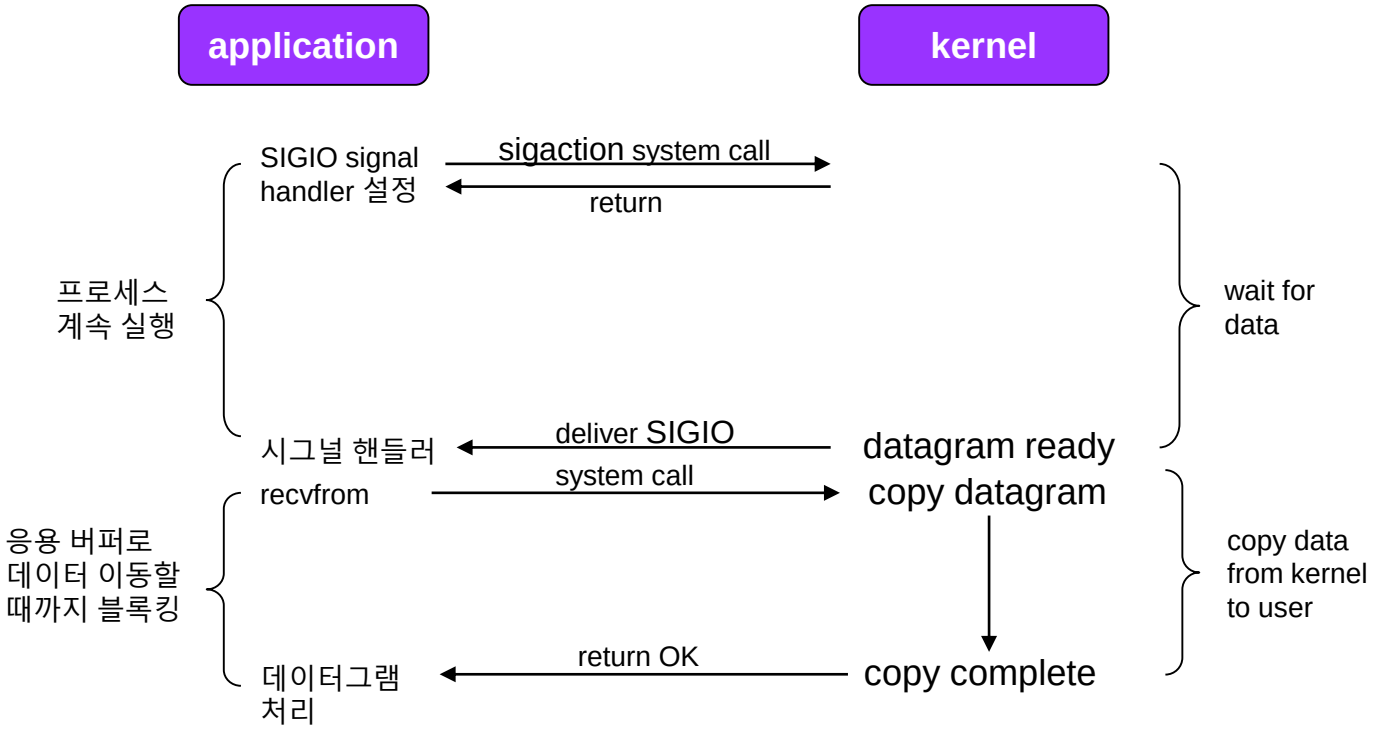
넌블럭 I/O 모델 개요

- 작업이 완료되지 않으면 EWOULDBLOCK(WSAEWOULDBLOC)를 리턴
  - 작업이 진행 중이라는 뜻으로 에러가 아님
- non-blocking I/O 사용법
  - Linux
    - fcntl(sock_fd, F_SETFL, O_NONBLOCK); // linux
  - Windows
    - unsigned long nb_flag = 1;
    - ioctlsocket(sock, FIONBIO, &nb_flag); // nb_flag = 0이면 off
  - 모두 polling을 하여 결과를 확인해야함
    - while (read(..) == EWOULDBLOCK) {}

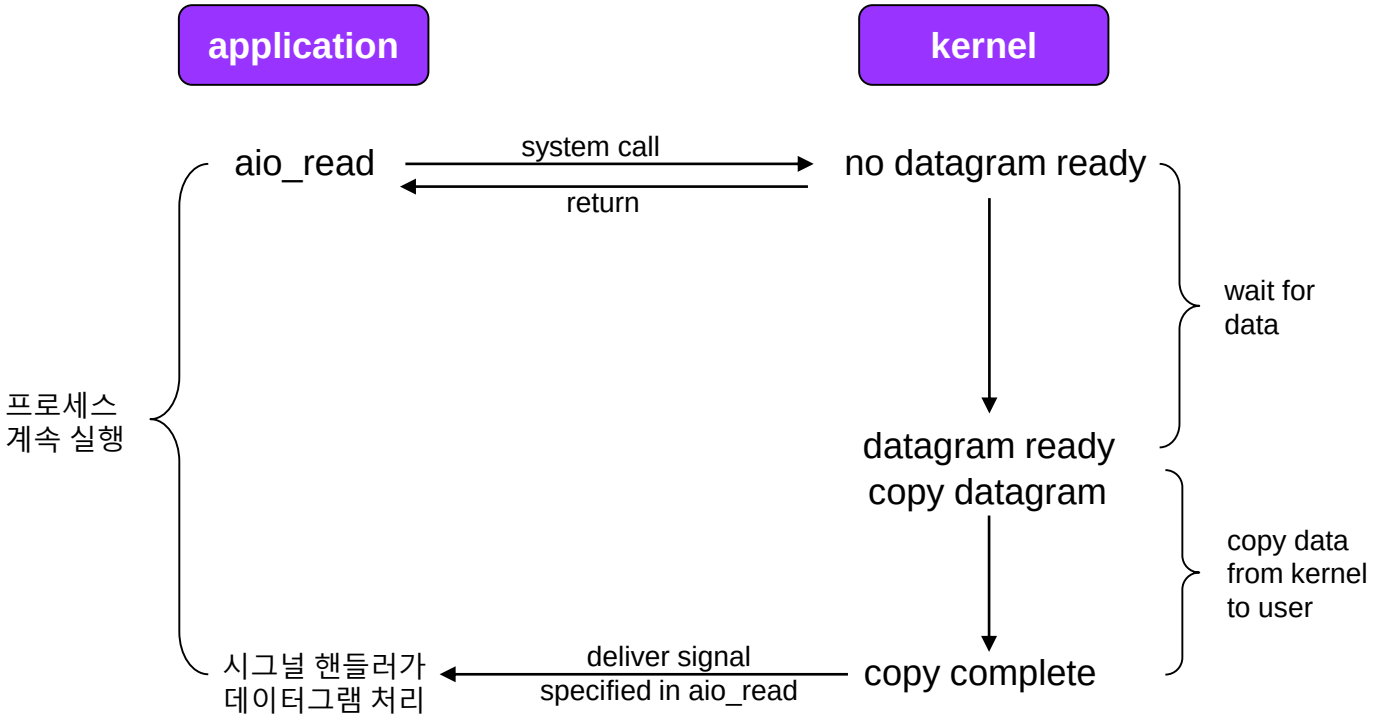
비동기 I/O - 시그널 인터럽트 방식

- SIGIO를 이용한 인터럽트 방식
  - Polling방식과 대비됨
- 소켓에서 I/O 변화가 발생하면 커널이 이를 응용프로그램에게 알려 원하는 동작을 실행
- 동작 과정
  - sigaction()를 이용 인터럽트 시그널의 종류를 시스템에 알림
  - fcntl()을 이용하여 자신을 소켓의 소유자로 지정
  - fcntl()을 통해 소켓에 FASYNC 플래그를 설정하여 소켓이 비동기 I/O를 처리하도록 수정

Signal Driven I/O Model



Asynchronous I/O Model



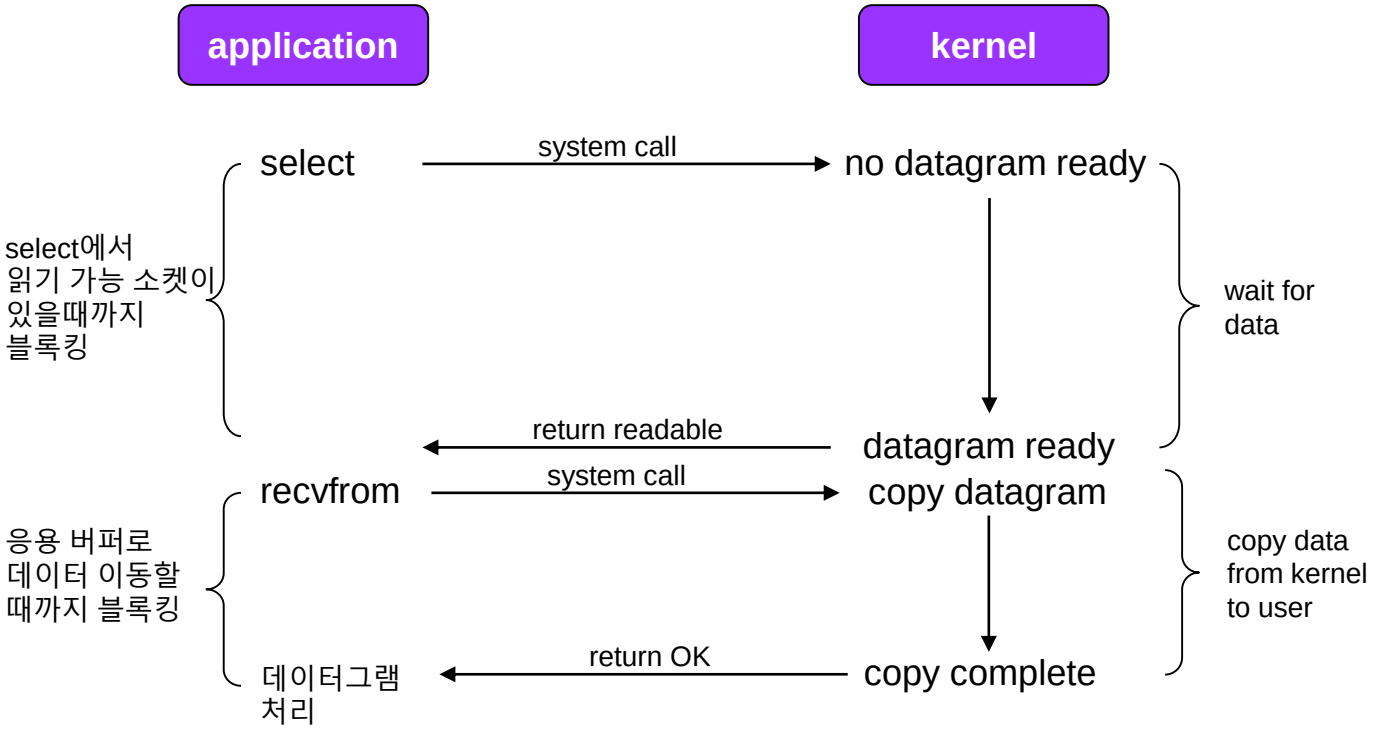
[실험 8] aio 조사 및 샘플 코드 확인

- man 또는 chatGPT로 aio 사용법 조사하여 정리
- man 또는 chatGPT로 aio 샘플 코드 작성 후 내용 정리

select( )를 이용한 멀티 플렉싱

- 멀티 플렉싱 (Multiplexing)이란?
  - 다수의 송수신자가 하나의 전송 채널을 공유하는 방식
- 소켓 프로그래밍에서의 멀티 플렉싱
  - 하나의 프로세스를 이용, 다수의 송수신 채널을 관리하는 방식
  - 주로 select()를 이용하여 처리 (LINUX의 epoll 다음 장에서 소개)
- 멀티 프로세싱 vs 멀티 플렉싱
  - 멀티프로세싱은 다수의 채널을 관리하기 위해 다수의 프로세스를 사용
  - 멀티플렉싱은 하나의 프로세스 내부에서 다수의 채널을 관리
- 용도
  - 멀티 프로세싱: 동시에 여러 채널의 I/O가 일어날 경우, 즉 하나의 프로세스의 실행 기간이 일정시간 지속될 경우
  - 멀티 플렉싱: 프로세스의 I/O처리 시간이 짧아 바로 다음 프로세스의 처리가 가능한 경우

I/O Multiplexing Model



select( )의 활용

- 단일 프로세스에서 여러 fd를 모니터링하는 방법을 제공

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

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

read, write 검사할 최대 fd+1

read(recv)를 감지할 fds

write(send)를 감지할 fds

예외상황을 감지할 fds

Select가 return될 시간
- 0=> 즉시 리턴
- Null => blocking

결과 값
0 : timeout
-1 : error
1 이상 : IO가 일어난 FD의 수

```
struct timeval {
        int tv_sec;
        int tv_usec;
};
```

select() 관련 매크로

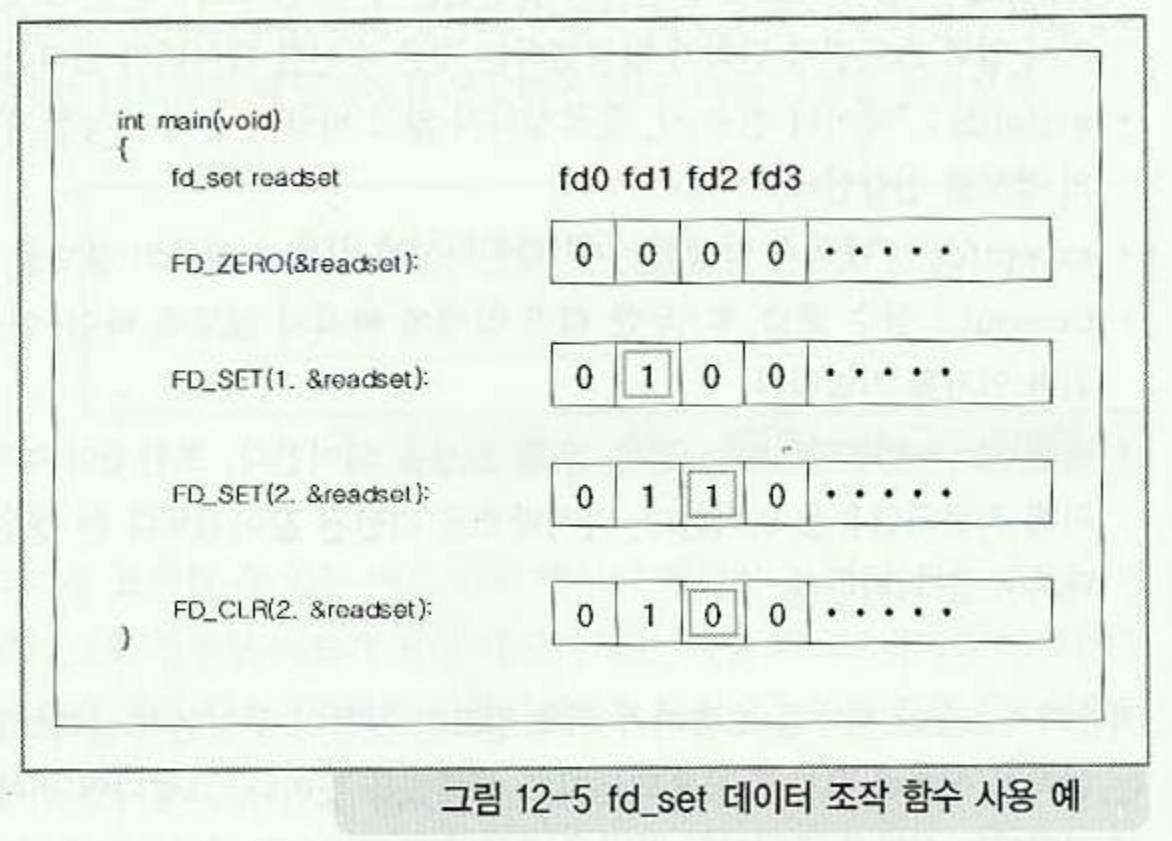


그림 12-5 fd_set 데이터 조작 함수 사용 예

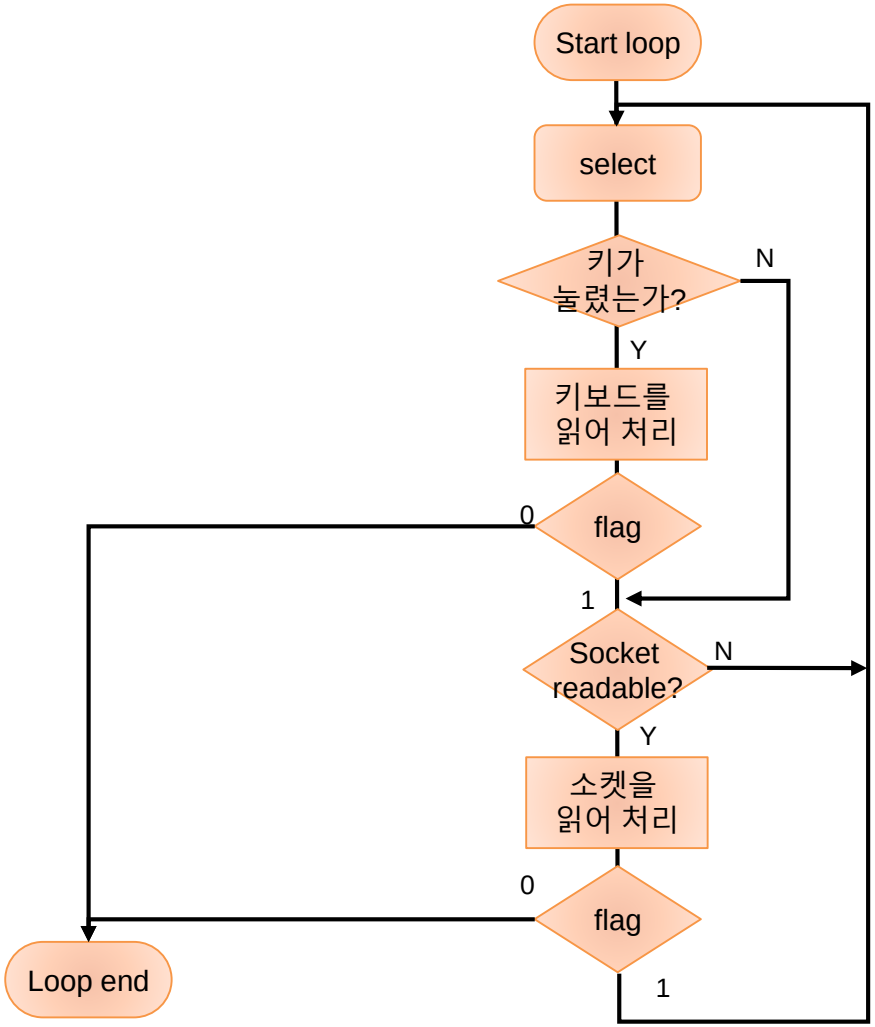
select() 관련 매크로

```
// macro
FD_ZERO(fd_set *set) - 파일기술자 집합을 소거한다
FD_SET(int fd, fd_set *set) - fd 를 set에 더해준다
FD_CLR(int fd, fd_set *set) - fd 를 set에서 빼준다
```

```
main(void)
{
        fd_set rfds;
        struct timeval tv;
        int retval;

        FD_ZERO(&rfds);
        FD_SET(0, &rfds);
        tv.tv_sec=5;
        tv.tv_usec=0;

        retval = select(1, &rfds, NULL, NULL, &tv);

        if(retval)      printf("Data is available now.\n");
        else            printf("No data within 5 seconed.\n");

        exit(0);
}
```

시간 내에 IO의 변화가 있을 경우, 값을 변경

select()를 이용한 키보드 입력과 데이터 수신의 비동기 처리



select()를 이용한 키보드 입력과 데이터 수신의 비동기 처리 (계속)

FD_ISSET(int fd, fd_set *set) - fd가 set안에서 active한지 확인

```
int game_end;
fd_set readOK;
flag=1;

while(1) {
        FD_ZERO(&readOK);
        FD_SET(0,&readOK); /* 표준 입력과 소켓의 디스크립터를 세팅 */
        FD_SET(sock,&readOK);
        select (maxfd+1, (fd_set*)&readOK, NULL, NULL, NULL);

        if(FD_ISSET(0,&readOK)) {
                send();
                …
                if(game_end) break;
        }

        if(FD_ISSET(sock, &readOK) {
                recv(sock, buf, sizeof(buf), 0);
                …
                if(game_end) break;
        }
}
```

하나씩 확인해야 함 !

Loop안에 있음을 확인 !

하나씩 확인해야 함 !